UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

Algoritmos para o Posicionamento de Células em Circuitos VLSI

por

RENATO FERNANDES HENTSCHKE

Dissertação submetida a avaliação, como
requisito parcial para obtenção do grau
de Mestre em Ciência da Computação

Prof. Dr. Ricardo Augusto da Luz Reis
Orientador

Porto Alegre, dezembro de 2002.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Hentschke, Renato Fernandes

Algoritmos para o Posicionamento de Células em Circuitos VLSI / por Renato Fernandes Hentschke. – Porto Alegre: PPGC da UFRGS, 2002.

137 p. : il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Porgrama de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2002. Orientador: Reis, Ricardo Augusto da Luz.

1. Microeletrônica. 2. Ferramentas de CAD. 3. Síntese Física. 4. Posicionamento. 5. Algorithms. I. Reis, Ricardo Augusto da Luz. II. Título.

Sumário

Agradecimentos

Em primeiro lugar, gostaria de agradecer a meus pais. A minha mãe Suzana Fernandes, por seu eterno e interminável carinho e amor. Durante todo o mestrado, me ajudou a superar os momentos de crise, e nos momentos de felicidade soube vibrar junto comigo. A meu pai Carlos Hentschke, que sempre foi um grande conselheiro para o meu sucesso profissional e acadêmico.

Ao meu excelente orientador Ricardo Reis, com o qual já trabalho desde 1998 (5 anos). Desde o começo, sempre foi um ótimo motivador. Depois da entrada no mestrado, mostrou-se um ótimo amigo acima de qualquer coisa, motivando-me a seguir com ele no doutorado. Durante todo este tempo, sempre meu ofereceu grandes oportunidades que foram fundamentais para que eu pudesse aprimorar minha formação.

À minha namorada Sabrina Coelho que, com seu amor, soube me animar sempre que passava por momentos de tensão, soube compartilhar comigo sonhos e principalmente vibrar com as vitórias. Nos momentos adequados, se mostrou totalmente prestativa, assistindo a treinos para apresentações difíceis, me acompanhando em congresso e estando sempre ao meu lado.

Ao meu ex-co-orientador Marcelo Johann por toda a experiência passada durante os 3 anos de bolsista de graduação e pelo modelo de aluno de pós graduação (e também pelo grande tecladista). Desde o começo, nos tornamos bons amigos e companheiros de trabalho. Graças ao conhecimento que recebi dele pude iniciar e concluir um curso de mestrado.

Aos demais membros da minha família. Minha irmã Cristina, meu tio Machado, minha tia Márcia e meu primo Gabriel, que sempre foram bons motivadores e amigos. E especialmente ao meu afilhado Arthur, que está torcendo por mim desde que nasceu, há três anos.

Ao meu amigo Fábio Cecin, que deu excelentes dicas para o aprimoramento do meu trabalho, como as perturbações mistas e cálculo rápido de *wirelength*. Além das eternas discussões sobre qual a melhor linguagem e modelo de programação (com ou sem ponteiros), que sempre enriqueceram meu conhecimento. Sem contar as caronas que dividimos e nos fizeram poupar bastante gasolina neste período do mestrado.

Aos meus companheiros de trabalho do GME, especialmente Diogo Fiorentin, Luigi Carro, José Guntzel, Fernanda de Lima, Felipe Marques, Érika Cota, Fernando Moraes, Cristiano Lazzari, Fernando Paixão Cortes e Alessandro Girardi. Ao Diogo pelas implementações do Cluster Growth para Posicionamento e pela implementação do leitor de arquivos spice com múltiplos níveis de hierarquia. Ao Luigi especialmente pelo grande amigo que mostrou ser, dando ótimos conselhos sobre o mestrado e a vida, além de ser um excelente companheiro de trabalho. Ao Guntzel pelo apoio na preparação da ferramenta de posicionamento. A Fernanda, Luigi e Felipe pelo auxílio na minha publicação mais importante até então. À Érika e Fernanda pela ajuda com o Mentor em pleno verão. Ao Moraes pelas ajuda com relação ao Tropic. À Felipe, Cristiano, Fernando e Alessandro, meus quatro companheiros de trabalho, pela amizade, pelo paddle e pelo nosso futuro que continua no mesmo rumo. Ao Cristiano também cabe um agradecimento relacionado à ferramenta de síntese física que estamos construindo juntos.

Aos meus colegas de graduação, especialmente Filipo Perotto, Eduardo D´Avila, Paulo Zaffari e Tales Heimfarth, velhos companheiros de Truco e Dragão Limão que, além de sempre amigos, são brilhantes computólogos. O Eduardo ainda teve participação direta neste trabalho, ajudando a definir as equações de variação da temperatura. Mas principalmente pelos seus fabulosos desenhos do Dragão Limão e *Mango Parrot*.

Resumo

Este trabalho faz uma análise ampla sobre os algoritmos de posicionamento. Diversos algoritmos são extraídos da literatura e de publicações recentes de posicionamento. Eles foram implementados para uma comparação mais precisa. Novos métodos são propostos, com resultados promissores.

A maior parte dos algoritmos, ao contrário do que costuma encontrar-se na literatura, é explicada com detalhes de implementação, de forma que não fiquem questões em aberto. Isto só foi possível pela forte base de implementação por trás deste texto. O algoritmo de Fidduccia-Mateyeses, por exemplo, é um algoritmo complexo e por isto foi explicado com detalhes de implementação.

Assim como uma revisão de técnicas conhecidas e publicadas, este trabalho oferece algumas inovações no fluxo de posicionamento. Propõe-se um novo algoritmo para posicionamento inicial, bem como uma variação inédita do *Cluster Growth* que mostra ótimos resultados. É apresentada uma série de evoluções ao algoritmo de *Simulated Annealing*: cálculo automático da temperatura inicial, funções de perturbação gulosas (direcionadas a força), combinação de funções de perturbação atingindo melhores resultados (em torno de 20%), otimização no cálculo de tamanho dos fios (avaliação somente das redes modificadas e aproveitamento de cálculos anteriores, com ganhos em torno de 45%). Todas estas modificações propiciam uma maior velocidade e convergência do método de *Simulated Annealing*.

É mostrado que os algoritmos construtivos (incluindo o posicionador do Tropic, baseado em quadratura com *Terminal Propagation*) apresentam um resultado pior que o *Simulated Annealing* em termos de qualidade de posicionamento às custas de um longo tempo de CPU. Porém, o uso de técnicas propostas neste trabalho, em conjunto com outras técnicas propostas em outros trabalhos (como o trabalho de Lixin Su) podem acelerar o SA, de forma que a relação qualidade/tempo aumente.

Palavras-Chave: Microeletrônica, Ferramentas de CAD, Síntese Física, Posicionamento, Algoritmos

TITLE: "ALGORITHMS FOR CELL PLACEMENT IN VLSI CIRCUITS"

Abstract

This work does a wide overview of placement algorithms. Various algorithms were extracted from the literature and from recent publications. They were implemented in a sunique tool for a precise comparison. New methods are proposed, with promising results.

Most algorithms are explained with several implementation details. The Fidducia-Mateyses algorithm, for example, is complex and because of that it was explained with implementation details.

Besides the review of known techniques, this work provides some innovations to the placement flow. A new initial placement algorithm is proposed. Also, new variations over the Cluster Growth algorithm were shown. This work presents some evolutions over the Simulated Annealing algorithm: automatic calculation of the initial temperature, greedy perturb functions (force directed), combination of perturbation functions reaching better results (20% in average), an optimized method for wirelenth calculation (evaluating only the modified nets, with gains around 45% in CPU time). All these modifications increase the speed and convergence of the Simulated Annealing method.

It is shown that the constructive algorithms (including the Tropic placer, based on Quadratic placement using Terminal Propagation) result in a worse wirelength compared with Simulated Annealing. Although, Simulated Annealing spends much more CPU time. This work combined with techniques found in other publications (such as Lixin Su work) points to strategies to optimize the CPU time, increasing the quality/time tradeoff.

Keywords: Microelectronics, CAD Tools, Physical Synthesis, Placement, and Algorithms

Lista de Abreviaturas

Alg.	Algoritmo
BB	Bounding Box
Bench.	Benchmark
CAD	Computer Aided Design
CG	Cluster Growth
Cong.	Congestionamento
CPU	Central Processing Unit
DL	Dragão Limão
DP	Desvio Padrão
E/S	Entrada e Saída
FD	Force Directed
FM	Fiduccia-Matteyses
FPGA	Field Programable Gate Array
KL	Kernighan Lin
LUT	Look Up Table
SA	Simulated Annealing
SCCG	Static CMOS Complex Gates
STL	Standard Template Library
WL	Wire Length

Lista de Figuras

Lista de Tabelas

1 Introdução

A industria de semicondutores está em constante crescimento e a complexidade dos circuitos integrados cresce da mesma forma à medida que as tecnologias vão avançando, com dezenas de milhões de transistores fazendo parte de um único circuito integrado. Ao mesmo tempo, o mercado exige que o tempo de projeto seja o menor possível. Juntando estes dois fatores percebe-se que é inviável o projeto de um CI (circuito integrado) completo sem o uso de ferramentas de CAD que automatizem este processo. Uma ferramenta de CAD de qualidade pode reduzir drasticamente os custos de projeto a medida que o tempo de projeto é reduzido.

Várias etapas de projeto podem ser automatizadas. Há décadas estudam-se algoritmos para ferramentas de CAD. Porém, conforme avança a tecnologia, são lançados novos desafios. Os problemas se tornam mais complexos, pois mais elementos são introduzidos. Diversas questões elétricas, que eram desprezadas em tecnologias anteriores começam a ser consideradas. Por exemplo, efeitos de uma conexão sobre outra, integridade do sinal, distribuição do consumo de potência, freqüência de relógio como um forte limitador, indutância, etc.. Por isto, tanto a indústria quanto a academia preocupam-se constantemente em atualizar-se. Observa-se, porém, que a pesquisa ainda é constante e muitas questões ainda estão em aberto. Há uma forte demanda por novas soluções.

O projeto de um circuito pode começar no nível de sistema, onde descreve-se o sistema completo (hardware e software, sem distinção) em uma linguagem de alto nível. A partir daí, são feitas diversas etapas de síntese, que visam tornar a descrição do circuito cada vez mais detalhada, mais próxima da descrição física de um circuito integrado em tecnologia CMOS. Entre elas, destacam-se a síntese de alto nível, síntese lógica e por fim síntese física. Esta última tem a função de gerar as máscaras do leiaute de forma que o circuito esteja pronto para fabricação.

São diversos os estilos de projeto, que dependem muito do estilo de leiaute alvo. Projetos comerciais de circuitos de alto desempenho são feitos, muitas vezes, parte manualmente e parte automaticamente. Projetos feitos preferencialmente a mão, em geral, não apresentam os padrões geométricos que são necessários em leiautes automatizados. Porém, projetos com menor exigência de *timing* e maior exigência de tempo de projeto podem ter praticamente todas as tarefas automatizadas.

O padrão mais usado para automatizar a síntese física é a utilização de células (estilo conhecido como *cell-based*). Há a formação de bandas de células. Uma banda de célula possui células de altura fixa. A partir daí, uma das duas seguintes metodologia é adotada: uso de bibliotecas de célula pré-desenhadas ou geração automática do leiaute. O estilo de leiaute *standard-cell* utiliza uma biblioteca de células de altura fixa. A vantagem de usar *standard cell* é conhecer as características elétricas das células previamente e precisamente. Em leiaute "*on-the-fly"* são utilizadas estimativas, que não são tão precisas quanto às medidas elétricas feitas em uma biblioteca pré-caracterizada, como *standard-cell*. Porém, permite o uso de portas com qualquer função lógica. Portas CMOS complexas, que são muito usadas nos projetos modernos devido a economia de área, atraso e potência dissipada, são a principal motivação para geração *on-the-fly*.

Os problemas de síntese física são, para estilos baseados em célula, cinco: planejamento topológico, particionamento, posicionamento, geração do leiaute e roteamento. O planejamento topológico é responsável por posicionar macro-blocos e macro-células, que são retângulos sem nenhum padrão de altura ou largura. Ainda é um problema do planejamento topológico o posicionamento de pinos de entrada e saída. Em seguida vem o particionamento dos elementos que compõem o circuito. As células que compõem o circuito são dividas entre os blocos. O objetivo é que fiquem no mesmo grupo as células que tem conexões ente si, minimizando o cruzamento de fios entre as partições. O posicionamento é tratado em separado para cada partição. O seu problema é alocar um espaço físico para cada célula dentro da partição, formando uma macro-célula. Em seguida vem a geração do leiaute. Por fim, a etapa de roteamento deve fazer todas as conexões.

Todos os problemas de síntese física são problemas de otimização, ou seja, que buscam encontrar a solução ótima em um conjunto de possíveis soluções. Considere o problema de planejamento topológico. Sua solução é trivial se for desconsiderada a otimização do circuito. Porém, a qualidade da solução do *floorplanning* influencia diretamente a dificuldade dos problemas seguintes, podendo até mesmo inviabilizá-los. Além disto, quanto mais otimizado o *floorplanning*, menor será a área do leiaute final. Além da área, o desempenho e o consumo de potência do circuito são diretamente afetados. Já no problema de roteamento, por outro lado, o maior desafio é encontrar uma solução válida, não necessariamente a ótima ou sub-ótima. Encontrar um roteamento válido é uma tarefa bastante complexa. Porém, sabe-se que o atraso de um circuito integrado está relacionado fortemente com o tamanho das conexões. Então é importante que o roteamento seja bem realizado, tentando equilibrar o tamanho das conexões. Seja qual for o seu objetivo, o processo de roteamento é totalmente dependente das etapas anteriores.

O problema de posicionamento é de fundamental importância para qualquer estilo de leiaute e tecnologia. Nas tecnologias modernas o atraso de um circuito causado pelas conexões é da mesma ordem de grandeza que o atraso causado pelas células. O posicionamento ocorre antes do roteamento, sendo que ele define a posição de todos os contatos que devem ser roteados. Por isto, o roteamento é plenamente dependente do posicionamento. Um mau posicionamento pode, e em muitos casos irá, inviabilizar o roteamento. Além disto, o tamanho das conexões entre as células é definido pelo posicionamento, o que influencia fortemente a freqüência de operação e consumo de potência do circuito.

O problema de posicionamento é definido, como já foi dito, simplesmente pela alocação de um espaço físico para cada elemento do circuito, desde que não haja intercessão de área entre dois ou mais elementos. Embora seja simplesmente definido e trivialmente solucionado não é trivialmente otimizado. Como a maior parte dos problemas de otimização, encontrar o posicionamento ótimo é classificado com np-completo ou np-hard (como mostra Garey). Para evitar algoritmos de complexidade exponencial que podem requerer até séculos de processamento, utilizam-se heurísticas. Desta forma não se está focado na solução ótima em si, mas em uma aproximação razoável.

A área de posicionamento é bastante pesquisada há bastante tempo (como os trabalhos de Gajski, Grover, Kling, Lam, Preas, Sechen, Malleta) A pesquisa por novas heurísticas é muito importante devido ao aparecimento de novas considerações ao posicionamento, conforme observa-se nos trabalhos de Madden (2002), Yang (2002), Wu (2001), Chen (2000), Caldwell (2000b), Caldwell (2000c) e outras publicações como demonstra a sessão 12). Patrick Madden observa que não há um consenso de qual o melhor algoritmo, a melhor técnica de posicionamento. Há uma necessidade de pesquisa por novas soluções em posicionamento, que possam tratar a grande quantidade de elementos que envolvem os projetos modernos, otimizando todos os parâmetros de interesse do projetista.

1.1 Objetivos deste Trabalho

Este trabalho busca dar uma visão geral dos algoritmos de posicionamento existentes na literatura e em artigos recentes, refletindo a novas questões e objetivos de posicionamento. Mostra-se que as principais idéias são mantidas do passado, porém acrescentam-se novos elementos, como estimativas de *timing*, congestionamento e potência. Há, porém, na literatura como um todo, uma indefinição de qual algoritmo de posicionamento é o melhor. Os trabalhos mais recentes variam bastante na escolha do algoritmo base.

O texto procura dar um panorama de trabalhos novos que estão tentando resolver estes problemas. São apresentadas variações em algoritmos e novas modelagens de timing, congestionamento e potência, bem como referências a trabalhos que aprofundam a sua análise.

Os algoritmos mais utilizados para posicionamento nos dias de hoje são baseados em idéias do passado, de décadas atrás. Este trabalho procura resgatar algumas técnicas que ainda são

utilizadas, através de uma pesquisa detalhada e da implementação dos algoritmos. Os algoritmos de posicionamento, da maneira que são apresentados na literatura, não descrevem como tratar exceções que devem ser totalmente especificadas na implementação. Desta forma, este trabalho busca dar uma visão de "baixo nível" dos algoritmos, observando detalhes de implementações e estruturas de dados. Para alguns algoritmos, são propostas alterações que visam melhorar sua eficiência e convergência. Um novo algoritmo, ainda em fase de estudo, é apresentado. São feitas comparações entre algoritmos usando diversos benchmarks ISCAS´89, conforme o anexo 1.

Todas as execuções do programa são feitas usando a ferramenta de posicionamento do *Mango Parrot* (*Mango Parrot Didactic Placement*). Nela, foram implementados praticamente todos os algoritmos de posicionamento discutidos neste trabalho, de forma que seja possível fazer uma comparação usando métricas idênticas para cada algoritmo. A ferramenta é capaz de desenhar o leiaute resultante de um determinado posicionamento, usando um gerador de leiaute e um roteador próprio. Desta forma, as estimativas são comprovadas na prática, dando uma maior credibilidade aos dados obtidos. A ferramenta faz parte do projeto FUCAS de leiaute automatizado, que é visto na próxima sessão.

São usados dois sistemas para gerar o leiaute a partir do posicionamento: TROPIC e LDCG. O Tropic (apresentado por Moraes em 1999) é um sistema de geração de leiaute automático partindo-se de uma descrição em nível lógico mapeado. Ele inclui uma etapa de posicionamento, de geração do leiaute e por fim de roteamento em um único sistema. Neste trabalho, porém, trabalha-se com uma versão modificada do Tropic, que permite o posicionamento ser realizado por um sistema externo, no caso, o *Mango Parrot*. O LDCG (apresentado por Lazzari em 2002) é um gerador de leiaute em desenvolvimento, cujo objetivo é gerar leiaute dentro do projeto FUCAS. Ele cobre somente a etapa de geração das células, previamente posicionadas (pelo *Mango Parrot Didactic Placement* por exemplo). Exige um roteador para o desenho das conexões. O LDCG usa o estilo de leiaute conhecido por *linear matrix*, sem canais de roteamento entre as bandas. Todo o roteamento será feito sobre as células. O primeiro nível de metal é usado para conexões internas da célula apenas. O segundo é usado para conexões internas, alimentação e para algumas conexões externas. As linhas de alimentação passam sobre os transistores, facilitando a inserção de contatos na periferia da célula. Os demais níveis de metal estão livres para roteamento. A área é minimizada através da união do leiaute de duas células, como pode ser visto no exemplo da figura 1.

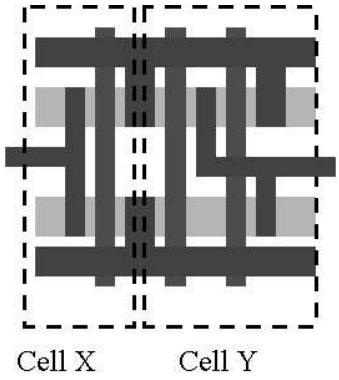

FIGURA 1– União do leiaute de duas células

Observe na figura 1 que a célula X compartilha a mesma difusão com a célula Y. Isto só ocorre porque a X termina com uma conexão com a alimentação, enquanto que a Y começa desta maneira. O algoritmo de procura pelo caminho de Euler, na geração do leiaute, deve levar isto em conta.

O último passo da síntese física é o roteamento. O algoritmo implementado é um *maze router* muito utilizado e conhecido. Ele usa o algoritmo A*, apresentado por Hart em 1968, e árvores de Steiner. A vantagem dele é sua generalidade, permitindo o uso de *n* níveis de metal e

inclusive usando níveis ocupados por quaisquer tipos de obstáculos. Para circuitos grandes, porém, este algoritmo não deve funcionar adequadamente devido à falta de roteamento global. Assim, usando tecnologias de ponta, com metal 3 e 4 disponíveis para roteamento, é possível usar o sistema GURIA (para roteamento global) e LEGAL (para roteamento detalhado), que é discutido em diversos trabalhos de Hentschke (2001), Johann (2001 e 2002).

1.2 Projeto FUCAS (texto extraído de www.inf.ufrgs.br/gme)

"O projeto trata do desenvolvimento de um sistema de síntese automática de leiaute que forneça ao projetista a possibilidade de implementar no silício as equações lógicas na sua forma mais otimizada, sem ter que restringir o mapeamento às células existentes em uma biblioteca de células. Para tanto será desenvolvido um sistema de síntese de leiaute baseado em um gerador automático de células original segundo a metodologia TRANCA. Pretendemos com uso deste sistema otimizar o número de transistores de um circuito, diminuindo área, consumo e possibilitando o tratamento dos caminhos críticos, de forma a aumentar a freqüência máxima de operação do circuito em desenvolvimento.

O sistema a ser desenvolvido estará apto a tratar de tecnologias submicrônicas, com até 8 camadas de metal e com contatos e vias empilhados. O leiaute do circuito será gerado automaticamente de forma a atender às especificações de desempenho, consumo e área. Este sistema terá interface com sistemas de CAD comerciais, permitindo a síntese automática a partir de uma descrição comportamental do circuito."

1.3 Ferramenta de posicionamento do *Mango Parrot*

O processo de síntese, como já foi visto, se divide em Posicionamento, Geração do Leiaute e Roteamento. No passado, foi desenvolvido o primeiro trabalho de síntese física do Instituto de Informática da UFRGS, como a de Lubaszewski em 1990. Desde lá, outras ferramentas foram e estão sendo desenvolvidas, como é o caso do *Mango Parrot Didactic Placement*.

A ferramenta de posicionamento oferece ao usuário vários algoritmos e parâmetros que podem ser explorados didaticamente. Alguns exemplos são: número de iterações, variação da temperatura (para o Simulated Annealing), algoritmo de particionamento (para posicionamento em Quadratura), função de estimativa de fios e muitos outros, como detalhado ao longo deste texto. Desta forma, modificando estes parâmetros e tentando diferentes algoritmos, o usuário pode fazer experiências e ver, em uma interface gráfica, os resultados. O código da ferramenta é feito em C++, usando recursos de templates e orientação a objetos. Do ponto de vista de pesquisa, a ferramenta é muito interessante por permitir facilmente que novos algoritmos sejam inseridos e testados em um fluxo real de síntese.

Além do aspecto didático, a ferramenta é eficiente para a síntese real de circuitos. A interface do programa é bastante intuitiva, fazendo com que ele seja fácil de aprender e usar. A entrada da ferramenta é um arquivo que descreve os transistores e suas ligações, no formato spice. Diferentemente do estilo standard-cell, todas as células são geradas automaticamente. Isto permite a geração de circuito contendo qualquer tipo de porta CMOS complexa (SCCG). Ainda, o processo posterior de geração de leiaute tem mais liberdade, podendo juntar células economizando quebras na difusão. O usuário está livre para ajustar o tamanho dos transistores e estilo de disposição das tiras de polissilício que compõem o transistor.

A etapa de posicionamento começa com a execução de um algoritmo construtivo, que pode ser escolhido entre os seguintes: Aleatório, Plic-Plac, Cluster Growth, Force Directed Placement (em uma versão construtiva) e Posicionamento Baseado em Particionamento. Depois de gerado o posicionamento inicial, refina-se ele com algum algoritmo iterativo. Foram implementados: Simulated Annealing, Algoritmos Genéticos, Force Directed Placement (em uma versão iterativa) e Greedy (Guloso). O posicionamento é feito antes da geração do leiaute e, por

isto, a largura das células não é conhecida. Assim, este parâmetro deve ser estimado com base no número de transistores da célula.

1.3.1 Detalhes de Implementação & Estruturas de Dados

A ferramenta de posicionamento do *Mango Parrot* está sendo implementada em C++, no ambiente Borland C++ Builder 5.0 (para Windows). A modelagem das estruturas do circuito é feita com classes C++. Na seqüência serão mostradas as principais classes da ferramenta para que o leitor possa ter uma idéia mais concreta de como os resultados de simulação apresentados ao longo deste texto foram obtidos.

Na figura 2 há um diagrama das classes de representação do circuito e sua hierarquia.

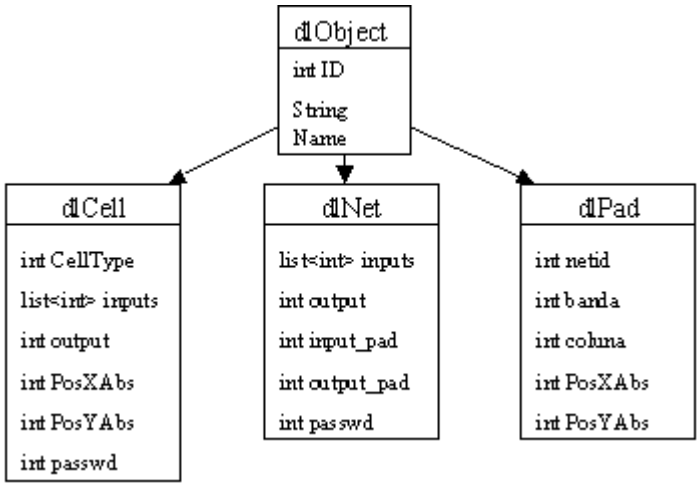

FIGURA 2 – Diagrama de classes de representação do circuito

Observe que há uma classe base, chamada de objeto. Nela estão armazenados os dados em comum às células, redes e pads. São eles: o ID, que é um identificador inteiro e único para cada instância de redes, células e pads; o nome da instância. O nome é um String, classe da biblioteca STL do C++.

A classe de célula guarda os seguintes dados:

- CellType: dá o tipo de célula da instância. Os tipos são definidos pelo arquivo de entrada da ferramenta. Exemplos comuns são: Nand2, Nor2, Inversor.
- list<int> inputs: é uma lista de IDs de redes de entrada da célula. A classe list<> é da biblioteca STL do C++;
- int output: é o ID da rede de saída da célula. Refere-se à rede de saída da célula
- int PosXAbs, int PosYAbs: indica a posição física absoluta da célula. Este valor é calculado em função da posição relativa da célula, que é informada pela classe dlPosic.
- int passwd: é um identificador utilizado para registrar quando foi a última atualização de alguma informação referente a esta célula. É usado para que algoritmos não repitam o mesmo procedimento diversas vezes.

A classe de rede possui os seguintes dados:

- list<int> inputs: é uma lista de IDs de células de entrada que recebem esta rede como entrada. Observe que uma rede pode alimentar várias entradas de células, mas somente uma saída.
- int output: é o ID da célula de saída da rede.
- int input_pad: é o ID do pad de entrada. Este valor somente será válido quando o campo *output* (descrito acima) for inválido (-1).
- int output_pad: é o ID do pad de saída. Um pad de saída faz o papel de uma célula de entrada. Se o valor for –1, a rede não está ligada a pads de saída.

A classe de pad possui os seguintes dados:

- netid: é o ID da rede conectada ao pad.
- int banda: indica a qual banda o pad pertence. Na ferramenta do *Mango Parrot*, os pads ocupam uma banda inteira. Não há restrições, porém, quanto a um pad ocupar o mesmo espaço que outro.
- int coluna: indica a qual coluna (tamanho igual a altura de uma banda) pertence o pad.

Além das classes para representar o circuito, outras classes fazem o seu gerenciamento. Elas estão representadas na figura 3.

FIGURA 3 – Classes para gerenciamento de recursos

A classe dlCellTypes armazena os dados de cada tipo de célula. Desta forma, a instância de uma célula precisa apenas armazenar um identificador de tipo, que é usado como índice para conhecer os seguintes dados:

num_ins: o número de entradas da célula
color: a cor da célula (para desenhar na tela)
width: a largura da célula (em micro metros)
descr: a descrição, em formato spice, dos transistores da célula
nome: o nome do tipo da célula.

A classe dlObjManager guarda todas as instâncias de células, redes e pads. Basicamente, ela é um ponto único de armazenamento de ponteiros para objetos, para que, quando um objeto for apagado, haja somente um lugar para atualizar. Desta forma tem-se um código menos suscetível a falhas. A classe oferece métodos para o retorno de ponteiros através da ID do objeto desejado.

O programa ainda contém classes de armazenamento do Posicionamento. Estas classes são observadas na figura 4.

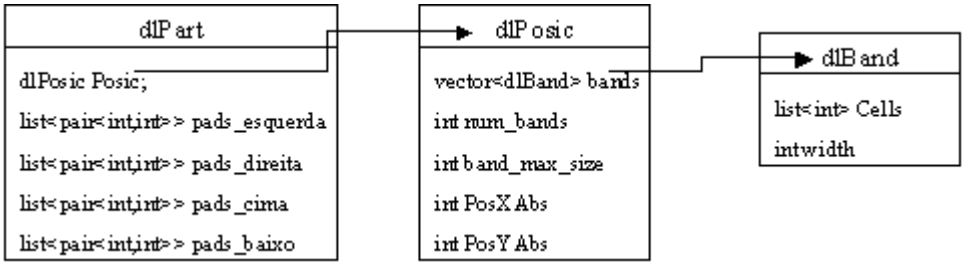

FIGURA 4 – Classes de Posicionamento

A classe dlPart representa uma partição do circuito, ou um bloco, que deve ser gerado automaticamente. Os dados dela são os pads e um ponteiro para o seu Posicionamento correspondente.

A classe dlPosic contém os seguintes dados:

- vector<dlBand> bandas: é um vetor de bandas. Assim, o número de bandas é fixo no início do processo. Por ser um vetor, há um acesso rápido a qualquer banda.
- int num_bands: é um inteiro que diz quantas bandas estão no vetor de bandas descrito acima.
- int band_max_size: diz qual a largura máxima de uma banda.

Por fim, a classe dlBand contém os seguintes campos:

- list<int> cells: é uma lista encadeada de células. Por ser uma lista, não possui um tamanho fixo. Esta estrutura facilita a inserção e remoção de células de uma banda. Ainda, permite que facilmente se manipulem células de tamanhos variados. A desvantagem é que não há um acesso rápido a todos os elementos, como no caso de um vetor.
- int width: informa o tamanho da banda. É calculado pela soma das larguras de todas as células. Como este cálculo é demorado, a variável armazena o valor que pode ser facilmente atualizado no caso de uma inserção ou remoção.

2 Problemas Computacionais e Algoritmos de Propósitos Gerais

2.1 Classificação de problemas computacionais

Problemas de computação podem ser classificados pela complexidade do melhor[1] algoritmo que é capaz de resolvê-lo, como mostra Garey em 1979. Baseado neste critério, temos o conjunto de problemas P, para o qual existe um algoritmo determinístico e polinomial capaz de resolver este problema. Em síntese física, alguns problemas específicos de roteamento, como *river routing*, pertencem a este conjunto. O conjunto NP é formado por problemas para os quais existe um algoritmo não-determinístico capaz de resolvê-lo em tempo polinomial. Todo algoritmo determinístico é também não-determinístico (i.e. $P \subseteq NP$). Não foi provada a existência de problemas NP que não sejam P, ou seja, que não exista um algoritmo determinístico polinomial, porém todas as pesquisas indicam que existam. O teorema de Cook, encontrado em Garey 1979, prova que, se um determinado problema (SAT) classificado como NP for resolvido por um algoritmo polinomial determinístico, haverá um outro algoritmo P para todos os demais problemas NP, ou seja P seria igual a NP. Porém, depois de décadas de pesquisa na área ainda não se encontrou um algoritmo P que fosse capaz de resolvê-lo. Assim, a hipótese que $P \subset NP$ é mais forte. Depois do problema SAT, apresentado por Cook, diversos outros problemas NP foram provados com a mesma característica através de um mapeamento em tempo polinomial destes problemas para o SAT. Problemas desta classe são chamados de np-completo. A figura 5 mostra o diagrama destes conjuntos.

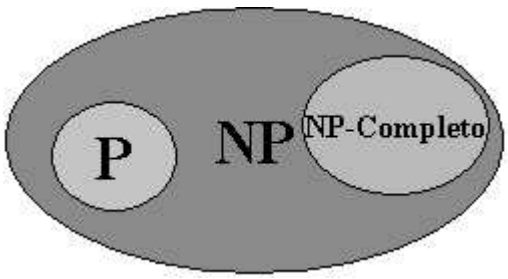

FIGURA 5 – Problemas P,NP e NP-Completo

Como é impraticável a implementação de algoritmos não-determinísticos[2], os únicos algoritmos possíveis em máquinas reais e que são capazes de resolver problemas np-completos são de tempo exponencial. Isto pode significar que, dependendo do tamanho da entrada (no caso de posicionamento, a quantidade de células), o processo pode durar até séculos de

[1] Considera-se o melhor algoritmo aquele que tem a menor complexidade. Uma complexidade é dita menor que outra quando a ordem do polinômio for menor (no caso de complexidades polinomiais). Uma complexidade exponencial é sempre maior que qualquer outra.

[2] Neste texto, o termo "algoritmos não-determinísticos" tem dois significados. Neste capítulo, discute-se algorítmos não-determinísticos do ponto de vista de informática teórica, conforme Garey. Neste contexto, estes algoritmos podem-se dividir em infinitos processos paralelos, o que torna a sua implementação impraticável. Mais adiante neste texto, são apresentadas implementações de algoritmos não-determinísticos, que se referem a algoritmos probabilísticos.

processamento. Algoritmos apropriados para resolver problemas np-completos são vistos nas sessões 2.4, 2.5 e 2.6.

2.2 Problemas de otimização

Uma classe de problemas de computação é caracterizada por buscar um valor ótimo para uma determinada função. Este problema é equivalente a percorrer uma árvore inteira em busca da folha que melhor satisfaça uma determinada função de avaliação. Um algoritmo não-determinístico polinomial para resolver este problema é trivial: basta disparar um processo para cada ramo da árvore e escolher a melhor folha. Assim, este problema é NP. Porém, tipicamente, o melhor algoritmo determinístico para resolver este problema é uma busca exaustiva, de tempo exponencial, em uma árvore n-ária de possibilidades. Assim, na maior parte dos casos, problemas de otimização são np-completos.

Por exemplo, encontrar o posicionamento ótimo dado um *netlist* de células é um problema np-completo.

2.3 Busca Exaustiva

A busca exaustiva é o algoritmo mais genérico para solução de um determinado problema de otimização. É simplesmente uma varredura de todas as possibilidades, sendo que ao fim é escolhida a melhor. A busca exaustiva assemelha-se a percorrer uma árvore inteira em profundidade. Como já foi mencionado anteriormente, sua complexidade de tempo é exponencial. Em problemas pequenos, este tempo é, em geral, aceitável. Considerando o problema de posicionamento, por exemplo, é comum (conforme mostra a sessão 5.4 deste trabalho) o uso de particionamento para que se divida o problema de posicionamento em partes menores, de menor complexidade. No trabalho de Caldwell em 2000, usa-se, com sucesso, esta técnica para particionar e posicionar poucas células provenientes de um grande processo de particionamento.

Muitas vezes, porém, não é aceitável trabalhar-se com algoritmos exponenciais. O tempo de execução gasto por um algoritmo exponencial pode ser bastante elevado, dependendo do tamanho da entrada. Mas, conforme cresce o tamanho do problema, o tempo cresce exponencialmente. Em um caso extremo, o tempo de resposta de um algoritmo exponencial pode ser de séculos, ou até milênios. Para acelerar o processo, diversas técnicas são empregadas. É comum que se busque, em tempo polinomial, uma solução dentro de um conjunto de soluções sub-ótimas.

2.4 Heurísticas

Heurísticas são métodos que não garantem que seu resultado será uma solução ótima. São, porém, mais eficientes que as pesquisas exaustivas. Em geral, exploram características específicas do problema e resolvem ele de uma forma simplificada, de modo que se atinja uma solução rapidamente. Por exemplo, considere uma famosa heurística para resolver o problema do posicionamento que chama-se Force Directed Placement (sessão 7). Nela, supõe-se que posicionando as células seguindo a sua ordem de conectividade obtêm-se um mínimo custo global. Porém, para descobrir a melhor ordem de fato, é preciso fazer uma pesquisa exaustiva. A heurística é claramente muito mais rápida e baseia-se em um conhecimento prévio, específico do problema. A isto se chama heurística. Quanto mais características do problema forem agregadas ao algoritmo heurístico, melhor e mais eficiente será o algoritmo.

2.5 Algoritmos Gulosos

Algoritmos gulosos são um tipo de heurística aplicável a diversos problemas de otimização. Caracterizam-se por procurar mínimos locais, por tomarem decisões precipitadas, baseadas apenas em uma iteração do algoritmo. A figura 6 mostra um exemplo de algoritmo guloso de pesquisa em grafos. Observe que o algoritmo toma a decisão do caminho a seguir baseado apenas no próximo nodo, sem olhar para possibilidades futuras. O caminho ótimo teria um custo total de 3, enquanto que o caminho escolhido tem um custo global de 8. Isto ocorre porque um algoritmo guloso é incapaz de voltar atrás, desfazendo um passo. A decisão, no começo do processo, pelo caminho aparentemente mais promissor comprometeu os passos seguintes. Neste caso, disse que o algoritmo ficou preso a um **mínimo local**, pois no primeiro passo houve a escolha pelo caminho de menor custo local. No entanto, a escolha não foi pelo caminho de menor custo global, fazendo com que o algoritmo não tenha encontrado o **mínimo global**.

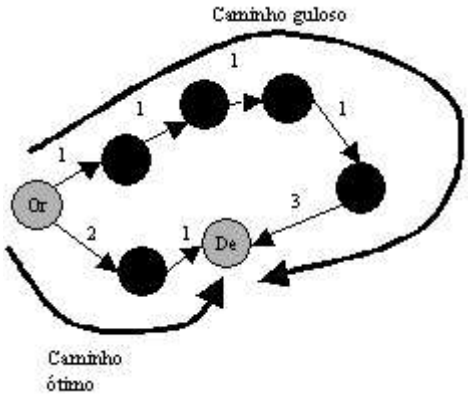

FIGURA 6 – Exemplo de algoritmo guloso na pesquisa por caminhos em um grafo

Aplicando-se um algoritmo guloso para posicionamento, pode-se pensar em um método que muda a posição de uma célula para um lugar em que diminua o comprimento global dos fios. Porém, o algoritmo não seria capaz de fazer um movimento que piore o custo de comprimento dos fios temporariamente, com um movimento futuro que atinja o mínimo global. Neste caso, diz-se que o algoritmo ficou preso em um mínimo local. Em muitos casos, porém, o algoritmo guloso tem uma convergência maior a boas soluções por não perder tempo com soluções pouco promissoras.

2.6 Meta Heurísticas

Meta-heurísticas são métodos heurísticos genéricos, ou seja, podem se aplicar a qualquer problema de otimização. São buscas não exaustivas, de tempo polinomial que usam métodos gulosos para convergir a uma otimização global. Para evitar mínimos locais elas exploram funções aleatórias. Para tanto, algumas meta-heurísticas procuram "imitar" processos de otimização naturais, como, por exemplo, a cristalização de metais pelo processo de têmpera (2.6.1).

Todas as meta-heurísticas são iterativas, iniciando por uma solução inicial, gerada aleatoriamente, por exemplo. São usadas funções aleatórias para modificar a solução intermediária. Porém, nem todas as modificações são aceitas. Existe uma função de avaliação que irá selecionar quais modificações serão concretizadas. Este processo é repetido por diversas iterações, até que se tenha uma solução final satisfatória. A figura 7 ilustra este método.

FIGURA 7 – Meta heurísticas vistas em um fluxograma

Observe que, a cada iteração, o algoritmo transformou uma solução válida em outra válida através de uma modificação aleatória. Esta modificação, portanto, merece um cuidado especial para que não gere soluções intermediárias inválidas. Em posicionamento, uma solução inválida seria onde houvesse sobreposição de duas células, por exemplo.

Este texto buscou investigar duas meta-heurísticas: Simulação de Têmpera (Simulated Annealing) e Algoritmos Genéticos. Elas foram escolhidas por serem as mais utilizadas para posicionamento.

2.6.1 Simulação de Têmpera

O Simulated Annealing (SA) faz a analogia ao processo de cristalização dos metais. Na natureza, este processo consiste no seguinte: inicialmente se faz um aquecimento do metal para altas temperaturas. Neste estágio, as moléculas estarão agitadas, obedecendo a nenhuma organização. Aos poucos, o metal é submetido a um resfriamento, para que, quando a temperatura for suficientemente baixa, as moléculas estejam arranjadas de maneira organizadas, formando uma rede cristalina. A figura 8 ilustra este método.

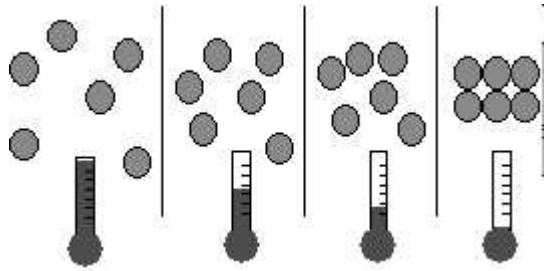

FIGURA 8 – Agitação de Moléculas na Têmpera

O algoritmo de Simulated Annealing funciona com uma variável chamada de temperatura que controla o "grau de agitação" do algoritmo. O algoritmo trabalha com uma solução temporária de posicionamento que é alterada por uma função de perturbação. A temperatura está relacionada com a tolerância do algoritmo às perturbações. Quanto mais alta a temperatura, maior é a aceitação do algoritmo, enquanto que em temperaturas baixas o SA aceita somente modificações julgadas como boas. Quem julga uma solução é a função de custo. O custo deve refletir o estado atual do posicionamento com uma nota. Dependendo da variação da nota e da temperatura, o algoritmo de Simulated Annealing vai aceitar a perturbação a qual foi submetido.

Parte-se de uma solução inicial, que em geral é aleatória. A cada iteração, é calculada uma pequena perturbação no estado atual. Diferentemente dos algoritmos gulosos, admite-se uma solução pior que a solução anterior com uma probabilidade, que varia ao longo do algoritmo. Esta probabilidade é calculada em função do parâmetro temperatura. Assim como no resfriamento de metais, a temperatura começa alta, fazendo com que a probabilidade de aceitar resultados piores seja alta e dando um comportamento praticamente aleatório ao algoritmo. Conforme passam as iterações, são calculados novos valores de temperatura, cada vez menores. A função que faz este cálculo chama-se *Scheduling*. No final do processo a temperatura é baixa, fazendo com que o algoritmo comporte-se praticamente como um algoritmo guloso.

A probabilidade é calculada conforme mostra a equação abaixo. A variação do custo, chamada de delta, é simplesmente uma subtração do custo da solução atual pelo custo da solução modificada. Isto faz com que o algoritmo seja sensível também à qualidade da nova solução.

$$prob = e^{\frac{-delta}{temperatura}} \qquad [1]$$

A figura 9 mostra o algoritmo de Simulated Annealing.

Simulated Annealing

```
Temperatura = temperatura_inicial;
Solução = Solução Aleatória
Enquanto Temperatura>temperatura_final
  Para cada iteração na mesma temperatura
    NovaSolução = Perturbação da Solução;
    ΔCusto = Custo(NovaSolução) – Custo(Solução);
    Se (ΔCusto < 0)
       Solução = NovaSolução;
    Senão Se Probabilidade exp(ΔCusto / Temperatura) é
obedecida
        Solução = NovaSolução;
   Temperatura = Schedule(Temperatura);
```

FIGURA 9.a – Algoritmo Simulated Annealing em pseudo-linguagem de programação

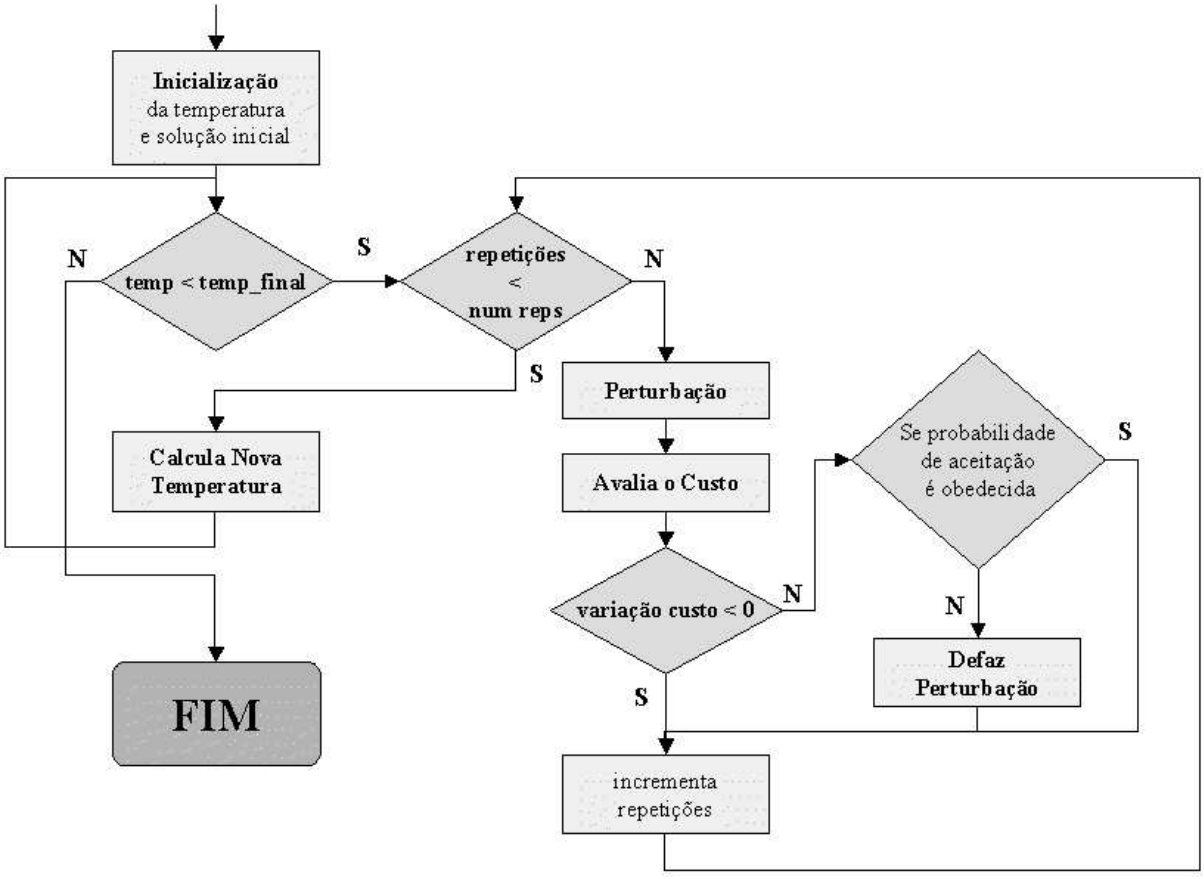

FIGURA 9.b - Algoritmo Simulated Annealing em fluxograma

2.6.2 Algoritmos Genéticos

Os algoritmos genéticos inspiram-se no processo natural de evolução das espécies. A idéia é seguir o princípio base da teoria da evolução que diz que os mais fortes sobrevivem. No

processo natural, os indivíduos mais capazes conseguem sobreviver e reproduzir-se, gerando filhos com características semelhantes. Os indivíduos mais fracos morrem logo, fazendo com que seus genes não sejam transmitidos. Há, portanto, uma seleção natural, fazendo com que a espécie evolua ao passar do tempo. Em algumas etapas pode ocorrer uma mutação, devido a algum fator externo, que modifica levemente um gene. A mutação garante para a população o ingresso de novas características, assim como uma maior variabilidade de indivíduos.

Os algoritmos genéticos tentam copiar este processo. Trabalha-se com uma pequena população de indivíduos, que mantém o mesmo tamanho durante todo o processo. Um indivíduo é uma possível solução consistente para o problema que se busca resolver. No caso de posicionamento, um indivíduo é uma solução intermediária do Simulated Annealing, por exemplo. O código genético dos indivíduos representa as características da solução representada pelo indivíduo. No caso de posicionamento, representa a posição dos blocos. O algoritmo executa em diversas iterações, até que um determinado critério seja atingido (ex.: número máximo de iterações). Cada iteração significa uma nova geração de filhos dos indivíduos da população. Esta geração é feita de forma que se mantenha a população de tamanho fixo. Há, portanto, uma operação de reprodução, conhecida como *crossover*. Para cada indivíduo é feita uma escolha de um par, baseado nas suas qualidades, para que haja uma reprodução. Neste momento, ou o pai ou a mãe são descartados para manter a população com o mesmo número de indivíduos. A reprodução é uma das operações elementares dos algoritmos genéticos que são capazes de modificar o código genético de um indivíduo, ou seja, as características de uma solução para o problema em questão. A outra operação é de mutação. Ela trabalha sobre um indivíduo somente, e altera uma pequena porção do seu código genético aleatoriamente.

Para aplicar um algoritmo genético, a solução de um problema qualquer de otimização, é preciso definir uma função de *crossover*, ou seja, de combinação de dois elementos, mantendo propriedades hereditárias. Ainda, é preciso definir qual o método para a escolha do par a se reproduzir. A técnica mais utilizada é a da roleta. Cada indivíduo é avaliado pela função de avaliação (como no Simulated Annealing). Sorteia-se uma solução com probabilidade proporcional a sua nota. Assim, priorizam-se soluções melhores, mas admite-se que seja escolhida uma solução ruim. Desta forma, mantém-se o caráter aleatório da evolução, obedecendo à lei que só os mais fortes sobrevivem. Com o passar das iterações, a tendência é que os indivíduos tenham características melhores, ou seja, que a solução para o problema seja refinada. Outra operação a ser definida é a mutação. Ela ajuda ao algoritmo a evitar mínimos locais. Um exemplo de mutação é uma alteração em um bit da representação do indivíduo. A figura 10 mostra um pseudocódigo que representa o algoritmo genético.

Algoritmo Genético

```
Montar uma população inicial aleatória
Enquanto Critério de Parada não é obedecido
    Para Cada Indivíduo
        Se Probabilidade de CrossOver satisfeita
               Escolher outro indivíduo pela técnica da roleta
               Realizar Crossover
        Se Probabilidade de Mutação satisfeita
               Realizar Mutação
```

FIGURA 10.a – Algoritmo Genético em pseudolinguagem de programação

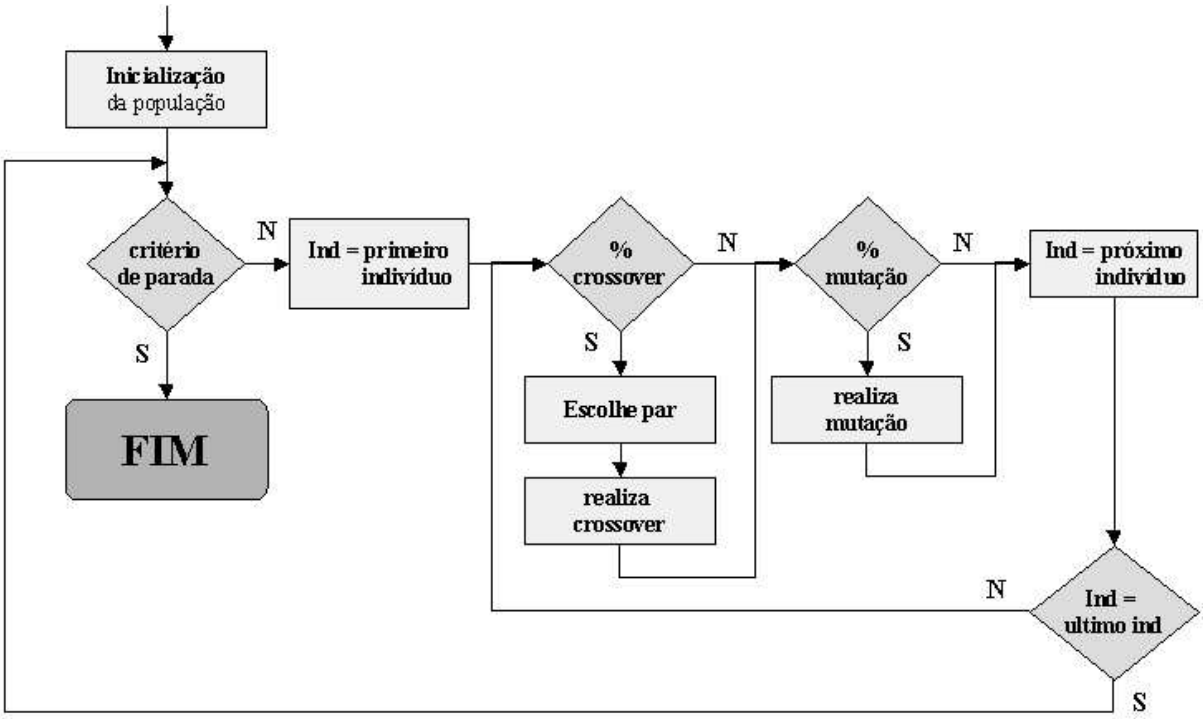

FIGURA 10.b – Algoritmo Genético em fluxograma

3 Posicionamento

O posicionamento pode ser definido, informalmente, como a etapa de síntese responsável por encontrar uma posição física para cada célula no circuito. Não pode haver intercessão de área das células. O principal objetivo do posicionamento é ter um circuito roteável, ou seja, que seja possível, com a área disponível, traçar fios para todas as conexões entre as células, conforme especificado no *netlist* de entrada. Para tanto, o posicionamento procura aproximar as células conectadas entre si. O problema torna-se complexo à medida que o número de células aumenta, pois estas ocupam espaços e aumentam a demanda por fios de roteamento. O posicionamento deve saber evitar que algumas regiões estejam excessivamente congestionadas[3], pois estas áreas prejudicarão o roteamento, podendo até inviabilizá-lo. Além disto, regiões congestionadas degradam a eficiência dos algoritmos de roteamento, pois exigem uma pesquisa maior por espaço para as conexões.

Além de ser uma etapa fundamental para que o roteamento seja completo, o posicionamento define outras variáveis de extrema importância para o bom funcionamento de um circuito integrado. Primeiramente, analisemos a freqüência de operação do circuito como função do posicionamento. Nas tecnologias atuais, o atraso das conexões é da mesma ordem de grandeza que o atraso das portas lógicas. Isto leva a crer que o *timing* é definido pela soma destas duas variáveis. O atraso dos fios é definido, indiretamente, pelo posicionamento. Ao posicionar as células dos caminhos críticos próximas uma da outra, o atraso dos mesmos pode ser significativamente reduzido. O tamanho das conexões, porém, pode ser estimado, mas não medido com precisão durante o posicionamento. Desta forma, torna-se importante definir métodos e modelos de atraso de conexão. É interessante que haja um prévio conhecimento do algoritmo de roteamento, sabendo que tipo de conexão ele vai traçar, que níveis de metal serão usados, e outras informações pertinentes ao cálculo da constante RC[4] do fio.

Depois da freqüência de relógio, a dissipação de potência é outra função do posicionamento. Primeiramente consideremos a dissipação total do circuito. Ela é função das capacitâncias de saída das portas e das capacitâncias dos fios onde há transição de nível lógico. Assim, a dissipação total de potência está relacionada com o tamanho total dos fios, ou seja, com o roteamento, que é definido pelo posicionamento. Outra questão fundamental relacionada à dissipação da potência é a sua distribuição ao longo do chip. Não é desejável que haja focos de alta dissipação de potência enquanto há outras áreas com pouca dissipação. Assim, além de distribuir as conexões igualmente ao longo do circuito, é preciso considerar a freqüência de operação das células.

Assim, a pesquisa por algoritmos, métodos e modelos para posicionamento é crescente e de fundamental importância devido à rápida evolução da tecnologia. Novas considerações devem ser tomadas:

- Diminuição do tamanho dos elementos. Um chip pode conter milhões de transistores, sendo que todos eles devem ser devidamente posicionados, sempre com a preocupação de minimizar área, potência dissipada e atraso. A diminuição dos componentes ainda traz problemas elétricos que antes não ocorriam, como degradação do sinal, *cross-talk* e outros, que devem ser evitados para o correto funcionamento do circuito.
- Aumento do número de camadas de metal para roteamento. Antigamente, as tecnologias ofereciam dois ou três níveis de metal para roteamento. Isso forçava o

[3] Diz-se que uma região está congestionada quando há uma demanda por conexões maior que o espaço disponível.

[4] A constante RC é dada pela resistência do fio multiplicado pela capacitância. Em circuitos integrados, ela define o atraso característico do fio.

aparecimento de canais de roteamento entre as bandas, o que efetivamente aumentava a área do circuito. Mas, hoje, se tem quatro ou mais níveis de metal para roteamento, o que possibilita um posicionamento com um espaço mínimo entre as bandas.

- Diminuição do atraso dos transistores. Hoje, o atraso das conexões é tão importante quanto o atraso dos transistores.
- Diminuição da capacitância de saída dos transistores com relação a capacitância das conexões. O consumo de potência do circuito agora não está centrado nas transições de sinal dos transistores, mas também nas transições das conexões.

As estimativas de congestionamento tornam-se fundamentais para que o circuito seja roteável. Além disto, nem todos os algoritmos são capazes de lidar com milhões de células. A estimativa de *timing* também é fundamental, na medida que as conexões tornam-se menores em dimensão, mas maiores com relação aos transistores. Uma conexão que cruza o circuito pode alterar significativamente a freqüência de operação do circuito. A mesma conexão poderá sofrer de degradação de sinal, sendo que pode ser necessária a inserção de *buffers* ou repetidores. Porém, na etapa de posicionamento já estão definidas as células, sendo que inserir mais células implicaria em reiniciar o processo. Assim, conexões longas demais devem ser evitadas.

O *cross-talk* é um problema bastante significativo nas tecnologias modernas. Ele pode afetar o funcionamento do circuito de duas maneiras diferentes. Primeiro, uma conexão pode sofrer um atraso mais prolongado de propagação em função de sua conexão vizinha. Assim, a estimativa de *timing* é significativamente prejudicada. Segundo, uma conexão pode alterar o seu valor lógico devido ao *cross-talk*. Esta situação deve ser evitada a todo custo.

O desafio do posicionamento hoje é tratar todas estas novas considerações de tecnologia, atender às especificações de mercado (freqüência de operação e consumo de potência) e atender a tempos de CPU aceitáveis para processar todas as informações.

3.1 Definição do Problema

A fim de definir formalmente o posicionamento, considere o seguinte modelo para descrever funções:

Retorno Função(Parâmetros)

O problema de posicionamento pode ser definido formalmente da seguinte forma: seja N um netlist de células, formado por um conjunto C de células e um conjunto R de redes. Para simplificar, sem perda de generalidade, considera-se que cada célula possui somente uma saída. São definidas duas funções:

list<c> Input(r) retorna uma lista de células que tem como entrada a rede r.
c Output(r) retorna a célula cuja saída é a rede r.

Para melhor compreensão do modelo, considere o circuito de exemplo, na figura 11. Neste caso:

Input(1) = {D};
Input(2) = {D,E};
Input(3) = {E};
Output(1) = A;
Output(2) = B;
Output(3) = C;

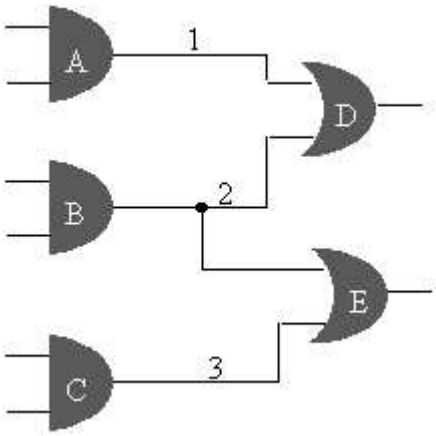

FIGURA 11 - Exemplo de circuito lógico com três redes

O posicionamento deve definir as seguintes funções:

Int PosX(c) retorna a posição x da célula c.
Int PosY(c) retorna a posição y da célula c.

Assim define-se o problema básico de posicionamento, mas não as variáveis a otimizar. Classicamente, o principal objetivo do posicionamento é minimizar o tamanho total das conexões. Para tanto, é desejável que o valor de **abs[5](Posx(i) - Posx(j))** e **abs(Posy(i) – Posx(j))** seja mínimo, onde i,j pertencem a mesma rede. Para pertencer a mesma rede *n*, i e j devem estar inclusos em uma das seguintes situações:

i ∈ Input(n) e **j ∈ Input(n)**
i ∈ Input(n) e **j ∈ Output(n)**
i ∈ Output(n) e **j ∈ Input(n)**

Modelos de timing, potência e congestionamento são vistos com mais detalhes nas sessões seguintes.

3.2 Classificação de Posicionamento

O posicionamento possui diferentes algoritmos dependendo do estilo de leiaute alvo. Este trabalho discute apenas o posicionamento para um modelo baseado em células, que é definido a seguir. Os transistores do circuito são agrupados em células. As células possuem uma altura fixa e são inseridas em bandas de células que são fileiras horizontais conforme observa-se na figura 12. Não há espaçamento entre as bandas, sendo que todo o roteamento deve ser realizado por cima das células. Havendo a inserção de canais de roteamento entre as bandas, os algoritmos apresentados neste trabalho não terão o mesmo resultado, podendo até mesmo inviabilizar que seja feita a síntese completa.

[5] Abs é uma função que retorna o valor absoluto de um número. Exemplo: abs(1) = 1 e abs(-1) = 1.

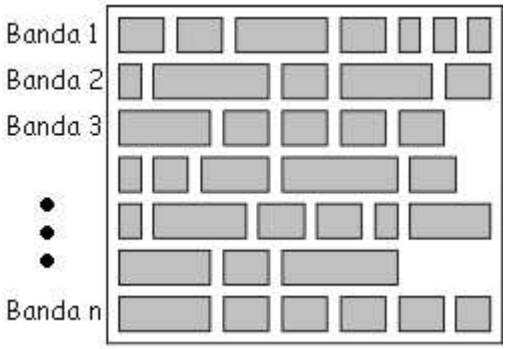

FIGURA 12 – Células posicionadas em Bandas

Baseado no modelo acima, consideram-se dois estilos de síntese do leiaute:

1 – Uso de bibliotecas de células chamadas *standard cell*. Há um prévio desenho de todas as células que poderão ser usadas no circuito. As células podem ser desenhadas de diferentes maneiras, usando diferentes tamanhos de transistores, para que possam ser inseridas adequadamente no circuito.

2 – Geração automática de todo o leiaute. O leiaute das células é gerado *on-the-fly*, ou seja, em tempo de execução.

A principal vantagem do estilo 1 é a possibilidade de pré caracterização das células. Neste caso, pode-se saber com precisão qual o atraso de uma célula, qual sua dissipação de potência e todos os parâmetros que possam ser úteis na síntese. Para a análise de *timing*, por exemplo, ter o valor exato do atraso de uma porta é de importância significativa.

A principal vantagem do estilo 2 é o fato de não estar preso há um número limitado de equações lógicas. Com a geração automática é possível que o circuito possua qualquer tipo de porta CMOS, que usa qualquer tipo de equação lógica que obedeça ao número máximo de transistores em série[6]. Além disto, a geração do leiaute automática propicia que os transistores sejam dimensionados livremente, desde que respeitem a altura da banda. Existem técnicas de duplicação do *w* de um transistor através da inserção de outro transistor em paralelo, mas não estão no escopo deste trabalho.

Para a etapa de posicionamento os algoritmos podem mudar de um estilo para o outro. Usando-se o estilo standard cell (1), o posicionamento deve encontrar a posição x e y exata das células. Isto ocorre porque as células já estão desenhadas e se conhece com precisão o tamanho delas. Ao posicionamento que encontra a posição x,y das células dá-se o nome de **posicionamento absoluto**.

Usando-se o estilo (2), duas são as possibilidades: ou o posicionamento é realizado antes da geração do leiaute das células ou é realizado depois. A vantagem de realizar depois é contar com o tamanho preciso das células, assim como no **standard cell**, citado no parágrafo anterior. Porém, ao realizar o posicionamento antes, dá-se uma maior liberdade para a etapa de geração do leiaute, permitindo que se compacte o leiaute. A figura 1 (na sessão 1.1) ilustra um caso onde duas células podem compartilhar uma mesma difusão.

Porém a etapa de posicionamento é bastante diferente do posicionamento absoluto, que foi discutido até então. Primeiro, o tamanho das células deve ser estimado para que seja possível estimar o tamanho dos fios e definir o formato do circuito com maior precisão. Segundo, não é possível definir uma posição (x,y) para as células, sendo necessário que o posicionamento

[6] É de costume em um projeto de circuitos integrados limitar o número de transistores em série de uma determinada porta. Isto ocorre para definir um atraso máximo para as portas, assim como evitar degradação do sinal.

trabalhe apenas com uma ordenação das células em cada banda. Assim, a coordenada y das células passa a ser qual banda ela ocupa, enquanto que a x é substituída pela ordem da célula (da esquerda para direita) na banda. Chama-se de **posicionamento relativo**.

O posicionamento relativo permite que as células sejam dispostas em diferentes estratégias, seguindo a mesma ordem. Pode-se, por exemplo, alinhar as células à margem esquerda, alinhar à direita ou centralizar. Estes três alinhamentos supõem que as células ficarão uma ao lado da outra, sem nenhum espaço entre elas, conforme observa-se na figura 13. Assim, a geração do leiaute poderá compactar as células da maneira que desejar. Seria possível, no entanto, dar um espaçamento entre as células, fazendo com que a relação de aspecto do circuito seja conforme o usuário deseja. Outra vantagem é aliviar o congestionamento, pois inserem-se espaços em branco ente as células. A figura 14 mostra células dispostas desta forma.

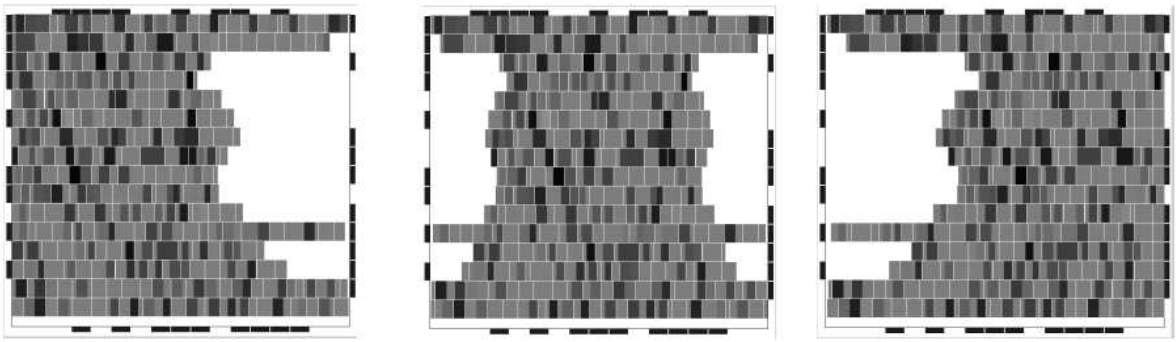

FIGURA 13 – O mesmo circuito em três estratégias de alinhamento: esquerda, centralizado, direita

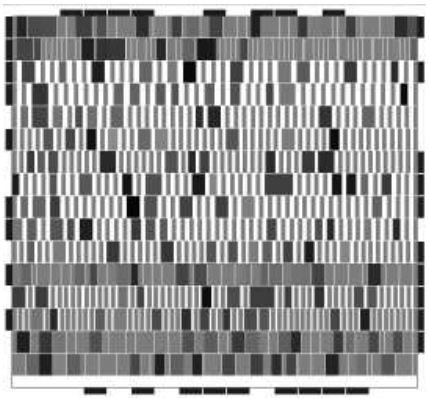

FIGURA 14 – O circuito da figura 13 com espaços entre as células.

A tabela 1 mostra dados de *wirelength* de um mesmo posicionamento usando alinhamento à esquerda, centralizado, a direita e justificado para dois *bechmarks*. Os resultados mostram, como é esperado, que o estilo centralizado apresenta o menor somatório de distâncias entre as células e, portanto, um menor WL necessário. E o estilo justificado é o que apresenta maior WL. Além disto, o estilo justificado é ruim pelas excessivas quebras na difusão, degradando a performance e dissipação de potência. A área, porém, é a mesma, sendo melhor aproveitada por dissolver as áreas congestionadas.

TABELA 1 – Tamanho médio dos fios em estratégias diferentes de alinhamento das células

Benchmark	Esquerda	Centralizado	Direita	Justificado
C1908_3x3	445205	433972	439772	462763
Alu4_2x2	11072150	10899392	11140179	11633212

3.3 Objetivos do posicionamento

3.3.1 Roteabilidade

Como já foi dito anteriormente, o principal objetivo do posicionamento é que o circuito seja roteável. Ser roteável é uma função do posicionador e do roteador. Porém, para circuitos mal posicionados, nenhum roteador será capaz de completar todas as conexões. Desta maneira, o principal responsável pela roteabilidade do circuito é o posicionador.

Para buscar este objetivo, busca-se otimizar o tamanho total das conexões, conhecido em inglês como *wirelength*. O tamanho total das conexões é simplesmente o somatório do tamanho de todas as conexões do circuito. Minimizar o somatório é equivalente a minimizar a conexão média. Porém a minimização da conexão média não garante que haja uma distribuição equivalente da demanda por fios ao longo do circuito, fazendo com que haja um cuidado especial para evitar o congestionamento de algumas regiões.

3.3.1.1 Tamanho dos fios

O posicionamento, a princípio, não sabe o tamanho das conexões. Sabe apenas a posição das células. Além disto, as células são consideras como caixas pretas, sem a informação da posição dos contatos de entrada e saída. Desta forma, trabalha-se com estimativas do tamanho das conexões. A estimativa do tamanho da conexão de uma célula A com uma célula B pode ser calculada simplesmente pela distância *manhatan*[7] das duas células, como mostra a figura 15.

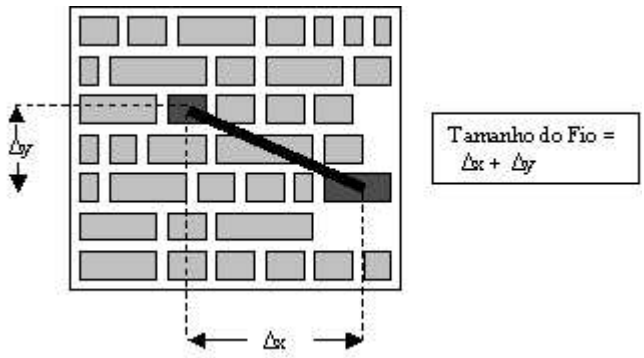

FIGURA 15 – Distância de duas células

Observe, porém, que as conexões de qualquer circuito integrado são multi-ponto, ou seja, conectam mais de duas células. Assim, não basta estimar o tamanho dos fios pela distância entre eles. Sait, em 1995, apresentadas algumas métricas para estimar o tamanho dos fios para redes multi-ponto, as quais são reproduzidas aqui.

- **Semi Perímetro**. Encontra a menor *bounding box* que inclui todas as células envolvidas em uma determinada conexão e calcula o seu semiperímetro (altura + largura). Repete este cálculo para todas as redes do circuito. O resultado final é a soma de todos os resultados parciais. É largamente utilizada por ser computada rapidamente e apresentar bons resultados.
- **Grafo Completo**. Calcula a distância entre todos os pontos a conectar, dois a dois. Esta distância pode ser a distância Manhatan ou Euclidiana, como na figura 18. É menos

[7] Distância Manhatan é o cálculo da distância usando apenas retas de 90 graus, como as quadras da ilha de Manhatan em Nova Yorque. Assim, a distância entre dois pontos no plano é dada por $\Delta x + \Delta y$.

eficiente que o semiperímetro por efetuar n^2 cálculos de distância, para uma rede de *n* pinos.

- **Origem para destino**. Toda rede tem uma saída e diversas entradas. Esta estimativa calcula a distância da saída para as entradas. É interessante para o cálculo de *timing* por refletir o atraso de uma conexão corretamente. Porém, não é uma boa estimativa de tamanhos dos fios por fazer uma diferenciação da célula de saída. Os algoritmos de roteamento em geral não sabem que pino é de saída e se sabem desconsideram esta informação. É, porém, uma estimativa calculada muito rapidamente.

- **Menor Cadeia**. Uma cadeia é uma seqüência de retas que parte de uma célula e passa por todas as outras, ligando as células por um fio único. Encontrar a menor cadeia não é um problema trivial, sendo que esta estimativa não acrescenta nenhum realismo. Portanto é um método pouco eficiente e que não traz nenhuma vantagem, sendo portanto um método pouco usado para estimativa dos fios.

- **Menor *Spanning Tree***. Spanning Tree é uma árvore que une todos os pontos a conectar. É uma estimativa realista, pois os algoritmos de roteamento trabalham com este tipo de estrutura, na sua maioria. Porém encontrar a menor *spanning tree* é um cálculo mais aprimorado que os anteriores, exigindo mais processamento.

- **Árvore de Steiner**. Um problema de roteamento é justamente encontrar a árvore de Steiner mínima para todas as redes, com a restrição que não podem se cruzar. Usar esta estimativa para posicionamento, sem a restrição do cruzamento, é extremamente realista e muito próximo do resultado de um algoritmo de roteamento clássico baseado em *Maze Routers*. O problema desta estimativa é o tempo de processamento. O problema de encontrar a menor Árvore de Steiner é NP-Completo, portanto temos somente algoritmos exponenciais para resolvê-lo. Como se trata de um problema de otimização, podemos tratá-lo com heurística e meta-heurísitcas. Uma solução é usar um *Maze Router* sem restrições de cruzamento de fios. Esta solução não pode ser usada durante o posicionamento, mas é muito adequada para estimar a qualidade do resultado no final do posicionamento.

A figura 16 mostra graficamente os métodos citados acima.

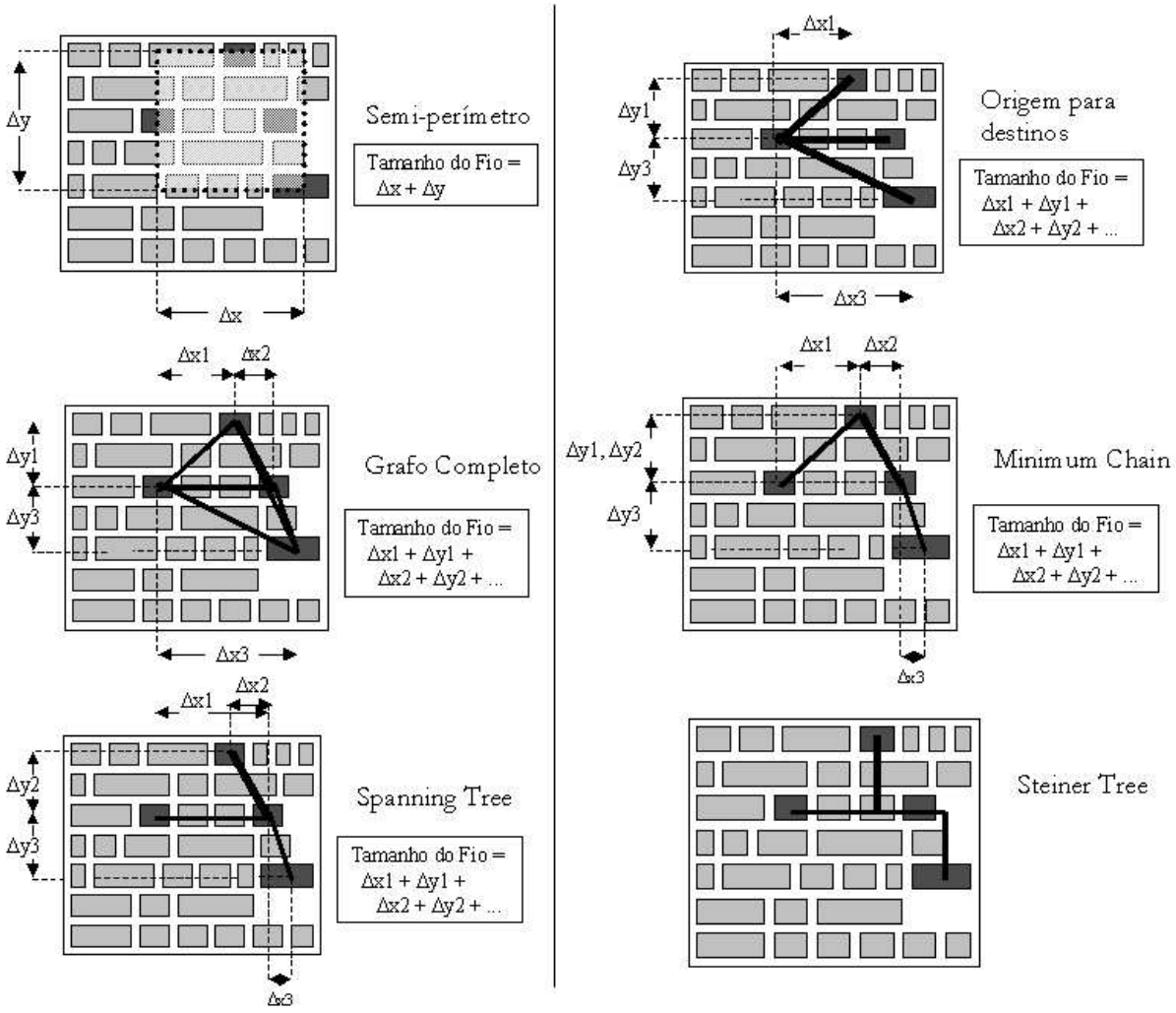

FIGURA 16 – Estimativas de tamanho dos fios para redes multi-ponto.

Um outro método de estimar o tamanho dos fios pode ser implementado com base no algoritmo de posicionamento chamado Posicionamento Direcionado a Forças (sessão 7 deste trabalho). Este algoritmo se baseia em uma função que define as coordenadas ideais para posicionar uma célula.

Considere uma situação de posicionamento qualquer, como o exemplo da figura 15. É possível saber a posição ótima de uma célula considerando que as outras estão fixas em suas posições. Esta posição é conhecida na literatura como a **posição de força zero**. O algoritmo de posicionamento direcionado a forças faz uma analogia a um espaço físico de células que fazem forças de atração e repulsão entre si. A posição de força zero é calculada pelo somatório destas forças, de acordo com as leis de Newton da mecânica. Em posicionamento, a posição de força zero é calculada conforme as equações abaixo.

$$Posx(i) = \frac{\sum_n PosX(Input(n)) + PosX(Output(n))}{\sum_n Count(Input(n)) + 1} \qquad [2]$$

$$Posy(i) = \frac{\sum_n PosY(Input(n)) + PosY(Output(n))}{\sum_n Count(Input(n)) + 1}$$

onde:

n representa as redes Input(i) + Output(i)
PosX(Input(n)) retorna a soma das posições X das células de entrada da rede n
Count(Input(n)) diz quantas entradas possui a rede n

Sabendo qual a posição ideal para uma célula, uma função de estimativa de tamanho dos fios adequada seria a diferença desta posição pela posição atual da célula. A figura 17 ilustra este caso graficamente.

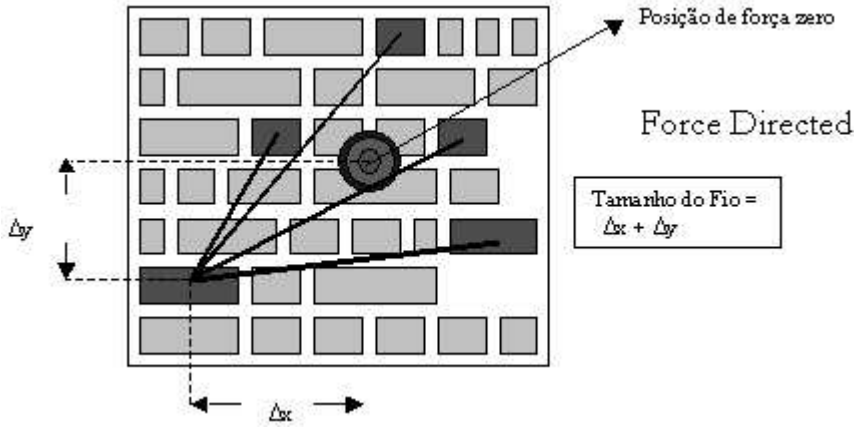

FIGURA 17 – Estimativa de *wirelength* com base na posição de força zero.

Os métodos de estimativa de fios do semiperímetro e de Origem para Destinos, além do Force Directed, apresentado logo acima, foram implementados na ferramenta do *Mango Parrot*. A tabela 2 mostra os resultados de tempos de processamento em um Pentium III 850 MHz rodando Windows 2000. As funções foram repetidas 300 vezes para uma medida mais acurada. Os bechmarks usados são o circuito C1903_3x3 (783 células) e o C6288_4x4 (3073 células). Eles são apresentados com mais detalhe na sessão 8 deste trabalho.

TABELA 2 – Tempo de processamento de estimativas de tamanho dos fios

Algoritmo	C1903_3X3	C6288_4X4
semiperímetro	0,47s	2,65s
Origem para Destinos	0,45s	2,60s
Grafo Completo	0,98s	5,12s
Force Directed	1,41s	7,01s

Observe na tabela 2 como os métodos mais rápidos são o semiperímetro e o Origem para Destinos. Em diversos algoritmos, como é o caso do Simulated Annealing, é muito importante que o cálculo do comprimento das conexões seja o mais rápido possível. O método do semiperímetro é o mais utilizado por ser rápido e por ser um modelo mais preciso que Origem para Destinos.

3.3.1.2 Congestionamento

Além de minimizar o tamanho das conexões é preciso que haja uma distribuição equilibrada das conexões ao longo do circuito. Não podem haver pontos de alto congestionamento, pois, neste caso, o algoritmo de roteamento pouco poderá fazer para traçar o fio corretamente.

Minimizar o tamanho dos fios indiretamente diminui o congestionamento. Mas não há nenhuma garantia que as conexões estarão espalhadas de maneira equilibrada pelo circuito. Observe na figura 18.a, o circuito C1908_3x3 posicionado aleatoriamente. Toda área está congestionada. Ao lado, na figura 18.b o mesmo circuito posicionado com outro algoritmo mais

eficiente. O tamanho dos fios decresceu de aproximadamente 827000 para 294000. Porém ainda observam-se áreas congestionadas, que estão circuladas, enquanto observa-se que outras áreas estão relativamente livres.

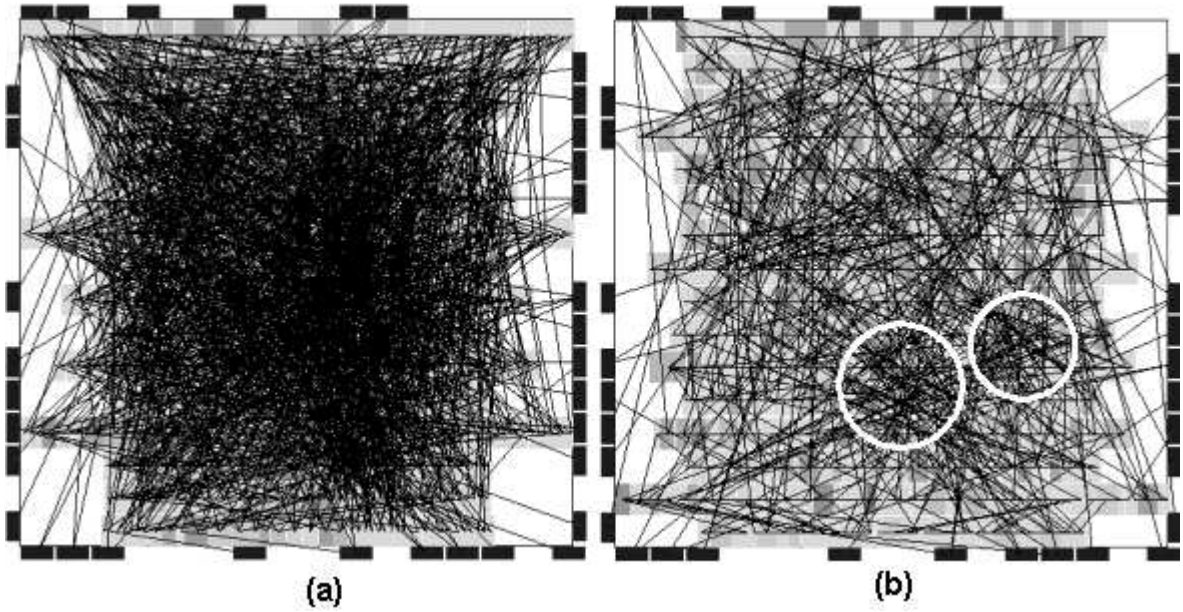

FIGURA 18 – Congestionamento pode persistir ao diminuir o tamanho total dos fios

O caso citado acima reflete-se diretamente no roteamento. O mesmo circuito foi posicionado posteriormente pela Simulação de Têmpera, visando uma otimização de tamanho dos fios. Depois de 900 segundos de processamento, o algoritmo atingiu uma solução de custo 139120. Foi gerado o seu leiaute usando a ferramenta de geração de Lazzari e roteado usando o roteador do Dragão Limão. O resultado é que 17 redes não foram completadas. A figura 19 mostra o desenho das conexões concluídas.

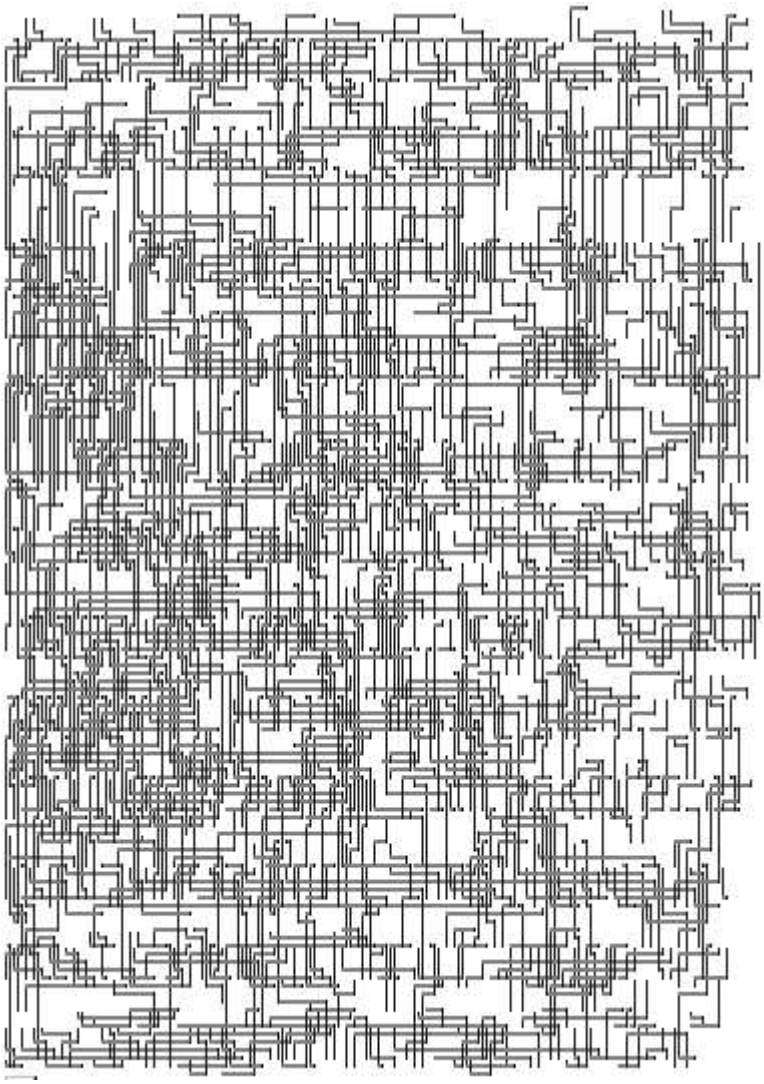

FIGURA 19 – Roteamento de um circuito congestionado.

Observe, na figura 19, que há muitos espaços vazios, não justificando o fato de ter redes não roteadas. Porém, no lado esquerdo da figura, há uma concentração de conexões que inviabiliza a passagem de outras conexões por ali. Portanto, se houvesse uma distribuição equilibrada de demanda por conexões, o circuito poderia ser roteado, pois há uma série de espaços vazios.

Congestionamento pode ser definido simplesmente pela relação de demanda por conexões com a oferta de espaço para elas. A demanda de conexão de uma área pode ser estimada através de modelos de congestionamento. Porém, qualquer estimativa que desconheça o algoritmo de roteamento será imprecisa. Observe a figura 20, que mostra como é prejudicada a estimativa de congestionamento no posicionamento. Ela mostra duas situações de roteamento, onde uma conexão pode passar por uma área congestionada ou não. Sem conhecer o algoritmo de roteamento, o posicionamento não pode dizer por onde a conexão irá passar de fato.

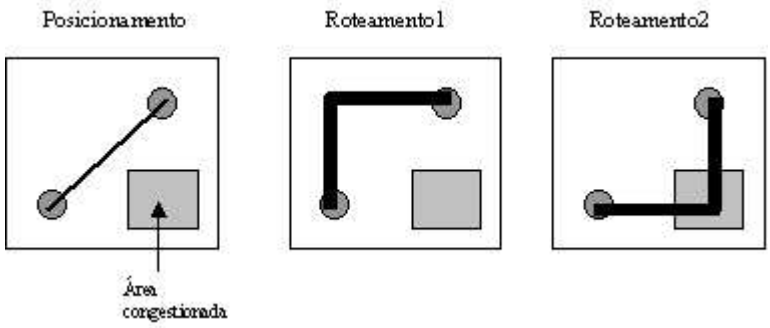

FIGURA 20 – Estimativa de congestionamento prejudicada

É comum, porém, o uso de um algoritmo de roteamento global (como mostram os trabalhos de Johann em 2001 e Hentschke em 2001) antecipadamente, durante a etapa de posicionamento. Este método é melhor discutido na sessão 3.3.1.2.3.

Devido a estas dificuldades, é necessário que se criem bons modelos de congestionamento. Diversos artigos propõem diferentes modelos, apresentando algumas vantagens para cada um deles. Este trabalho reúne eles em uma breve explicação dos modelos considerados mais relevantes pelo autor deste texto. É importante observar que os modelos ainda não conseguem solucionar o problema de congestionamento eficientemente. Wang, em 2000, apresenta alguns modelos utilizados classicamente e outros próprios do autor. Conclui o artigo dizendo que nenhum dos modelos pôde solucionar adequadamente o problema de congestionamento. Ele observa que uma etapa posterior ao posicionamento pode resolver o problema eficientemente.

O primeiro método exposto neste texto se baseia no corte do circuito como um todo. Os demais baseiam-se em uma divisão do circuito em macro-blocos retangulares de tamanhos idênticos. O trabalho de Wang em 2000 também usa a divisão em blocos, porém com a noção de posicionamento global (sessão 3.4), ou seja, sem definir a posição exata das células dentro do bloco. Ele assume que as células estão posicionadas exatamente no centro do bloco, havendo intercessão de área entre elas. Porém, no mesmo artigo, Wang mostra que a estimativa de congestionamento usando posicionamento global não se contradiz, em nenhum caso apresentado, com a estimativa de congestionamento detalhada.

3.3.1.2.1 Corte Máximo

Este método procura o ponto do circuito onde há uma maior quantidade de conexões passando. Para isto, são feitos cortes no circuito, conforme observa-se na figura 21. Os cortes são feitos horizontalmente e verticalmente, abrangendo toda a área do circuito. Dentre os cortes horizontais é escolhido o corte que cruza com mais conexões. O mesmo é feito verticalmente e por fim seleciona-se o maior corte global. No exemplo da figura 21, temos que o maior corte tanto no sentido vertical quanto no horizontal é de 3 conexões. Assim, max(3,3) = 3, e portanto a estimativa de congestionamento final vale 3. O somatório de todos os cortes é equivalente a uma medida de tamanho de conexões, vistas na sessão anterior.

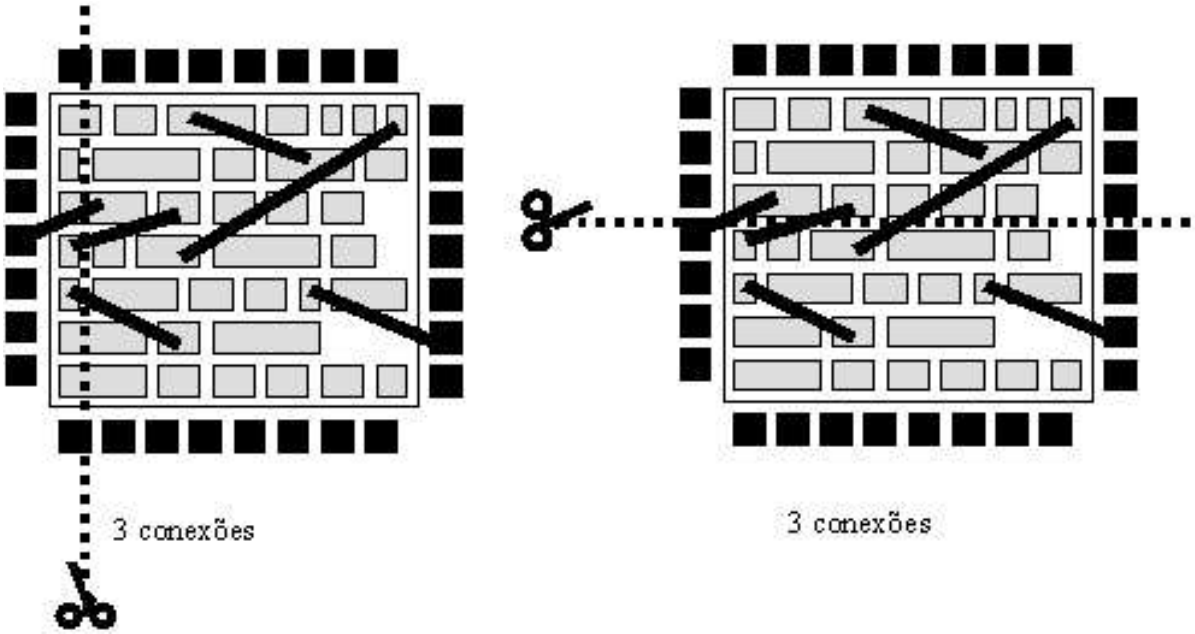

FIGURA 21 – Estimativa de congestionamento por corte

Para implementar este algoritmo, é necessário definir o passo, ou seja, em que pontos discretos do circuito serão feitos os cortes. O tempo do algoritmo varia inversamente com o tamanho do incremento. A tabela 3 mostra alguns valores de tempo calculados para o circuito Alu4 4x4, em um Pentium III 850 MHz rodando Windows 2000.

TABELA 3 – Tempos de execução da estimativa de congestionamento por corte

Tamanho do passo (μm)	Tempo de execução (s)
1	0,1906
5	0,0407
10	0,0219
15	0,0156

Observa-se que os tempos de execução na tabela 3 são muito altos para um posicionamento guiado por congestionamento. Por isto, o algoritmo deve ter algum mecanismo de atualização mais rápida.

Esta estimativa de congestionamento é tanto quanto imprecisa. Ela reflete a quantidade de conexões, mas é incapaz de diferenciar duas conexões próximas de duas distantes. No entanto, como é desconhecido o algoritmo de roteamento, a estimativa é bastante útil por ser totalmente independente do caminho traçado para o fio.

3.3.1.2.2 Densidade Máxima por Bounding-Boxes

Este método discretiza o circuito em regiões retangulares idênticas, como mostrado anteriormente na figura 20. Para cada região é feita uma avaliação de quantas redes passam por ali. A região com mais redes determina o congestionamento do circuito. Para determinar se uma rede *n* cruza uma região *r* é calculada a *bounding-box* que envolve todos pinos da rede *n*. Se há alguma intercessão da caixa que envolve a rede com a região é considerado que a rede passa pela região. A figura 22 ilustra um exemplo de uma rede cruzando uma determinada região.

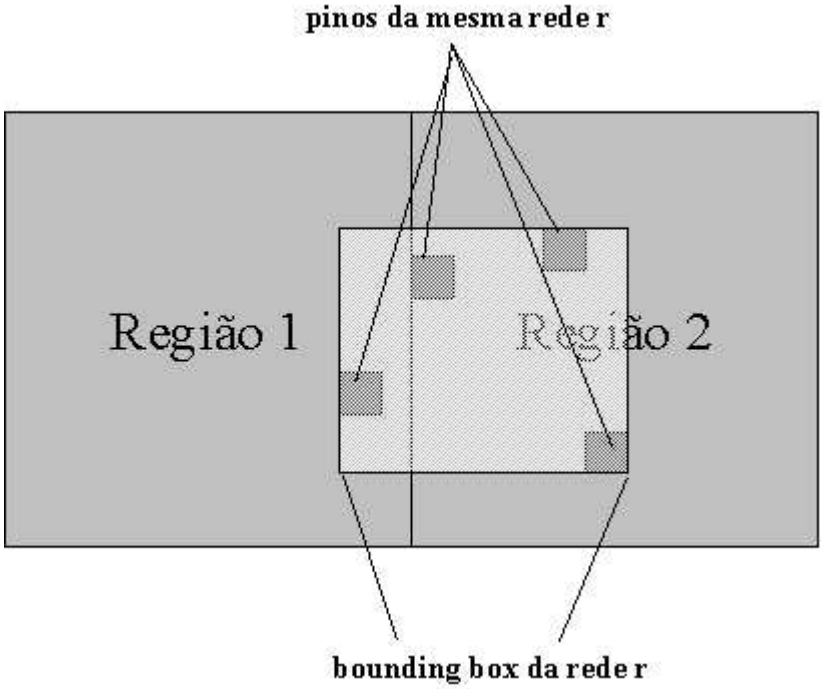

FIGURA 22 – Estimativa de congestionamento por *bounding-boxes*

As fórmulas a seguir dizem se uma rede delimitada por (x1,y1) e (x2,y2) cruza uma região delimitada por (X1,Y1) e (X2,Y2).

```
if ( X2>x1 && X1<x2 && Y2>y1 && Y1<y2)
        as regiões se cruzam
else
        as regiões não se cruzam
```

Encontrar um bom tamanho das regiões é um problema interessante. Não faz sentido usar um tamanho da região próximo ao tamanho do circuito, pois neste caso não há variável para minimizar. Quanto menor a região, mais preciso é o cálculo. Em geral usam-se regiões quadradas. Porém, se a região for muito pequena, a eficiência do algoritmo é prejudicada significativamente ($o(n^2)$ onde n é o tamanho do lado do quadrado), como mostra a tabela 4. O tamanho ideal será uma combinação satisfatória de tempo de processamento com eficiência na estimativa.

A tabela 4 apresenta os tempos de execução desta estimativa para o circuito alu4 4x4, rodando em um Pentium III 850 MHz com Windows 2000.

TABELA 4 – Tempos de execução para a estimativa de congestionamento por *bounding boxes*

Tamanho do lado (μm)	Tempo de execução (s)
10	3,3469
50	0,1406
75	0,0641
100	0,0375

A primeira observação pertinente é o altíssimo tempo de execução requerido para tamanhos pequenos de lado. Comparando-se com a tabela 3 (sessão anterior), somente com quadrados de lado 75 tem-se a mesma ordem de grandeza de tempo de execução.

3.3.1.2.3 Densidade Máxima Usando Roteamento Global

Está claro que, pela figura 20, o desconhecimento do algoritmo de roteamento leva o desenvolvimento de modelos imprecisos, como a intercessão de *bounding boxes*, citado na sessão anterior. O ideal seria conhecer o algoritmo de roteamento e saber por quais regiões exatamente passará a conexão.

Wang, em 2000, mostra através de alguns experimentos, que a estimativa de congestionamento baseada em *bouding box* é contraditória (em alguns casos) em relação à estimativa por roteamento global. Sabendo que o roteamento global é um modelo mais realista, supõe-se que as *bounding boxes* são um modelo impreciso.

3.3.1.2.4 Overflows

Overflow é um conceito bastante utilizado na estimativa de congestionamento. Basicamente, overflow é a quantidade de conexões excedentes em uma determinada região. É calculado através de duas variáveis: demanda e oferta. A demanda é calculada pela quantidade de conexões que precisam passar pela região. A oferta é o número de trilhas oferecidas para a passagem de conexões.

O número de trilhas é dependente do algoritmo de roteamento. Supondo um roteamento *gridless*[8], o número de trilhas estaria limitado somente pelas regras de tecnologia, conforme é visto na figura 23.

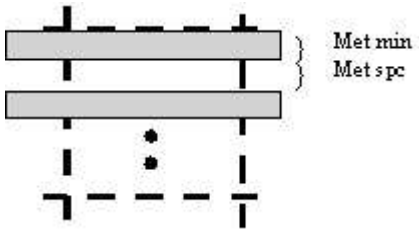

FIGURA 23 – Roteamento sem grade (*gridless*)

Assim, conhecendo-se o número de trilhas horizontais e verticais, o cálculo do overflow de uma região é dado pela seguinte equação:

overflow =	**demanda – oferta**	**se demanda > oferta**
	0	**se demanda ≤ oferta**

O overflow total do circuito é calculado pela soma de todos os overflows do circuito. Wang, em 2000, relata que sua experiência usando roteadores da indústria e posicionamento guiado pelo overflow total apresenta resultados satisfatórios.

O cálculo de overflow, porém, desconsidera os casos de alto congestionamento parcial, ou seja, abaixo da oferta. Nem sempre um roteador vai completar as conexões se a oferta é maior ou igual à demanda. Para que se possa penalizar congestionamentos parciais, é usada uma técnica conhecida como **overflow look ahead**, que simplesmente usa um valor menor que a oferta para calcular o overflow. A figura 24.a mostra o gráfico da função de overflow em função da demanda e a figura 24.b mostra o gráfico do **overflow look ahead** em função da demanda.

[8] A maior parte dos algoritmos de roteamento usa uma grade fixo por onde devem passar todos os fios. Este método torna mais fácil o roteamento. Algoritmos *gridless* não usam nenhuma espécie de grade.

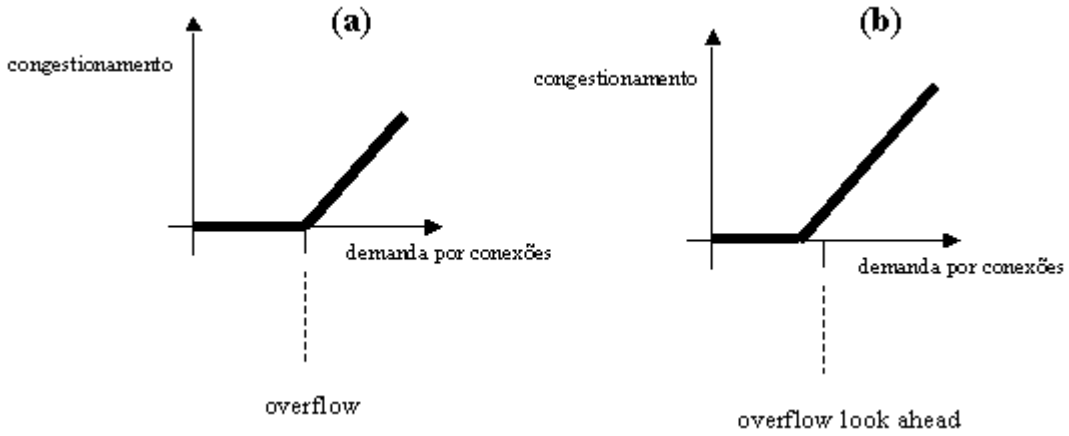

FIGURA 24 – Estimativa de congestionamento por *overflow* e *overflow look ahead*

3.3.1.3 Regra de Rent

Foi descrita pela primeira vez por Landman em 1971. É uma observação empírica das células e suas conexões externas em uma partição do circuito. Está relacionada, portanto, ao posicionamento global. Em uma região com G células e T conexões cruzando o bloco, a lei relaciona G com T da seguinte maneira:

$$T = tG^{p} \qquad [3]$$

onde *t* é o número médio de rede por bloco e *p* é um expoente, denominado de número de Rent, que varia entre 0,4 e 0,8 em circuitos reais.

Esta regra é usada em estimativa de tamanho dos fios e congestionamento, conforme o trabalho de Yang em 2002. Em geral, um expoente alto resulta em um comprimento de fios maior, o que, em média, aumenta o congestionamento do circuito. Além disto, a regra permite a observação de congestionamento em uma pequena região. O objetivo é diminuir ao máximo o expoente e não permitir que haja regiões com expoente muito maior que a média.

3.3.2 Dissipação de Potência

O posicionamento pode buscar a otimização de duas variáveis com relação à dissipação de potência:

- Potência total dissipada
- Distribuição da dissipação de potência

O tamanho dos fios de roteamento está diretamente relacionado à capacitância dos mesmos e, portanto, com a dissipação de potência de um circuito. Assim, a minimização do tamanho total dos fios também vale para a dissipação total de potência.

Porém, minimizar a potência total dissipada pode acarretar em áreas com maior dissipação que outras. Por esta razão, a distribuição da dissipação é uma variável que deve ser modelada em separado. Ching-Han Tsai, em 2000, apresenta um algoritmo de posicionamento que visa equilibrar a dissipação de potência ao longo do circuito juntamente com a minimização do tamanho total dos fios. Tsai desenvolve um modelo de dissipação de temperatura baseado apenas na posição das células (não considera dissipação das conexões). Este modelo pode ser aplicado a qualquer posicionador, de forma que o posicionamento seja guiado por ele e combinado com outros modelos. O trabalho apresenta resultados satisfatórios de redução da

distribuição de calor ao longo do circuito com pouco impacto no tamanho dos fios e área do circuito.

3.3.3 Timing

Ao observar as tecnologias modernas de fabricação de circuitos integrados, observa-se uma série de diferenças que tornam o atraso das conexões um dado muito relevante para o atraso total do circuito. Primeiro, a resistência das conexões aumenta, em função da diminuição, na mesma proporção, das suas três dimensões. A resistência das conexões que cruzam um chip aumenta ainda mais, pois os chips são cada vez maiores. O mesmo vale para a capacitância das conexões que cruzam um chip. Ainda, o canal dos transistores diminui, o que implica que as células lógicas operam cada vez mais rápidas. Hoje em dia, o atraso das conexões é na mesma ordem de grandeza do atraso dos transistores. Por esta razão, o *timing* das conexões deve ser considerado por qualquer ferramenta de CAD, o que implica em modificações na etapa de posicionamento.

A etapa de posicionamento é a responsável por definir a posição das células do caminho[9] crítico. Assim, o tamanho das conexões é significativamente influenciado por esta decisão. Se o posicionamento não fizer a aproximação correta, o roteamento pouco poderá fazer para diminuir o atraso de uma conexão crítica que cruza o circuito.

Além do tamanho das conexões, outros fatores podem influenciar o atraso das mesmas. O principal é o efeito de *cross-talk*. Uma linha de roteamento pode influenciar a outra, fazendo com que haja degradação do sinal e maior atraso de propagação. Na pior das hipóteses, o *cross-talk* poderia influenciar em uma linha de forma a trocar o seu valor lógico.

Sendo assim, são três as metodologias de posicionamento dirigido a *timing*:

- Avaliação dos caminhos críticos, sem considerar as conexões, antes do posicionamento;
- Avaliação dos efeitos elétricos das conexões durante o posicionamento.
- Nenhuma avaliação prévia de *timing*. É feita apenas uma prevenção que procura equilibrar o tamanho das conexões.

a) Avaliação prévia dos caminhos críticos

A primeira metodologia escolhe os caminhos de maior atraso com base apenas no atraso das células. Apesar de ser uma estimativa incorreta, ela aponta os caminhos mais promissores para serem críticos, depois de terminado o roteamento. Para estes k caminhos selecionados, o posicionamento deve buscar minimizar o atraso das conexões, ou seja, o tamanho delas. Observe que são escolhidos k caminhos, diferentemente de um caminho apenas, como é a estimativa tradicional. Isto é necessário, pois um outro caminho mal posicionado pode se tornar crítico pelo atraso das suas conexões. Escolhendo mais caminhos, esta probabilidade reduz, fazendo com que o posicionador preocupe-se com os caminhos mais promissores.

No projeto FUCAS, a pesquisa por caminhos críticos é baseada nos trabalhos de Güntzel (1998 e 2000). Basicamente buscam-se os caminhos através de uma pesquisa **best-first**, usando o algoritmo A*. O grafo contém custos, referentes ao atraso de cada porta. Porém, usa-se um modelo que trata dois casos de atraso: subida e descida..

A partir da definição das redes críticas[10], várias alternativas podem ser seguidas pelos algoritmos de posicionamento. A mais tradicional é aumentar o peso destas redes no cálculo do tamanho dos fios. No caso de posicionamento baseado em particionamentos sucessivos (sessão 5.4), esta informação pode ser utilizada para manter juntas as células de um caminho crítico. Yih-Chih Chou, em 2002, apresenta uma técnica baseada em *Force Directed Placement* (ver sessão 7.3).

[9] Um caminho é formado por um conjunto de redes que unem um Flip-Flop (ou um pino de E/S) a outro

[10] Uma rede crítica é uma rede que faz parte do caminho crítico

Basicamente, o algoritmo acrescenta um falso *link* entre os *flip-flops* dos caminhos críticos, fazendo com que haja uma "força" tentando os aproximar. Desta forma, as redes críticas vão ficar mais próximas entre si. Chou compara sua técnica com uma ferramenta de posicionamento comercial. Ele mostra que, em média, diminui em 10% o atraso do caminho crítico, e 11,5% o tamanho das conexões.

b) Estimativa do atraso das conexões "on-the-fly"

A segunda metodologia de posicionamento guiado a *timing* busca bons modelos elétricos para caracterizar as conexões estimadas pelo posicionamento. Isto faz com que hajam estimativas mais precisas de atraso, guiando o processo de posicionamento. Um modelo que interessa bastante é de capacitância e resistência dos fios. Outro modelo interessante é de *cross-talk*, evitando, por exemplo, que redes críticas sofram deste efeito.

A partir de uma estimativa precisa da constante RC das conexões, é possível fazer dimensionamento dos transistores das portas lógicas de forma que possam considerar a capacitância parasita das conexões adequadamente.

O trabalho de Chen, em 2000, une as duas metodologias citadas acima. Inicialmente ele faz uma análise buscando os *k* caminhos críticos. Através de uma formulação matemática é estimado o atraso das conexões. A técnica junta as informações em um algoritmo iterativo que dimensiona as células de forma a tornar o atraso dos caminhos mais críticos os menores possíveis. Se o dimensionamento não foi satisfatório, uma nova iteração é executada, até que encontre-se uma solução satisfatória. Chen compara seu método com o software TimberWolf, mostrando uma melhora média de 15% (considerando-se delay e área).

c) Não usar análise de timing

A terceira metodologia baseia-se no princípio que nenhum caminho pode ser muito maior que outro. Observa-se que em um circuito, o atraso é definido pelo caminho de atraso maior. Assim, o ideal é que haja um equilíbrio no tamanho dos caminhos. Sait, em 2001, mostra um modelo de atraso, aplicado em algoritmos genéticos e busca tabu, exatamente desta forma. É feita a estimativa de tamanho de cada rede de cada caminho do circuito. O atraso de um caminho é a soma dos atrasos das redes pertencentes ao caminho. O algoritmo de posicionamento busca minimizar o maior atraso de um caminho.

O trabalho de Shih-Lian, em 2000, parte do mesmo princípio. Baseia-se em posicionamento por particionamentos sucessivos (sessão 5.4). Ele busca evitar que os caminhos do circuito sejam divididos entre as partições mais do que um número máximo de cortes. O posicionamento é dividido em etapas, começando-se com um posicionamento global e seguindo-se de um posicionamento detalhado. Shih compara os resultados com outro sistema de posicionamento chamado Timing-QUAD, mostrando uma melhora de 23% em média.

3.3.4 Área

O posicionamento de células deve ainda se preocupar com questões de área do circuito. Não há como diminuir a área das células, porém o objetivo é diminuir a área do retângulo envoltório do bloco gerado. Para tanto, é importante equalizar a largura das bandas.

Outra consideração importante é quanto à relação de aspecto do bloco. Ela deve ser definida antes do posicionamento. Em geral, escolhe-se um quadrado (largura igual a altura).

Em uma execução do software TROPIC usando-se um posicionamento da ferramenta do *Mango Parrot*, verificou-se que bandas muito pequenas prejudicam significativamente o leiaute. Um experimento mostra que a densidade de transistores/mm^2 cresceu de 53,3K para 64,6K (20%) eliminando-se uma banda demasiadamente pequena. Este tipo de banda prejudica a alocação de *feedthroughts* e também força o aparecimento de áreas vazias. Ao mesmo tempo, atrapalha o algoritmo de roteamento por se tratar de elementos isolados. Contudo, deve haver

uma preocupação especial do posicionamento em limitar o tamanho mínimo e máximo de cada banda, com o objetivo de equalizar o tamanho das bandas.

No software TROPIC, a geração do leiaute é feita juntamente com o roteamento, sendo que as bandas são afastadas no caso de necessidade de espaços para fios. Por esta razão, o processo sempre termina o roteamento, porém a área pode ser significativamente aumentada. Por esta razão, estabelece-se uma relação importante entre roteabilidade e área do bloco.

3.4 Posicionamento Global

Posicionamento global é uma simplificação do problema de posicionamento. Não se está interessado na posição física precisa das células, mas sim em uma distribuição delas ao longo do espaço de posicionamento. O objetivo é que se simplifique o problema de posicionamento de forma que seja tratável em tempos de execução menores.

O problema de posicionamento global varia conforme o estilo de projeto desejado. Este trabalho se foca em **posicionamento de área**, ou seja , sem nenhum canal específico para roteamento. Assim, o espaço de posicionamento é uma área retangular totalmente livre de obstáculos.

A figura 25 mostra um exemplo de posicionamento global de área. Note que a área é dividida em retângulos regularmente espalhados, sendo que as células são apenas assinaladas a um dos retângulos. No caso da figura 25, as células são posicionadas no centro de cada retângulo, havendo intercessão de área entre elas. Porém esta questão é irrelevante para o posicionamento global. O problema de posicionamento global visa espalhar as células ao longo do circuito. O posicionamento detalhado, sem intercessão de células é realizado em uma etapa posterior.

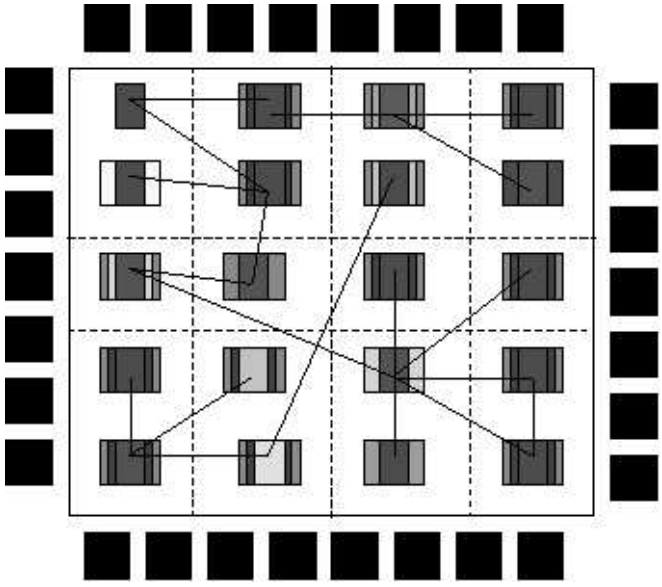

FIGURA 25 – Posicionamento Global em Área

Posicionamento Global se aplica em diversas áreas:

a) Estimativa eficiente de congestionamento. Na sessão 3.3.1.2 comenta-se que trabalhar com congestionamento global dá resultados semelhantes ao congestionamento a nível de posicionamento detalhado, porém com maior eficiência em tempo de processamento gasto.

b) Estimativa de distribuição da dissipação de potência (sessão 3.3.2). A partir do congestionamento das conexões é possível estimar onde vai haver focos de

dissipação de potência devido às conexões. Ainda, a distribuição das células permite analisar a distribuição da dissipação ao longo do circuito.

c) Estimativa de timing (sessão 3.3.3). A partir da distribuição das conexões por posicionamento global, sabe-se o formato das maiores conexões do circuito, o que é uma informação de extrema relevância para análise de Timing.

Caldwell, em 2000, apresenta um algoritmo de posicionamento baseado em posicionamento global. O princípio é de particionamento em células globais, de 15 células (aproximadamente) cada. Como a quantidade de células é pequena, o algoritmo trabalha com uma pesquisa exaustiva de forma que é capaz de encontrar o posicionamento detalhado ótimo para cada bloco.

4 Classificação de algoritmos de posicionamento

Um algoritmo de posicionamento é uma seqüência finita de passos que deve encontrar um posicionamento válido. Como já foi visto, encontrar um posicionamento qualquer é uma tarefa trivial, porém otimizar um posicionamento para qualquer conjunto não vazio de variáveis é um problema np-completo[11]. Daí a dificuldade em encontrar algoritmos eficientes, com tempos de CPU aceitáveis e que sejam capazes de considerar e otimizar *timing*, potência, área e roteabilidade de um circuito.

Os algoritmos de posicionamento se classificam, por sua entrada, como:

- Algoritmos Construtivos
- Algoritmos Iterativos

Os algoritmos construtivos recebem apenas um *netlist* de entrada. A partir apenas do *netlist*, devem gerar um posicionamento, conhecido como inicial. Os algoritmos iterativos recebem como entrada o *netlist* e também um posicionamento inicial, que será melhorado iterativamente. Os algoritmos construtivos, em geral, são utilizados como entrada para um algoritmo iterativo, que irá iterar a fim de melhorar o resultado final.

Desta forma, o fluxo de posicionamento é dividido, tipicamente, em três etapas:

1- Particionamento do problema. Um circuito com 1 milhão de células não pode ser tratado como um único bloco, pois iria necessitar de muito uso de CPU e memória, acima dos padrões aceitáveis por uma ferramenta de CAD. A etapa de particionamento cria subproblemas de posicionamento, que serão tratados na seqüência.

2- Posicionamento inicial, ou construtivo. Basicamente, parte de um *netlist* gerado pelo particionamento, visando um posicionamento inicial, que será entrada para o posicionamento iterativo. Para esta etapa, é usado um algoritmo construtivo.

3- Posicionamento Iterativo. Parte do posicionamento inicial com o objetivo de melhorar ele para uma determinada função de otimização, conforme visto no capítulo 3. Para isto, é utilizado um algoritmo iterativo.

Outra classificação dos algoritmos de posicionamento é por serem:

- Determinísticos
- Não-Determinísticos (Probabilísticos)

Os algoritmos determinísticos têm sempre a mesma saída para uma mesma entrada. Os algoritmos não-determinísticos baseiam-se em cálculos de probabilidade e funções aleatórias e, por isto, variam sua saída para a mesma entrada. Em algumas aplicações, como algoritmos genéticos, é desejável gerar um conjunto de soluções diferentes entre si. Nos algoritmos genéticos, é desejado que haja uma diversidade de soluções no começo do processo, para que o algoritmo possa convergir para soluções melhores, baseado nas soluções iniciais. Neste caso é necessário utilizar um algoritmo construtivo não-determinístico.

Neste trabalho, serão apresentados e detalhados diversos algoritmos, de acordo com as classificações acima. Inicialmente serão abordados os algoritmos construtivos:

[11] Parte da literatura afirma que posicionamento é np-completo, enquanto que outra parte afirma que é np-difícil. Em ambos casos, o algoritmo ótimo possui tempo exponencial.

Cluster Growth é o algoritmo intuitivo para posicionamento, pois apenas faz um esforço de posicionar, gradativamente, as células próximas. Este trabalho mostra que mesmo o CG apresenta dificuldades de implementação, à medida que não está totalmente especificado na literatura. Este trabalho apresenta diferentes maneiras de implementá-lo e discute as vantagens de cada uma. Está em aberto inclusive o seu caráter determinístico ou não. Na ferramenta do *Mango Parrot*, ele foi implementado de maneira probabilística. É feito um cálculo probabilístico para escolher a banda destino da célula.

Plic-Plac é um algoritmo apresentado neste trabalho. Ele parte de um princípio simples: faz uma ordenação das células a partir dos *pads*. De acordo com esta ordenação, calcula a melhor posição para a célula. No capítulo 9.1 são apresentados resultados de simulação observando seus bons resultados comparados com algoritmos de tempo de execução semelhante. Há, porém, bastante trabalho de pesquisa para ser desenvolvido sobre este algoritmo. É um algoritmo determinístico.

Posicionamento baseado em particionamento é o algoritmo construtivo mais clássico e com resultados de WL mais satisfatórios. Há muita pesquisa a respeito deste tipo de algoritmo (Caldwell, , Parakn, Ou, Wang, por exemplo). Este trabalho procura dar uma visão ampla desta classe de algoritmos, focando-se em implementações específicas feitas na ferramenta do *Mango Parrot*. Este tipo de algoritmo é uma área aberta para pesquisas em congestionamento (Parakn 1998), timing (Ou 2000), distribuição de potência, e outros problemas das tecnologias modernas. Há, portanto, pouco consenso de como implementar o posicionador ideal baseado em particionamento.

As **meta-heurísticas** são algoritmos muito usados para posicionamento. Especialmente a **Simulação de Têmpera** (Simulated Annealing) tem longa história. Este trabalho apresenta implementações do *Simulated Annealing* e de um **Algoritmo Genético**, fazendo algumas comparações entre ambos. As meta-heurísticas são iterativas e probabilísticas.

Por fim, o **posicionamento direcionado a forças** é outra área de pesquisa em aberto (Chou 2002). É um conjunto de algoritmos baseados em uma idéia muito simples: calcular uma posição considerando que as outras células estão fixas. A literatura usa de uma analogia a um sistema de forças para explicar didaticamente este algoritmo. A posição de força zero é a posição calculada por meio de fórmulas, como foi visto na sessão 3.3.1. O capítulo 7 deste trabalho apresenta os algoritmos direcionados a força em dois estilos diferentes: um deles construtivo e o outro iterativo.

Em resumo, a tabela 5 mostra os algoritmos abordados neste texto, e sua classificação quanto a entrada (Construtivo ou Iterativo) e comportamento (Aleatório ou Determinístico). Os algoritmos determinísticos podem ser modificados para serem aleatórios, como é o caso do Crescimento de Aglomerados (Cluster Growth) (sessão 5.3). A tabela 5 mostra a classificação do algoritmo da maneira que foi tratado por este trabalho. Outro exemplo é o caso do algoritmo baseado em particionamento. Usando um algoritmo de particionamento não-determinístico, obviamente que o posicionamento também será não-determinístico. Porém, neste trabalho foi usada a heurística determinística de Fidducia-Mateyeses.

TABELA 5 – Classificação de algoritmos de posicionamento

Algoritmo	Construtivo	Iterativo	Determinístico	Aleatório
Aleatório	X			X
Plic-Plac	X		X	
Crescimento de Aglomerados	X		X	
Baseado em Particionamento	X		X	
Force Directed Construtivo	X			X
Force Directed de Chou	X		X	
Force Directed Iterativo		X	X	
Simulação de Têmpera		X		X
Algoritmo Genético		X		X

5 Algoritmos construtivos

5.1 Introdução

Esta sessão destaca alguns algoritmos construtivos que solucionam o problema de posicionamento. Estes algoritmos são heurísticos, e procuram minimizar o comprimento total dos fios. Eles foram implementados na ferramenta de síntese do *Mango Parrot* para fins comparativos.

Este trabalho apresenta um novo algoritmo, que está em fase inicial, mas apresenta resultados preliminares promissores. Chama-se de Plic-Plac. Depois dele, destacam-se alguns algoritmos bastante conhecidos:

- Aleatório
- *Cluster Growth*
- Posicionamento baseado em Particionamento
- *Force Directed Placement*

O algoritmo aleatório simplesmente gera um posicionamento qualquer. Não há otimização, mas é o mais rápido. Se aplica perfeitamente para algoritmos iterativos que independem da solução inicial, como, por exemplo, o *Simulated Annealing* (usando ***high annealing*** **-** ver sessão 6.2). A principal utilidade dos algoritmos construtivos é ser entrada dos algoritmos iterativos. Chama-se de posicionamento inicial.

As sessões seguintes apresentam os algoritmos com detalhe. No capítulo 9, é feita uma comparação do tempo de execução e roteabilidade dos posicionamentos gerados por algoritmos construtivos.

5.2 Plic Plac

É um algoritmo de posicionamento inicial bastante eficiente. Sua proposta é substituir o algoritmo aleatório, que é ainda usado hoje para fazer posicionamento inicial para algoritmos iterativos como o *simulated annealing*. Os dados do capítulo 9 mostram que este algoritmo é tão rápido quanto o algoritmo aleatório e o tamanho dos fios é significativamente menor. Mostra-se também, no capítulo 9, que a combinação Plic-Plac seguido de *Simulated Annealing* pode resultar em posicionamentos menores que qualquer outro algoritmo construtivo implementado na ferramenta do *Mango Parrot* usando o mesmo tempo de CPU.

Baseia-se na idéia de percorrer todas as células partindo dos pinos de entrada e saída, de forma que se compute a distância que elas têm dos pinos. Esta pesquisa é feita com o intuito de ordenar as células horizontalmente e verticalmente. A figura 26 mostra as células ordenadas a partir dos pads da esquerda e de cima.

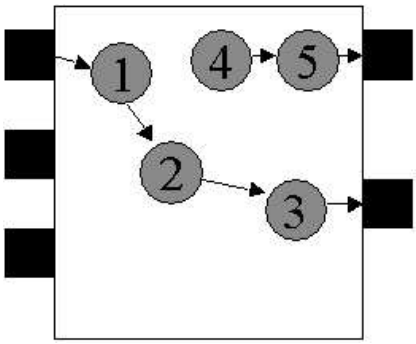

FIGURA 26 – Células ordenadas desde os pinos de E/S pelo algoritmo Plic-Plac

A algoritmo se divide em 4 etapas:
1 – Encontrar a ordem Horizontal de todas as Células
2 – Encontrar a ordem Vertical de todas as Células
3 – Mapear cada célula para uma banda de acordo com a ordem vertical
4 – Inserir as células nas bandas seguindo a ordem horizontal.

A etapa 1 é resolvida percorrendo as células a partir dos pinos da esquerda. Duas estratégias de pesquisa são consideradas: DFS e BFS. Em ambos estilos não é permitido a visita a um nodo que já foi pesquisado. Desta forma, encontra-se uma *spanning tree*[12] que cobre boa parte do grafo das células com suas vizinhanças. Observa-se, porém, que a pesquisa DFS encontra árvores mais altas (maior profundidade máxima) e magras (menor número médio de filhos por nodo). Já a pesquisa BFS encontra árvores mais baixas e gordas. A figura 27 mostra duas *spanning trees*. 27.a mostra uma árvore gerada por pesquisa DFS e 27.b BFS.

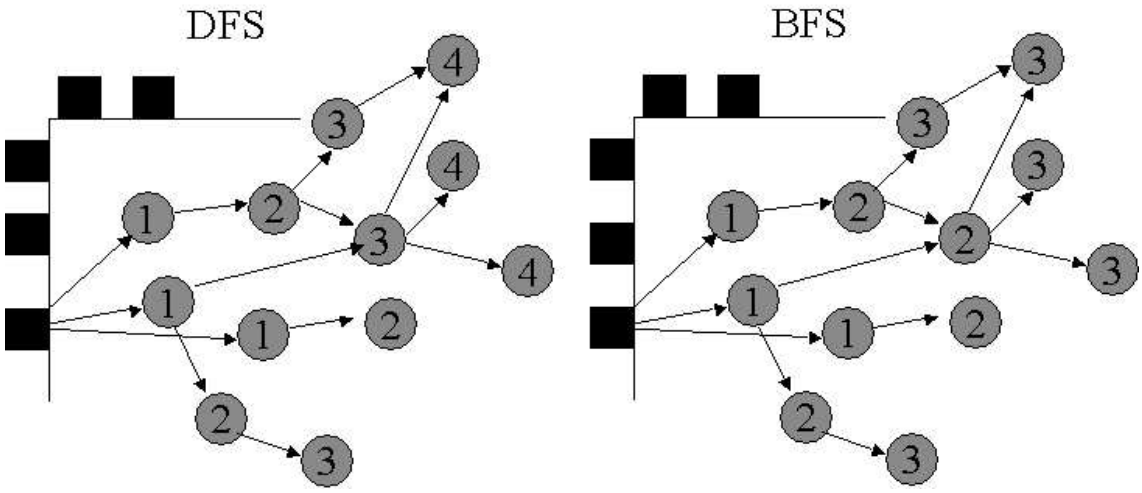

FIGURA 27 – Células ordenadas por pesquisa DFS e BFS

A pesquisa BFS encontra árvores com uma quantidade de filhos mais equilibrada, o que é bastante interessante para a pesquisa horizontal. Já para a pesquisa vertical, BFS não é apropriada por encontrar árvores de tamanhos menores, deixando de ocupar áreas do circuito (ver figura 28). O ideal, portanto, é usar BFS para pesquisa horizontal e DFS para pesquisa vertical. A figura 28

[12] *Spanning Tree* é um termo usado para uma árvore que cobre todos os nodos de um grafo. No contexto onde foi utilizado neste texto, o termo Spanning Tree se refere a uma árvore que cobre parte dos nodos do grafo, pois nem todos são necessariamente atingidos pela pesquisa.

mostra o resultado de posicionamento usando as pesquisas propostas (da esquerda para a direita: exclusivamente DFS, exclusivamente BFS e mista).

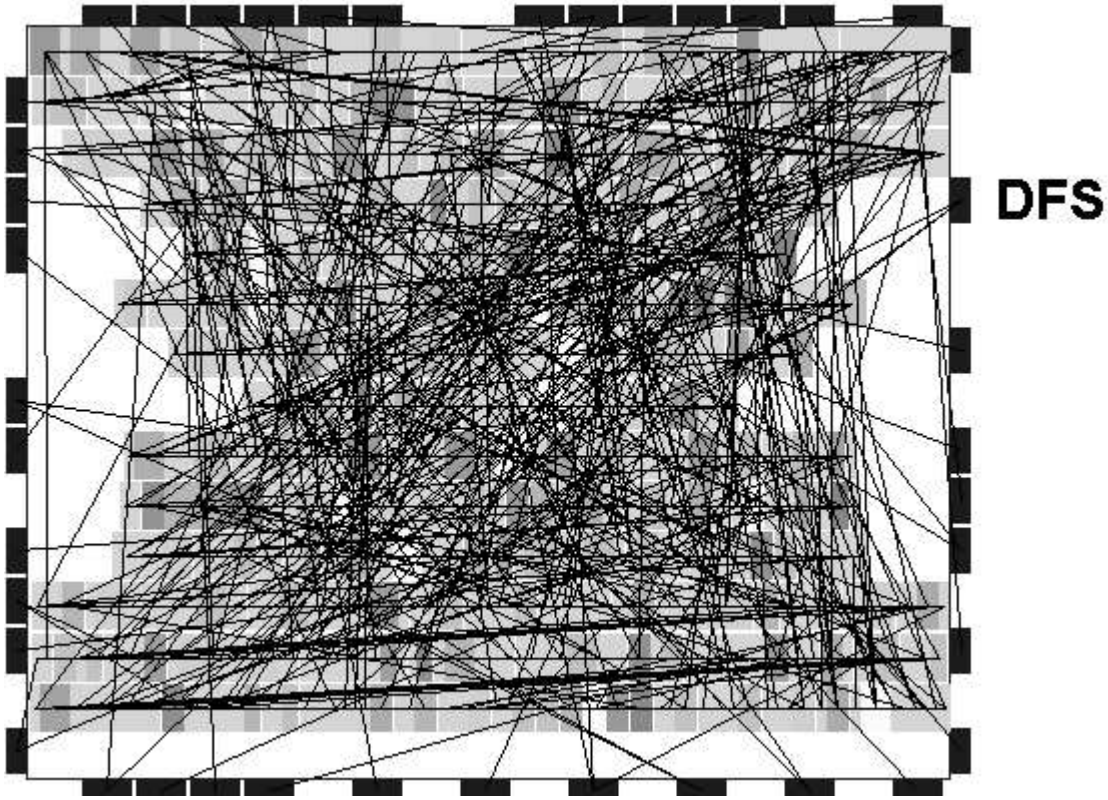

FIGURA 28.a– Posicionamento gerado por uma pesquisa DFS

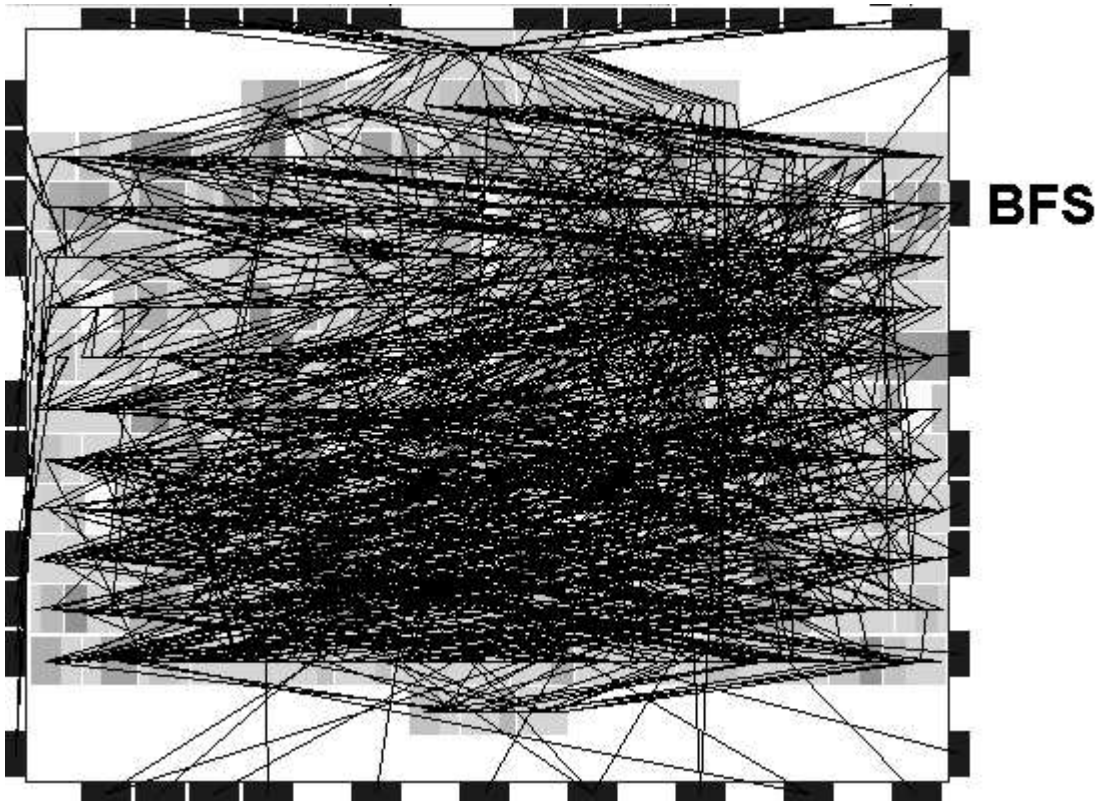

FIGURA 28.b– Posicionamento gerado por uma pesquisa BFS

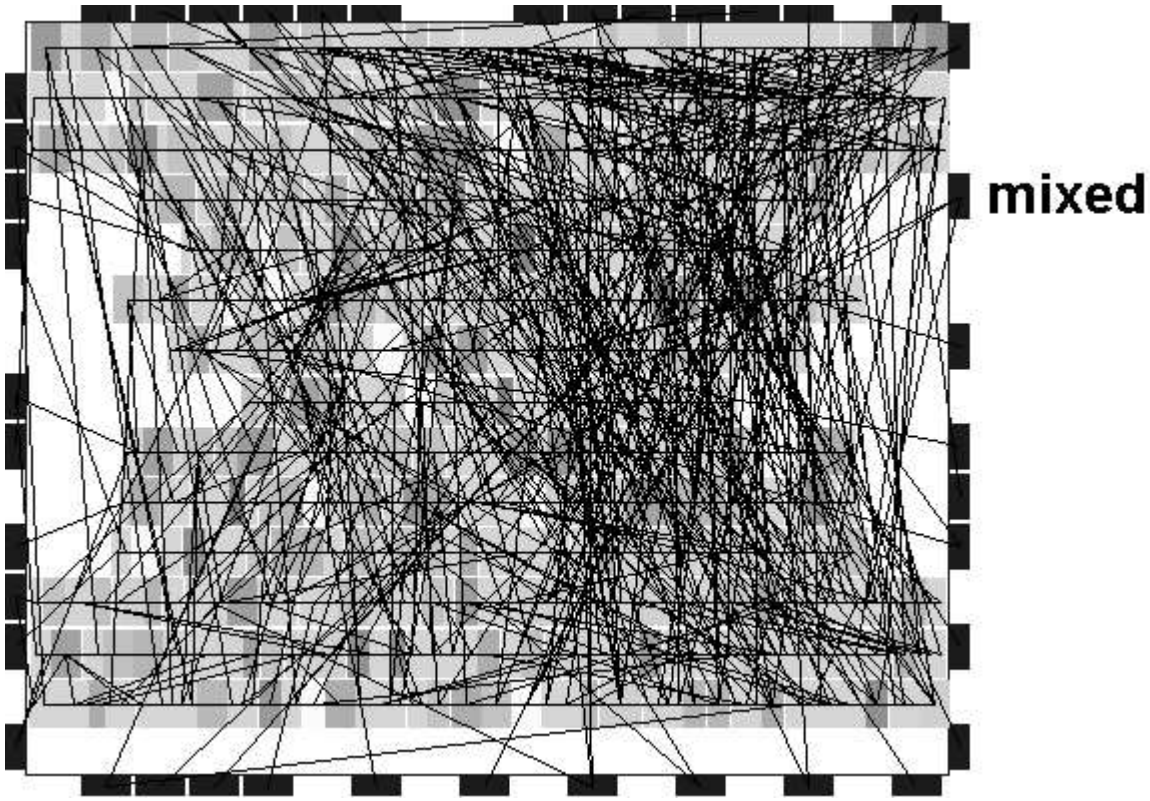

FIGURA 28.c– Posicionamento gerado por uma pesquisa mista DFS/BFS

Ambas as pesquisas podem deixar de visitar algumas células. Assim, algumas células ficam sem valor de ordem. Para corrigir o problema, são propostas algumas técnicas:

- Fazer uma pesquisa no sentido contrário, partido dos pinos da direita. Todos os nodos que já foram visitados não são modificados. A pesquisa inversa coloca INT_MAX nas células vizinhas aos pinos e vai decrementando o valor para os vizinhos. Depois de terminadas as duas pesquisas, as células estarão ordenadas. É possível ainda que algumas células não tenham sido atingidas por nenhum pesquisa. Neste caso, estas células serão posicionadas aleatoriamente no sentido horizontal.
- Fazer a mesma pesquisa no sentido contrário citada acima, porém armazenar nos nodos que já foram visitados uma média das duas pesquisas. Assim, os nodos visitados pelas duas pesquisas tenderiam a ficar no centro, enquanto que os demais tenderiam a ficar na periferia.

Note que ambos os casos acima resultam em uma ordenação irregular, com alguns "buracos" entre os números. Por exemplo, um caso típico de ordenação da primeira proposta seria: 0,1,2,3...,7,1000,1001,1002. No segundo caso haveria ainda mais buracos, como por exemplo os seguintes valores: 0,1,2,3,500,501,502,1000,1001. Porém esta diferença drástica entre duas células não reflete a realidade do posicionamento, pois o desejado é que estas células sejam posicionadas lado a lado. Assim, uma etapa posterior deve eliminar as falhas na ordenação.

A figura 29 ilustra os dois casos citados acima.

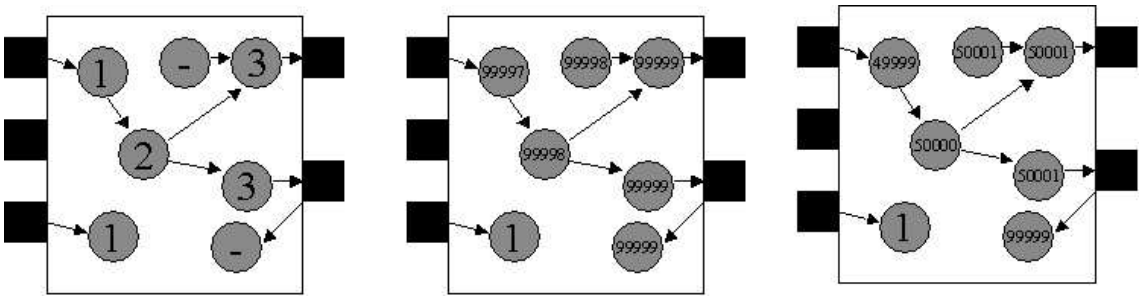

FIGURA 29 – Dois casos de ordenação irregular

A etapa 2 do algoritmo é análoga a etapa 1, porém a pesquisa parte dos pinos de cima (ou de baixo).

A etapa 3 consiste no mapeamento das células para uma banda. Para isto é usada a ordem vertical das células. O intervalo [0 ; max_ordem_vertical] deve ser mapeado para o intervalo [0 ; num_bandas - 1]. Usa-se a seguinte equação:

$$banda = \frac{ordem\_y \times num\_bands}{máxima\_ordem\_vertical + 1} \qquad [4]$$

É possível fazer uma verificação de tamanho máximo, para que nenhuma banda seja larga demais. Neste caso, o ideal é que seja selecionada uma banda próxima à banda calculada.

A etapa 4 do algoritmo realiza a inserção das células em suas bandas correspondentes. Parte-se, portanto, de um posicionamento vazio (i.e. estruturas de dados que armazenam nenhuma célula). A inserção deve seguir a ordem horizontal. Assim, primeiro são inseridas as células cuja ordem horizontal é 1, depois 2, e assim por diante. Após esta etapa, o posicionamento estará concluído. A figura 30 mostra dois posicionamentos gerados com a estratégia mista BFS/DFS. O primeiro usa limite de tamanho horizontal, atingindo um custo final de 442421. O segundo não usa limite, atingindo um custo de 480829.

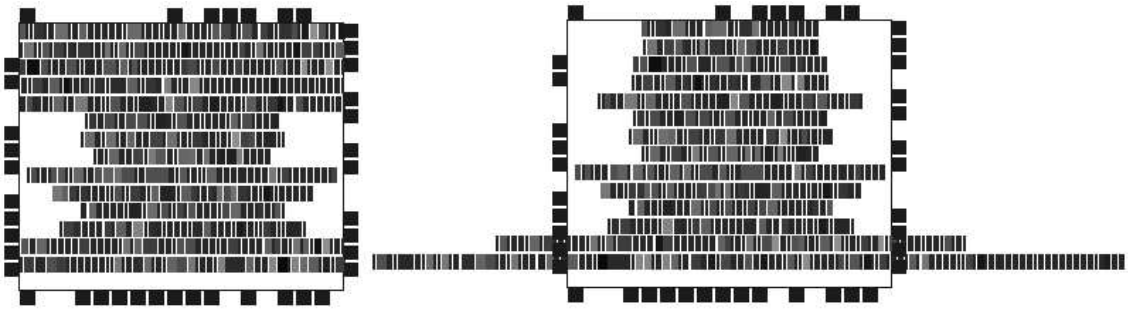

FIGURA 30 – Posicionamento gerado pelo Plic-Plac com e sem limite de largura de bandas

5.3 Crescimento de Aglomerados

Em inglês, como é mais conhecido, chama-se de *Cluster Growth*. A idéia é de formar aglomerados de células que estão conectadas. Parte-se de uma célula denominada semente. Todas as células vizinhas são inseridas da maneira mais próxima possível. O processo é iterativo até que se insiram todas as células.

Sherwani mostra o algoritmo conforme a figura 31.

```
Algoritmo de Formação de Aglomerados

Selecionar a Semente
Criar o Conjunto X de células a serem
Posicionadas
Posicionar a Semente
Enquanto o Conjunto X não for vazio
  Escolher uma Célula B pertencente ao
conjunto X
  Posicionar a Célula B
  Retirar B do conjunto X

O(n)
```

FIGURA 31 – Algoritmo de Cluster Growth conforme Sherwani

Observa-se na figura 31, porém, que quando o conjunto X ficar vazio, é necessário escolher outra semente. Este algoritmo, na figura, deixa algumas questões em aberto ainda. Por exemplo, como é posicionada a célula. O que é feito depois de posicionados todos os filhos da semente (i.e. como é escolhida a próxima semente)?

A figura 32 mostra o algoritmo de crescimento de aglomerados em uma versão mais genérica, em um fluxograma.

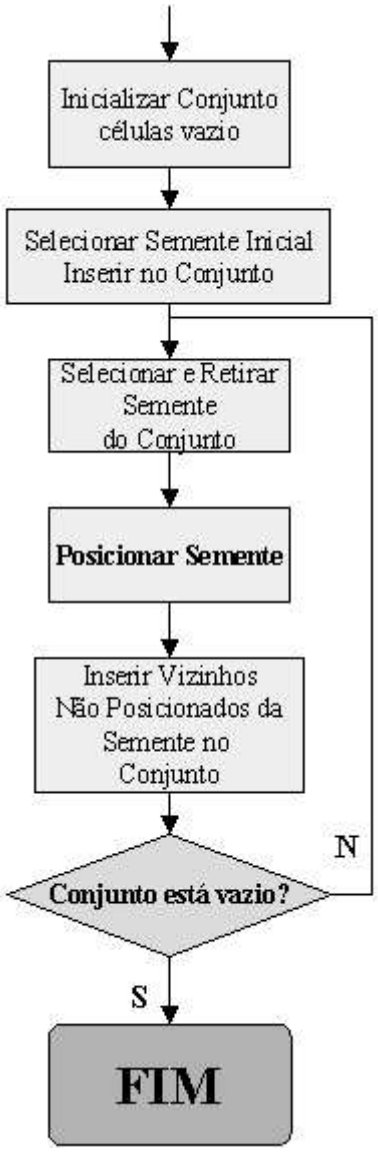

FIGURA 32 – Algoritmo de Crescimento de Aglomerados em Fluxograma

Na ferramenta do *Mango Parrot*, foram feitas quatro implementações diferentes deste algoritmo: A,B,C e D. Em todas elas, é feita uma lista de possíveis sementes inicialmente, ordenada por ordem de conectividade. A célula com mais conexões é escolhida inicialmente. A partir daí as implementações são diferentes.

Na implementação A, a semente é inserida no final de uma banda escolhida aleatoriamente. Em seguida, suas vizinhas são inseridas no final de uma banda escolhida aleatoriamente também, mas restrita às três bandas vizinhas da semente. Depois de inseridas as vizinhas da semente, é escolhida uma nova semente, repetindo-se o processo até que todas as células sejam inseridas. Este processo tende a distribuir os aglomerados ao longo do circuito, fazendo com que haja um certo equilíbrio do tamanho das bandas. Porém, não é muito regular, pois os aglomerados têm tamanhos variados. A figura 33 mostra o posicionamento final gerado por estes algoritmos para o circuito C1908_3x3:

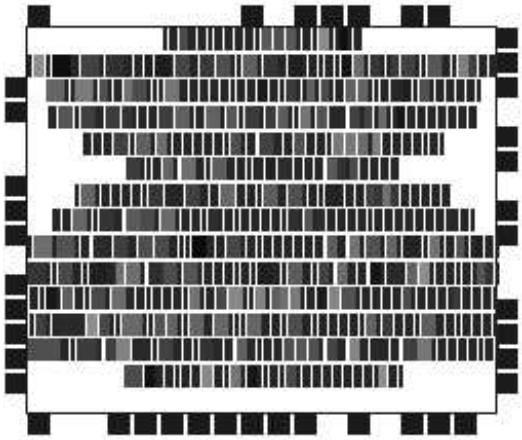

FIGURA 33 – Posicionamento com *Cluster Growth* A

A segunda implementação (impl. B) é um processo semelhante. A diferença está na escolha das novas sementes. A segunda semente será uma das vizinhas da primeira semente. Todas as vizinhas da primeira semente serão inseridas, antes que as células netas (vizinhas das vizinhas) da semente sejam inseridas. Caracteriza-se, portanto, por ser uma pesquisa em largura (BFS). O método irá inserir todas as células em um único aglomerado, a menos que haja partes desconexas. Na ferramenta do *Mango Parrot*, foi implementada uma limitação de altura da árvore de pesquisa, a fim de evitar o aglomerado único. Chama-se de limite de profundidade. Se o limite for 1, o algoritmo comportar-se-á como a implementação A. Se o limite for 0, comportar-se-á como um algoritmo aleatório. A tendência deste algoritmo é que formem-se aglomerados grandes, ocupando completamente algumas bandas. A figura 34 mostra alguns posicionamentos usando este algoritmo (respectivamente CGB2, CGB5, CGB20).

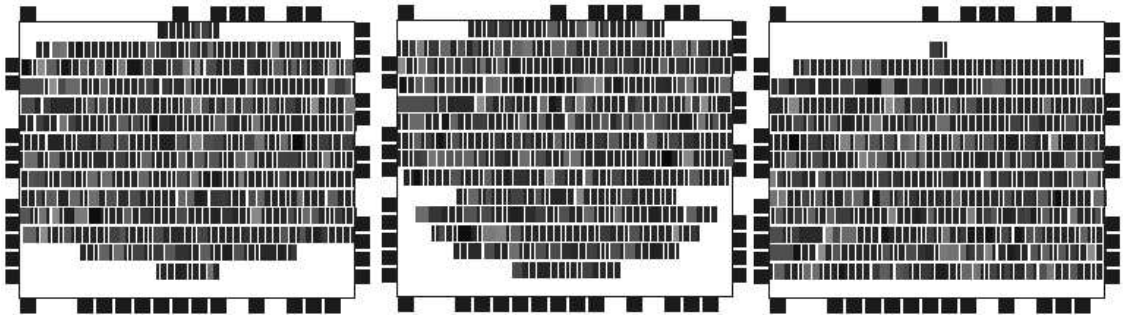

FIGURA 34 – Posicionamentos usando *Cluster Growth* B

A terceira implementação (implementação C) modifica o algoritmo significativamente, atingindo resultados interessantes (tabela 6). Ela baseia-se na implementação B, porém o cálculo

da banda onde a célula será inserida é feito através de uma média das posições de suas vizinhas (caso já tenham sido posicionadas). Assim, a primeira semente será inserida aleatoriamente. As vizinhas da primeira vizinha da semente serão inseridas em uma banda próxima a da semente (como nas implementações A e B). Porém, nos demais casos, a banda de destino é calculada em função das células vizinhas que já foram inseridas. Assim, no começo do processo, o algoritmo comportar-se-á semelhante ao B, porém no final, as células assumirão bandas médias com relação às suas vizinhas Quanto maior o bloco, a tendência é que as células sejam bem distribuídas verticalmente, fazendo com que as bandas tenham tamanhos equalizados, o que é muito interessante para um algoritmo construtivo. A figura 35 mostra alguns blocos gerados por este algoritmo, variando-se a profundidade máxima (respectivamente CG C2, CG C5, CG C20).

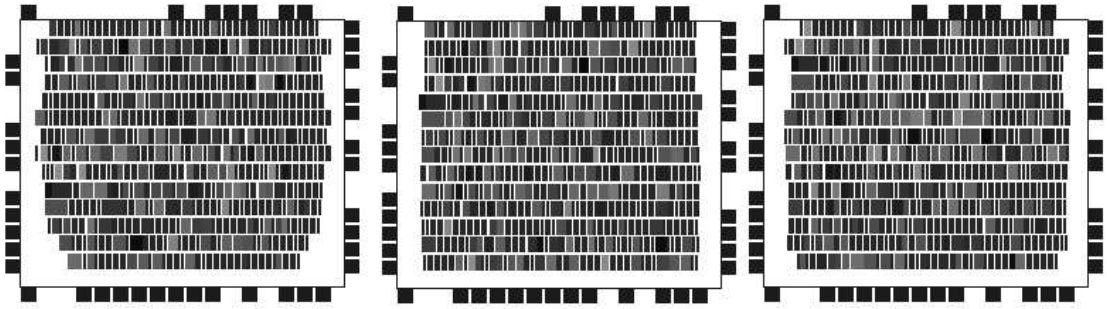

FIGURA 35 – Posicionamentos usando *Cluster Growth* C

A implementação D é semelhante à A, porém, assim como a implementação B, difere na escolha da próxima da próxima semente. A próxima semente será uma das filhas da semente inicial, mas somente uma. Caracteriza-se, portanto, por uma pesquisa em profundidade. A figura 36 mostra um posicionamento gerado por este algoritmo.

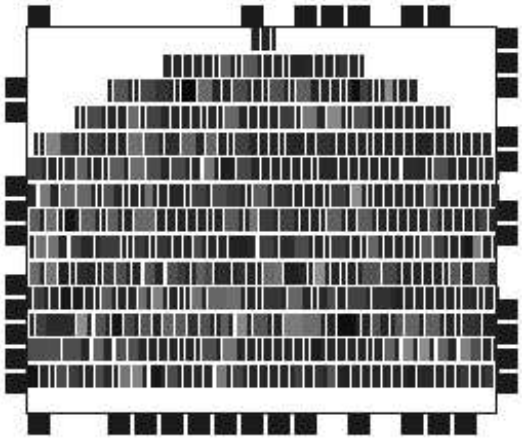

FIGURA 36 – Um posicionamento usando o *Cluster Growth* D

Foram feitas diversas execuções da ferramenta do *Mango Parrot* nos benchmarks C1908_3x3, Alu2_4x4 e Alu4_4x4, a fim de caracterizar os métodos. Os resultados estão na tabela 6.

TABELA 6 – Dados de execução das quatro variações propostas no algoritmo Cluster Growth

Algoritmo	C1908_3X3		Alu2_4x4		Alu4_4x4	
	WL	CPU (s)	WL	CPU (s)	WL	CPU (s)
CG A	446082	0	255232	0	5535579	5
CG B0	849795	0	297877	0	6667289	3
CG B1	430404	0	263189	0	5530945	5
CG B2	441137	0	263893	0	5692462	5
CG B5	448096	0	261973	0	5723610	5
CG B10	484975	0	261546	0	5721248	4
CG B20	493896	0	260749	0	5746936	4
CG C1	523532	0	240356	0	5191940	3
CG C2	448017	0	229814	0	5066699	4
CG C5	356986	0	227870	0	5232436	4
CG C10	324265	0	226834	0	5314137	4
CG C20	323729	0	225302	0	5290302	4
CG D	436630	0	256828	0	5347816	5

Inicialmente, vamos avaliar o que acontece no circuito C1908. O melhor algoritmo foi o CG C20. Observe que o grupo de Algoritmos CG C tem desempenho superior em relação a todos os outros algoritmos. Isto se deve ao cálculo da banda de destino, que é uma média aritmética das bandas dos vizinhos. Quanto maior o aglomerado, melhores os resultados para este caso, mas depende muito da organização do circuito. No Alu2, por exemplo, observa-se o mesmo fenômeno, mas no alu 4 os aglomerados maiores foram piores que o aglomerado de 2 níveis. Em todos os casos, porém, a implementação C é a vencedora.

5.4 Algoritmos Baseados em Particionamento

O algoritmo de posicionamento baseado em particionamento desenvolve uma seqüência de particionamentos com o objetivo de isolar as células mais conectadas em partições. Em geral, a seqüência é realizada hierarquicamente. Esta sessão trata apenas de implementações hierárquicas (recursivas) dos algoritmos de corte.

Os cortes, feitos hierarquicamente, baseiam-se em diversos bi-particionamentos da área de posicionamento. Está em aberto, porém, a ordem e o sentido (horizontal ou vertical) do corte. A estratégia de corte é um problema muito interessante que pode modificar significativamente o resultado do posicionamento, como observa Yildiz em 2001. Yildiz mostra, através de um modelo matemático, que usar biparticionamentos sucessivos sem sentidos alternados é a melhor alternativa.

Depois de escolhida a direção do corte, faz-se um particionamento das células visando minimizar o corte, ou seja, a quantidade de redes que cruzam as partições. Para isto, são usados algoritmos de particionamento. A sessão 5.4.2 apresenta uma revisão de alguns dos algoritmos mais utilizados para particionamento.

Outra variação possível, portanto, é o algoritmo de particionamento escolhido. Alguns algoritmos de particionamento equilibram a área das partições, enquanto outros equilibram o número de elementos. Para o posicionamento, o ideal é que seja equilibrada a área. Ainda, os algoritmos construtivos de particionamento podem ser combinados, assim como os algoritmos de posicionamento, com os iterativos.

Os algoritmos baseados em particionamento devem ainda considerar a posição dos *pads*. Eles devem, portanto, acrescentar as redes ligadas aos *pads* no cálculo do corte. É preciso, portanto, escolher, para cada *pad*, a que partição o mesmo pertence, ou a qual mais se aproxima. A ferramenta do *Mango Parrot* implementa uma heurística no algoritmo de particionamento para considerar *pads* no particionamento (ver seção 5.4.2.3, passo 1). Outra alternativa é encontrada no

trabalho de Andrew Caldwell, em 1999, que trata a questão de particionamento com vértices fixos, como é o caso de *pads*.

5.4.1 Estratégias de Corte

São conhecidas na literatura (Sherwani) as seguintes estratégias de corte, apresentadas inicialmente por Breuer:

1- Orientado a corte
2- Quadratura
3- Bisseção em bandas hierarquicamente
4- Bisseção em fatias

As estratégias mais utilizadas são **Quadratura** (2) e **Bisseção em Bandas Hierarquicamente** (3). Elas serão apresentadas nas sessões 5.4.1.2 e 5.4.1.1 respectivamente. Ambas sempre fazem bipartições em partições de tamanhos iguais, o que torna-se uma facilidade para o algoritmo de particionamento.

A estratégia orientada a corte (1) é ilustrada na figura 37.a Ela faz bipartições sucessivas de maneira seqüencial e não hierárquica. Assim, as bipartições devem tratar duas partições de tamanhos diferentes. Ela é pouco usada pelo fato de que algoritmos de particionamento não lidam adequadamente com este tipo de estratégia. A minimização do corte na primeira partição será máxima, porém as demais serão prejudicadas.

A estratégia de bisseção em fatias (2) é ilustrada na figura 37.b. Sherwani afirma que esta estratégia diminui o congestionamento na periferia do circuito. No caso de muitas conexões de I/O esta estratégia teria uma suposta vantagem, portanto. Mas ela apresenta a mesma dificuldade da estratégia (1) em função de tamanhos diferentes de partições.

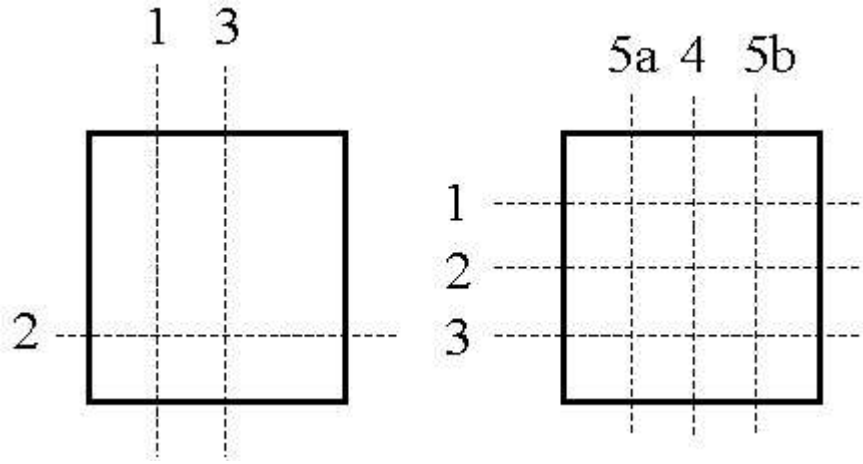

FIGURA 37 – Estratégias de corte por fatias

5.4.1.1 Bandas Hierarquicamente

É uma seqüência de cortes hierárquica como mostra a figura 38. Ela baseia-se na idéia de dividir o circuito em bandas e depois organizar as bandas com particionamentos verticais. Assim, primeiro é feito uma partição do circuito em duas fatias horizontais. Para cada fatia, a função é aplicada recursivamente, até que o circuito esteja dividido em fatias horizontais da altura de uma banda de células. A partir daí, cada banda é particionada no sentido vertical. Primeiro divide-se a banda ao meio e recursivamente repete-se o processo para cada metade. Termina-se o processo baseado em algum dos seguintes critérios de parada:

- Número fixo de níveis de hierarquia
- Partições com menos que X elemento(s)
- Espaço menor que Y micro metros

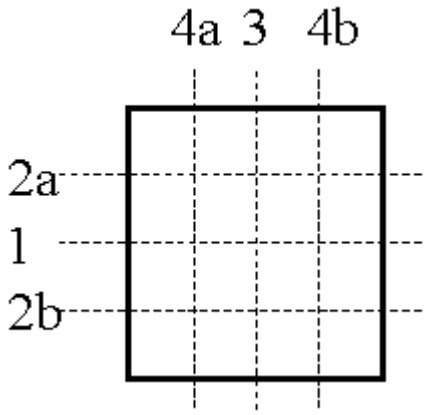

FIGURA 38 – Estratégia de cortes por Bi-Particionamentos horizontais

A primeira possibilidade de parada não é boa, pois circuitos grandes exigem mais níveis do que circuitos pequenos. Porém, é importante que haja este tipo de limitação, juntamente com outro, para que não haja estouro de memória devido a recursividade.

Limitar o tamanho físico da partição, como aponta a terceira alternativa, resolve o problema da diferença de níveis exigido nos circuitos grandes em relação aos pequenos. Porém, isto pode fazer com que o algoritmo perca tempo com partições sem elementos.

A alternativa mais usada é a segunda, onde o algoritmo pára quando as partições tiverem um número X de elementos. Quanto maior X, maior será a liberdade do posicionador que vem após o particionamento. No trabalho de Caldwell (2001) usa-se X próximo de 15, juntamente com um posicionador baseado em pesquisa exaustiva. Desta forma, garante-se que o posicionamento será ótimo para cada partição. Por experimentos, Caldwell constatou que 15 células são adequados para a eficiência que eles desejam alcançar. Outra alternativa de X seria usar X =1, ou seja, parar o particionamento quando tiver somente uma célula.

Este trabalho buscou investigar a primeira alternativa , misturada com a segunda para X = 0. Ou seja, o algoritmo pára de particionar quando chegar a uma partição vazia ou quando atingir um certo nível de hierarquia. Pelas simulações executadas, observa-se que o critério que define realmente a velocidade é o número de níveis de hierarquia. Então, buscou-se investigar o impacto do limite de profundidade da recursividade no tamanho das conexões e no tempo de processamento. Estes dados são apresentados na tabela 7 para três benchmarks. O primeiro, alu2 4x4, é considerado pequeno. O segundo, C1908 3x3 é de tamanho médio, enquanto que o alu4 4x4 é considerado grande. Em todos os casos, utilizou-se o algoritmo Cluster Growth B 2 (ver na sessão 5.4.2.1) seguido do Fidduccia-Matteyeses (sessão 5.4.2.3) para particionamento. A tabela mostra também o número de chamadas recursivas para cada caso. Os algoritmos estão desprezando os pads.

TABELA 7 – Impacto do Limite de Recursividade em Posicionamento por Bi-Particionamentos Horizontais Sucessivos

	Alu2 4x4			C1908 3x3			Alu4 4x4		
limite	Tempo (s)	WL	Chamadas	Tempo (s)	WL	Chamadas	Tempo (s)	WL	Chamadas
1	0	229436	32	0	343068	44	18	4288639	101
2	0	223476	59	1	313306	74	18	3874224	169
4	0	218925	199	1	305360	254	27	3425645	577
6	0	223284	496	3	304687	866	37	3559465	2137
8	1	215638	812	5	303325	1694	168	3583108	6135
10	1	220247	1120	8	301482	2529	554	3574313	11609
12	2	228524	1461	15	301473	3364	1254	3665179	17005

O limite igual a um (1) significa que não há particionamento vertical. Os dados, para este caso, são muito interessantes, pois mostram que os particionamentos iniciais têm maior peso no comprimento dos fios em relação aos finais. No circuito menor, alu2, observa-se que não há ganho significativo com o aumento da profundidade. No circuito C1908, observa-se a partir do limite 4 o ganho não é significativo e o tempo cresce rapidamente. No caso do maior circuito, o ganho é mínimo também, podendo até haver perda (indicando um comportamento aleatório).

O maior problema deste estilo de corte é a falta das vizinhanças verticais. Observe que o particionamento minimiza o corte entre duas bandas vizinhas. Porém, deseja-se que o corte seja pequeno entre células distantes e grande em células próximas. Observe o tamanho de uma banda. O algoritmo vai preferir manter duas células conectadas na mesma banda do que em bandas diferentes, desconsiderando a possibilidade de proximidade vertical.

5.4.1.2 Quadratura

Baseia-se em particionamentos quádruplos da região (figura 39). A sessão 5.4.2.6 mostra como evoluir de um algoritmo de biparticionamento para um algoritmo que trata múltiplas partições, como é o caso da quadratura. Usualmente, utiliza-se o método hierárquico. Neste caso, a quadratura se confunde com uma bisseção em direções alternadas. Yildiz mostra que a bisseção alternada é a melhor estratégia de corte.

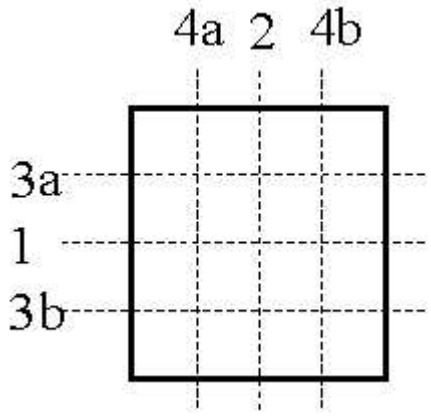

FIGURA 39 – Estratégia de corte por Quadratura

O particionamento em quadratura é repetido até que se tenha uma partição com a altura de uma banda. Após isto, pode-se parar o particionamento ou seguir com cortes verticais. A opção de parar cedo demais exige uma inteligência maior do posicionador. Pode ser interessante para o uso de pesquisa exaustiva, como Caldwell (2001), por exemplo. A opção de seguir com cortes verticais leva ao uso de um dos critérios de parada citados na sessão anterior (5.4.1.1). Porém é uma questão interessante definir qual o ponto de parada ideal para escolher.

Na ferramenta do *Mango Parrot* foi implementado o esquema de seguir com cortes verticais, seguindo-se por X níveis de hierarquia máximos, pré-definidos. Na sessão anterior, foi visto que biparticionamento em bandas necessitava de 4 níveis, no mínimo, de particionamento horizontal (com cortes verticais) para atingir bons resultados no circuito alu4. No caso da quadratura, os cortes verticais serão feitos em regiões muito menores, de forma que menos níveis de hierarquia são requeridos. A tabela 8 mostra, para três benchmarks, a evolução do tempo de execução, repetições de recursividade e *wirelength* para diferentes níveis de profundidade de cortes verticais. Vale lembrar que o posicionamento das células dentro do bloco é aleatório. Porém, aumentar a profundidade da recursividade dos cortes para tornar os blocos ainda menores pode ser inútil, demandando muitos recursos computacionais.

TABELA 8 - Impacto do Limite de Recursividade em Posicionamento por Quadratura

	Alu2 4x4			C1908 3x3			Alu4 4x4		
Limite	Tempo (s)	WL	Chamadas	Tempo (s)	WL	Chamadas	Tempo (s)	WL	Chamadas
1	0	265483	373	1	301436	541	27	4512012	2837
2	0	269411	516	2	304379	899	41	4577670	4185
4	0	273729	826	4	295988	1705	210	4422484	7081
6	1	262421	1115	6	299939	2488	342	4528856	10057

Os dados da tabela 8 mostram que os cortes verticais surtiram pouco efeito no WL de todos os circuitos, sendo em muitos casos um efeito aleatório. Comparando os dados com a tabela 7 (na sessão anterior), observa-se que a quadratura foi levemente melhor em C1908 e bem pior nas duas alus. Ainda, o número de chamadas recursivas é maior na quadratura com limite 1 com relação a bisseção em bandas com limite 4. Isto justifica o maior tempo de processamento e aponta um maior consumo de memória para armazenar as variáveis locais. O tamanho final dos fios é significativamente afetado pela falta de *terminal propagation* (sessão 5.4.3).

Uma vantagem da quadratura, segundo Sherwani, é que este método reduz a concentração de conexões no centro do circuito.

5.4.1.3 Outras estratégias de corte

Mehemet Yildiz, em 2001, apresenta um trabalho muito interessante para a definição da seqüência ótima de cortes. A principal contribuição do trabalho é mostrar que a seqüência de corte por bisseções sucessivas em direções alternadas é de fato a melhor estratégia. Sempre houve a especulação de que esta seqüência era de fato a melhor, porém Yildiz reforça esta hipótese com um modelo formal, assumindo algumas outras hipóteses.

O trabalho de Yildiz define um método para encontrar a seqüência ótima de corte. Inicialmente, ele define como estimar o tamanho dos fios de um dado corte antes de efetivamente dar o corte, usando a regra de Rent (sessão 3.3.1.3). A partir daí, ele modela, através de programação linear, como encontrar a melhor posição para o corte, e com qual porcentagem de área para cada partição.

A partir deste modelo, foram feitos alguns experimentos que mostraram duas propriedades surpreendentes: 1- todos os cortes foram bisseções de área totalmente balanceadas, ou seja, 50% para cada partição. 2- se a relação de aspecto de bandas por colunas exercer um certo valor, o particionamento é feito horizontalmente; caso contrário, verticalmente. Isto leva a uma conclusão muito importante: **a definição da direção do corte pode ser encontrada apenas com uma análise da relação de aspecto da região**. Ainda, **o corte sempre deve ser dado na metade da área**

Claramente, o cálculo de programação linear é muito complexo e não deve ser usado durante a fase de posicionamento. Porém, mostra-se que de uma maneira muito rápida pode-se saber qual a melhor direção de corte a ser dada. No caso de uma área quadrada, esta direção será dada sempre de maneira alternada.

5.4.2 Algoritmos de Particionamento

Algoritmos de particionamento têm o objetivo de distribuir as células por duas ou mais partições (conjuntos de células) de forma que haja um equilíbrio de área e que sejam minimizadas as conexões que cruzam partições. Este trabalho vai apresentar dois algoritmos para bipartição. Na sessão 5.4.2.6 é mostrado como eles podem ser estendidos para *n* (*n*>2) partições.

Formalmente, o bi-particionamento é definido da seguinte forma:

Seja H um hipergrafo (V,E), onde V é um conjunto de células e E é o conjunto de redes, ou seja, conexões multi-ponto. Sejam A e B duas partições (conjuntos de células). Seja *cut*() uma função que diz quantas redes estão cruzando as partições. O problema de posicionamento é encontrar um mapeamento V=> A e V=>B de forma que A∩B = {}, A∪B = V e que *cut()* seja mínimo. Assim como o posicionamento, o particionamento mínimo é um problema np-completo. Os algoritmos apresentados nesta sessão são, portanto, heurísticos.

Os algoritmos de particionamento se classificam, quanto a técnica utilizada, desta forma:

- Migração de grupos (2 células): Iterativamente, uma célula de cada partição é escolhida para trocar de grupo, ou seja, migrar.
- Migração de grupos (1 célula): Neste caso, uma célula apenas é escolhida.
- Replicação de componentes. Não há movimentações de células de uma partição para outra, mas sim replicação de células que estão prejudicando o corte.
- Crescimento de aglomerados: é um método construtivo de criar dois ou mais grupos de células.

As técnicas de migração de grupos são mais utilizadas que replicação de componentes por não aumentarem a área resultante do circuito. A migração de duas células foi proposta por Kernighan-Lin e se caracteriza por manter a quantidade de células em cada partição durante todo o processo. Desta forma é fácil equilibrar as partições por quantidade de componentes. Porém, a técnica mais utilizada é de migração de um componente somente, onde é possível equilibrar as partições por área, por exemplo. As Figuras 40 e 41 mostram as duas técnicas graficamente. A técnica de crescimento de aglomerados pode ser usada para particionamento inicial[13].

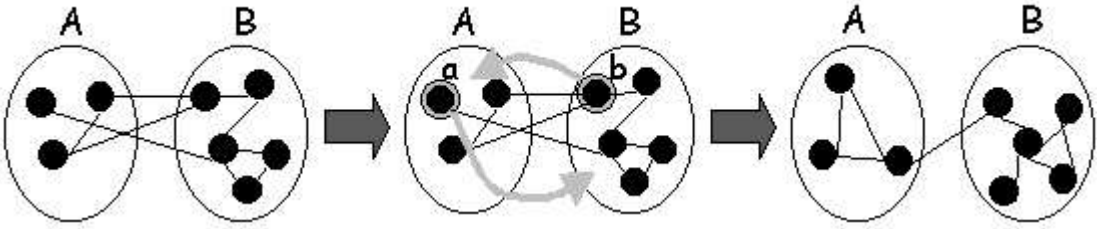

FIGURA 40 – Migração de 2 células

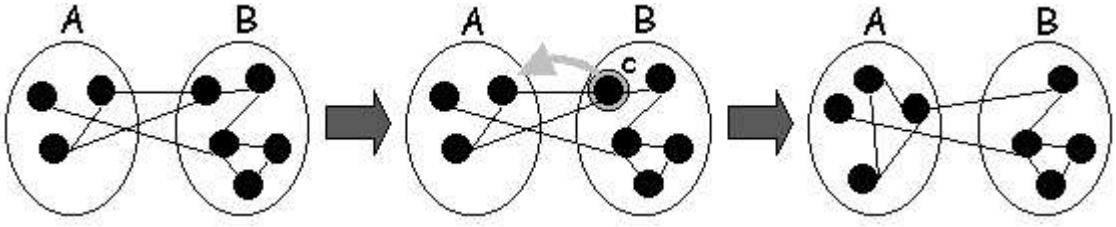

FIGURA 41- Migração de 1 célula

[13] Chama-se o resultado de um algoritmo de particionamento construtivo de particionamento inicial, pois ele será entrada para um refinamento iterativo.

Dois algoritmos iterativos foram escolhidos por sua importância na história das ferramentas de CAD. O de Kernighan-Lin, que foi um dos pioneiros em ferramentas de CAD e o Fiduccia-Matteyses, que é utilizado desde que foi proposto até hoje por ser uma heurística muito rápida com bons resultados de particionamento. Depois de muitos anos de pesquisa, foram propostas diversas alterações nos algoritmos. Algumas são consideradas por este trabalho.

5.4.2.1 Algoritmos Construtivos

Assim como no posicionamento, os algoritmos de particionamento são classificados em construtivos e iterativos. Há, porém, uma diferença significativa. Os algoritmos de particionamento são praticamente independentes da solução inicial (ver sessão 5.4.2.). A qualidade de um particionamento inicial é medida pelo equilíbrio e pelo corte. Por esta razão, os algoritmos construtivos abaixo foram escolhidos pela sua eficiência e capacidade de equilibrar as partições em quantidade de elementos:

- Algoritmo Aleatório
- Algoritmo de Crescimento de Aglomerados

O algoritmo Aleatório é simplesmente uma partição aleatória que mantém as duas partições com o mesmo número de elementos (ou uma diferença de um elemento apenas). É extremamente eficiente (em tempo de CPU) e por esta razão é adotado como particionamento inicial para o Fidduccia-Matteyeses (sessão 5.4.2.3).

O segundo algoritmo é derivado do crescimento de aglomerados para posicionamento. Funciona da seguinte maneira: primeiro escolhe-se uma célula qualquer como semente. Esta célula e todos os seus vizinhos são colocados na partição A. A partição alvo é trocada e repete-se este processo enquanto alguma célula não for assinalada a alguma partição.

O crescimento de aglomerados, assim como nos algoritmos de posicionamento, apresenta variações de implementação. Neste trabalho tentaram-se algumas alternativas interessantes. Primeiro, mudou-se o modelo descrito acima, para um modelo mais radical: não só as células vizinhas da semente devem se mover para determinada partição, mas também todos os vizinhos dos vizinhos, até que se ache um grupo desconexo. Este modelo, porém, ainda não é satisfatório por não haverem muitos grupos desconexos (às vezes não há grupo desconexo). Este método levaria a partições muito desbalanceadas.

Por esta razão, o modelo escolhido e implementado limita a profundidade de vizinhança. Os vizinhos da semente têm profundidade 1. Os vizinhos de células de profundidade 1 têm profundidade 2, e assim por diante. Limitando-se em profundidade 2 ou 3, por exemplo, observou-se que este algoritmo pode ser razoavelmente melhor do que a alternativa proposta inicialmente, como mostram os resultados da tabela 9 e 10.

Por razões didáticas, chama-se de Cluster Growth A (CGa), o algoritmo proposto inicialmente e Cluster Growth B (CGb) o algoritmo modificado para aceitar *n* níveis de profundidade.

A fim de comparar os algoritmos, eles foram implementados e testados em dois circuitos diferentes: C1908 3x3 e C5315 4x4. O segundo circuito é particularmente ruim para o posicionamento com particionamento implementado devido a presença maciça de *pads*. Os *pads* são elementos fixos e de difícil tratamento por algoritmos de migração de grupos como o FM. Os resultados são de uma execução completa do posicionamento, incluindo diversos processos de particionamento. Foram tentadas as duas alternativas de estilo de corte exploradas neste texto nos capítulos 5.4.1.1 e 5.4.1.2.

A tabela 9 apresenta os dados de execução para C1908. 3x3, *wirelength* e balanceamento de células. O balanceamento diz qual a porcentagem média de células que ficou em cada partição. A tabela 10 apresenta os mesmos dados para o circuito C5315 4x4. Em todos os casos, nenhum

algoritmo de particionamento iterativo após o construtivo. Também, em todos os casos, o tempo de execução do algoritmo foi menor que um segundo.

TABELA 9 – Dados de execução de posicionamento baseado em particionamento usando somente algoritmos construtivos de particionamento no circuito C1903_3x3

Quadratura			Bisseção em Bandas		
Algoritmo de Particionamento Inicial	Estimativa de Tamanho das Conexões (WL)	Balanceamento	Algoritmo de Particionamento Inicial	Estimativa de Tamanho das Conexões (WL)	Balanceamento
Random	850552	49	Random	853403	44
Cluster Growth A	549833	36	Cluster Growth A	532257	35
CG B – 2 níveis	494110	29	CG B – 2 níveis	457524	30
CG B – 3 níveis	484433	29	CG B – 3 níveis	438383	28
CG B – 4 níveis	490109	29	CG B – 4 níveis	435677	28
CG B – 10 níveis	532515	30	CG B – 10 níveis	408953	22

TABELA 10 – Dados de execução de posicionamento baseado em particionamento usando somente algoritmos construtivos de particionamento no circuito C5315_4x4

Quadratura			Bisseção em bandas		
Algoritmo de Particionamento Inicial	Estimativa de Tamanho das Conexões (WL)	Balanceamento	Algoritmo de Particionamento Inicial	Estimativa de Tamanho das Conexões (WL)	Balanceamento
Random	3120684	42	Random	3047599	43
Cluster Growth A	2600347	34	Cluster Growth A	2355830	32
CG B – 2 níveis	2747047	34	CG B – 2 níveis	2144234	24
CG B – 3 níveis	2787947	34	CG B – 3 níveis	2174511	24
CG B – 4 níveis	2809472	34	CG B – 4 níveis	2220324	24
CG B – 10 níveis	3215026	36	CG B – 10 níveis	2141458	21

As tabelas revelam dados surpreendentes. Primeiro, analise o balanceamento dos algoritmos construtivos. O algoritmo *Cluster Growth* produz um desbalanceamento maior, como era esperado. Porém, se comporta diferente na quadratura com relação a bisseção no circuito C5315_4x4, que possui mais *pads*. Esta é provavelmente a explicação para este fato. Note que estes dados são médias de dez execuções e, por isto, não são casos particulares de uma ou outra execução do algoritmo. No circuito C1903 há uma pequena diferença de comportamento no caso do Cluster Growth com 10 níveis. Outro dado muito interessante é que, no caso da quadratura, o aumento dos níveis de profundidade do Cluster Growth pioram o resultado da solução em WL, enquanto que no bisseção eles melhoram Estes dados revelam que há bastante investigação a ser feita nestes algoritmos.

5.4.2.2 Kernighan-Lin

Baseia-se em migração de grupos, envolvendo uma célula de cada grupo. A cada iteração do algoritmo, duas células são selecionadas de forma que o corte seja diminuído. Para realizar este cálculo, o algoritmo calcula duas funções para cada célula: *inedge(cell)* e *outedge(cell)*. A primeira dá o número de redes da célula *cell* que não cruzam as partições. A segunda dá o número de redes de *cell* que cruzam as partições. A partir das funções é calculado o ganho G(cell) de cada célula conforme a equação abaixo.

G(cell) = *outedge(cell) – inedge(cell)*

Depois de calculado o ganho de cada célula, é efetuada a troca das duas células de maior ganho de cada partição. O processo termina quando não houver mais trocas que possam diminuir o corte. A figura 42 mostra o algoritmo em uma pseudo-linguagem.

Algoritmo Kernighan-Lin

Criar um particionamento inicial (aleatório, em geral)
Enquanto A Solução Estiver Melhorando
  Enquanto Houverem Vértices Trancado
  Escolher *a* ε A e *b* ε B, sendo que *a* e *b* não estão trancados e G(a) + G(b) é máximo
  Trocar *a* por *b*, ou seja, *a* ε B e *b* ε A.
  Trancar *a* e *b*
  Avaliar Resultado Atingido na Iteração Anterior
  Destrancar todos os vértices

$O(n^3)$

FIGURA 42 - Algoritmo de Kernighan-Lin - Textualmente

O algoritmo apresenta uma série de desvantagens. A sua ordem de complexidade é cúbica. Isto ocorre porque não há um mecanismo inteligente de atualização dos ganhos. Ou seja, a cada iteração, o ganho de todas as células é recalculado. Mas a principal desvantagem é o fato de não conseguir balancear as partições por área. Isto não seria um problema se as células tivessem tamanhos semelhantes, mas nos circuitos modernos isto não ocorre.

5.4.2.3 Fiduccia-Matteyses

Mais tarde, foi proposta uma modificação do algoritmo KL. Fiduccia-Matteyses (FM), como ficou conhecido o novo algoritmo, faz migração simples de grupos (uma célula) e apresenta uma complexidade de tempo inferior ($o(n^2)$).

Uma modificação, talvez a mais importante, com relação a KL é a consideração do corte de redes. Para FM, não importa quantas células de uma rede existem em cada partição, mas sim se a rede atravessa as partições ou não. A figura 43 ilustra o corte de redes contra o corte tradicional. Observe que há uma rede formada por três pinos. Dois deles encontram-se em um conjunto (A, no caso) e o terceiro está no outro conjunto (B). No caso do corte tradicional diz-se que duas conexões cruzam as partições (corte = 2). No caso do corte de redes, uma rede cruza as partições (corte = 1). A vantagem do corte de redes, como será visto adiante, é maior facilidade no seu cálculo, permitindo um algoritmo mais rápido e com uma complexidade de tempo menor. Além disto, o corte de redes é um corte mais realista, porque em um circuito real é necessário somente um fio para conectar dois extremos de um circuito.

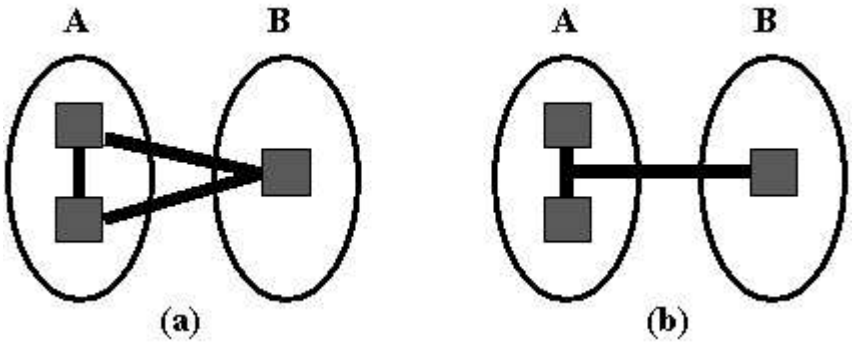

FIGURA 43 – Corte de redes

Antes de entender melhor o cálculo do ganho, devem ser introduzidos alguns conceitos. O **estado de corte** de uma rede diz se a rede corta ou não corta as partições. Uma **rede crítica** é uma rede que pode ter seu estado de corte alterado com o movimento de uma célula pertencente

à rede. Uma célula é dita **crítica** quando é responsável por uma rede ser crítica. No caso da figura 43, a rede é crítica, pois movendo-se a célula da direita para a esquerda o estado de corte se alterará. Assim, a célula na partição da esquerda é crítica.

O principal reflexo deste cálculo de ganho no algoritmo é que não está mais atrelado com as células, mas sim com as redes. Em FM, calcula-se o ganho com base em dois contadores (A e B), existentes em cada rede. Eles se referem a quantidades de células existentes em cada partição. Para o exemplo da figura 43, estes contadores valeriam 2 e 1. Neste caso, existe uma célula isolada na partição 2 (da esquerda, na figura 43). Desta forma, o cálculo do ganho de cada célula fica extremamente simples e eficiente, conforme será visto em seguida. Por este motivo também, a atualização dos custos é feita de maneira ainda mais eficiente.

Outra modificação é a possibilidade de equilíbrio de área das partições. FM exige que seja construída uma função de avaliação que diga se uma célula pode ser movida com base em um critério de balanceamento. Se a movimentação quebrar o critério, ela não será considerada. Sem isto, FM juntaria todas as células em uma mesma partição.

Este algoritmo é bastante complexo e para ser bem compreendido é preciso uma análise detalhada. A figura 44 coloca o algoritmo em uma pseudo-linguagem de programação. Em seguida cada passo é analisado com detalhe.

Algoritmo Fiduccia-Mattheyses
Passo 1: Computar o ganho para todas as células **Passo 2:** i = 1; Selecionar uma célula base. Se não há célula base possível, encerrar o algoritmo. **Passo 3:** Trancar a célula base. Atualizar os ganhos. **Passo 4:** Se não há células trancadas pegar uma nova base. Se há uma célula base, i = i + 1 e voltar para passo 3. **Passo 5:** Selecionar a melhor seqüência de movimentos $m_1, m_2, m_3 .... m_k$ $(1 \leq k \leq i)$, onde o somatório $m_1+m_2+m_3 +... + m_k$ seja máximo. **Passo 6:** Tornar os movimentos $m_1, m_2, m_3 .... m_k$ permanentes Liberar todas as células Voltar para o passo 1
$O(n^2)$

FIGURA 44 – Algoritmo FM em pseudo-linguagem

O algoritmo trabalha com dois *loops*. No *loop* externo, o valor de ganho de todas as células é computado. O *loop* interno (passos 3 e 4) faz uma série de *i* movimentos sem repetir células e sem recalcular o custo, apenas usando a função de atualização de custo (que é mais eficiente). O *loop* interno é repetido até que se troquem todas as células livres que não causam desbalanceamentos. Porém, as trocas não são efetivadas neste passo. Depois, voltando ao *loop* externo, o algoritmo escolhe um número *k* de movimentos que serão efetivados. O *loop* externo é repetido enquanto *k* >=1, ou seja, enquanto o *loop* interno for capaz de encontrar pelo menos uma célula para mover.

Passo 1:

Para calcular o ganho, primeiro os contadores A e B de cada rede devem ser atualizados. O contador A é referente à partição A, ou partição da esquerda. Para isto deve-se fazer uma varredura por cada rede e contar quantas células conectadas a ela estão em cada partição.

Depois disto, o ganho é calculado seguindo-se o seguinte algoritmo, onde:

F refere-se ao conjunto de origem da célula

T refere-se ao conjunto de destino da célula

F(n) diz quantas células da rede *n* estão no conjunto de origem.

T(n) diz quantas células da rede *n* estão no conjunto de destino.[14]

```
Ganho = 0;
Para cada rede n da célula
        Se F(n) == 1 então ganho = ganho + 1;
        Se T(n) == 0 então ganho = ganho – 1;
```

Os dois testes no algoritmo acima tentam descobrir se a célula em questão é crítica. Perceba que, para o algoritmo de FM, somente células críticas têm valores de ganho diferentes de zero.

No caso de considerar a existência de *pads*, uma célula pode deixar de ser crítica. Seja FP(n) uma função que diz se há um *pad* conectado a rede *n* no conjunto de origem e TP(n) para o conjunto destino. O cálculo do ganho ficaria assim:

```
Ganho = 0;
Para cada rede n da célula
        Se F(n) == 1 e não FP(n) então ganho = ganho + 1;
        Se T(n) == 0 e não TP(n) então ganho = ganho – 1;
```

Este algoritmo deu um ganho de 2% em média, para o circuito C1908 3x3.

Passo 2:

A variável "i" é um contador de movimentos feitos pelo *loop* interno do algoritmo. No passo 2 ele deve ser inicializado com 1.

Em seguida o algoritmo deve escolher uma célula para trocar com base no seu ganho. A célula escolhida é a que tem maior ganho e que não cria desbalanço. Esta etapa é descrita pelo algoritmo abaixo:

```
melhor_ganho = Int_Min;
melhor_celula = -1;
Para cada célula c
        Se ganho(c)>melhor_ganho e não_cria_desbalancemento(c) então
                melhor_ganho = ganho(c);
                melhor_celula = c;
```

Note que o valor do maior ganho pode ser negativo, dando ao algoritmo uma possibilidade de fugir de mínimos locais. Porém, se não forem encontrados passos posteriores capazes de compensar esta perda, eles serão descartados.

O critério de balanceamento, como já foi dito, evita que o algoritmo mova todas as células para uma dada partição. Ele pode ser expresso pela seguinte equação:

$$r \times |V| - sm \leq |F| \leq r \times |V| + sm \qquad [5]$$

onde:
V é o conjunto de todas as células
|V| é a área do conjunto V
F é o conjunto ao qual a célula pertence
r é coeficiente de balanceamento, calculado por |F| ÷ |V|. Em geral vale 0,5
sm é o tamanho da maior célula no circuito

Passo 3:

[14] Os valores de F (*n*) e T(*n*) são calculados com base nos contadores A e B da rede *n* e também com um indicador booleano que diz se a célula em questão pertence a A ou B.

Primeiro deve-se fixar a célula base escolhida para migrar de partição com o objetivo que ela não se mova novamente. Isto pode ser implementado com uma classe *map* oferecida na *Standard Template Library* do C++.

Depois há a atualização dos custos. A atualização segue o seguinte algoritmo:

```
Para cada rede n da célula base
        Se T(n) == 0 então
                Incrementar os ganhos de todas as células livres de n
        Se T(n) == 1 então
                Decrementar o ganho da única célula de n em T, se estiver livre.

        //Efetivar a troca
        F(n) = F(n) – 1;
        T(n) = T(n) + 1;

        Se F(n) == 0 então
                Decrementar os ganhos de todas as células livres de n
        Se F(n) == 1 então
                Incrementar o ganho da única célula de n em F, se estiver livre.
```

Passo 4:

Inicialmente, procura-se por células livres. Se há, procura-se por uma célula base que esteja livre e que satisfaça o critério de desbalanço, da mesma forma que no passo 2. Se foi encontrada a célula, incrementa-se o *i* e volta-se para o passo 3. Se não foi encontrada, segue-se para o passo 5.

Passo 5:

Este passo escolhe as *k* iterações (*k*<=*i*) do *loop* interno que serão concretizadas. Por este motivo, o algoritmo precisa armazenar o ganho de cada iteração. Note que *k* será igual a *i* se todos os ganhos forem positivos. E será menor que *i* se houver um ganho negativo que não seja compensado por iterações posteriores.

O algoritmo que realiza o passo 5 é descrito abaixo:

```
melhorganho = INT_MIN;
soma = 0;
for (int r=0; r<i; r++)
        soma += HistoricoGanhos[r].melhorganho;
        if (soma>melhorganho)
                k = r;
                melhorganho = soma;

if (melhorganho<=0)
        fim;
```

No algoritmo acima, o vetor HistoricoGanhos guarda o melhor ganho dos *i* passos realizados no *loop* interno.

Sait, em 1995, sugere em caso de empate que o desempate seja efetuado usando o critério de balanço.

Passo 6:

Todos os movimentos menores ou iguais a *k* devem ser efetivamente realizados, ou seja, a célula deve ser de fato movida. No passo 3 apenas os contadores de redes foram atualizados. No passo 6 elas serão movidas e ao voltar ao passo 1 os contadores serão recalculados. Depois, todas as células são liberadas.

5.4.2.4 Outros

A técnica de FM é a que mais foi utilizada e continua sendo largamente utilizada. Porém, muitos outros trabalhos aprimoraram e criticam esta técnica, como por exemplo Wang em 2000, Karypis em 1999 e Caldwell em 1999b.

Goldberg, em 1983, observou que a qualidade dos algoritmos de particionamento por migração de grupos depende fortemente no número médio de arestas por nodo, ou seja, do *fanout* médio do circuito. Por exemplo, KL funciona bem se a esta média for maior do que 5. Porém, em circuitos observados por Goldberg, esta média fica 1,8 e 2,5. Por isto, Goldberg desenvolveu uma técnica para unir vértices e, com isto, aumentar a sua conectividade. Uma alternativa a técnica de Goldberg é usar o algoritmo de crescimento de aglomerados para criar grupos de células, que podem ser abstraídos para um único elemento.

Outra vantagem desta idéia de aglutinar células é o fato de diminuir o número total de elementos a serem tratados pelo algoritmo de particionamento. Fidduccia-Mateyeses possui tempo quadrático e KL cúbico, o que indica a importância de reduzir o número de células. As tabelas 11 e 12 mostram a comparação de tempos de execução para o circuito Alu4 e Alu2, com uma diferença drástica entre os dois.

Outra técnica bastante conhecida é a de replicação de componentes (Sherwani em 1995). Basicamente, se uma determinada célula é requisitada em ambas as partições, ela pode ser replicada, com o objetivo de diminuir o corte. Isto só vai acontecer se o *fanin* da célula for menor que o *fanout*. O trabalho de Wai-Kei Mak, em 2002, usa uma variação, que ele chama de *Functional Replication*. A replicação funcional, segundo Mak, pode levar a resultados de corte melhores que a replicação tradicional por considerar a dependência lógica de diferente sinais de saída de uma porta em sua entrada.

Outras técnicas bastante usadas são baseadas em meta-heurísticas, como *Simulated Annealing* e Algoritmos Genéticos. Para instancias pequenas de problemas de particionamento, é proposto por Caldwell em 2001 o uso de busca exaustiva para particionamento mostrando alguns resultados de melhora significativos.

Este trabalho sugere uma implementação mista do KL com FM, chamada de BagFM. Basicamente, este algoritmo possui a mesma complexidade de tempo do KL, pois faz a atualização completa dos ganhos a cada turno. Os ganhos são computados como em KL também. A única diferença é que o algoritmo move uma célula por vez, como o FM, permitindo o balanceamento das partições. A comparação com o BagFM é interessante para mostrar a vantagem significativa que o FM tem com a computação rápida dos ganhos.

5.4.2.5 Resultados de execução

A fim de comparar algumas técnicas apresentadas nas sessões anteriores, elas foram implementadas na ferramenta do *Mango Parrot*. Inicialmente deseja-se observar o efeito de um bom particionamento no comprimento das conexões do posicionamento. Para isto, usa-se a quadratura como estratégia de corte e varia-se o algoritmo de particionamento. Como circuito de teste, usa-se o C1908_3x3. Os dados experimentais estão na tabela 11. Na tabela 12 usa-se o circuito alu4 como teste. As tabelas 13 e 14 são análogas, mas usam o método de bi-particionamentos em bandas.

TABELA 11 – Quadratura no Alu2_4x4. Comparação de algoritmos de particionamento.

Algoritmo de Particionamento Inicial	Algoritmo de Particionamento Iterativo	Tempo de Execução do Posicionamento (s)	Estimativa de Tamanho das Conexões (WL)	Balanceamento	Profundidade da Recursividade
Random	Bag FM	4	407400	38	541
Random	FM	2	285930	37	541
CG A	FM	2	286453	37	541
CG B2	FM	2	292532	34	541
CG B3	FM	2	293083	34	541
CG B4	FM	2	303354	33	541
CG B10	FM	2	337199	34	541

TABELA 12 – Quadratura no Alu4_4x4. Comparação de algoritmos de particionamento.

Algoritmo de Particionamento Inicial	Algoritmo de Particionamento Iterativo	Tempo de Execução do Posicionamento (s)	Estimativa de Tamanho das Conexões (WL)	Balanceamento	Profundidade da Recursividade
Random	Bag FM	701	7609433	43	2837
Random	FM	88	3310479	38	2837
CG A	FM	62	3738205	36	2837
CG B2	FM	44	4517400	34	2837
CG B3	FM	44	4656866	33	2837
CG B4	FM	42	4967648	33	2837
CG B10	FM	37	6412203	35	2837

TABELA 13 – Biparticionamento no Alu2_4x4. Comparação de algoritmos de particionamento.

Algoritmo de Particionamento Inicial	Algoritmo de Particionamento Iterativo	Tempo de Execução do Posicionamento (s)	Estimativa de Tamanho das Conexões (WL)	Balanceamento	Profundidade da Recursividade
Random	FM	2	301868	42	254
CG A	FM	2	289122	42	254
CG B2	FM	2	291552	39	254
CG B3	FM	2	296996	37	254
CG B4	FM	2	292240	36	254
CG B10	FM	2	288527	30	254

TABELA 14 – Biparticionamento no Alu4_4x4. Comparação de algoritmos de particionamento.

Algoritmo de Particionamento Inicial	Algoritmo de Particionamento Iterativo	Tempo de Execução do Posicionamento (s)	Estimativa de Tamanho das Conexões (WL)	Balanceamento	Profundidade da Recursividade
Random	FM	29	3551131	40	577
CG A	FM	25	3594258	40	577
CG B2	FM	25	3601557	37	577
CG B3	FM	25	3688960	35	577
CG B4	FM	24	3794774	33	577
CG B10	FM	26	3849015	27	577

O primeiro resultado interessante destas duas tabelas é o tempo de processamento do BagFM (que é um derivado de KL) na tabela 12. É extremamente alto comparado com a tabela

11 e também com o FM. Outro dado interessante é a pouca dependência do FM ao particionamento inicial. Embora seja mais rápido particionar a partir do *Cluster Growth*, o FM não consegue melhorar o desbalanceamento inicial gerado. Isto explica o tempo menor, pois o FM executa menos iterações. Assim, o algoritmo aleatório mostra-se mais adequado devido ao seu melhor balanceamento de células (ver tabelas 9 e 10).

É interessante analisar que não há metodologia ideal. A quadratura ganha em alguns casos, perde em outros. Ao mesmo tempo, o particionamento inicial é melhor aleatório na maioria dos casos, mas o *Cluster Growth* mostra melhoras significativas em outros.

5.4.2.6 Multi-Way Partitioning

A maior parte dos algoritmos de particionamento trabalham com duas partições somente. Porém, em circuitos integrados com mais de 1000 células, bi-particionar apenas não é suficiente para reduzir a complexidade do problema de posicionamento. É desejável que se particione o circuito em diversas partições, de tamanhos variados ou não, dependendo do problema.

São muito comuns adaptações de algoritmos de bipartição, como o FM, para trabalhar com n partições (Karypis, em 1999, por exemplo). Esta sessão apresenta três métodos de uso e adaptação dos algoritmos de bipartições para múltiplas partições.

O primeiro método é uma extensão direta do FM clássico. Cada nodo pode se mover para todas as outras partições. Assim, um valor de ganho é calculado para cada possibilidade, e é escolhida a possibilidade com maior ganho. A desvantagem é a grande quantidade de memória necessária para armazenar todos os ganhos. A figura 44 ilustra este método.

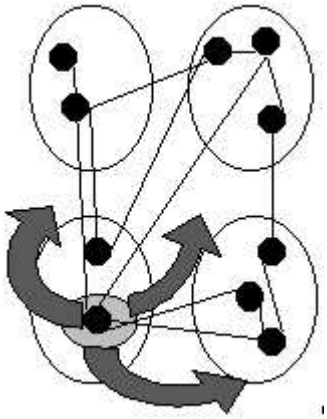

FIGURA 44 – Fiduccia-Mateyeses adaptado a múltiplas partições – primeiro método

O segundo método é iterativo. Começamos com uma solução de particionamento em n partições. Depois são pegos diversos pares de partição para realizar um bi-particionamento somente nas duas partições selecionadas. Neste método, redes que cortam as partições são minimizadas, mas redes que cortam somente uma das partições são desconsideradas. A figura 45 ilustra o segundo método.

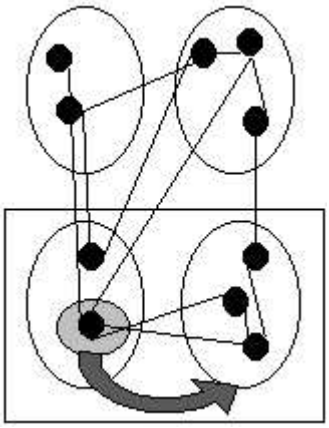

FIGURA 45 - Fiduccia-Mateyeses adaptado a múltiplas partições – método iterativo

O terceiro método é hierárquico e construtivo. Ele parte de uma solução com somente uma partição e faz um biparticionamento. Depois, pega as duas partições em separado e biparticiona elas, de forma a dividir o circuito em n partições hierarquicamente. É ilustrado na figura 46.

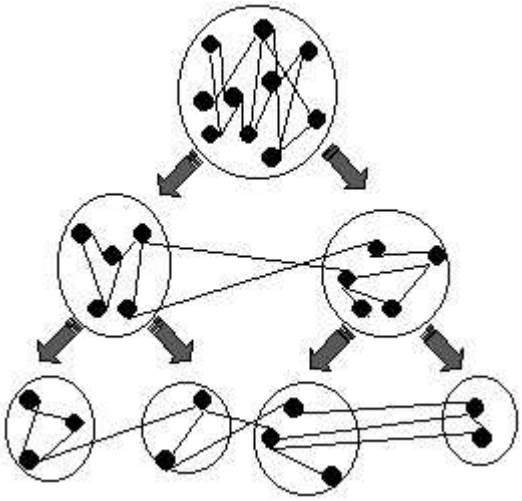

FIGURA 46 – FM em múltiplas partições – método hierárquico

O primeiro método, além de necessitar de muita memória, precisa de mais tempo para computar mais valores de ganho e percorrer todos eles. No trabalho de Wang mostra-se que ele pode ser 500% mais lento que o terceiro método. Desta forma, embora pareça intuitivo, o primeiro método é de fato uma solução ruim.

Wang analisa detalhadamente desta questão. Observa que o método hierárquico é um método guloso e não consegue desfazer um erro que cometeu no começo do processo. Desta forma será comum ficar preso em mínimos locais. Resultados experimentais em permitem concluir que a abordagem hierárquica é mais rápida (144 vezes) e produz melhores resultados (7,1%).

5.4.3 *Terminal Propagation*

O posicionamento por particionamento, como foi apresentado nos capítulos anteriores, é um modelo incompleto para o posicionamento que busca minimizar o comprimento das conexões. Imagine a situação onde duas células conectadas A e B são assinaladas a partições diferentes. A partir deste momento, as células serão particionadas e posicionadas independentemente desta ligação. É como se o algoritmo ignorasse esta conexão. A figura 47, reproduzida de Sherwani, ilustra um caso onde as células A e B são posicionadas de maneira ruim e a solução alternativa usando *Terminal Propagation*.

A solução, chamada de *Terminal Propagation*, quebra as conexões, inserindo, no meio delas, um terminal extra, fazendo o papel de um *pad* de entrada e saída. A partir daí, o próprio algoritmo de minimização de corte vai manter as células o mais próximas possível. Observe que a inserção de terminais é uma etapa extra, mas barata em termos de processamento. Ainda, esta inserção não reduz o potencial de paralelismo do posicionamento baseado em particionamentos.

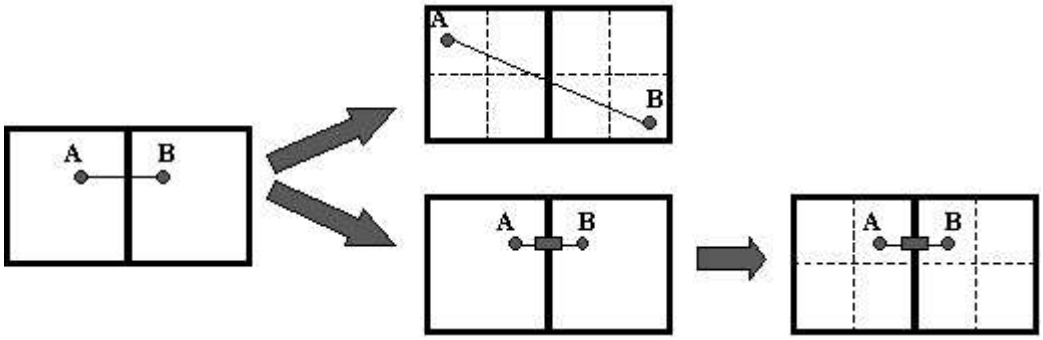

FIGURA 47.- Terminal Propagation

A posição do terminal é um problema relativamente complexo a ser resolvido. A maneira mais simples é simplesmente sortear uma posição para o terminal. Outra solução é de escolher heuristicamente a melhor posição para o terminal com base nas células vizinhas das células A e B. A posição de força zero da célula seria estimada com base nos particionamentos anteriores.

As implementações dos algoritmos de posicionamento baseados em particionamento para a ferramenta do *Mango Parrot* não usam *terminal propagation*. Isto é uma desvantagem significativa que deve ser levada em conta na análise dos dados do capítulo 5.5.

6 Meta-heurísticas Aplicadas a Posicionamento

6.1 Introdução

Meta-heurísticas são métodos de otimização genéricos e não determinísticos. Como já foi demonstrado no capítulo 2, as meta-heurísticas partem de soluções iniciais e iteram sobre elas com perturbações aleatórias. O problema de posicionamento se adapta muito bem a estes métodos por três principais razões:

1 – É simples de encontrar-se um posicionamento inicial. Como já foi dito, encontrar um posicionamento válido é um problema trivial. As meta-heurísticas, como visto na sessão 2.6, necessitam de uma solução inicial a fim de otimizá-la.

2 – É fácil de fazer uma perturbação no algoritmo. Principalmente no caso da Simulação de Têmpera e da Mutação nos algoritmos genéticos, a função de perturbação pode ser trivialmente definida.

3 – Simples avaliação de custo. Foram vistas no capítulo 3 uma série de variáveis do posicionamento que devem ser avaliadas. A estimativa de tamanho dos fios é, em geral, usada em conjunto com outras.

Há décadas existe pesquisa por algoritmos de posicionamento. No princípio, a quantidade de células para posicionar era bem inferior ao que se tem hoje. Além disto, não havia tanta preocupação com desempenho e potência dissipada como se tem hoje, com o advento dos chips embarcados. As tecnologias da época ofereciam menos riscos de problemas elétricos, como *cross-talk* e enfraquecimento de um sinal. No entanto, diversos trabalhos, com o passar dos anos, utilizam meta-heurísticas para resolver os problemas, pois são de simples implementação e os poucos elementos a tratar não comprometiam o desempenho do algoritmo.

Como qualquer algoritmo de otimização genérico, as meta-heurísticas são reconhecidamente algoritmos lentos. Muitas iterações são necessárias para atingir resultados satisfatórios. E, provavelmente, muitas iterações são desperdiçadas por más perturbações. Lixin Su, em 2001, observa que estes algoritmos não têm "inteligência", já que não conseguem se adaptar a um problema em particular e ser eficiente para ele. Com a idéia de embutir conhecimento na estrutura do SA visando um aumento de performance, alguns métodos (como o de Su) foram propostos. Alguns trabalhos são: diminuição do tempo de CPU de cada movimento, geração de movimentos válidos somente, o uso de funções de *schedule* adaptativas e eficientes (Lam 1988, Aart 1985), e particionamento do problema (Caldwell). Porém, hoje o problema é mais sério devido a grande quantidade de células existentes em um único chip. A perturbação em si continua com o mesmo tempo de CPU, mas a avaliação do custo agora é mais complexa e mais demorada. Ainda, mais iterações são necessárias, já que temos mais elementos a movimentar. Um circuito, hoje, pode ter mais de 10 milhões de células. A questão é como tratar tantos elementos de maneira eficiente.

Há, porém, uma série de vantagens a serem exploradas em algoritmos genéricos como são as meta-heurísticas. A mais importante é a generalidade da função de otimização. Existe uma série de variáveis que o posicionamento deve se preocupar em otimizar, como: área, potência, freqüência, etc. Ainda devem ser considerados problemas como *cross-talk*, degradação de sinal, estruturas de teste. Todas estas otimizações são complexas, como já foi visto no capítulo 3. Lixin Su, em 2001, observa que os algoritmos determinísticos procuram incluir estas otimizações em versões simplificadas, onde os objetivos são expostos em termos de um pequeno número de variáveis bem definidas. Ele observa também que estes algoritmos começam a ter dificuldades quando os problemas crescem e as funções de objetivo incluem questões reais de projeto. As

meta-heurísticas podem tratar estes assuntos da maneira simples, pois baseiam-se em movimentos aleatórios usando uma avaliação de custo genérica.

6.2 *Simulated Annealing*

6.2.1 Introdução

Em posicionamento, o algoritmo de *simulated annealing* é utilizado como uma algoritmo iterativo, que melhora uma solução inicial com pequenas modificações no posicionamento. O algoritmo básico já foi visto na sessão 2.6.1. É necessário definir uma função de perturbação, que modifica o estado atual do algoritmo levemente. A função de perturbação deve ser suficientemente aleatória para que seja possível atingir qualquer outra solução, que será avaliada por uma função de custo. É necessário, portanto, definir a função de custo. O tipo de solução de interesse é definido por ela. As diversas otimizações a serem feitas pelo algoritmo de posicionamento, como área, potência, atraso, roteabilidade, etc. são variáveis avaliadas pela função de custo. Por fim, é necessário definir a função de *schedule*. Ela define a temperatura inicial, a final e toda a variação da temperatura durante o processo. A importância da função de *schedule* está na definição do comportamento do algoritmo. Com altas temperaturas, o *Simulated Annealing* se comporta como um algoritmo aleatório, ou seja, faz perturbações aleatórias e aceita praticamente todas elas. Em baixas temperaturas o algoritmo se comporta como guloso, ou seja, aceita somente as perturbações que melhoraram o estado atual do algoritmo. Portanto, a variação da temperatura é fundamental para definir o quão guloso e quão aleatório será o algoritmo ao longo do tempo. Quanto mais guloso maior é a probabilidade de ficar preso em um mínimo local, porém mais rápida será a convergência para uma solução satisfatória.

6.2.2 Função de Perturbação

A função de perturbação deve possuir as seguintes propriedades:

- Simples: deve fazer uma perturbação de granularidade pequena para que, através dela, atinja-se qualquer estado de posicionamento.
- Aleatória: deve ser aleatória para que todas as células sejam posicionadas e que o espaço de soluções seja eficientemente pesquisado.
- Consistente: deve gerar somente soluções consistentes de posicionamento.

Em posicionamento, geralmente usa-se uma das funções de perturbação abaixo:

1 – Mover uma célula escolhida aleatoriamente para uma posição aleatória. (figura 48.a)

2 – Trocar duas células escolhidas aleatoriamente de lugar. (figura 48.b)

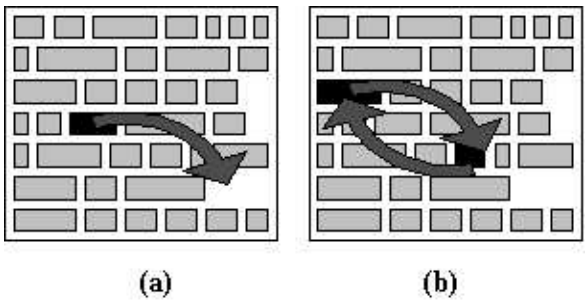

FIGURA 48 – Funções de perturbação para *Simulated Annealing*

Ambos os algoritmos obedecem às duas primeiras propriedades: são sucintos, pois tratam somente de uma célula, e são aleatórios. Porém anda devem ser consistentes, ou seja, devem gerar

somente posicionamentos válidos. Um posicionamento será válido quando respeitar a todas as restrições impostas a ele. Usualmente, restringe-se o número de bandas e a largura máxima da banda, de forma a manter o tamanho do circuito o mais quadrado possível. Da mesma forma, evita-se que bandas sejam pequenas em demasiado, restringindo-se o tamanho mínimo. Foi visto, na sessão 3.3.4, que é ruim manter bandas de tamanhos muito variados.

No caso do algoritmo 1, o sorteio da célula de origem deve respeitar a restrição de tamanho mínimo de banda. O sorteio da banda de destino deve procurar uma banda onde a célula possa ser inserida sem extrapolar a largura máxima da banda. Isto pode ser feito criando-se um vetor de possibilidades e restringindo o sorteio da banda a este vetor. A seleção de uma célula aleatoriamente e o cálculo de sua banda de destino são mostrados na figura abaixo, em pseudo-código. O mesmo vetor de possibilidades pode ser usado para a escolha da célula de origem.

```
Int celula = seleciona_uma_celula_aleatoria_qualquer();

Int cont = 0;
Int vetor_de_possibilidades[num_bandas];

For (int banda2 = 0; banda2<num_bandas; banda2++)
  If (largura(banda2) + largura(celula) <= largura_max_banda)
    Vetor_de_possibilidades[cont] = banda2;
    Cont = cont + 1;

BandaSelecionada = vetor_de_possibilidades[random(cont)];
```

No caso do algoritmo 2, há o sorteio de duas células. A primeira célula deve ser escolhida aleatoriamente, porém a segunda deve pertencer a uma banda onde a primeira célula possa se encaixar. Ao mesmo tempo, a primeira banda deve também suportar a segunda célula. Assim, o algoritmo deve repetir sorteios até que encontre um sorteio válido. Isto faz com que o tempo de execução da seleção dupla dependa do espaçamento horizontal oferecido pelo problema. Porém, a restrição não é forte, pois uma célula será retirada para dar lugar a outra. Este algoritmo é descrito abaixo em pseudo-código.

```
Do
  Banda1 = seleciona_uma_banda_não_vazia();
  Celula1 = seleciona_uma_celula(Banda1);

  Banda2 = seleciona_uma_banda_não_vazia();
  Celula2 = seleciona_uma_celula(Banda2);
While( (largura(banda1)-largura(Celula1)+largura(Celula2)<=largura_max_banda) and
       (largura(banda2)-largura(Celula2)+largura(Celula1)<=largura_max_banda) and
       (largura(banda1)-largura(Celula1)+largura(Celula2)>=largura_min_banda) and
       (largura(banda2)-largura(Celula2)+largura(Celula1)>=largura_min_banda));
```

O algoritmo 1 apresenta uma significativa vantagem de ser independente de solução inicial. Considere o posicionamento inicial mostrado pela figura 49. Há uma disparidade na quantidade de células por banda. A primeira banda apresenta 10 células, enquanto que a terceira banda apresenta somente 1 célula. O algoritmo 1 não seria limitado por este problema. Em algumas iterações, o posicionamento inicial pode ser completamente corrigido. No caso do segundo algoritmo, existe uma dependência muito forte do posicionamento inicial, pois a primeira banda teria sempre 10 células, enquanto que a terceira sempre teria uma célula. No entanto, é comum que o posicionamento seja suficientemente equilibrado para que a troca dupla seja eficiente. Na sessão 3.3.4, há uma proposta de algoritmo dedicado exclusivamente em equilibrar a largura das bandas do posicionamento inicial a fim de preparar para o posicionamento iterativo.

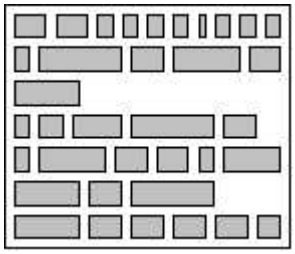

FIGURA 49 – Posicionamento Inicial Desbalanceado

As duas funções de perturbação não exploram nenhuma característica de problemas de posicionamento. Na próxima seção, são apresentadas duas técnicas baseadas na equação de força zero vista na sessão 3.3.1.

6.2.2.1 Perturbações baseadas em Force-Directed Placement

A sessão 3.3.1 deste trabalho apresenta uma maneira de calcular a melhor posição para uma determinada célula assumindo que as demais células estão paradas. Esta hipótese claramente não é válida durante o processo de posicionamento, porém, repetindo este cálculo a cada iteração pode-se convergir para soluções melhores em menos tempo que usando perturbações totalmente aleatórias. Propõem-se então novas funções de perturbação, baseadas nas funções anteriores:

3 – Mover uma célula escolhida aleatoriamente para sua posição de força zero

4 – Trocar duas células de lugar, sendo a segunda célula escolhida com base no cálculo da posição de força zero da primeira.

O algoritmo 3 é baseado no algoritmo 1. Primeiro escolhe-se uma célula aleatoriamente. Depois, é calculada a banda de destino desta célula com base no cálculo de posição de força zero. Se não for possível inserir a célula nesta banda por falta de espaço, são tentadas bandas adjacentes alternadamente em cima e em baixo, até que se encontre uma banda livre. Este procedimento é mostrado abaixo, em pseudocódigo.

```
Banda1 = seleciona_uma_banda_não_vazia();
Celula1 = seleciona_uma_celula(Banda1);

Y = CalculaPosicaoYForceDirected(Celula1);
Banda2 = Y / AlturaDaBanda;

Int Cont = 0;
While (largura(Banda2) + largura(Celula1) > largura_max_banda)
  Cont = Cont + 1;
  Banda2 = abs(band2 + (cont%2?1:-1)*cont)%num_bands; //% eh o operador modulo

X = CalculaPosicaoXForceDirected(Celula1);

Encontrar_a_posicao_entre_duas_celulas_mais_proxima_de_X(Banda2,X);
```

Observa-se que neste algoritmo as células não são movidas exatamente para sua posição de força zero. Primeiro calcula-se a banda mais próxima desta posição. Em seguida encontra-se um espaço entre duas células que mais se aproxime da posição x de força zero.

A posição de força zero de uma célula, como já foi visto na sessão 3.3.1, tende para o meio do circuito. Desta forma, diminui-se significativamente a aleatoridade do algoritmo. Torna-se mais guloso, e portanto, a chance de cair em mínimos locais aumenta. Ainda, existe um novo

problema que é a concentração de células no meio da área. A tendência é que esvasie as bandas da periferia e que aumentem as bandas do centro, até o máximo. Mas este comportamento depende também da função de custo escolhida. Se a função de custo penaliza bandas de tamanhos variados este comportamento pode ser alterado.

O algoritmo 4 é bastante parecido com o 2. A diferença é que a segunda célula é calculada em função da posição de força zero da primeira célula. Assim como o algoritmo 2, é necessário que seja repitido o procedimento até que sejam encontradas duas células que possam trocar de lugar respeitando a largura máxima da banda. Um problema deste algoritmo é que a concorrência por espaços no meio do circuito implica em muitas movimentações de células ali posicionadas, diminuindo a eficiência do algoritmo. Outro problema é que não há como aumentar a quantidade de células nas bandas. No caso do algoritmo 4, uma célula localizada no meio do circuito será movida diversas vezes, o que o torna mais ineficiente.

Outra constatação importante é em relação à eficiência destas funções. O cálculo da força zero custa um tempo insignificante. Porém, quando o meio do circuito está ocupado por muitas células, o cálculo pode demorar mais em função de procurar uma banda liberada.

6.2.2.2 Comparação das técnicas

Foram feitas execuções do algoritmo de Simulated Annealing Guloso (sessão 6.2.7), usando 10000 iterações e estimativa de custo baseada em semiperímetro a fim de comparar as funções de perturbação. A figura 50 mostra um gráfico da variação do custo do algoritmo guloso utilizando as quatro funções de perturbação propostas. Todos partem da mesma solução inicial de custo 527539. A tabela 15 mostra os dados finais de custo atingidos pelas quatro execuções.

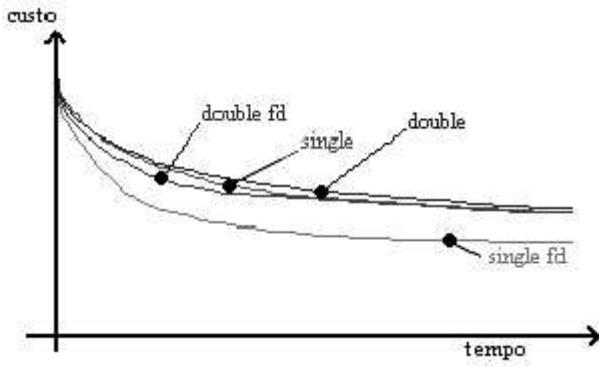

FIGURA 50 – Gráfico de variação do custos de cada função de perturbação

TABELA 15 – Custo final das execuções referentes a figura 50.

Algoritmo de perturbação	Custo Final (após 10000 iterações)
Single	254031
Double	261878
Single FD	195106
Double FD	252939

Observe como o Single FD converge mais rapidamente para bons posicionamentos, além de chegar a uma solução final melhor nos casos mostrados. Porém, eles são exemplos pequenos, pois o número de iterações é muito baixo. A tabela 15, porém, mostra casos onde a convergência.das perturbações FD é maior, mas rapidamente o algoritmo pára em um mínimo local. Isto mostra que para poucas iterações as perturbações FD terão melhor desempenho, mas para muitas iterações terão pior desempenho se comparadas as perturbações aleatórias.

Outra comparação válida a ser feita é com o tempo médio de execução para alguns *benchmarks*. Todos os *benchmarks* utilizados por este texto estão detalhados no anexo1. Para este

exemplo, o *bechmark* utilizado é o C1908, com 783 células. A tabela 16 mostra o tempo de 300 execuções das duas funções de perturbação em um Pentium III 800MHz. Observa-se que a diferença de tempo de execução é uma pequena vantagem da troca dupla.

TABELA 16 – Comparação do tempo de execução das quatro funções de perturbação estudadas

Algoritmo	Tempo de execução (s)
Single	0,04
Double	0,02
Single FD	0,03
Double FD	0,03

Observa-se que os tempos de execução, na média, são muito semelhantes. A função de perturbação simples (*single*) demora um pouco mais devido à manipulação de listas encadeadas, pois um elemento deve ser inserido de uma lista e apagado de outra. Mas a conclusão mais importante a ser tirada da tabela 16 é que o cálculo da posição de força zero não significa um aumento considerável de tempo de CPU.

Agora, analisemos as perturbações direcionadas por outro ângulo. O tempo do algoritmo de *Simulated Annealing* está relacionado com o número de movimentos que são rejeitados pois, conforme será mostrado adiante, são feitas mais chamadas a funções de *undo* (defaz). As perturbações direcionadas a força, porém, tendem a gerar movimentos bons, e por isto são menos rejeitadas. A tabela 17 mostra o tempo de execução do *Simulated Annealing* em baixa temperatura inicial, rodando os dois tipos de perturbações. Porém, fixa-se o número de iterações (150 iterações com 1000 repetições). A comparação será quanto ao tempo do algoritmo simplesmente. Observe que o tempo está diretamente relacionado com o número de movimentos rejeitados. Em todos os casos, o posicionamento inicial é gerado pelo Plic-Plac. A temperatura inicial é calculada pela probabilidade de 0,025 e a sua variação é uma curva cujo ponto de inflexão é (0,25*NumIteracoes,0,25*TemperaturaInicial).

TABELA 17 –Comparação das técnicas de perturbação do *Simulated Annealing* isoladamente

Bench	Perturbação	CPU (s)	*wirelength*	Congestionamento
C499 364 células 405 redes	Single	106	70606	41
	Greedy Single	87	82255	46
	Double	33	77017	44
	Greedy Double	28	97308	57
C1908_3x3 783 células, 816 redes	Single	371	156152	37
	Greedy Single	269	175254	54
	Double	112	178644	49
	Greedy Double	72	171379	49
Alu4 4x4 2523 células, 2537 redes	Single	2869	2312079	277
	Greedy Single	2342	2871695	348
	Double	1606	2395316	279
	Greedy Double	1219	2619805	322
Bw 468 células 473 redes	Single	165	65480	33
	Greedy Single	127	65120	30
	Double	41	71655	34
	Greedy Double	29	71338	34
C53 3249 células 3427 redes	Single	4302	1184910	239
	Greedy Single	2444	1196380	269
	Double	1936	1316547	265
	Greedy Double	1226	1210824	264
Mult 1408 células 1468 redes	Single	944	641440	97
	Greedy Single	765	806386	135
	Double	494	681375	127
	Greedy Double	365	698816	102

Inicialmente, observe (na tabela 17) que as perturbações duplas são bastante rápidas em relação às perturbações simples, em todos os casos. Mas o dado que mais chama a atenção é que as perturbações gulosas, que são mais espertas que perturbações aleatórias, perdem na maior parte dos casos. Isto acontece porque a perturbação gulosa fica presa em mínimos locais facilmente. Conforme mostra o gráfico da figura 51, a convergência deste tipo de função é muito maior no começo do processo, mas em seguida o algoritmo não consegue mais melhorar o posicionamento.

Ainda observando a tabela 17, percebe-se que as perturbações duplas atingem piores resultados em *wirelength* em todos os casos. Isto se deve a dependência forte das perturbações duplas ao posicionamento inicial, conforme discutido anteriormente.

Para observar a convergência das diferentes perturbações, estuda-se o caso do circuito Mult isoladamente. A figura 51 mostra a variação do custo das perturbações aleatórias contra as perturbações direcionadas à força. O gráfico da esquerda é de movimentos simples, e à direita os movimentos duplos. Observe como a convergência inicial dos movimentos direcionados a força é significativamente maior. Porém, a convergência dos movimentos aleatórios é mais constante, sendo que no final do processo atinge-se um melhor resultado do que usando os movimentos direcionados a força.

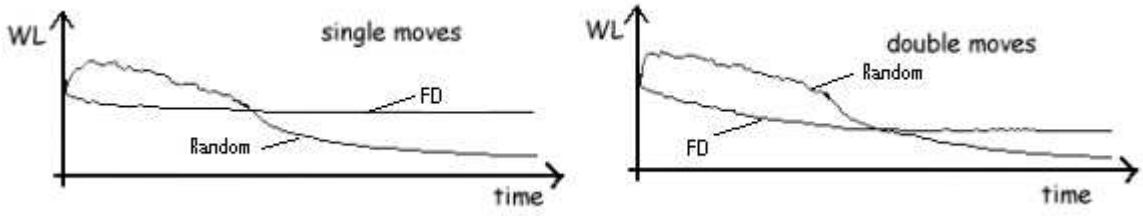

FIGURA 51 – Variação do custo dos métodos direcionados à força contra os métodos tradicionais.

6.2.2.3 Mistura das técnicas

Foi discutido na sessão anterior que tipo de vantagens e desvantagens cada método de perturbação apresenta. Basicamente, as perturbações simples são mais lentas e mais presas ao limite máximo da banda, pois é preciso que uma célula inteira possa ser inserida na banda sem que outra seja retirada (como na perturbação dupla). As perturbações duplas, por outro lado, dependem demasiadamente da solução inicial, pois nunca mudarão a quantidade de células de cada banda. As perturbações gulosas são interessantes por rejeitar menos movimentos e convergirem mais rapidamente, porém podem ficar presas a mínimos locais. Então, uma maneira de ganhar a vantagem de todas as técnicas é simplesmente usar as quatro perturbações ao mesmo tempo. É dada uma probabilidade para cada uma delas, e o algoritmo de posicionamento sorteia uma das quatro a cada iteração. Desta forma, resolve-se o problema das trocas simples, pois trocas duplas podem ajudar a liberar bandas. Resolve-se o problema das trocas duplas, pois as simples garantirão a migração de células para bandas mais vazias. E evitam-se os mínimos locais das técnicas *Force Directed*, pois técnicas menos gulosas são usadas em conjunto. Ao mesmo tempo, mantém-se a boa convergência, pois as técnicas de Force Directed funcionam bem usando poucas iterações (nas primeiras iterações especialmente), como mostra as figura 51 da sessão anterior.

A fim de verificar o resultado prático desta união, este mecanismo foi implementado na ferramenta do *Mango Parrot*. A interface permite que se escolha a probabilidade de cada uma das funções. Usam-se 150 iterações de 1000 movimentos, exatamente com os mesmos parâmetros que a tabela 17 (anterior).A temperatura inicial é calculada pela probabilidade de 0,025 e a sua variação é uma curva cujo ponto de inflexão é (0,25*NumIteracoes,0,25*TemperaturaInicial).

Primeiro experimentou-se usar uma probabilidade de 25% para cada função de perturbação. Depois, 35% para as perturbações direcionadas a força, e 15% para as demais. O próximo experimento inverte este dado, dando 15% para as direcionadas a força. Por fim, usa-se somente perturbações direcionadas a força, com 50% cada. A tabela 18 mostra os dados de execução.

TABELA 18 - Comparação das técnicas de perturbação mistas do Simulated Annealing

Bench	Perturbação	CPU (s)	*wirelength*	Congestionamento
C499 364 células 405 redes	25% each	63	59475	32
	35% greedy	63	64885	38
	15% greedy	65	61686	35
	Only greedy	64	77770	41
C1908_3x3 783 células 816 redes	25% each	191	131616	35
	35% greedy	186	127260	37
	15% greedy	218	121172	33
	Only greedy	193	155697	40
Alu4 4x4 2523 células 2537 redes	25% each	1884	2466637	332
	35% greedy	1669	2400587	267
	15% greedy	1963	2318945	274
	Only greedy	1560	2368556	264
Bw 468 células 473 redes	25% each	84	62873	29
	35% greedy	84	60937	27
	15% greedy	95	61872	30
	Only greedy	89	61309	30
C53 3249 células 3427 redes	25% each	2408	772454	167
	35% greedy	2080	826390	180
	15% greedy	2486	655343	148
	Only greedy	2302	955994	222
Mult 1408 células 1468 redes	25% each	584	530129	82
	35% greedy	508	525264	78
	15% greedy	641	481736	70
	Only greedy	601	591951	88

Quanto maior a probabilidade de movimentos direcionados a força, maior é o caráter guloso do algoritmo. Porém, a existência de movimentos aleatórios permite que a fuga de mínimos locais.

No circuito menor (C1908) observou-se uma pequena vantagem das perturbações direcionadas a força, enquanto que no maior a vantagem está nas perturbações aleatórias. As mesmas conclusões tinham sido observadas na tabela 17 (da sessão anterior). Mas o interessante é que a diferença entre as diferentes probabilidades é pequena, valorizando a idéia de puramente combinar os dois tipos de funções de iteração.

O resultado mais interessante, porém, é mostrado na comparação da tabela 18 com a tabela 17 (sessão anterior). O melhor caso (em *wirelength*) para cada circuito é, respectivamente: 70606 contra 59475 (C499), 156152 contra 121172 (C1908), 2395312 contra 2318945 (Alu4), 65120 contra 60937 (Bw), 1184910 contra 655343 (C53) e finalmente 641440 contra 481736 (Mult). As perturbações mistas sempre vencem. Em média, o ganho é de 21,6%. Na comparação do congestionamento, temos: 41 contra 32 (C499), 37 contra 33 (C1908), 279 contra 264 (Alu4), 30 contra 27 (Bw), 239 contra 148 (C53) e 97 contra 70 (Mult). Novamente, as perturbações mistas ganham em todos os casos, como um ganho médio de 19,3%. Destes resultados, nós podemos concluir que as perturbações mistas aumentam a capacidade de busca do *Simulated Annealing*.

Os resultados do parágrafo anterior são reproduzidos, graficamente, nas figuras 52 e 53.

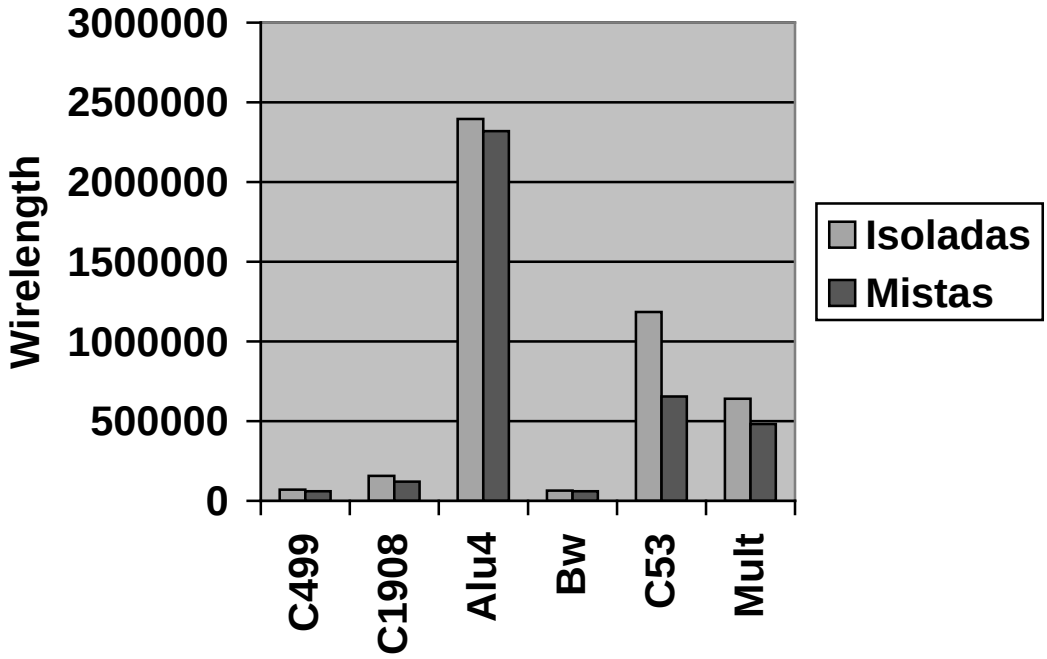

FIGURA 52 – Vantagem, em *wirelength*, das perturbações mistas em relação às isoladas

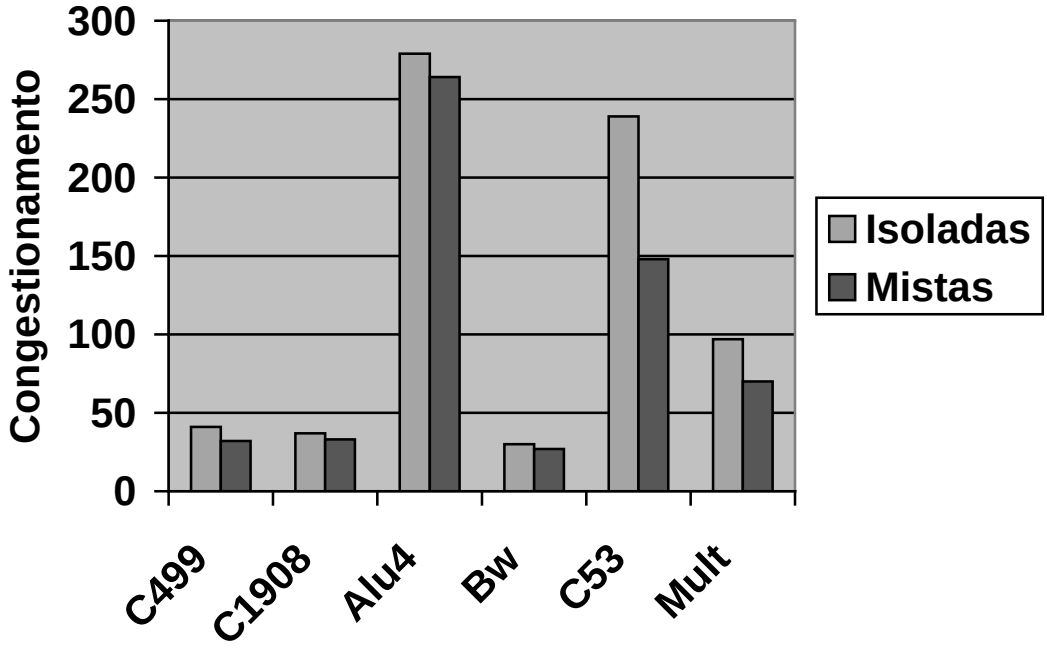

FIGURA 53 – Vantagem, em *wirelength*, das perturbações mistas em relação às isoladas

Considerando o tempo de processamento da técnica mista, observa-se pelas tabelas 17 e 18 que é um valor intermediário das técnicas simples e duplas, o que é um resultado esperado. Porém, é importante observar que há ganho em *wirelength* enquanto que não há perda de desempenho.

É importante avaliar também a convergência do algoritmo ao longo do tempo. Para isto, usa-se o mesmo exemplo da figura 51, na sessão anterior. Agora, porém, acrescenta-se a curva de variação das perturbações mistas, na figura 54.

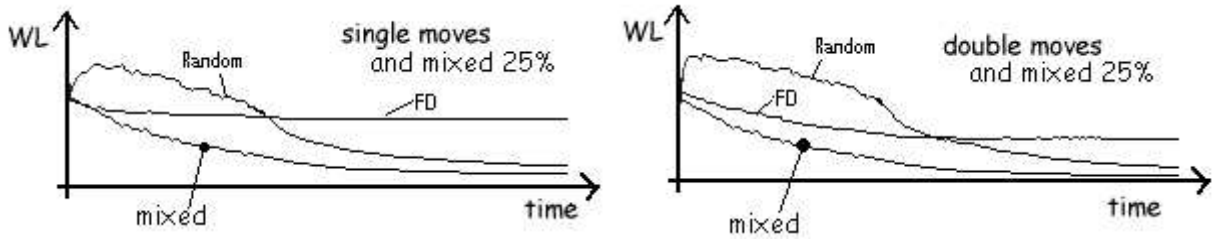

FIGURA 54 - Variação do custo dos métodos isolados contra perturbação mista.

Com uma última análise, considere o caso onde só há uso de perturbações gulosas (tanto dupla quanto simples). Observe, na tabela 18, como os resultados são melhores do que o uso isolado de qualquer uma das técnicas gulosas (tabela 17). Este dado mostra que, simplesmente combinando perturbações simples com duplas, pode-se atingir resultados melhores que qualquer técnica isolada.

Os bons resultados deste uso misto de técnica apontam para um novo trabalho de pesquisa. As probabilidades usadas para cada perturbação (25%, 35%, 15% ...) foram valores escolhidos ao acaso. Porém, ao observar a tabela 18, percebe-se que em alguns casos um determinado conjunto de probabilidades é superior ao outro. Porém, este caso não se repete em todos os circuitos. Cabe, portanto, um trabalho de pesquisa que busque investigar este fenômeno e a relação que ele tem com o tipo de circuito em questão. Técnicas heurísticas poderiam determinar probabilidades até mesmo dinâmicas para um melhor desempenho do *Simulated Annealing*.

6.2.2.4 Funções de *Undo*

Toda função de perturbação deve ser revertida no caso de não haver uma aceitação pelo algoritmo de *Simulated Annealing*. Para que esta reversão seja feita eficientemente, devem ser desenvolvidas funções de *Undo* para cada uma das funções de perturbação acima. Os algoritmos 1 e 3 compartilham a mesma função de *Undo*, assim como o 2 e o 4.

No caso dos algoritmos de troca simples, há uma inserção e remoção em lista. Para que esta operação possa ser revertida, deve-se guardar o número das bandas de cada um dos elementos, além dos dois índices. Se os dois elementos pertencem à mesma banda, a troca simples não precisa inserir e remover elementos, apenas fazer um *swap* de duas posições de memória. Se os elementos não pertencem à mesma banda, a função de *undo* deve percorrer as listas (no caso de lista encadeada de células) até encontrar a posição do elemento a ser movido de uma lista para outra.

No caso dos algoritmos de troca dupla, a função de *undo* é extremamente simples. Deve apenas fazer um *swap* entre as duas posições trocadas pela função de perturbação. As informações armazenadas são simplesmente dois ponteiros para as posições trocadas, fazendo com que a função de *undo* seja extremamente eficiente. Execuções repetidas de 300 vezes mostram um tempo desprezível para um *undo* de trocas duplas (menor que 0,01 segundo para a soma de todas as execuções), enquanto que um *undo* de movimento simples levou 0,29 s (para a soma das execuções).

6.2.3 Função de Custo

O maior desafio da função de custo é de unir múltiplos objetivos de posicionamento, como foi visto no capítulo 3 deste trabalho, em um único número inteiro.

Primordialmente, deve haver preocupação em tamanho dos fios, afinal, é o principal objetivo. Porém, foi visto que o congestionamento é muito importante também. *Timing* é

fundamental. E se pudermos dissipar menos potência seria muito interessante também. E ainda aspectos de área, como a variabilidade da largura das bandas e a relação de aspecto. O problema é: como combinar duas funções que retornam números com unidades diferentes? Por exemplo, como somar litros, quilogramas e bananas? A primeira alternativa mais interessante é transformar todos os valores para valores entre 0 e 1, tirando as unidades. Porém, seria necessário descobrir o máximo *Timing* e o máximo WL a fim de fazer uma divisão. Como descobrí-los?

O trabalho de Sait, em 2001, mostra uma função ***fuzzy*** que combina um valor para WL, outro para *Timing* e um terceiro para **potência**. Não fica claro em seu texto, porém, como é feito o tratamento para unidades de medidas diferentes.

6.2.4 Função de *Schedule*

A função de *schedule* computa o valor da temperatura inicial e sua variação ao longo do algoritmo. Ela determina o comportamento de aceitação da simulação de têmpera. Como já foi visto, o SA tolera soluções ruins com probabilidade proporcional a sua temperatura.

Um *scheduling* adequado é divido em três etapas:

1- Como encontrar a temperatura inicial?
2- Como variar a temperatura?.
3- Quando parar a Simulação de Têmpera.

Neste trabalho, o critério de parada é fixo. Outros *schedules*, como de Aart, usam um critério variável. Ou seja, o algoritmo pára quando define que a sua solução está adequada. A *schedule* adotado na ferramenta *Mango Parrot Didactic Placement* usa um número fixo de iterações, pré-determinado. Este número é determinado, tipicamente, pelo número de células no circuito. Desta forma, usa-se uma temperatura final fixa em 0, e a variação da temperatura é simplesmente uma interpolação entre a temperatura inicial e a final. Ficam em aberto, portanto, o método para encontrar a temperatura inicial (6.2.4.1) e a sua variação ao longo do tempo (6.2.4.2).

6.2.4.1 Encontrando a temperatura inicial

Primeiro, vamos observar a fórmula matemática que calcula a probabilidade de aceitação em função da temperatura:

$$prob = e^{\frac{-delta}{temperatura}} \qquad [6]$$

Observe que ela é dependente de duas variáveis: delta e temperatura. Delta é a variação causada no custo do posicionamento devido a uma perturbação. Assim, no caso da ferramenta do *Mango Parrot*, delta é medido em WL. Para manter uma mesma probabilidade inicial, a temperatura inicial deve variar conforme varia o *delta*. Porém, quanto maior é o circuito, maiores são os valores de delta. Isto faz com que a temperatura inicial seja diferente para cada circuito.

A ferramenta do *Mango Parrot* implementa um mecanismo de encontrar automaticamente a temperatura inicial com base em uma probabilidade inicial desejada. A probabilidade inicial determina dois tipos de algoritmos de simulação de têmpera: dependentes ou independentes da solução inicial. Quanto menor a probabilidade inicial, maior é a dependência. Porém, a dependência não significa uma desvantagem, e sim uma vantagem em muitos casos. O *simulated annealing*, em sua concepção tradicional (conhecido como *high annealing*), é independente da solução inicial, pois começa em temperaturas altas o suficiente para "destruir" a solução inicial. Teoricamente, ele estaria mais apto a fugir de mínimos locais desta forma. Porém, o *low annealing*, como é conhecido, pode partir de uma solução inicial heuristicamente gerada, como visto no capítulo 4, usando menos iterações.

O primeiro passo é de encontrar um *delta médio* para um dado circuito. Para isto, faz-se uma série de execuções do Simulated Annealing, sem nenhuma aceitação, somente perturbações e avaliação do custo. Depois, basta aplicar a seguinte fórmula:

$$temperatura\_inicial = \frac{-delta\_médio}{\ln(prob)} \quad [7]$$

Este procedimento, porém, apresentou falhas de arredondamento. O algoritmo implementado mistura os dois passos, calculando um valor de delta, seguido de sua temperatura inicial correspondente. É feita a média das temperaturas encontradas.

6.2.4.1 Variação da temperatura

Considere, inicialmente, a abordagem clássica de Simulated Annealing, começando com altas temperaturas. Neste caso, a variação de temperatura é de fundamental importância, pois ela determina quanto tempo o algoritmo será aleatório e quanto será guloso. E determinará também o quão suave será esta mudança. Já em uma abordagem de baixa temperatura inicial, a variação tem papel secundário.

A ferramenta do *Mango Parrot* implementa duas rotinas para variação de temperatura: linear e curva de terceiro grau. A rotina linear funciona da seguinte maneira:

1 – Divide-se a temperatura inicial pelo número de iterações
2 – A cada iteração, é subtraído da temperatura o valor calculado no passo 1

O *schedule* linear se aplica a *low annealing*, pois oferece uma redução lenta da temperatura. Também se aplica para temperaturas altas, desde que use-se um número bastante elevado de iterações.

A curva de terceiro grau é bastante complexa, por outro lado. Basicamente, é uma curva com um ponto de inflexão. Até ele, a segunda derivada é negativa e a partir dele é positiva. É feito através da união de duas parábolas. Considere os seguintes valores:

X1 = 0;
Y1 = Temperatura Inicial
X2 = X do ponto de Inflexão (deve estar entre 0 e o Número de Iterações)
Y2 = Y do ponto de Inflexão (deve estar entre 0 e a Temperatura Inicial)
X3 = Número de Iterações
Y3 = 0

A fórmula de cálculo da temperatura, com base em X é dado pelo código em C++ a seguir.

```
double CurveSchedule(x, x1, y1, x2, y2, x3, y3)
{
  double y;

  if(x <= x2)
    y = (y2 - (y2-y1)*sqrt(double(x2-x)/double(x2-x1)));
  else
    y = (y2 - (y2-y3)*sqrt(double(x-x2)/double(x3-x2)));

  return y;
}
```

A figura 55 mostra alguns exemplos de curvas para diferentes X2,Y2. Na figura 55.a, X2 = 0,25*X3 e Y2 = 0,25*Y1. Na figura 55.b, X2 = 0,25*X3 e Y2 = 0,5 * Y1. Na figura 55.c, X2 = 0,8*X3 e Y2 = 0,5 * Y1.

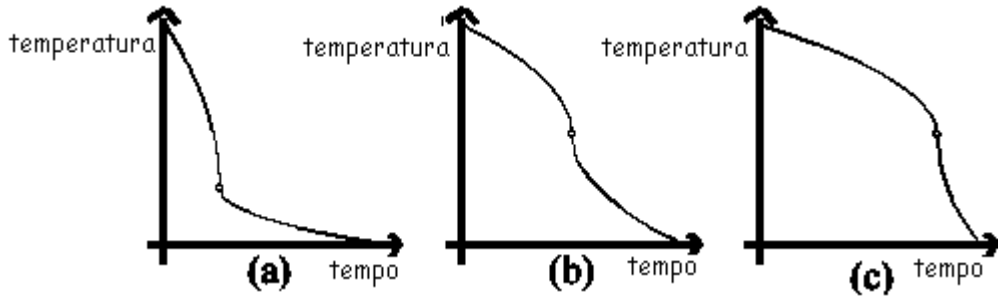

FIGURA 55 – Gráficos de schedule curvo

É interessante observar o comportamento do algoritmo apenas a partir da análise destes gráficos. A figura 55.a mostra um caso onde o algoritmo irá convergir rapidamente para uma solução compacta, e estará apto para fugir de alguns mínimos locais. A solução b poderá fugir de mais mínimos locais, porém a convergência será demorada, e por isto, pode sobrar pouco tempo em baixas temperaturas, quando o posicionamento irá se concretizar de maneira otimizada. É uma boa opção para os mesmos casos do *schedule* linear. A opção c será somente uma boa escolha em temperaturas iniciais muito baixas. Neste caso, é desejável que o algoritmo demore para baixar a temperatura. Na maior parte do tempo, ele estará apto a fugir de alguns mínimos locais, enquanto que no final do processo será totalmente guloso.

Hentschke, em 2002, foi apresenta uma ferramenta didática de *Simulated Annealing*, onde o usuário tem controle sobre a variação da temperatura e pode observar o resultado em uma interface gráfica. Este artigo mostra uma figura muito interessante, que é reproduzida aqui (figura 56). Trata-se de três execuções de *Simulated Annealing*: A primeira usa uma variação de temperatura semelhante à figura 55.a; A segunda é semelhante a 55.c; a terceira apresenta um *schedule* diferenciado, conforme será visto na sessão 6.2.7 deste texto (a temperatura é fixa em zero, fazendo com que o algoritmo seja 100% guloso). A convergência é alta, porém é fácil de se prender em mínimos locais. Os resultados das simulações acompanham a figura.

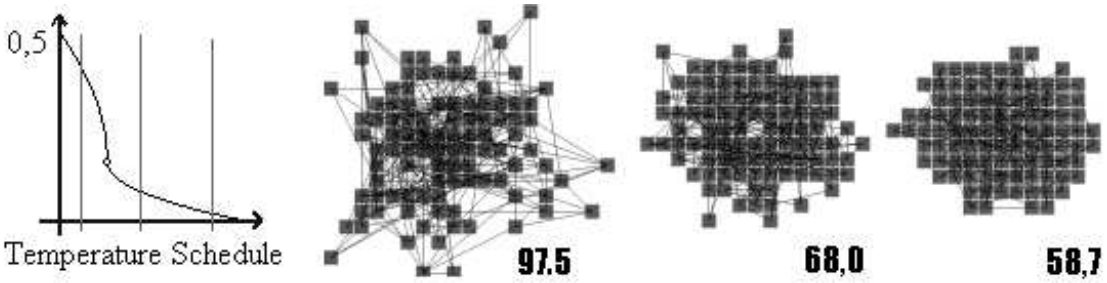

FIGURA 56.a – Simulação de uma *schedule* em um ambiente didático de posicionamento

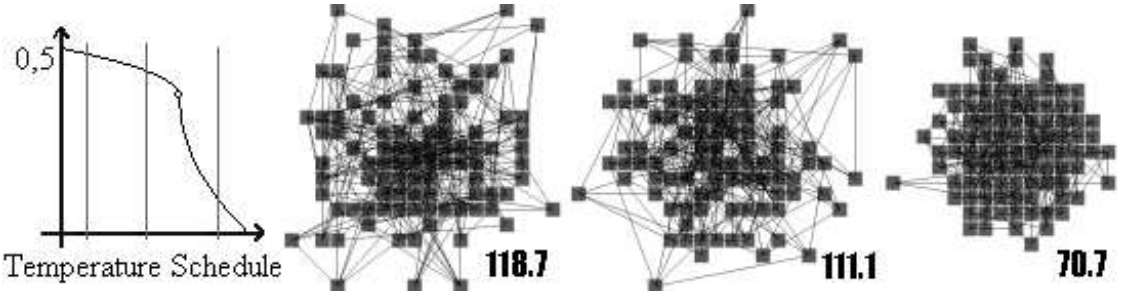

FIGURA 56.b – Simulação de uma *schedule* em um ambiente didático de posicionamento

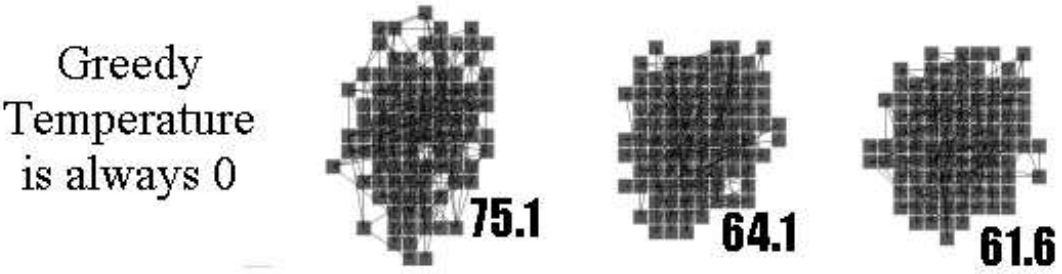

FIGURA 56.c – Simulação de uma *schedule* gulosa em um ambiente didático de posicionamento

Observe que o algoritmo guloso terminou sua execução com custo de 61,6, enquanto que o SA terminou com custo final de 58,7 para o mesmo número de iterações. A figura mostra a variação do custo graficamente. O número 1 indica a figura 56.a. O número 2 indica a figura 56.b e o número 3 a figura 56.c. Um outro caso foi acrescentado na figura 57, indicado pelo número 4. Ele usa uma temperatura inicial mais alta (3 vezes maior). Observe o salto de qualidade da solução no início. Note que a solução inicial é aleatória. Então o SA atingiu uma solução muito pior do que a aleatória, ou seja, a temperatura foi alta demais.

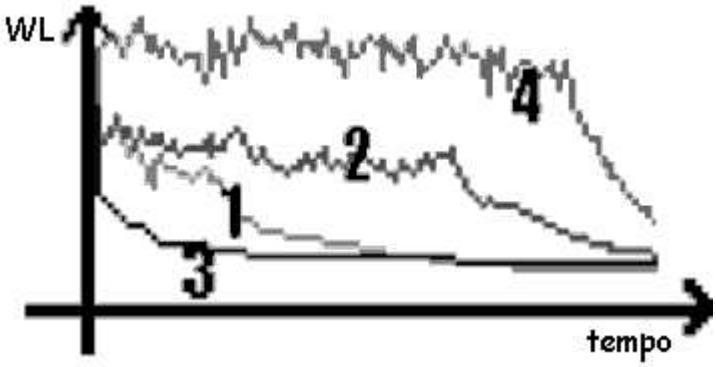

FIGURA 57 – Variação do custo das *schedules* da figura 56 adicionada de uma quarta *schedule*

A figura 57 mostra o impacto da função de *schedule* sobre a execução do algoritmo. É interessante como o gráfico de variação do custo apresenta semelhanças com o gráfico da *schedule* mostrado na figura 56. Nota-se, porém, que no final as execuções não se encontram, mostrando que algumas *schedules* podem ser melhores que outras.

6.2.5 Análise de tempos – encontrando o gargalo do sistema

A simulação de têmpera é reconhecidamente uma heurística lenta. Esta sessão busca identificar os gargalos do algoritmo: quais funções ocupam a maior parte do tempo de CPU? As sessões seguintes vão propor técnicas para otimizar o algoritmo como um todo.

A figura 58 mostra uma versão simplificada do algoritmo visto na sessão 2.6.1, deixando apenas as funções que interessam para o cálculo do tempo de CPU. As funções que são executadas fora do *loop* não interessam, pois não haverá repetição.

Simulated Annealing (*loop*)

```
Enquanto Temperatura>temperatura_final
  Para cada iteração na mesma temperatura
    Perturbação da Solução;
    ΔCusto = Custo(NovaSolução) - Custo(Solução);
    Se Não Aceita()
       Solução.Undo();
   Temperatura = Schedule(Temperatura);
```

FIGURA 58 – *Simulated Annealing*: principais funções para análise do gargalo

Destacam-se 5 funções: Perturbação, Custo, Aceita, Undo e Schedule. Vamos analisar o tempo de cada uma: Para isto, rodou-se a ferramenta do *Mango Parrot* com foco para cada um dos algoritmos em um Pentium III 800Mhz rodando Windows 2000. Foram realizadas 300 execuções de cada função no benchmark C1908_3x3. O resultado apresentado é a soma das 300 execuções.

a) Função de Perturbação: como visto na sessão 6.2.2, existem dois grupos: baseadas em trocas simples e baseadas em trocas duplas. A função de perturbação simples consumiu 0,03 segundos, enquanto que a dupla também consumiu 0,03 segundos.

b) Função de Custo: A sessão 3.3.1.1 deste trabalho mostrou algumas funções de estimativas de WL. Elas são usadas, na ferramenta do *Mango Parrot*, como função de custo para o *Simulated Annealing*. O tempo de execução do cálculo do semiperímetro é de 0,47 segundos.

c) Aceita: Esta função é de tempo constante e não causa impacto no tempo de CPU. As simulações mostram que o tempo consumido é inferior à 0,01 segundos (em 300 execuções).

d) Undo: Assim como a perturbação, a função de Undo também tem duas versões. A função de undo simples é mais custosa que a função de Undo dupla por envolver manipulação de listas encadeadas. Os tempos medidos são: 0,23 segundos para Undo simples e 0,03 para undo duplo.

e) *Schedule*: O tempo de função de *schedule* é constante e totalmente desprezível. Em 300 execuções, ficou bem abaixo de 0,001 segundos.

Montam-se dois fluxos de funções: para perturbações simples e para perturbações duplas. A partir destes dados, dois gráficos são gerados (figura 59 (a) e (b), respectivamente).

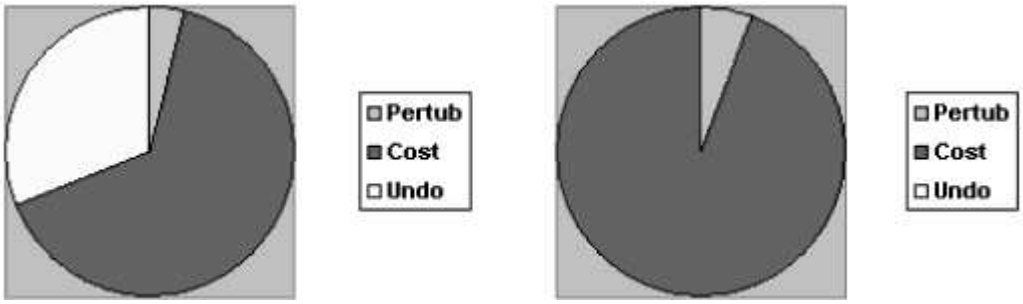

FIGURA 59 – Papel de cada função no tempo de execução do *Simulated Annealing* para C1908_3x3

Observa-se que a função de Custo é sempre predominante. No caso de perturbações simples, ela ocupa mais da metade do tempo. No caso de perturbações duplas, é uma ordem de grandeza superior às demais. Note que a função de *Undo* apresenta uma contribuição significativa também. Porém, é importante lembrar que ela não é chamada em todas iterações e sim nas iterações rejeitadas. Já a função de custo é sempre chamada.

Em um circuito maior, o C7552_4x4, que tem 1800 células aproximadamente, o impacto do custo é ainda maior. Isto é devido à complexidade de tempo da função de custo maior em

relação às outras. Os dados para o circuito C7552_4x4 estão na figura 60. A tendência é que, quanto maior o tamanho do circuito, maior seja a dominação da função de custo sobre as demais.

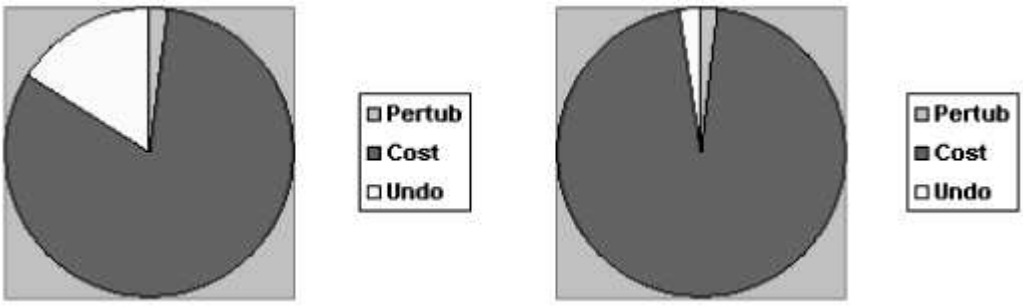

FIGURA 60 – Papel de cada função no tempo de execução *Simulated Annealing*, para C7552_4x4

Depois de analisar os tempos de execução de cada parte do SA, observa-se que o gargalo de tempo de uma iteração é realmente a função de custo. Assim, otimizações sobre o SA devem focar-se em melhorar a função de custo ou reduzir o número total de iterações. Na próxima sessão são expostas algumas técnicas experimentadas na ferramenta do *Mango Parrot* e também por outros autores.

6.2.6 Técnicas para acelerar o *Simulated Annealing*

6.2.6.1 Diminuindo o tempo da função de custo

Na ferramenta do *Mango Parrot* tentou-se, com sucesso, reduzir o tempo de execução da função de custo. Isto partiu da observação de que, a cada iteração, o *Simulated Annealing* calcula o custo de todas as redes, inclusive das que não foram tocadas pela perturbação. Assim decidiu-se implementar uma função de perturbação que retornasse a variação do custo que ela causou. A função de perturbação conhece exatamente o que foi alterado, e por isto pode calcular o delta da maneira mais rápida possível. Basicamente, esta idéia será explorada nesta sessão.

Então, o algoritmo do *Simulated Annealing* fica alterado, como mostra a figura 61.

Simulated Annealing

```
Temperatura = temperatura_inicial;
Solução = Solução Aleatória
Enquanto Temperatura>temperatura_final
  Para cada iteração na mesma temperatura
    ΔCusto = Perturbação da Solução;
    Se Não Aceita(ΔCusto)
      Desfaz a Perturbação;
   Temperatura = Schedule(Temperatura);
```

FIGURA 61 – Otimização proposta na Simulação de Têmpera

A nova função de perturbação tem condições de saber quais células foram modificadas. No caso de uma perturbação simples, todas as células que ficam a direita da célula de origem são modificadas. E todas as células que ficam a direita do local de destino também são modificadas. No caso da perturbação dupla, tem-se o mesmo cálculo. Observe a figura 62.

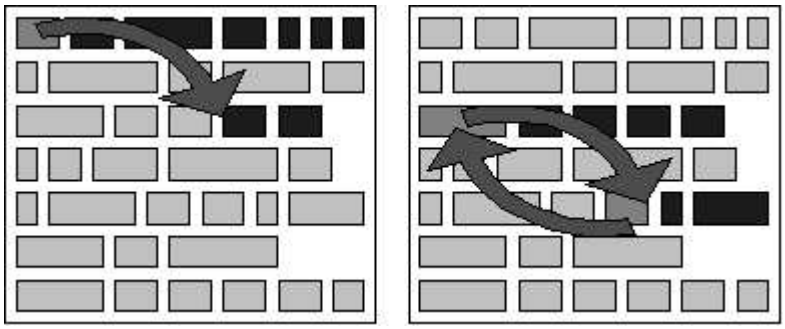

FIGURA 62 – Células com posição modificada pela execução de uma perturbação

A ferramenta do *Mango Parrot* considera, de maneira simplificada, que a modificação se reflete nas duas bandas afetadas, integralmente. A função de perturbação deve calcular o custo das células antes e depois de efetuada a perturbação. Assim, o cálculo do delta é feito da seguinte maneira:

```
int delta = 0;
Para todas as células da banda de origem
  delta = delta – CustoDaCélula();
Para todas as células da banda de destino
  delta = delta – CustoDaCélula();

Efetuar Troca!

Para todas as células da banda de origem
  delta = delta + CustoDaCélula();
Para todas as células da banda de destino
  delta = delta + CustoDaCélula();
```

Agora, é preciso conhecer a função **CustoDaCélula**(**cell**). Esta função deve calcular o custo de todas as redes conectadas à célula *cell*.. Assim, a função simplesmente soma a estimativa de WL de cada rede conectada à célula. Deve haver um cuidado para evitar a repetição do cálculo de uma rede. Dois mecanismos são considerados:

a) Armazenar uma variável booleana (*flag*) chamada "**calculou"** em cada rede. Neste caso, todas as células devem começar com a variável **calculou** em *false*. Quando fosse efetuado o cálculo, a *flag* **calculou** deve ser setada para *true*. Porém, sempre que se for recalcular o delta, deve-se atualizar todas as redes para *false* novamente. Este cálculo é bastante custoso, sendo que a otimização perde seu sentido.

b) Criar uma variável inteira para cada rede, chamada "**password"**. A ferramenta deve possuir um **password** próprio também. O **password** de todas as redes começa em 0 e o da ferramenta começa em 1. Sempre que for necessário acessar uma rede, é verificado o seu *password*. Se for diferente do *password* da ferramenta, a rede é processada e tem seu *password* atualizado (com o da ferramenta). Se é igual, a ferramenta simplesmente descarta a rede. Para passar para o próximo cálculo de delta, não é preciso setar todos os *passwords* das redes para 0. Simplemente incrementa-se o valor do *password* da ferramenta.

A segunda possibilidade é implementada na ferramenta do *Mango Parrot*. Esta otimização é bastante simples, mas muito eficaz. A tabela 19 compara execuções usando a otimização com execuções sem a otimização. Observa-se que o ganho é muito significativo.

TABELA 19 – Resultados experimentais da implementação proposta no cálculo do *wirelength*

Benchmark	Tempo de CPU **sem** otimização	Tempo de CPU **com** otimização	Ganho
C499	69	33	53%
C1908	193	314	39%
Alu4	1794	3492	49%
Bw	91	96	6%
C53	2267	4940	54%
Mult	561	1175	53%

O ganho não é constante. A variação é explicada pelo tamanho do circuito. Quanto maior o circuito, maior é o impacto da otimização, pois a função de custo original percorre todas as células enquanto que a otimizada não. Os resultados da tabela 19 usam perturbações mistas com 25% para cada uma das quatro funções. No caso de perturbações isoladas, as trocas de duas células sofrem um impacto maior devido a sua rápida função de Undo.

6.2.6.2 Perturbação Otimizada

Outra maneira de otimizar o tempo do *Simulated Annealing* como um todo é criar uma função de perturbação restritiva, que tenha alguma inteligência e gere posicionamentos com maior probabilidade de aceitação. Desta forma, reduz-se o número de *undos* e também é possível trabalhar com menos iterações.

Usar os limites mínimo e máximo do tamanho da banda é um exemplo de uma perturbação inteligente, pois restringe a perturbação de forma a evitar posicionamentos que sabidamente serão ruins. As perturbações direcionadas a força e mistas são outros exemplo, pois também diminuem a taxa de rejeições.

Existem outros trabalhos interessantes nesta linha. O trabalho de Lixin Su implementa um mecanismo de aprendizado *on-the-fly*. A idéia é usar informações de iterações anteriores para melhorar a função de perturbação, usando um método estatístico (regressão linear). Lixin dá resultados que melhoram em até 28% para o conjunto de *benchmarks* testado.

6.2.7 Greedy Simulated Annealing

Uma variação interessante da Simulação de Têmpera é o algoritmo Guloso (*Greedy*). É definido, simplesmente, pelo *Simulated Annealing* com temperatura constante em zero. Em outras palavras, o algoritmo Guloso nunca aceita um movimento que piore o estado atual do algoritmo. Assim, ele facilmente fica preso em um mínimo local. Um movimento ruim temporário pode levar a um segundo movimento muito bom, fazendo com que seja possível alcançar mais variações de estado e, consequentemente, ampliando o espaço de busca. A tabela 20 mostra (para C1908_3x3) diversos casos onde o algoritmo guloso é pior que o *Simulated Annealing*, justificando que não seja usado para posicionamento iterativo sem a ajuda de outro algoritmo. Serão vistas, porém, algumas boas aplicações para o Algoritmo Guloso: pós-processamento do posicionamento inicial visando equalizar o tamanho das bandas; pós-processamento do posicionamento iterativo, a fim de um ajuste final de algumas posições; pós-processamento do posicionamento iterativo visando redução do congestionamento do circuito.

TABELA 20 – Comparação entre Algoritmo Guloso e Simulated Annealing para posicionamento iterativo

Algoritmo construtivo	WL inicial	Algoritmo Iterativo	WL final	Tempo (s)
Plic-Plac	449961	SA	190989	95
		Greedy	225764	99
CG C20	324407	SA	184740	99
		Greedy	199911	105
Quadratura	289137	SA	181915	95
		Greedy	214004	97

A figura 63 mostra a evolução do custo da Simulação de Têmpera e do Algoritmo Guloso para o posicionamento inicial do Plic-Plac. Observe que o algoritmo guloso inicia com alta convergência, mas pára em seguida, preso em um mínimo local.

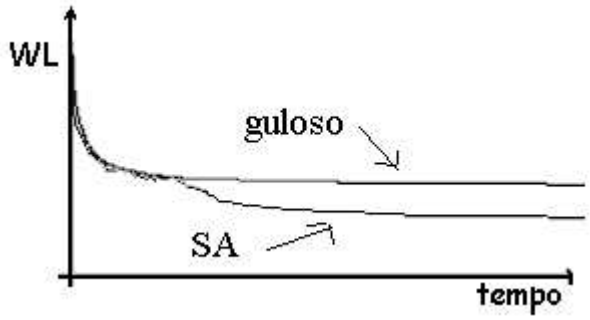

FIGURA 63 – Variação do custo da Simulação de Têmpera comparada com do Algoritmo Guloso

a) Pós processamento para posicionamento inicial.

Uma característica deste algoritmo é o alto índice de rejeições das perturbações. Isto degrada o desempenho do algoritmo, pois a função de *undo* é exaustivamente utilizada. Porém, o algoritmo guloso apresenta uma convergência muito grande, pois sempre melhora o estado atual. Por esta razão, o algoritmo Guloso pode ser utilizado como um pós processamento para o **posicionamento inicial**. No momento que chegar a um mínimo local, o processo deve parar, e começar um refinamento com o *Simulated Annealing*.

Esta idéia pode ser estendida para um algoritmo que modifica o posicionamento inicial para tornar-se mais compatível com o algoritmo iterativo. O *Simulated Annealing* implementado na ferramenta do *Mango Parrot* possui um limite mínimo e máximo de tamanho das bandas. Os limites são dados para que não haja grande desbalanceamento da largura das bandas. Então, é interessante que o posicionamento inicial respeite estas duas restrições. Para tanto, implementa-se uma função de custo que combina WL e variância do tamanho das bandas (visto na sessão 3.3.4).

Existem diversas maneiras de combinar a variância com o tamanho dos fios. Na ferramenta do *Mango Parrot*, simplesmente é feita uma soma dos dois valores, ignorando a sua incompatibilidade (μm de fio e μm^2 da variância de largura de bandas). Para verificar a qualidade do posicionamento e sua relação com o posicionamento construtivo, ele foi testado em seis exemplos de posicionamento inicial: Aleatório (1), Plic-Plac restrito (2), Plic-Plac não restrito (3), Quadratura CG10 (4), Quadratura FM (5) e ClusterGrowth C20 (6). O Posicionamento Aleatório é bastante equilibrado. O Plic-Plac restrito é relativamente equilibrado, pois há uma restrição de largura máxima. Já o Plic-Plac irrestrito é bastante desequilibrado. A Quadratura FM é bem equilibrada, com exceção da primeira banda. A Quadratura CG10, porém, é extremamente desequilibrada. O Cluster Growth C20 é extremamente equilibrado.

Os desenhos dos posicionamentos iniciais estão na figura 64, em ordem da esquerda para a direita e de cima para baixo.

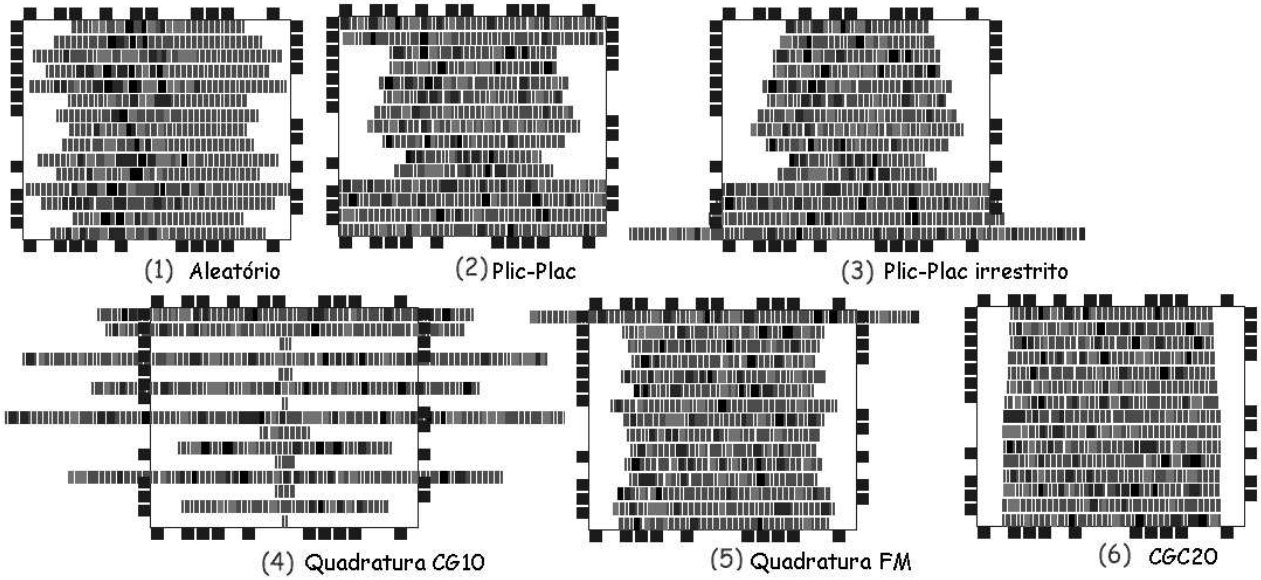

FIGURA 64 – Posicionamento inicial executado por diversos algoritmos

Os resultados do pós-processamento (com 10000 iterações) da figura 64 estão na figura 65.

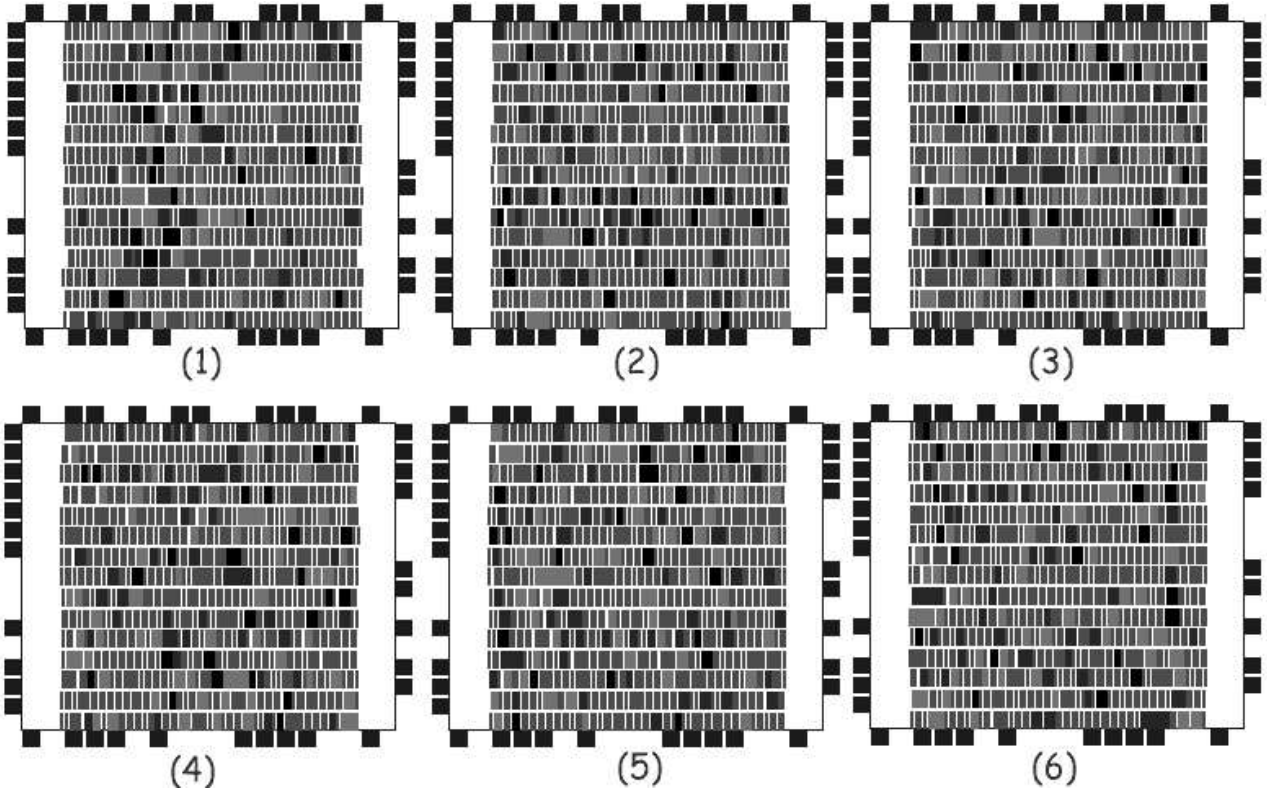

FIGURA 65 – Pós processamento usando a variância

A tabela 21 mostra a diferença em WL antes e depois da execução do pós-processamento usando a variância. Foram feitos dois testes usando a variância: o primeiro executa apenas 1000 iterações, e outro com 10000 iterações.

TABELA 21 –Comparação da Equalização por Variância e por Desvio Padrão

Algoritmo Construtivo	WL Inicial	Variância		Desvio Padrão	
		1000 iterações	10000 iterações	1000 iterações	10000 iterações
Random (1)	819284	686741	469204	641255	404336
Plic-Plac (2)	425262	495495	397491	386886	327044
Plic-Plac R (3)	404401	1010977	403322	384552	312085
Quad CG10 (4)	518482	3523161	461410	465988	337472
Quad FM (5)	280206	356996	308011	277462	262898
CGC10 (6)	404401	338324	316303	322750	295621

Há uma série de observações interessantes a fazer nas figuras 64, 65 e na tabela 21. Primeiro, observa-se que houve uma melhora drástica no equilíbrio de todas as bandas. Outra observação interessante é que os circuitos inicialmente equilibrados mantiveram-se assim, sem comprometer o tamanho dos fios. Porém, nos circuitos inicialmente desequilibrados, a equalização por variância prejudica o tamanho total das conexões. Usando 1000 iterações, os circuitos desequilibrados obtiveram um aumento drástico de tamanho dos fios. Isto acontece porque a variância é um valor excessivamente alto, que sobressaiu-se completamente em relação ao custo de WL. O uso de 10000 iterações ajudou o algoritmo, pois ele pode dedicar mais iterações para melhorar o WL quando as bandas já estavam equalizadas e a variância assume valores menores.

Observando o exagerado equilíbrio gerado pelo custo de variância, modificou-se o algoritmo para que fosse computado o desvio padrão (raiz quadrada da variância). Desta forma, aumenta-se, indiretamente, o peso do *wirelength* em relação à variação das bandas, fazendo com que o algoritmo preocupe-se mais com o tamanho dos fios e menos com a variabilidade das bandas. Observou-se, de fato, que o equilíbrio final foi reduzido, como mostra a figura 66, mas em nenhum caso foi comprometido o tamanho dos fios. Pelo contrário, o pós processamento pode melhorar a sua qualidade, como mostra a tabela 21. A única dificuldade foi verificada para os casos onde o desequilíbrio inicial era muito acentuado, como é o caso (3) e (6). No caso (3), o desequilíbrio se manteve por até 30000 iterações (na tabela 21 estão apenas as primeiras 10000), melhorando a WL até 290053.

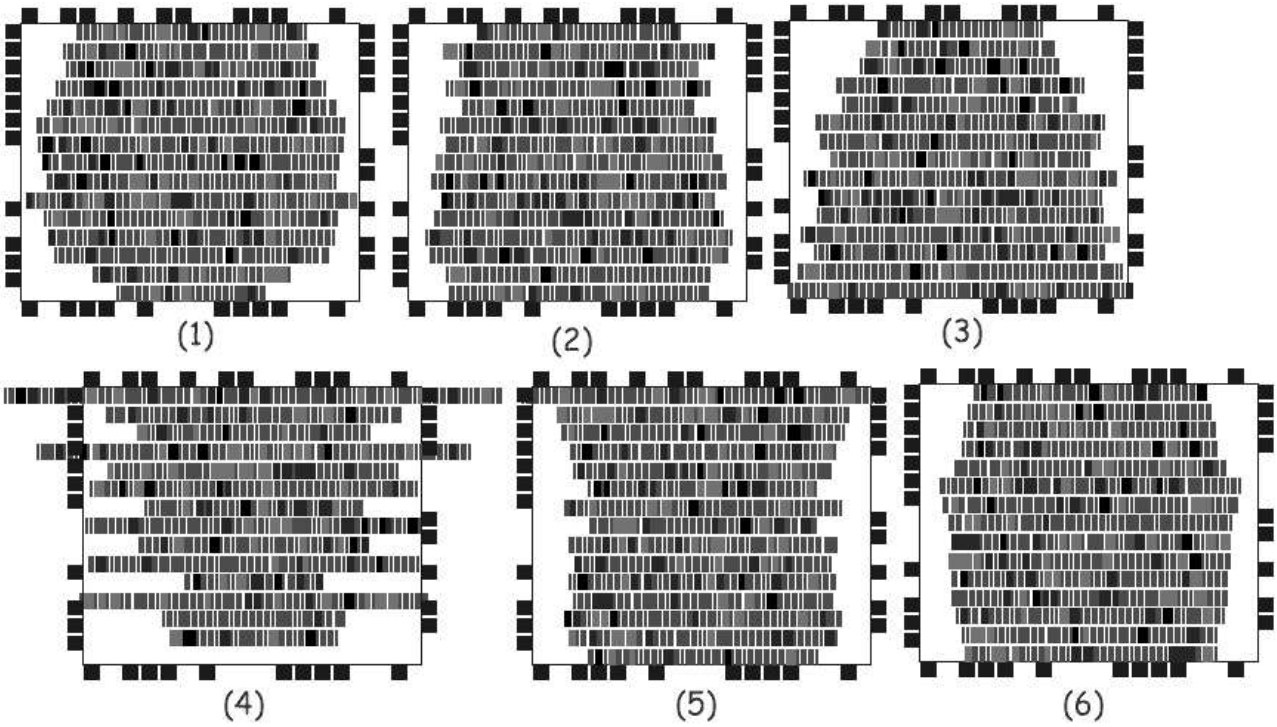

FIGURA 66 – Pós-processamento usando o desvio padrão.

b) Pós-processamento do posicionamento iterativo

Outro uso interessante para o algoritmo guloso é aplicá-lo ao **final do processo de posicionamento**. O objetivo é um ajuste final nas posições que ainda podem ser melhoradas. O algoritmo guloso garante que o estado final do posicionamento não é deteriorado, fazendo com que seja possível fazer um "acabamento" ao posicionamento.

A tabela 22 mostra alguns resultados de execução do Algoritmo Guloso, mostrando sua aplicação como pós-processamento do *Simulated Annealing*. Observe que o WL final é melhorado levemente. O circuito utilizado é o C1908_3x3, usando perturbações mistas (sessão 6.2.2.3).

TABELA 22 – Pós processamento do posicionamento iterativo usando o algoritmo Guloso

Num Iterações do SA	WL obtido	Num Iterações do pós-processamento	WL final	Tempo de execução do pós-processamento
150 x 100	248832	10000	232681	6
150 x 1000	190989	10000	190146	6
300 x 1000	182061	10000	181966	6
500 x 2000	164959	10000	164924	6
150 x 100	248832	100000	215767	70
150 x 1000	190898	100000	189062	69
300 x 1000	182061	100000	181163	68
500 x 2000	164959	100000	164455	65

Quanto pior é o WL de saída do SA, melhor será o impacto do pós-processamento. Em circuitos bastante otimizados (na tabela 22, os casos de 500 x 2000 iterações), o ganho é mínimo, tanto para 1000 quanto 10000 iterações.

c) Pós-processamento do posicionamento iterativo para reduzir o congestionamento

Uma terceira aplicação para o *Greedy*, relatada por Wang em 2000, usa o algoritmo para reduzir o congestionamento de um circuito depois de finalizado o posicionamento. O artigo introduz três métodos interessantes para redução de congestionamento *post-placement* O algoritmo guloso, é o que Wang chama de *Cell-Centric Greedy Algorithm*. Sua abordagem é um pouco diferente:

1) A primeira diferença é que o algoritmo é aplicado ao nível de posicionamento global. Em outras palavras, as células estão assinaladas a regiões (*global bins*) e posicionadas no centro destas regiões, com sobreposição. O posicionamento global já foi concluído, porém. É usado o algoritmo guloso para reduzir o congestionamento e, por fim, ele usa o *Simulated Annealing* com baixa temperatura inicial para posicionar as células corretamente dentro das regiões (de posicionamento global para posicionamento detalhado).
2) A função de custo do algoritmo não leva em conta WL, e sim puramente um modelo de congestionamento, como os que foram vistos na sessão 3.3.1.2. Precisamente, Wang utiliza A* para determinar as rotas das conexões e calcular o *overflow*, com *look ahead*, de cada região.
3) A função de perturbação deve ter mais inteligência que as funções de perturbação tradicionais, pois deve garantir que o posicionamento feito até então não seja estragado, já que a função de custo não considera o tamanho dos fios. O ideal é que sejam usadas funções de perturbação baseadas em cálculos de congestionamento, pegando células de regiões congestionadas e movendo para regiões vazias. Para evitar que o movimento seja aleatório, podem-se usar o cálculo da força.

Wang mostra em alguns exemplos de execução que pode reduzir o congestionamento total do circuito em 40% em média usando esta técnica. Outras duas técnicas são relatadas, com desempenho um pouco superior.

6.3 Algoritmos Genéticos

Assim como a Simulação de Têmpera, os algoritmos genéticos são genéricos, podendo ser aplicados em qualquer problema de otimização. Esta sessão vai apresentar algumas considerações na aplicação deles para posicionamento.

O algoritmo base foi visto na sessão 2.6.2. Esta sessão discute algumas implementações utilizadas em outros trabalhos. A ferramenta do *Mango Parrot* implementa os algoritmos genéticos para posicionamento, porém os resultados foram péssimos. Infelizmente, a pesquisa bibliográfica realizada foi insuficiente para avaliar eficientemente este algoritmo. Desta forma, esta sessão preocupa-se mais em discutir algumas desvantagens e problemas encontrados neste algoritmo, dentro da pesquisa que foi realizada. Fica para um próximo trabalho uma pesquisa aprofundada neste tema, visando um desempenho superior, como em Sait em 2001.

A sessão 6.3.1 mostra como deve ser a população. A sessão 6.3.2 discute as possibilidades de elitismo e quais se aplicam melhor em posicionamento. A sessão 6.3.3 apresenta algumas funções para realizar a mutação. A sessão 6.3.4 mostra possibilidades de representação do cromossomo visando a reprodução. A sessão 6.3.5. mostra algumas funções de *crossover*. O *crossover* é a parte mais importante do algoritmo, pois nele está contida a inteligência do algoritmo, que é a combinação de dois indivíduos, na geração de um terceiro com uma combinação das características de ambos.

6.3.1 População

Já foi visto que, para um problema de otimização qualquer, a população será formada por soluções viáveis para o problema em questão. No caso de posicionamento, têm-se uma população de posicionamentos, portanto. O tamanho da população é uma variável determinante para o bom desempenho do algoritmo. Uma pequena população de indivíduos vai levar a pouca diversidade, o que implica em baixa convergência. Uma população grande demais, por outro lado, leva ao uso de muita memória por parte do algoritmo. Tipicamente, o tamanho da população é determinado experimentalmente e fica entre 30 e 100 indivíduos.

Durante todo o processo, a população é mantida com o mesmo número. Desta forma, o algoritmo não precisa alocar e desalocar memória. A melhor estrutura de dados para armazenar os indivíduos é um vetor de tamanho fixo.

Uma questão importante é a determinação da população inicial. Ela deve ser aleatória, no sentido que deve haver grande diversidade. Quanto maior a diversidade, melhores são as chances de boa convergência do algoritmo. Usam-se, portanto, algoritmos construtivos e não-determinísticos para gerar a população inicial. Dentre os algoritmos vistos no capítulo 4, somente o posicionamento aleatório e o Crescimento de Aglomerados são não-determinísticos. Pode-se, porém, trocar a ordem da lista de células, que é entrada para os algoritmos, fazendo com que os resultados sejam diferentes.

6.3.2 Elitismo

O elitismo é um artifício dos algoritmos genéticos para garantir que as melhores soluções atingidas se mantenham, garantindo que nunca vai haver perda de uma boa solução atingida. A idéia é de garantir que um grupo de indivíduos, a "elite" da população, não seja descartado ao final da geração. O elitismo também garante que a elite não sofrerá mutação.

Na natureza, o elitismo não existe. Ninguém tem vida eterna. Porém, a reprodução garante a perseverança das qualidades das gerações anteriores. A tendência é que, de fato, haja um crescimento da qualidade da população, sem o uso de artifícios. Porém, os problemas computacionais, em geral, não têm o mesmo comportamento, como é o caso do posicionamento. O operador de *crossover* (como é visto na sessão 6.3.4) é um operador aleatório, ou seja, não mantém adequadamente as qualidades da geração anterior. Neste caso, o elitismo é fundamental.

A questão está centrada no tamanho da elite. O usual é usar elite unitária, ou seja, se preserva somente o indivíduo mais capaz. Uma elite muito grande diminui o tamanho efetivo da população que poderá sofrer mudanças, comprometendo a variabilidade da população e consequentemente a convergência do algoritmo. A relação correta é: quanto melhor for o seu *crossover*, menor deve ser sua elite. É usual, em posicionamento, usar elites de tamanho igual à metade da população , porque o operador de *crossover* é praticamente aleatório.

6.3.3 Mutação

A mutação deve dar ao processo uma maior variabilidade de indivíduos e suas características, auxiliando na fuga de mínimos locais. A taxa de mutação determina a quantidade de mutações. Muitas mutações deixam o algoritmo demasiadamente aleatório, enquanto que poucas mutações podem viciar o algoritmo e deixá-lo preso às soluções que possui.

No caso de posicionamento, a mutação pode ser idêntica à perturbação do *Simulated Annealing*, vista na sessão 6.2.2. A única diferença que deve ser considerada é que uma perturbação pode ser insuficiente para o caráter aleatório desejado. Assim, o ideal é realizar *n* perturbações. Na ferramenta do *Mango Parrot*, este mecanismo foi implementado, sendo que *n* é um parâmetro de entrada do usuário. N deve estar relacionado com o número de células do circuito.

6.3.1 Representação do cromossomo

O cromossomo é a estrutura que contém todas as informações de um indivíduo, ou seja, um dado posicionamento. O fenótipo é o posicionamento. O genótipo é a maneira como este posicionamento está codificado. Esta sessão procura apresentar algumas representações para o cromossomo.

A representação do cromossomo está relacionada com a função de *crossover*. Na ferramenta do *Mango Parrot*, usa-se uma classe para representar um posicionamento. Esta classe, vista na sessão 1.3.1, apresenta a estrutura de bandas, com uma lista de células para cada banda. Porém, para a maior parte das funções de *crossover* conhecidas, esta estrutura é incompatível. O normal é usar uma *string* contínua de células, como observa-se na figura 67. Observe que são acrescentados caracteres de fim de linha, para saber onde a banda termina.

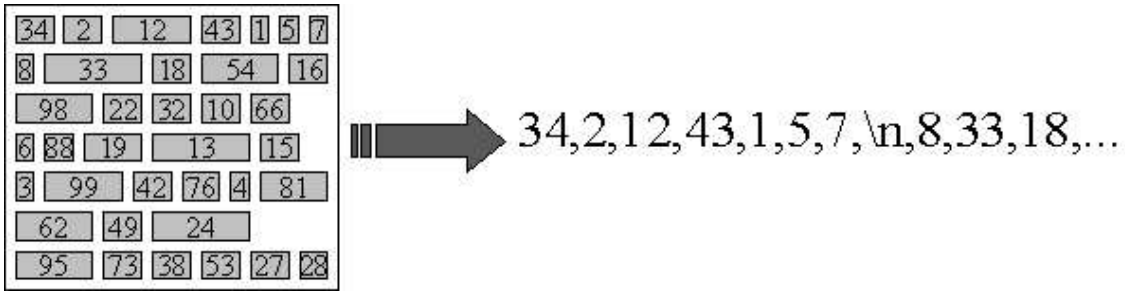

FIGURA 67 – Montagem do Cromossomo com caracteres de fim de linha

Este método, porém, apresenta o caracter de fim de linha, com o qual é difícil de lidar no *crossover*. Muitas combinações fatalmente gerarão posicionamentos com maior número de bandas que o máximo. Ainda, pode acontecer de haver bandas pequenas, em função de um caracter de fim de linha no começo da banda.

Para resolver este problema, eliminam-se os caracteres de fim de linha. Neste momento, o algoritmo perde a informação de vizinhanças verticais. Esta mudança acarreta em uma falsa vizinhança da última célula de uma banda para a primeira da banda seguinte. O melhor é montar a *string* de células de maneira alternada, como mostra a figura 68.

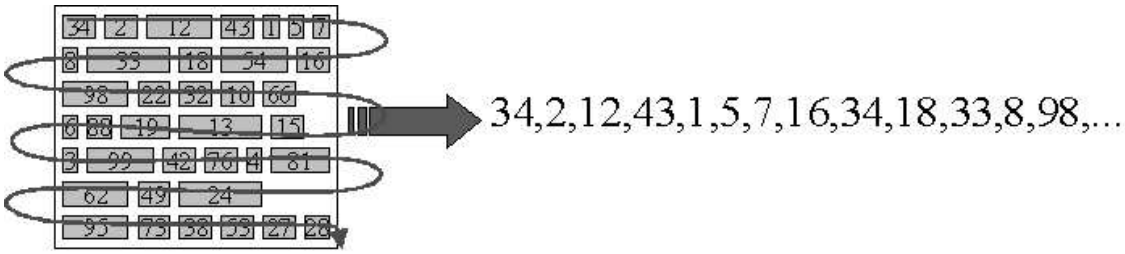

FIGURA 68 – Montagem do Cromossomo sem definição de fim de linha

Depois de codificado o cromossomo e realizado o *crossover*, é necessário que se monte novamente o posicionamento a partir do mesmo tipo de informação. Este processo pode ser realizado de duas maneiras:

a) Preenchimento das bandas superiores até sua lotação. Este procedimento pega o *string* de células e preenche as bandas superiores até que estejam na sua capacidade máxima. O problema deste método é que as bandas inferiores estarão vazias. Outro problema é com relação as vizinhas verticais. Desta forma, células vizinhas verticalmente provavelmente ficarão distantes horizontalmente.
b) Preenchimento de bandas de acordo com seu tamanho médio. Este procedimento tenta manter algumas vizinhanças verticais, pois deixa as bandas de tamanhos

próximos aos seus tamanhos originais. O problema é que a última banda pode apresentar super lotação, provavelmente passando do tamanho máximo.

Ambos os métodos estão ilustrados na figura 69. A ferramenta do *Mango Parrot* implementa o método b.

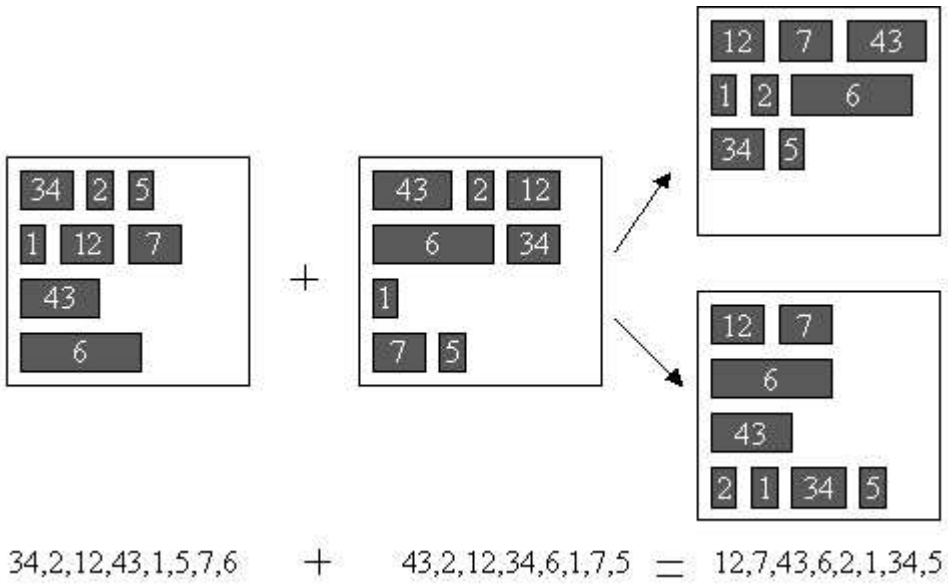

FIGURA 69 – Interpretação de um cromossomo codificado sem fim de linha

6.3.4 *Crossover*

A função de *crossover* é responsável pela combinação de uma solução do problema com outra solução. Ela deve manter informações das duas soluções combinadas em uma terceira. Para alguns problemas, como é o caso do posicionamento, esta combinação é difícil de ser feita de maneira inteligente.

As técnicas tradicionais de *crossover* trabalham com *strings* de dados, fazendo um corte em algum ponto aleatório desta *string*. O conteúdo da *string* de dados do filho será uma cópia da parte esquerda da *string* da mãe com a direita da *string* do pai. Porém, em posicionamento, não pode-se fazer este procedimento, pois certamente levaria a repetições de células em diversas posições, o que não é um posicionamento válido. Para que seja possível fazer um *crossover* eficientemente, são conhecidas na literatura algumas técnicas, que são resumidamente vistas aqui:

a) **PMX (*Partially Mapped Crossover*):** Trabalha com um ponto de corte, como os métodos tradicionais de *crossover*. Tudo o que está à direita do corte do pai é copiado para a direita do filho. A parte que está a esquerda do corte na mãe é copiada, excetuando-se repetições. No caso de repetições, é procurada iterativamente uma célula que não seja repetida. O algoritmo é descrito detalhadamente na figura abaixo (em C++). Algumas funções e variáveis auxiliares são utilizadas:

- Corte é um número inteiro que determina a posição do corte
- Pos(valor,cromossomo) é uma função que diz a posição do valor no cromossomo. Se o valor não for encontrado, Pos() retorna –1.

```
//lado direito do pai
for (int r=corte; r<sz; r++)
  filho[r] = pai1[r];

//lado esquerdo da mae
int valor;
for (int r=0; r<corte; r++) {

  // "valor" é o nosso valor legal
```

```
            valor = pai2[r];

            // enquanto encontrar o alelo duplicado no filho...
            while (Pos(valor, &filho) != -1) {

              // determina posicao no pai1
              int p1pos = Pos(valor, &pai1);

              // valor é o valor de pai2[p1pos]
              valor = pai2[p1pos];
            }

            // finalmente copia
            filho[r] = valor;
```

b) *Cycle Crossover*

Não trabalha com pontos de corte. Inicia escolhendo uma ID do pai. Considere o exemplo:

Pai: **1** 10 3 2 4 6 7 5 9 8
Mãe: 9 2 10 1 7 3 5 4 6 8

Escolheu-se a ID "1" do pai. Forma-se o seguinte cromossomo:
Filho: 1 - - - - - - - - -

O próximo passo é procurar em que posição *p* da mãe encontra-se o ID 1. No exemplo, *p* = 4. Então, a posição *p* do pai é copiada para o filho:

Filho: 1 - - 2 - - - - - -

Repete-se o algoritmo para o ID 2, até que se tenha um ciclo, como no exemplo:

Filho: 1 10 3 2 - 6 - - 9 -

Depois que foi concluído o ciclo, o restante do cromossomo é ocupado com os genes da mãe:

Filho: 1 10 3 2 7 6 5 4 9 8

c) *Order Crossover*

Trabalha com corte duplo de uma string de IDs. Acompanhe o exemplo:

Pai: 1 10 3 | 2 4 6 7 | 5 9 8
Mãe: 9 2 10 | 1 7 3 5 | 4 6 8

Primeiro, seleciona a string do pai entre os dois pontos de corte (2 4 6 7). Depois, apaga os IDs iguais na mãe:

Mãe: 9 – 10 1 – 3 5 - - 8

Depois copia a sub-string do pai nas posições vazias da mãe, seguindo a mesma ordem.

Filho : 9 2 10 1 4 3 5 6 7 8

Starkweather, em 1991, há uma análise e comparação dos métodos de crossover mais famosos, como os três apresentados nesta sessão. O artigo conclui que a qualidade do operador depende muito da sua aplicação. Starkwheather destaca que alguns operadores preservam melhor as adjacências (como o Order e PMX) enquanto que outros operadores (Cycle) não preservam. Algumas aplicações, como o Posicionamento, codificam na ordem dos genes uma informação muito importante que deve ser preservada.

7 Algoritmos direcionados a força

Esta sessão apresenta alguns algoritmos derivados da idéia de posição de força zero, vista na sessão 3.3.1. Basicamente, é feita uma analogia a um sistema físico de forças de atração e repulsão. As forças de atração ocorrem em células que estão conectadas, como seu houvesse um elástico amarrado a cada uma das células. As forças de repulsão ocorrem em células que se sobrepõem.

Na sessão 3.3.1 é apresentada uma equação, chamada de equação de força zero, que indica a posição ótima para uma célula, dado que as outras células estão paradas e que não há força de repulsão. Esta equação é usada em praticamente todos os métodos vistos nesta sessão.

Existem diversos algoritmos derivados dessa idéia de sistema de forças. Na ferramenta do *Mango Parrot* foram implementados dois algoritmos: um construtivo e outro iterativo. Além destes, na sessão 7.3, é discutida uma terceira idéia, publicada por Chou em 2002. Esta idéia exige a presença de forças de repulsão aliadas às forças de atração. Além disto, trata-se de um trabalho muito recente.

Este trabalho já apresentou, indiretamente, outros algoritmos baseados na idéia de algoritmos direcionados a força, que são as meta-heurísticas com movimentos *force directed*.

7.1 Algoritmo Construtivo

Este algoritmo inicia posicionando todas as células em duas bandas: a primeira e a última. A ordem em que as células estão dispostas é aleatória, sendo que não há intercessão de área. Após isto, é usada uma série de movimentos baseados na equação de força zero, até atingir uma condição de parada. São levantadas algumas possibilidades de parada:

- Número de iterações: utilizam-se um número fixo de iterações, que está relacionado com o número total de células.
- Qualidade desejada: escolhe-se o WL desejado. O algoritmo vai parar quando atingir a qualidade alvo. Deve haver, porém, um controle de número máximo de iterações em conjunto
- Distância mínima de movimento: o algoritmo termina quando todas as células não moverem-se mais do que um limiar definido anteriormente. Usualmente, usa-se o limiar igual à altura da banda.

O capítulo 9 mostra alguns resultados de simulação deste algoritmo, comparado com outros algoritmos construtivos.

7.2 Algoritmo Iterativo

Esta sessão descreve o algoritmo iterativo que foi implementado na ferramenta do *Mango Parrot*.

Na ferramenta do *Mango Parrot*, o algoritmo iterativo é simplesmente o algoritmo de *Simulated Annealing* sem a parte da aceitação. Ou seja, ele aceita qualquer alteração. Trata-se de um algoritmo guloso também, pois suas perturbações são direcionadas a força, e é incapaz de desfazer um passo anterior.

A condição de parada do algoritmo é semelhante às possibilidades vistas na sessão anterior, para o algoritmo construtivo. Na ferramenta do *Mango Parrot*, foi implementado a condição de parada por número de iterações.

7.3 Algoritmo Construtivo de Chou

Yih-Chih Chou apresenta um algoritmo baseado em forças. Ele é um algoritmo construtivo, embora Chou não use nenhuma espécie de refinamento iterativo no final do processo. O algoritmo possui estruturas específicas para otimização de *timing*. Nesta sessão, porém, será considerada somente a construção básica do algoritmo. Todo o texto desta sessão baseia-se no texto de Chou.

O algoritmo divide-se em três fases: 1- Inicialização; 2 – Equilíbrio de Forças; 3- Formação das bandas. O processo é iterativo. Repetem-se os passos 2 e 3 diversas vezes. Sempre são consideradas somente a primeira e a última banda. Depois disto, é reduzida a área de trabalho, excluindo-se as duas bandas posicionadas. A partir de então, as células desta banda são consideradas fixas, como *pinos* de E/S. O processo é repetido até que não haja mais bandas.

A etapa de inicialização (1) apenas posiciona todas as células no centro do circuito, com sobreposição de área. Na abordagem de Chou, permitem-se sobreposições que são resolvidas na etapa 3, de formação de bandas. A presença dos *pads* e as forças de repulsão entre células sobrepostas faz com que as células se espalhem ao longo da área.

O algoritmo, descrito em uma pseudo-linguagem, pode ser observado abaixo.

```
Procedure FDP
Begin
        Inicializar
        Force_balance();
        Repeat
                Posicionar a banda superior()
                Posicionar a banda inferior()
                Force_balance();
        Until all rows are placed
end
```

Etapa 2: Equilíbrio de Forças

Esta etapa gradualmente move todas as células para sua posição de força zero. A coordenada y deve se alinhar a uma banda, mas a posição x é livre, permitindo sobreposição. O processo é repetido até que nenhuma célula consiga se mover mais do que uma distância *H*. A figura 70 ilustra esta etapa.

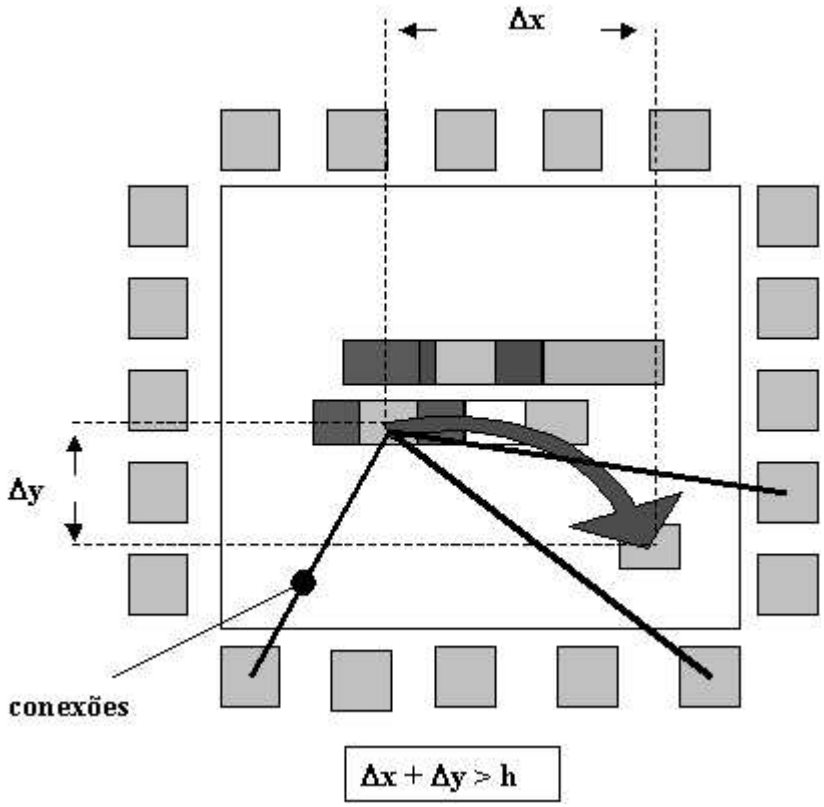

FIGURA 70 – Equilíbrio de forças no algoritmo de Chou

O algoritmo baseia-se em uma estrutura de fila. No início, todas as células adjacentes a pinos de entrada e saída são inseridas na fila. Enquanto a fila não estiver vazia, a célula do topo da fila é selecionada. É calculada a sua posição de força zero, assumindo que todas as outras células estão fixas. Se a distância é menor que um número *H* (pré-determinado), a célula não se move. Caso contrário, a célula é movida para a posição calculada. Chou utiliza a altura da banda como valor de x. A figura a seguir mostra o algoritmo em uma pseudo-linguagem de programação (reproduzida de Chou 2002).

```
Procedure force_balance
Begin
        Inicializar uma fila Q;
        Para cada célula c adjacente a um pino de E/S insere c no final de Q;

        Enquanto Q ≠ ϕ
                c = topo de Q;
                (x,y) = PosiçãoForçaZero(c);
                d = distância de c até (x,y);
                Se (d > H) /* H é a altura da banda*/
                        Mover c para (x,y)
                        Para cada célula v conectada a c e que ainda não esteja na fila
                                Inserir v no fim de Q;
end
```

Existe uma diferença entre o cálculo da posição de força zero apresentada aqui em relação ao cálculo de Chou. Na abordagem apresentada por Chou, existem forças de repulsão nas posições onde há sobreposição de células. A repulsão garante que as células que são inicialmente postas no centro da área se espalhem ao longo do circuito.

Etapa 3: Montagem das bandas

Esta fase descreve a montagem das bandas a partir da descrição das células com sobreposições.

Inicialmente, a banda superior é preenchida. Todas as células que possuem a coordenada y respectiva desta banda são consideradas. A ordem na banda é regida pela coordenada x das células. Como são permitidas sobreposições, possivelmente a banda esteja excessivamente cheia. Neste caso, são movidas as últimas células para a próxima banda. No caso contrário (muitos espaços vazios) são movidas células da bandas adjacente para a banda em posicionamento. O processo é idêntico para a última banda.

8 Comparação Geral

Esta sessão apresenta diversas comparações entre algoritmos de posicionamento. Primeiro, é feita uma comparação apenas dos algoritmos construtivos. Depois, o conjunto de algoritmos construtivos e iterativos, com pós-processamento de ambas as etapas. Buscam-se as melhores seqüências de algoritmos para posicionar adequadamente as células de um bloco. No final da sessão, é feita uma comparação dos posicionadores da ferramenta do *Mango Parrot* com o posicionador do TROPIC.

Algumas comparações são particularmente interessantes, por compararem circuitos mapeados para portas complexas (Alu4_4x4) com o mesmo mapeado para portas simples (Alu4_2x2). Pode-se esperar que o posicionamento do circuito com portas simples seja mais demorado, por apresentar mais elementos a tratar, aumentando a complexidade da etapa de posicionamento. Em FPGAs, por exemplo, é feito um agrupamento de LUTS em *clusters* para diminuir a complexidade do posicionamento e roteamento (Marquardt 2000).

Os algoritmos são comparados segundo as seguintes variáveis:

- Estimativa de tamanho dos fios.
- Estimativas de Congestionamento
- Tempo de CPU
- Roteabilidade no fluxo de síntese usando as ferramentas de Lazzari e Hentschke
- Roteabilidade no Tropic

Para estimar o tamanho dos fios, usa-se o método do semiperímetro (sessão 3.3.1.1). Para o congestionamento, usa-se o método da Densidade Máxima por *Bounding-Boxes* (sessão 3.3.1.2.2).

A roteabilidade é medida em função de dados de roteadores reais. Usa-se o gerador de células de Lazzari (2002) para a geração do leiaute do circuito. A partir do leiaute, são definidos pontos de roteamento, que são descritos em um arquivo de interface. O arquivo é lido pelo Roteador do Dragão Limão (Hentschke 2002) usando de 2 a 5 níveis de metal para roteamento. Diferentemente do roteador do Tropic, este roteador não pode atuar sobre o leiaute para abrir espaços. Assim, algumas conexões podem ser inviabilizadas pela falta de espaços disponíveis. A porcentagem de redes roteadas é a estimativa de roteabilidade considerada. O roteador do Dragão Limão baseia-se na técnica de *Maze Routing*, que é sabidamente uma técnica restrita para roteamento detalhado. Como baseia-se em um algoritmo fortemente guloso e dependente de uma ordenação, ao final do processo apresenta dificuldades para completar as conexões. Por esta razão, o método é ruim, mas é útil para avaliar o posicionamento. Os Maze Routers são bastante usados para roteamento global, onde não há restrições de cruzamento, sendo que a ordem das conexões não é tão relevante. É muito usado para estimar congestionamento a nível de posicionamento.

No gerador de leiaute do Tropic, o espaço dedicado ao roteamento varia conforme a complexidade do circuito, havendo canal de roteamento de altura variável entre as bandas. Assim, quanto melhor for o posicionamento, menor será a área de roteamento, aumentando, portanto, a densidade (transistores por milímetro quadrado). Assim, a comparação de roteabilidade no Tropic é feita pela densidade do leiaute. Os dados são obtidos a partir da execução do Tropic com um parâmetro especial para considerar o posicionamento da ferramenta do *Mango Parrot*. Algumas modificações são necessárias no posicionamento do *Mango Parrot* para seu uso integrado ao Tropic. Como existem canais de roteamento de alturas variáveis, a estimativa de altura será afetada. Ela deve ser bastante pessimista, para minimizar o roteamento vertical.

8.1 Algoritmos Construtivos

A fim de encontrar a melhor técnica de posicionamento inicial, foram feitas comparações de todos os algoritmos implementados na ferramenta de síntese do *Mango Parrot*.

As tabelas 25, 26, 27, 28, 29 mostram os dados de execução para os seguintes circuitos, respectivamente: C1908_3x3, C53, Alu4_4x4, Alu4_2x2, Misex3. Apenas para o circuito C1908_3x3 há dados de roteamento usando o gerador de Lazzari e Roteador do Dragão Limão. Isto ocorre por limitações destes sistemas, que estão em fase de implementação e não conseguiram processar os demais circuitos. Todas as tabelas citadas consideram pinos de entrada e saída no bloco. Eles são lidos do arquivo de entrada e posicionados aleatoriamente. A estimativa de altura e largura usada é a seguinte: 80 de altura, 5 de largura por transistor. Usam-se os algoritmos descritos na tabela 24 para comparação.

TABELA 23 – Algoritmos Construtivos em comparação

Algoritmo A	**Aleatório**
Algoritmo B	**Plic-Plac** com pesquisa DFS para altura, BFS para largura e restrição de tamanhos máximos de banda.
Algoritmo C	**Cluster Growth** C com 20 níveis de profundidade.
Algoritmo D	**Particionamento em Quadratura**, usando o particionamento aleatório seguido de Fidducia-Mateyeses
Algoritmo E	**Particionamento em Bisseção de Bandas**, com algoritmo de particionamento aleatório seguido de FM. Algoritmo A: Aleatório
Algoritmo F	**Posicionamento directionado a Forças** usando o algoritmo construtivo visto na sessão 7.1

TABELA 24– Comparação de algoritmos construtivos sobre o C1908_3x3 (783 células)

Alg.	WL	CPU	Cong	Roteamento com RotDL			
			(fios/BB)	2	3	4	5
A	828479	0	401	25%	-	-	-
B	390666	0	87	57%	76%	90%	90%
C	328624	0	70	67%	95%	97%	97%
D	309080	3	86	62%	79%	90%	89%
E	314865	2	87	57%	58%	80%	81%
F	348299	1	82	57%	81%	92%	91%

TABELA 25 – Comparação de algoritmos construtivos sobre o C53 (2307 células)

Alg.	WL	CPU	Cong
A	6091991	0	1307
B	4245944	0	867
C	4155273	2	827
D	4863666	31	900
E	4790069	15	872
F	-	-	-

TABELA 26 – Comparação de algoritmos construtivos sobre o Alu4_4x4 (2523 células)

Alg.	WL	CPU	Cong
A	6093522	0	1248
B	5506767	0	1005
C	5477311	4	958
D	3475286	56	517
E	3690933	27	535
F	3553700	14	400

TABELA 27 – Comparação de algoritmos construtivos sobre o Alu4_2x2 (4844 células)

Alg.	WL	CPU	Cong
A	11328331	0	1589
B	-	-	-
C	8817320	13	930
D	5405946	831	592
E	6118671	113	655
F	6123342	51	468

TABELA 28 – Comparação de algoritmos construtivos sobre o Misex3 (5477 células)

Alg.	WL	CPU	Cong
A	17830702	0	1765
B	17406626	0	1594
C	17019316	17	1439
D	11021284	1133	560
E	11172222	64	600
F	-	-	-

A primeira observação importante, em relação às tabelas, é em relação aos algoritmos que não foram executados, marcados pelo caracter " – ". Isto acontece por limitações dos sistemas e algoritmos implementados, que ainda estão com *bugs*. No caso da execução do Plic Plac no circuito Alu4_2x2, ocorreu o erro *stack overflow*, pelo uso da recursividade em um circuito muito grande. Este erro é uma limitação do sistema operacional Windows e da máquina utilizada, sendo que não é falha do algoritmo ou de sua implementação.

Outra observação importante é referente aos altos tempos de processamento para posicionamento usando particionamento nos circuitos maiores. Por limitações de máquina e sistema operacional, houve um intenso uso de memória virtual, aumentando demasiadamente o tempo de execução do algoritmo. Há uma necessidade de armazenamento de parâmetros de rotinas recursivas, o que implica em uma forte necessidade por memória, o que é uma desvantagem deste tipo de algoritmo.

Para análise dos dados válidos, o gráfico da figura 71 pode ser utilizado.

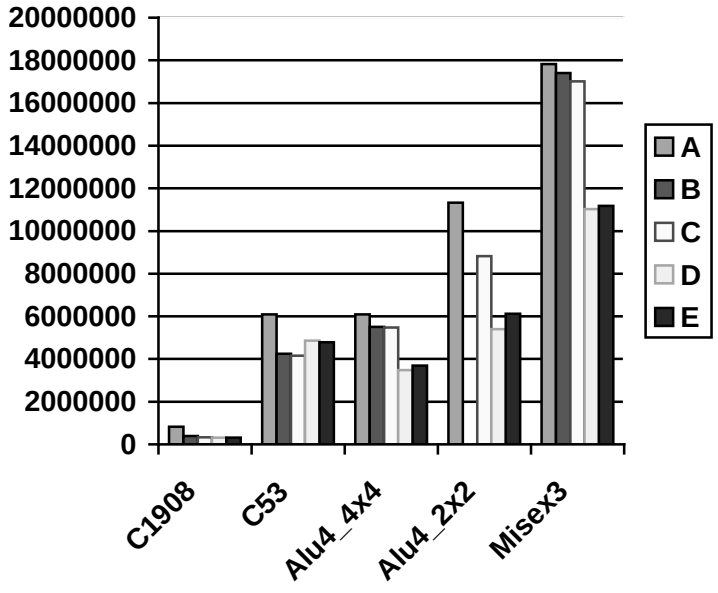

FIGURA 71 – Variação de *wirelength* em algoritmos construtivos

Analisando as tabelas juntamente com o gráfico, pode-se obter uma série de conclusões importantes.

Na tabela 24, os dados de roteamento são interessantes por mostrar que a estimativa de WL é uma boa estimativa para roteabilidade. O mesmo vale para a estimativa de congestionamento. O caso com menor congestionamento (70) é também o caso com o maior percentual de redes roteadas para dois níveis de metal (67%). Observou-se também, que o acréscimo de níveis de metal melhora a roteabilidade, mas há um limite prático. Para este circuito, o algoritmo com melhor desempenho foi o *Cluster Growth* C 20.

Nas demais tabelas, usaram-se circuitos maiores. Observa-se que o algoritmo Direcionado a Forças (F), e os baseados em particionamento (D e E) apresentam um melhor desempenho, porém com um maior tempo de processamento. O algoritmo Plic-Plac mostra seu desempenho equivalente ao algoritmo aleatório, porém com um bom resultado de WL. No circuito C53 (tabela 26), os algoritmos com melhor desempenho são o Plic-Plac e *Cluster Growt*h C 20.

Analisando-se puramente o gráfico, observa-se que, com o aumento da complexidade do circuito, a tendência é que o Plic-Plac e o *Cluster Growth* aproximem-se do posicionamento aleatório, enquanto que os baseados em particionamento se destacam. Note que os algoritmos baseados em particionamento estão implementados sem *terminal propagation* (sessão 5.4.3), o que melhoraria ainda mais seu desempenho. Entre estes dois casos, está o algoritmo direcionado a forças.

O melhor algoritmo deve ser escolhido pela sua aplicação principal. Caso o objetivo seja um processamento rápido e uso de pouca memória sem exigência de uma maior qualidade, os algoritmos Plic-Plac e *Cluster Growth* mostram-se adequados. Esta situação ocorre, por exemplo, no caso de uso posterior de *Simulated Annealing*. Porém, se não há um limite pequeno de recursos, os algoritmos baseados em particionamento são os mais indicados por apresentarem a melhor qualidade de congestionamento (roteabilidade) e WL (roteabilidade, *timing* e potência). O algoritmo direcionado a forças está no meio do caminho.

As tabelas seguintes (30 até 34) correspondem à execução de posicionamento dos mesmos circuitos anteriores, porém passam por um pré-processamento no Tropic para eliminar a hierarquia de células. O objetivo é fazer a mesma comparação de roteamento feita anteriormente usando o roteamento do Tropic. Estes testes não consideram pinos de entrada e saída. Ainda, eles usam diferentes estimativas de altura de banda e largura de célula, para que sejam mais compatíveis com o Tropic (200 de altura, 5 de largura por transistor). Por esta razão, os dados de WL são bastante diferentes em relação às tabelas anteriores. Alguns circuitos não puderam ser roteados por faltar espaços para *feedthroughts*. Na coluna de densidade de transistores por milímetro quadrado, eles são marcados por **Ñ** seguido pelo número de *feedthroughts* não posicionados. O algoritmo Plic-Plac não é comparado por considerar os pinos de E/S.

TABELA 29 – Comparação de algoritmos construtivos sobre o C1908 (783 células) mapeado para o Tropic

Alg.	WL	CPU	Cong.	Densidade Tropic
A	1153335	0	325	Ñ (1594)
C	469480	0	59	Ñ- (145)
D	396090	3	73	Ñ – (1)-
E	402400	2	72	37K
F	541850	1	83	Ñ – (274)

TABELA 30 - Comparação de algoritmos construtivos sobre o C53 (2307 células) mapeado para o Tropic

Alg.	WL	CPU	Cong.	Densidade Tropic
A	10024515	0	1055	Ñ (14028)
C	2448316	6	1080	Ñ (584)
D	2167350	40	659	Ñ (33)
E	2464495	40	671	24K
F	3226090	43	362	Ñ (2733)

TABELA 31 - Comparação de algoritmos construtivos sobre o Alu4_4x4 (2523 células) mapeado para o Tropic

Alg.	WL	CPU	Cong.	Densidade Tropic
A	8388175	0	890	Ñ (31748)
C	8321175	3	896	Ñ (22169)
D	5526985	34	438	Ñ (3463)
E	6174295	25	543	Ñ (3977)
F	5587970	22	366	Ñ (6957)

TABELA 32 - Comparação de algoritmos construtivos sobre o Alu4_2x2 (4844 células) mapeado para o Tropic

Alg.	WL	CPU	Cong.	Densidade Tropic
A	14842650	0	1055	Ñ (36100)
C	13933550	13	1080	Ñ (16558)
D	10032930	124	659	Ñ (8331)
E	10191290	113	671	Ñ (7301)
F	8703950	93	362	Ñ (13165)

TABELA 33 - Comparação de algoritmos construtivos sobre o Misex3 (5477 células) mapeado para o Tropic

Alg.	WL	CPU	Cong.	Densidade Tropic
A	31392595	0	2953	Ñ (59970)
C	14863830	42	421	Ñ (16158)
D	6379150	665	320	Ñ (606)
E	6847690	297	335	Ñ (117)
F	11042875	279	318	Ñ (16467)

A partir destes dados, foi montado o gráfico da figura 72. Os números negativos correspondem à quantidade de *feedthroughs* não roteados, enquanto que os números positivos correspondem à densidade do leiaute.

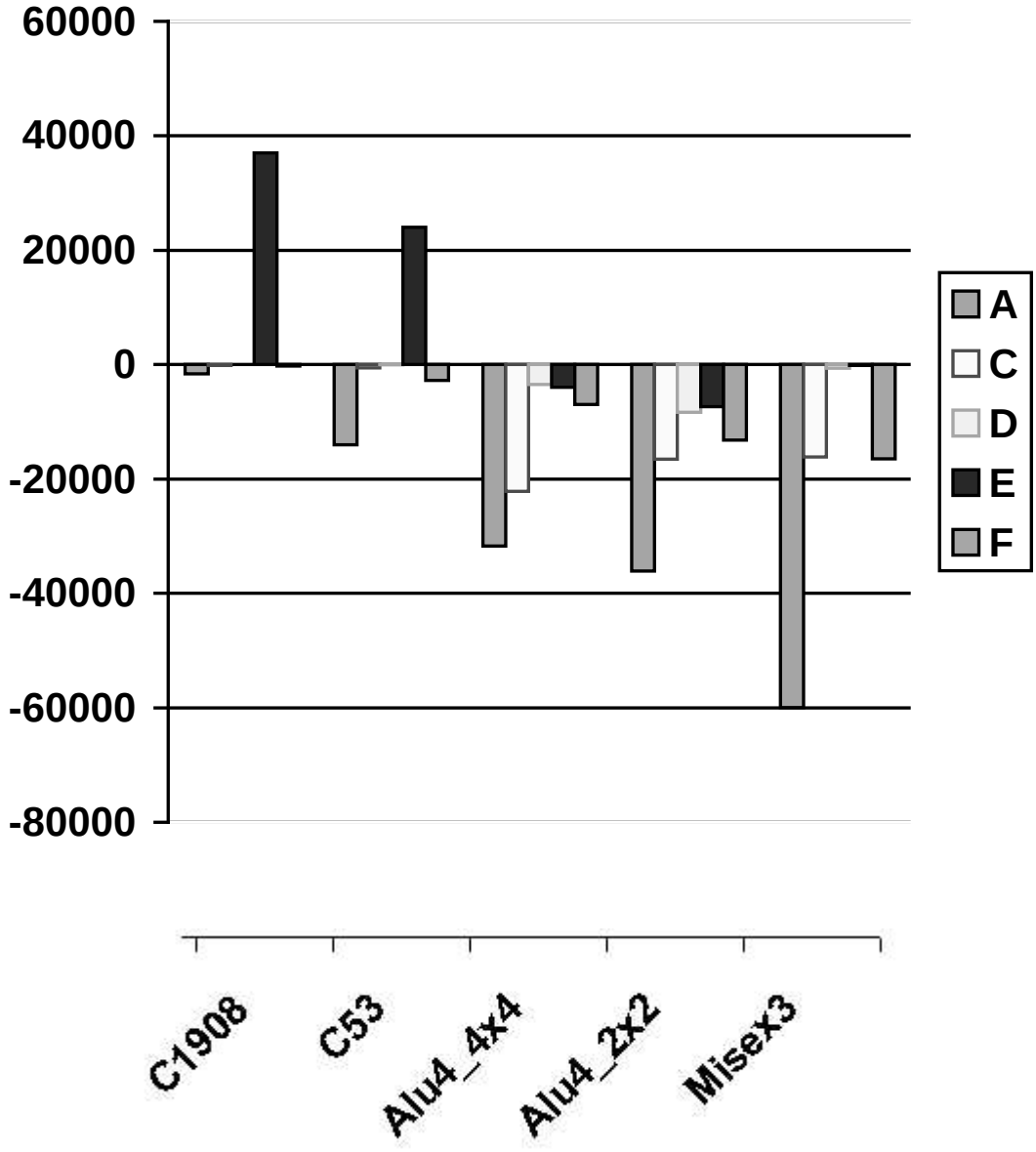

FIGURA 72 – Comparação de Roteabilidade no Tropic dos Algoritmos Construtivos A,B,C,D e E

Analisando o gráfico, observa-se que o algoritmo de bisseção em bandas apresenta a melhor roteabilidade para o Tropic. Isto pode ser explicado pela característica de leiaute do Tropic. Usa-se canais de roteamento de tamanhos variados entre as bandas, de forma que as vizinhanças verticais são mais distantes, prejudicando a estimativa de altura das bandas. Por isto, um algoritmo que minimiza as conexões verticais apresenta um melhor comportamento.

Outra característica notável é que os algoritmos de particionamento são os melhores, seguidos do algoritmo de FD (Algoritmo directionado a Forças) e CGC (Crescimento de Aglomerados, versão C). Conforme cresce a complexidade do problema, cresce esta vantagem, conforme tinha sido mencionado anteriormente.

Comparando-se o resultado de roteamento do Tropic com a estimativa de *wirelength*, observa-se que não há uma correlação forte entre estas duas variáveis. Alguns circuitos com menor *wirelength* são piores para rotear, e vice-versa. O problema na estimativa ocorre devido à altura de banda ser muito variada no Tropic. Na tabela 30, referente ao C1908 roteado com o

Roteador Do Dragão Limão, observa-se que a estimativa de *wirelength* corresponde com a roteabilidade. O mesmo fenômeno observa-se com relação à estimativa de congestionamento.

8.2 Algoritmos construtivos com pós-processamento

O uso do algoritmo de posicionamento inicial sozinho possui como problema o desequilíbrio do tamanho das bandas, conforme foi visto em alguns exemplos da sessão 3.3.4. Isto leva a problemas de roteamento, como pode ser observado na figura 73, que reflete o roteamento de C1908_3x3 com Quadratura.

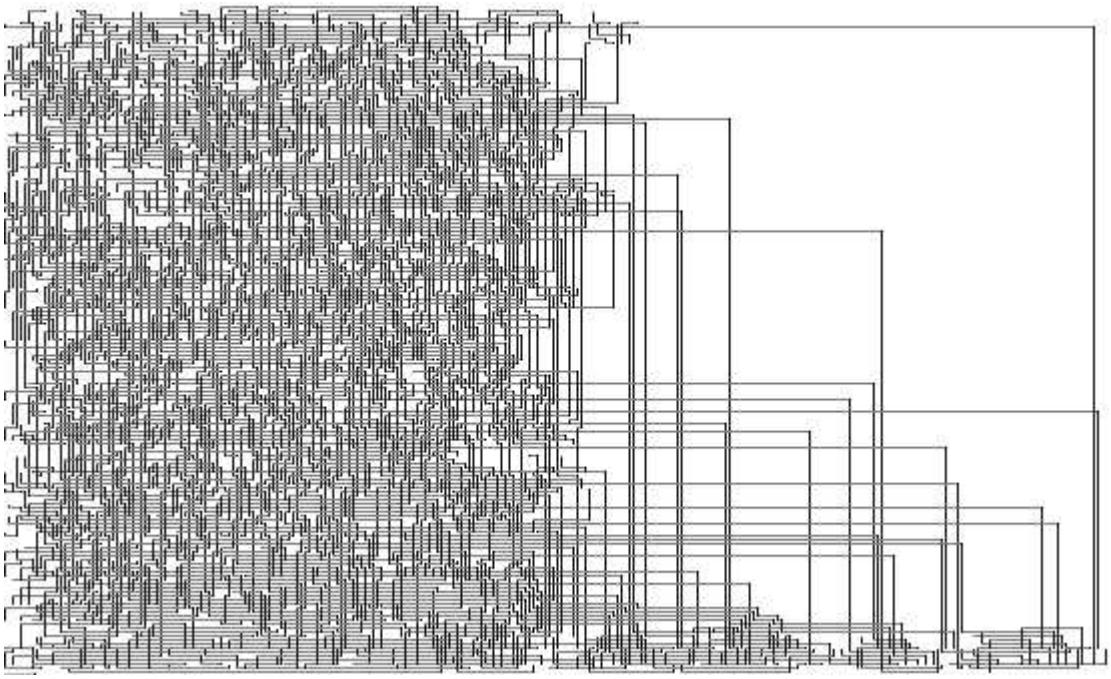

FIGURA 73 – Roteamento de um circuito com largura de banda desbalanceada (figura gerada pelo roteador do Dragão Limão)

Para evitar este problema, usa-se o pós-processamento proposto na sessão 6.2.7. A tabela 35 mostra os dados para o circuito C1908_3x3 usando equalização baseada na variância das bandas, enquanto que a tabela 36 mostra o mesmo dado usando o desvio padrão das bandas.

TABELA 34 - Comparação de algoritmos construtivos com pós-processamento (variância) sobre o C1908_3x3 (783 células)

Alg.	WL	CPU	Cong.	Roteamento com RotDL			
				2	3	4	5
A	385347	0+7	100	51%	74%	88%	88%
B	306172	0+6	65	63%	85%	94%	94%
C	270464	0+7	51	73%	95%	97%	97%
D	278840	3+6	62	68%	87%	95%	94%
E	302715	2+6	78	67%	80%	92%	92%
F	-	-	-	-	-	-	-

TABELA 35 - Comparação de algoritmos construtivos com pós-processamento (desvio padrão) sobre o C1908_3x3 (783 células)

Alg.	WL	CPU	Cong.	Roteamento com RotDL			
				2	3	4	5
A	329044	0+6	73	61%	78%	93%	93%
B	306172	0+6	54	74%	91%	96%	96%
C	238742	0+6	39	78%	98%	98%	98%
D	236238	3+6	57	79%	93%	95%	95%
E	274330	2+6	71	70%	93%	95%	95%
F	-	-	-	-	-	-	-

As tabelas mostram uma queda em WL com relação às tabelas anteriores (em média, 27% para variância e 35% para DP). Com o congestionamento também ocorre uma redução (em média, 48% para variância e 57% para DP). A comparação da roteabilidade é feita na figura 74.

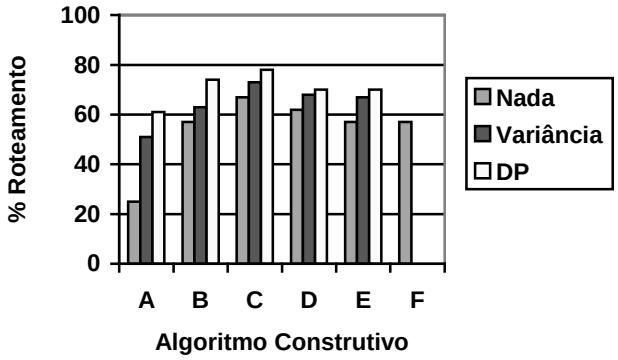

FIGURA 74 – Porcentagem de redes roteadas para C1908_3x3 com e sem pós-processamento do posicionamento construtivo

O gráfico e as tabelas apresentadas correspondem ao circuito C1908, que é um circuito pequeno. Por esta razão, os dados de roteamento mostram uma vantagem do Plic-Plac e do *Cluster Growth* sobre os baseados em particionamento. É interessante perceber, no entanto, que a roteabilidade aumenta significativamente com o acréscimo do pós-processamento. Ainda, percebe-se uma vantagem do Desvio Padrão com relação à variância. Lá, foram analisados os dois métodos, e apontou-se que a variância domina o *wirelength*, guiando excessivamente o algoritmo *Greedy* para equalizar as bandas, desconsiderando efeitos em *wirelength*.

O mesmo experimento é repetido no *software* Tropic. Os resultados estão nas tabelas 37 e 38.

TABELA 36 - Comparação de algoritmos construtivos com pós-processamento (variância) sobre o C1908_3x3 (783 células) mapeado para o Tropic

Alg.	WL	CPU	Cong.	Densidade Tropic
A	494025	0+10	86	Ñ (98)
C	329400	0+10	43	50K
D	415885	3+10	61	Ñ (18)
E	394275	4+10	65	42K
F	422630	2+11	59	Ñ (175)

TABELA 37 - Comparação de algoritmos construtivos com pós-processamento (desvio padrão) sobre o C1908_3x3 (783 células) mapeado para o Tropic

Alg.	WL	CPU	Cong.	Densidade Tropic
A	405680	0+10	63	Ñ (83)
C	279685	0+10	29	56K
D	327610	3+10	55	46K
E	314085	4+10	52	41K
F	309525	2+11	46	Ñ (34)

A partir destes dados, foi montado o gráfico da figura 75. Os números negativos correspondem à quantidade de *feedthroughs* não roteados, enquanto que os números positivos correspondem à densidade do leiaute dividida por 100.

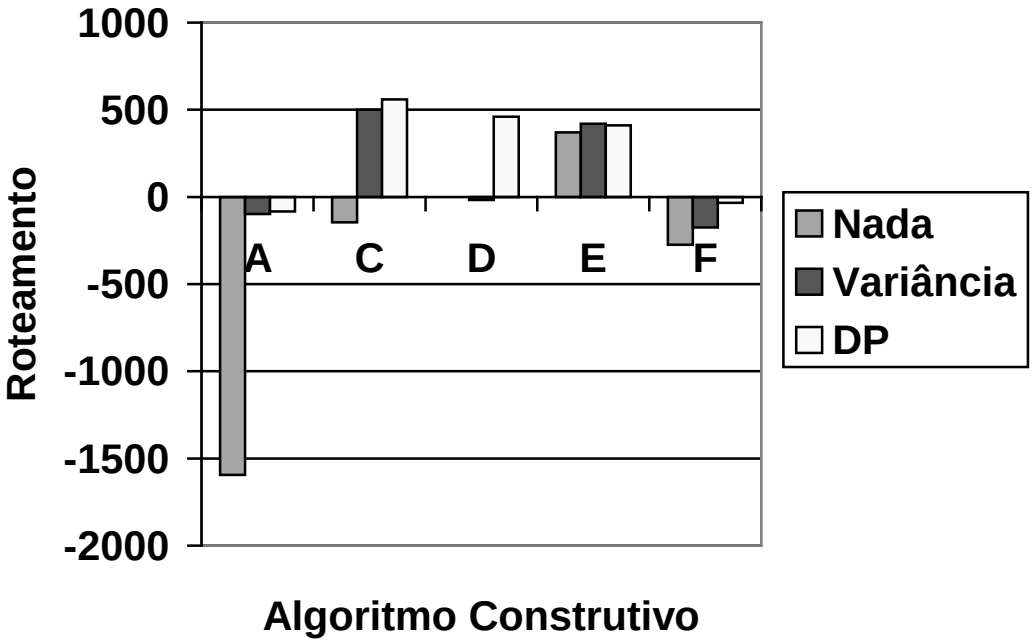

FIGURA 75 – Comparação da roteabilidade dos algoritmos construtivos com e sem pós-processamento no Tropic

Observa-se que a roteabilidade no Tropic da equalização por desvio padrão se sobressai a variância. Fica claro que a roteabilidade sempre melhora com pós-processamento. Outro fato interessante é que o algoritmo E sempre é superior aos demais, reforçando a idéia de que é melhor (em *wirelength*) para o roteador do Tropic.

8.3 Fluxo completo de posicionamento para *low-annealing*

Esta sessão apresenta resultados de execução de posicionamento usando o posicionamento inicial visto na sessão anterior seguido de *low annealing* (*Simulated Annealing* com baixas temperaturas iniciais). O objetivo de *low annealing* é manter o resultado obtido como solução inicial e trabalhar sobre ele, melhorando e escapando de mínimos locais.

Estudam-se, nesta sessão, alguns fluxos de execução. São feitas comparações dos algoritmos construtivos com *low annealing*. Depois é feita uma série de experimentos sobre o circuito C1908_3x3 (783 células) buscando a melhor quantidade de repetições das iterações. Inicia-se com 1/3 de repetições por célula (tabela 39), depois 1 repetição por célula (tabela 40), 3

repetições por célula (tabela 41) e por fim 10 repetições por célula (tabela 42). Será demonstrado como há um compromisso de tempo de execução com qualidade da solução.

Depois, será feita uma comparação, para o mesmo circuito, de possíveis temperaturas iniciais. A temperatura inicial tem um efeito diferente para cada algoritmo. No caso de algoritmos como a quadratura, onde o WL atingido é melhor, a temperatura inicial deve ser menor. Nos casos onde o algoritmo é semelhante ao aleatório, a temperatura inicial deve ser mais alta.

Por fim, serão feitos alguns testes para diferentes circuitos, mantendo a mesma temperatura inicial e algoritmo para posicionamento inicial.

8.3.1 *Low-Annealing* variando as repetições por iteração

Quatro experimentos são realizados nesta sessão, variando apenas a quantidade de repetições de cada iteração. O número de repetições, porém, é dado como uma função do número de células no circuito, de forma que se crie uma metodologia independente do circuito. O experimentos feitos são:

a) *Low-Annealing* com 1/3 de repetição por célula
b) *Low-Annealing* com 1 repetição por célula
c) *Low-Annealing* com 3 repetições por célula
d) *Low-Annealing* com 10 repetições por célula

Os resultados encontram-se, respectivamente, nas tabelas 39 até 42.

TABELA 38 – Fluxo de posicionamento terminando em *Low Annealing* com 1/3 de repetições por célula

Alg.	WL	CPU	Cong.	Roteamento com RotDL			
				2	3	4	5
A	210503	48	44	85%	97%	98%	98%
B	210545	53	46	87%	97%	98%	98%
C	194127	49	32	89%	98%	98%	98%
D	196363	49	37	92%	99%	99%	99%
E	210730	49	36	83%	94%	96%	96%

TABELA 39 – Fluxo de posicionamento terminando em *Low Annealing* com 1 de repetição por célula

Alg.	WL	CPU	Cong.	Roteamento com RotDL			
				2	3	4	5
A	182193	141	31	92%	97%	98%	98%
B	188717	153	33	90%	97%	99%	99%
C	173549	142	28	96%	99%	99%	99%
D	171731	137	26	94%	98%	98%	98%
E	180205	161	31	93%	98%	98%	98%

TABELA 40 – Fluxo de posicionamento terminando em *Low Annealing* com 3 de repetições por célula

Alg.	WL	CPU	Cong.	Roteamento com RotDL			
				2	3	4	5
A	175832	426	32	96%	99%	99%	99%
B	173640	424	27	96%	99%	99%	99%
C	170318	409	27	95%	98%	99%	99%
D	165454	458	27	95%	99%	99%	99%
E	171990	394	26	93%	97%	98%	98%

TABELA 41 – Fluxo de posicionamento terminando em *Low Annealing* com 10 de repetições por célula

Alg.	WL	CPU	Cong.	Roteamento com RotDL			
				2	3	4	5
A	172234	1352	34	85%	98%	98%	98%
B	166213	1288	27	94%	99%	99%	99%
C	163479	1335	27	96%	98%	99%	99%
D	162008	1365	36	94%	99%	99%	99%
E	159466	1348	29	92%	98%	98%	98%

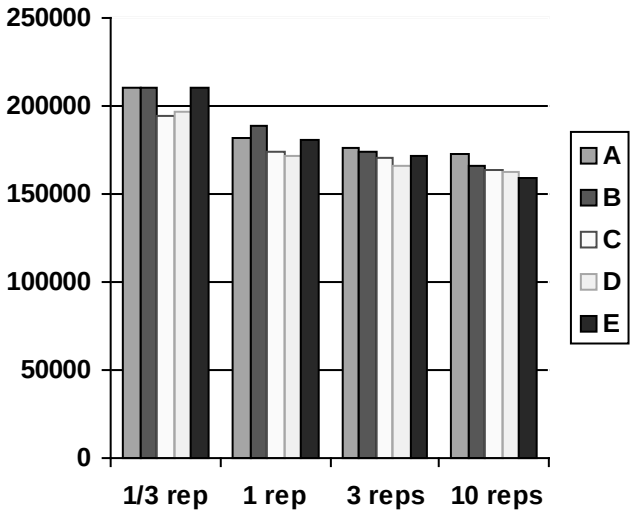

FIGURA 76 – Comparação do número de repetições executadas em *Low Annealing*

Observe que o custo final em *wirelength* diminui com o aumento das iterações. Porém, a diferença diminui- gradualmente, sendo que há pouco progresso de 3 para 10 repetições. Observando as tabelas 41 e 42, nota-se que a roteabilidade inclusive diminui. Isto não é uma regra. Aconteceu simplesmente pelo caráter não determinístico do algoritmo.

Observe também que o algoritmo C (CGC20) é o melhor para 1/3 repetições por célula e pouco evolui com o aumento de 1 repetição para 10 repetições. Isto sugere que, quanto melhor for o algoritmo de posicionamento inicial, menor é a necessidade por iterações de *low annealing*.

8.3.2 *Low-Annealing* variando a temperatura inicial

Quatro experimentos são realizados nesta sessão:
a) *Low-Annealing* com probabilidade inicial de 0,2 (que é alta)
b) *Low-Annealing* com probabilidade inicial de 0,02
c) *Low-Annealing* com probabilidade inicial de 0,002
d) *Low-Annealing* com probabilidade inicial de 0,0002

Usam-se três repetições por célula. Os dados para probabilidade 0,02 encontram-se na tabela 41, vista na sessão anterior. Os demais dados encontram-se, respectivamente, nas tabelas 43,44, e 45.

TABELA 42 – Fluxo de posicionamento terminando em *Low Annealing* com probabilidade inicial de 0,2

Alg.	WL	Cong.	Roteamento com RotDL			
			2	3	4	5
A	178101	28	92%	99%	99%	99%
B	174313	33	96%	98%	98%	98%
C	182262	33	91%	98%	98%	98%
D	169139	29	95%	98%	98%	98%
E	172260	25	94%	98%	99%	99%

TABELA 43 – Fluxo de posicionamento terminando em *Low Annealing* com probabilidade inicial de 0,002

Alg.	WL	Cong.	Roteamento com RotDL			
			2	3	4	5
A	184934	30	93%	98%	98%	98%
B	180861	30	95%	98%	99%	99%
C	162768	24	94%	97%	98%	98%
D	176054	32	95%	99%	99%	99%
E	165684	29	95%	98%	99%	99%

TABELA 44 – Fluxo de posicionamento terminando em *Low Annealing* com probabilidade inicial de 0,0002

Alg.	WL	Cong.	Roteamento com RotDL			
			2	3	4	5
A	178597	30	92%	99%	99%	99%
B	175181	28	91%	98%	98%	98%
C	170278	30	96%	98%	98%	98%
D	168277	27	95%	98%	98%	98%
E	169778	31	97%	98%	99%	99%

Os dados de roteamento com 2 níveis de metal das tabelas estão reproduzidos no gráfico da figura 77. Conforme diminui-se a temperatura inicial, o algoritmo está mais sujeito a ficar preso em mínimos locais.

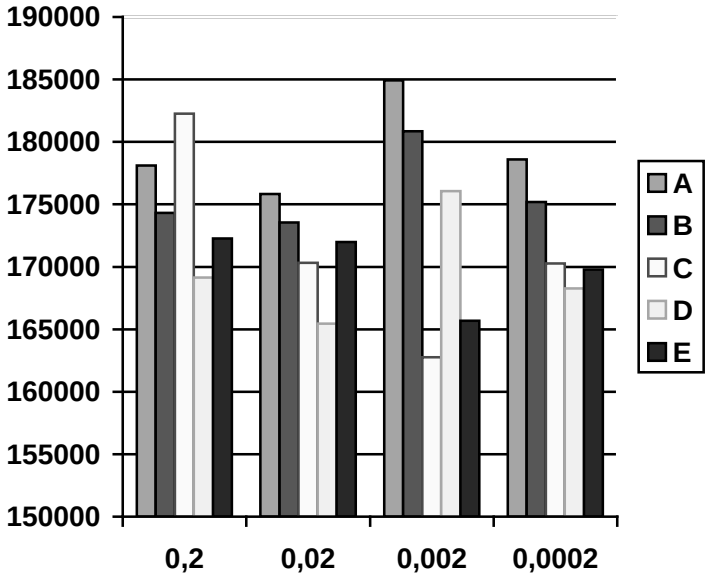

FIGURA 77 – Roteabilidade de *Low Annealing* variando a probabilidade de aceitação inicial

Observa-se no gráfico, porém, que o efeito final no *wirelength* não é totalmente determinado pela temperatura no circuito C1908. Há uma forte dependência do algoritmo de posicionamento inicial. A partir dos dados, observa-se que a probabilidade de 0,2 não é adequada para *Low Annealing*, por ser muito alta. O resultado final foi melhor que o inicial, porém percebe-se pelas outras probabilidades, que o resultado poderia ser melhor usando temperatura mais baixa. A probabilidade de 0,02 ainda é alta para os circuitos com melhor posicionamento inicial. Porém os circuitos com pior posicionamento inicial, gerados pelos algoritmos A e B, tem a sua melhor otimização (em relação as outras probabilidades). Isto significa que a probabilidade de 0,02 ainda foi suficiente para quebrar a estrutura inicial do posicionamento. A probabilidade de 0,002 apresenta resultados ruins para os circuitos menos otimizados e resultados ótimos para os circuito mais otimizados (com exceção da quadratura). Em outras palavras, uma probabilidade de 0,002 é muito baixa para recuperar um posicionamento inicial ruim, mas é suficientemente baixa para melhorar um bom posicionamento inicial. No caso da probabilidade de 0,0002, há uma vantagem razoável dos circuitos otimizados também, porém percebe-se que a temperatura é baixa demais, prendendo-se com mais facilidade em mínimos locais.

Ao analisar a curva de variação do custo (figura 78) de cada um dos métodos descritos acima, observa-se uma convergência gulosa dos algoritmos com baixa temperatura, enquanto que a probabilidade inicial de 0,02 mostra uma variação de custo sempre descendente.

A figura 78 mostra graficamente a evolução do custo de cada uma das execuções usando o algoritmo aleatório como inicial. Observe como a execução com probabilidade inicial de 0,2 começa aumentando o custo inicial, mas consegue convergir adequadamente no final. Quanto menor a temperatura inicial, maior é a convergência, aproximando-se de um algoritmo guloso. As tabelas 43-45 mostram, porém, que o caráter guloso implica, nestes casos, em mínimos locais, encontrando uma solução final inferior .

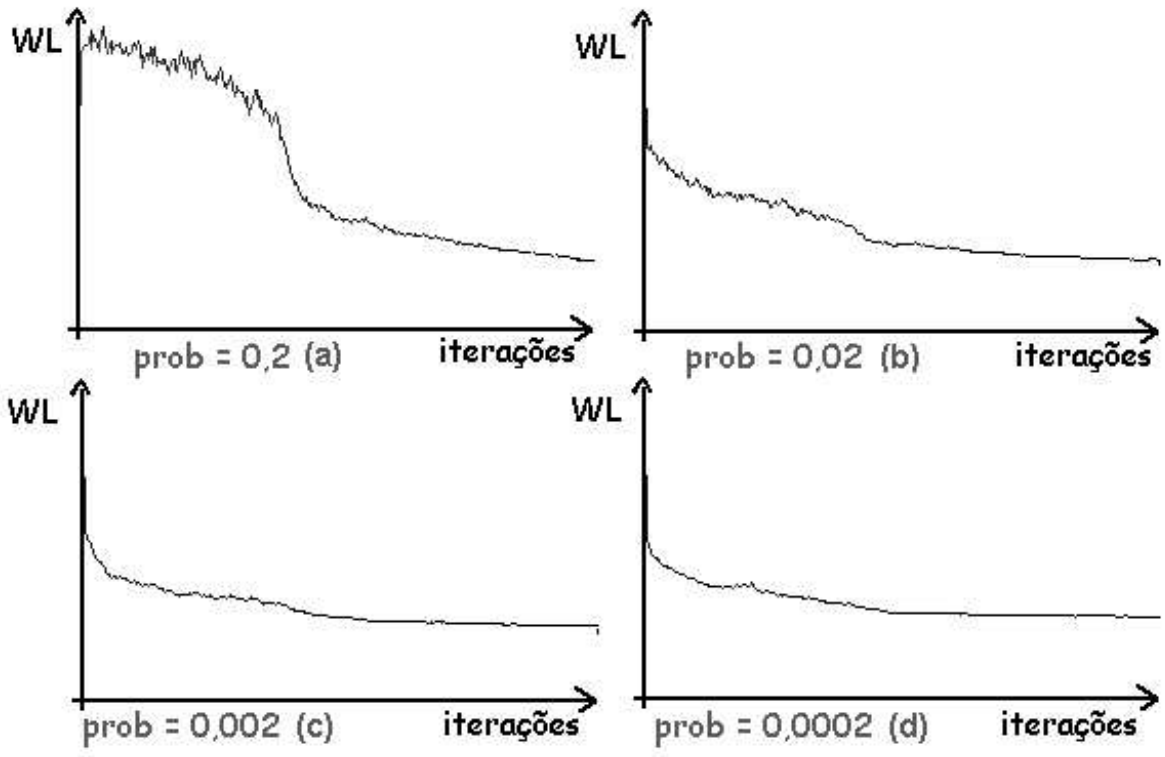

FIGURA 78 – Variação do custo de Low Annealing do algoritmo Aleatório

Observe agora o gráfico da figura 79. Ele corresponde a variação do custo da execução do *Simulated Annealing* com probabilidade inicial 0.2 (como na figura 78.a). Porém, usa-se o

algoritmo Plic-Plac como inicial. Compare com a figura 78.a. Repare como o custo inicial sobe mais no Plic-Plac, já que este algoritmo gera WLs melhores. Este experimento foi repetido para outros posicionamentos iniciais. Observou-se que, quanto melhor é o posicionamento inicial, maior é a elevação inicial da curva de custo. Portanto, a probabilidade inicial deve ser proporcional ao WL da solução inicial. Desenvolver um método automático para este procedimento é proposto como uma modificação futura, na sessão 6.2.4.

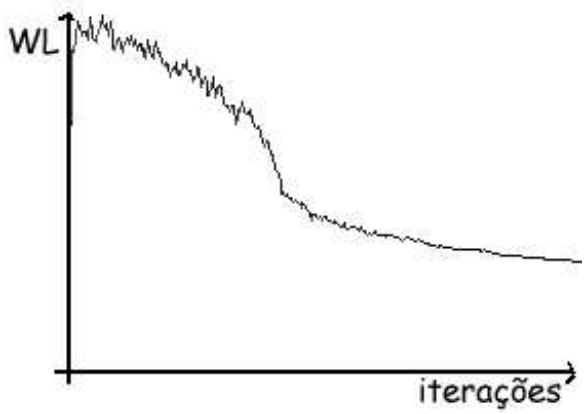

FIGURA 79 - Variação do custo de *Low Annealing* (0,2) do algoritmo Plic-Plac

8.3.3 Resultados gerais

Os próximos dados referem-se à execução de *Low Annealing* sobre diversos circuitos *benchmarks* com um algoritmo de posicionamento construtivo único. Usa-se quadratura, seguida de pós-processamento para equalizar o tamanho das bandas. Por fim, usa-se Simulação de Têmpera, com probabilidade inicial de 0,02 e variação de temperatura em curva com ponto de inflexão em (0,4*num_iteracoes; 0,4*temperatura_inicial). O *software* Tropic é utilizado para geração do leiaute e roteamento. Diferentemente das tabelas anteriores desta sessão, a tabela 46 apresenta dados de circuitos modificados para o Tropic. Basicamente, é feito um mapeamento do circuito para um nível de hierarquia. Outra diferença é na estimativa de altura das bandas, que é maior devido aos canais de roteamento.

TABELA 45 – *Low Annealing* de posicionamento com Quadratura de circuitos mapeados para o Tropic

Benchmark	Num Cells	WL	CPU (s)	Cong.	Densidade Tropic
C499	364	89344	161	31	68K
Bw	468	94829	219	27	63K
C1908	783	207652	588	39	58K
Mult	1408	577351	1625	49	63K
Alu4_4x4	2523	3811192	6142	265	Ñ (1565)
C53	3249	986913	7318	154	43K
Alu4_2x2	4844	4837485	18056	183	Ñ (2326)

Para melhor compreensão, os dados de roteamento (densidade de área final) são colocados no gráfico da figura 80.

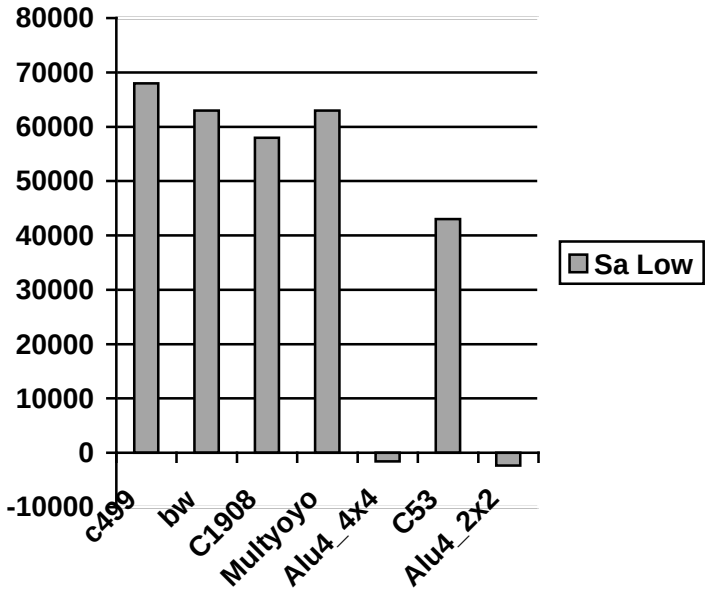

FIGURA 80 – Densidade de leiaute depois de *Low Annealing* (transistores por milímetro quadrado)

Os dados de roteamento de *low annealing* mostram que o *Simulated Annealing* melhorou significativamente os resultados apresentados nas sessões anteriores, onde havia somente um posicionamento construtivo (veja tabelas da sessão 8.1). Basicamente, esta informação mostra que o posicionamento iterativo executado pelo Simulated Annealing foi capaz de melhorar, de fato, o comprimento total de conexões e a roteabilidade do circuito.

É claro, pela figura 80, que a tendência natural é que circuitos mais complexos tenham menor densidade, devido ao aumento da necessidade de conexões. A exceção é o circuito Mult, que trata-se de um multiplicador e, por isto, é mais regular que o bloco de lógica aleatória C1908. Por ser mais regular, é mais simples de posicionar. Outra exceção é o bloco C53, que é maior do que o Alu4_4x4, porém menos complexo. O resultado decrescente de densidade indica a tendência de que espaços para roteamento são cada vez mais necessários. Porém, o fenômeno que ocorre é inverso. Nas tecnologias modernas, há uma diminuição dos espaços de roteamento horizontal e vertical. Ao mesmo tempo, há um aumento de número de camadas de metal. Porém, como foi visto em outras tabelas, o aumento do número de níveis de metal tem um limite de saturação. Por esta razão, os algoritmos de posicionamento são cada vez mais exigidos e precisam de constantes atualizações.

8.4 Fluxo completo de posicionamento para *high-annealing*

O processo de *High Annealing* é diferente do *Low Annealing* por ser totalmente independente da solução inicial. O processo inicia em altas temperaturas, fazendo com que a solução inicial seja totalmente destruída. A tabela 47 mostra alguns resultados de simulação usando *High Annealing* com 400 iterações e 2 repetições por célula. A variação da temperatura também é alterada. O ponto de inflexão passa para (0,22*num_iterações ; 0,22* temperatura_inicial), conforme mostra a figura 82.

TABELA 46 – High Annealing em diferentes posicionamentos iniciais

Alg.	WL	CPU	Cong.	Roteamento com RotDL			
				2	3	4	5
A	177835	385	31	93%	98%	98%	98%
B	169404	386	26	95%	98%	98%	98%
C	172780	370	27	93%	98%	98%	98%
D	169514	390	27	95%	99%	99%	99%
E	178256	393	31	93%	99%	99%	99%

A figura 81 mostra a variação do custo de todas as soluções. Repare como o custo inicial cresce demasiadamente, mostrando que a solução inicial é totalmente descartada.

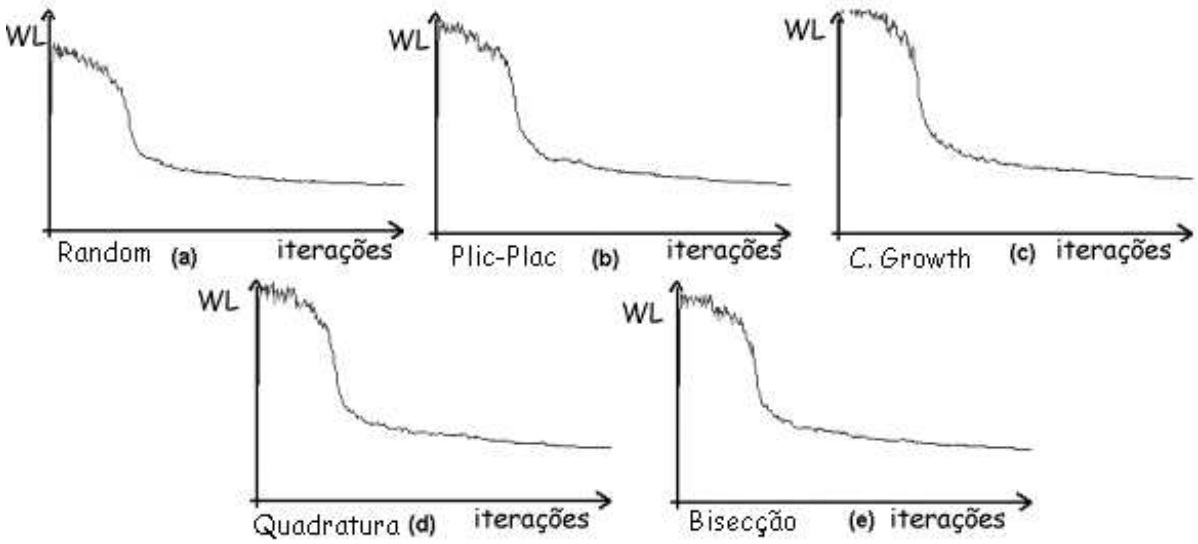

FIGURA 81 – Variação do custo no processo de *High Annealing*

Uma característica muito interessante do *High Annealing* é como a variação do custo acompanha a variação da temperatura. Observe a figura 82, que mostra alguns exemplos para o circuito C1908, posicionado inicialmente com o Plic-Plac.

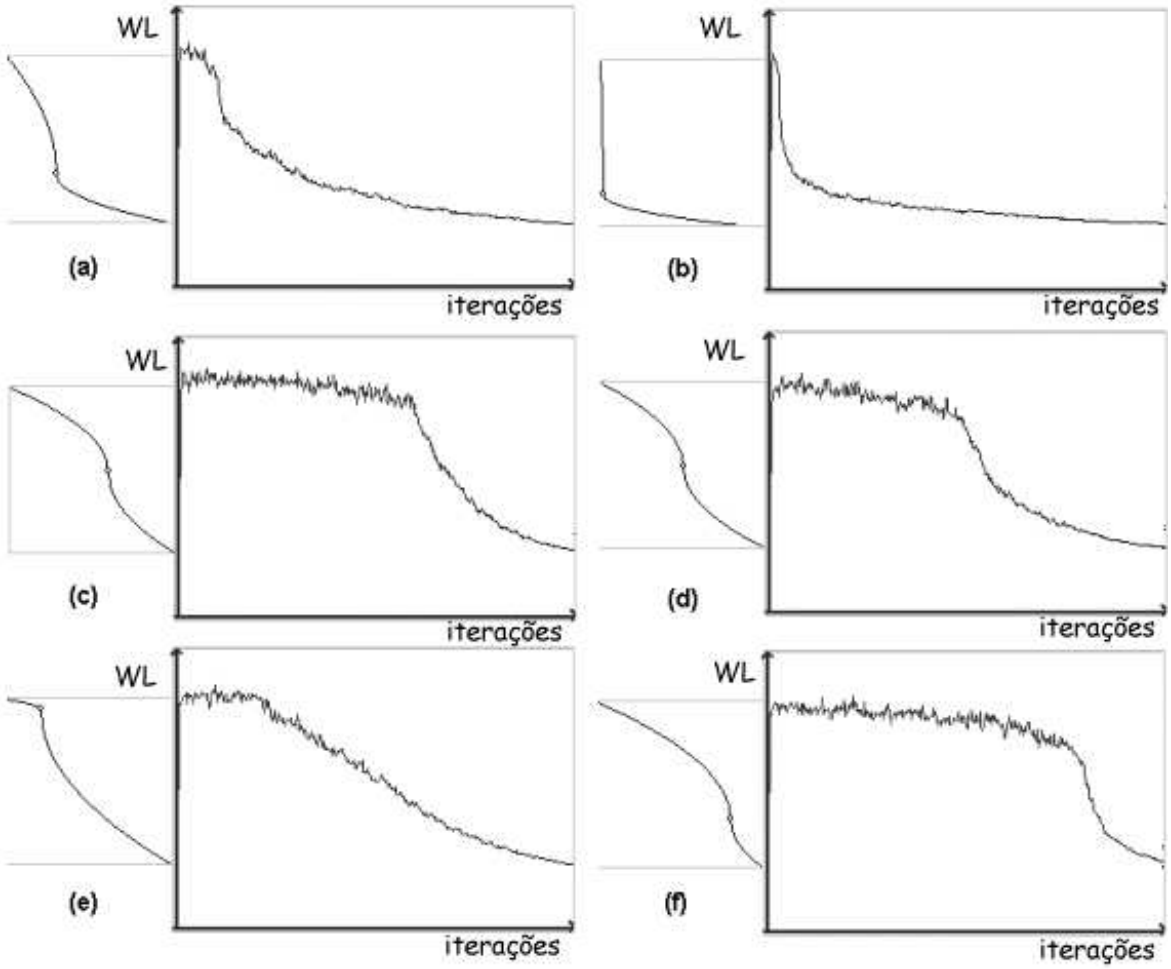

FIGURA 82 – Curva de variação do custo acompanha curva de variação da temperatura, em High Annealing

As altas temperaturas têm a função de tornar o algoritmo independente da solução inicial, fugindo de qualquer eventual mínimo local. Observando as figuras, nota-se que ficar muito tempo em altas temperaturas, porém, é inútil, pois o algoritmo não consegue melhorar a solução inicial. Assim, o escalonamento (*schedule*) de *High Annealing* deve permanecer pouco tempo nas mais altas temperaturas. Na figura 82 *c*,*d*,*f* o algoritmo perde muito tempo estragando a solução inicial. As soluções *a*,*b*,*e* aproveitam mais a quantidade de iterações determinada para melhorar a solução.

Foram coletados três conjuntos de dados para *High Annealing*. O primeiro usa 1 repetição por célula. Os segundo 3 e o terceiro 10 (todas usam 400 iterações).

TABELA 47 – Fluxo de posicionamento terminando em High Annealing (1 repetição por célula) para circuitos mapeados para o Tropic

Benchmark.	WL	Cong.	Densidade Tropic
C1908	218297	35	57K
Alu4_4x4	3834991	237	Ñ (1720)
Alu4_2x2	6267722	257	Ñ (5244)
C499	119259	47	62K
C53	1207520	146	40K
Mult	795649	82	-
Bw	104256	30	59K

TABELA 48 – Fluxo de posicionamento terminando em High Annealing (3 repetições por célula) para circuitos mapeados para o Tropic

Benchmark.	WL	Cong.	Densidade Tropic
C1908	198976	35	57K
Alu4_4x4	3637810	262	Ñ (1093)
Alu4_2x2	4659951	187	Ñ (1311)
C499	87590	32	67K
C53	986373	133	49K
Mult	529440	46	62K
Bw	95376	32	63K

TABELA 49 – Fluxo de posicionamento terminando em High Annealing (10 repetições por célula) para circuitos mapeados para o Tropic

Benchmark.	WL	Cong.	Densidade Tropic
C1908	168315	22	61K
Alu4_4x4	3276835	237	-
Alu4_2x2	4735165	210	Ñ(826)
C499	112494	47	74K
C53	898533	142	49,4K
Mult	526470	42	62,2K
Bw	95376	32	68,5K

Os resultados mostram que, como já era esperado, a roteabilidade (densidade) melhora conforme aumentam-se o número de iterações. Observa-se também que a complexidade e o tamanho do circuito prejudicam o desempenho dos algoritmos de posicionamento.

8.5 Comparações com o Posicionador do Tropic

Este trabalho valida a qualidade de seus posicionamentos através de uma comparação com o posicionador do Tropic. Moraes, em 1999, explica detalhadamente como funciona o posicionador. Trata-se de uma implementação de particionamento em quadratura, usando *terminal propagation* (sessão 5.4.3). Não há nenhuma espécie de refinamento iterativo, porém. Os tempos de execução serão, portanto, ordens de grandeza menores que execuções de *Simulated Annealing*, o que é certamente uma vantagem significativa. Não faz sentido, porém, comparar o tempo de execução de ambas estratégias, pois não rodam na mesma CPU (Mango Parrot roda em um Pentium IV 1.8GHz, Tropic numa Ultra Sparc 10).

Foram comparados os seguintes parâmetros:

- Dimensões do bloco - XxY (em μm)
- Área do Bloco – Área (em μm^2)
- Densidade - D(em transistores por milímetro quadrado)
- Comprimento Médio das Conexões – C (em μm)
- Porcentagem de conexões menores que 100 μm - % 0-100
- Porcentagem de conexões menores que 200 μm e maiores que 100 μm - % 100-200
- Número de Bandas

Todas as métricas avaliam a qualidade do posicionamento, em diferentes perspectivas. Tanto área, quanto dimensões e densidade são medidas de roteabilidade, pois estão relacionadas com a demanda por recursos de roteamento. Comprimento Médio das Conexões também é uma

métrica de roteabilidade com relação indireta a *timing* e dissipação de potência. A porcentagem de conexões menores que 200μm está relacionada com a proximidade de células conectadas, qualificando adequadamente o posicionamento, e analisando o *timing* do circuito devido às conexões, uma vez que conexões longas demais podem significar muito atraso para as conexões críticas do circuito.

A tabela 51 mostra os resultados de execução em alguns circuitos.

TABELA 50 – Comparação do Simulated Annealing com o Posicionador do Tropic

Bench	Plac	Area	XxY	C	%0-100M	%100-200M	D	Rows
C1908_3x3	Low 3	0,04228	259,7 x 162,8	59,92	89,1%	9,9%	57.9K	9
	High 1	0,04316	258,45 x 167	61,78	87,6%	10,8%	57K	9
	High 9	0,04030	259 x 155	53,91	92,8%	6,8%	61K	9
	Tropic	0,04252	144,25 x 294,80	65,92	83,7%	13,4%	57.8K	16
	Tropic	0,04556	258,00 x 176,60	61,69	86,5%	12,3%	53,9K	9
Mult	Low 3	0,13550	433,45 x 312,60	71,82	88,5%	7,7%	63,3K	16
	High 3	0,13709	433,00 x 316,60	69,91	89,9%	6,5%	62,6K	16
	High 9	0,13795	433,00 x 318,60	71.08	90,0%	6,2%	62,2K	16
	Tropic	0,14757	432,00 x 341,60	80,59	79,0%	15,6%	58,1K	16
Bw	Low 3	0,01972	168,25 x 117,20	48,22	91,8%	5,5%	63,6K	7
	High 1	0,02122	169,45 x 125,20	54,67	91,6%	4,0%	59,2K	7
	High 9	0,01833	169,45 x 108,20	45,60	93,7%	4,0%	68,5K	7
	Tropic	0,01874	106,70 x 175,60	49,08	89,9%	6,1%	67,0K	11
	Tropic	0,01993	168,60 x 118,20	47,98	91,6%	4,6%	63,0K	7
C499	Low 3	0,02281	193,00 x 118,20	58,14	84,7%	11,5%	68,2K	7
	High 1	0,02481	193,50 x 128,20	66,12	84,3%	9,4%	62,7K	7
	High 9	0,02102	192,45 x 109,20	54,03	87,9%	8,8%	74,0K	7
	Tropic	0,02279	194,45 x 117,20	57,62	85,5%	11.9%	68,2K	7
C53	Low 3	0,24761	530,45 x 466,80	86,11	85,6%	5,6%	43,0K	18
	High 1	0,26492	533,25 x 496,80	101,87	81,2%	7,5%	40,2K	18
	High 9	0,21566	531,45 x 405,80	73,07	88,3%	4,9%	49,4K	18
	Tropic	0,26206	530,70 x 493,80	103,09	81,1%	8,1%	40,6K	18
C432_4x4	Low 3	Feeds Not Routed						8
	High 1	0,01088	76,00 x 143,20	67,19	78,7%	15,5%	63,5K	8
	High 9	0,01006	77,25 x 130,20	58,15	83,3%	15,5%	68,8K	8
	Tropic	0,01115	101,70 x 109,60	66,11	79,5%	18,4%	62,0K	6
	Tropic	Feeds Not Routed						8

Considere, inicialmente, a porcentagem de conexões menores que 200μm como métrica para qualidade do posicionamento. O gráfico da figura 83 mostra as porcentagens para os circuitos da tabela 51 ordenados por complexidade. No caso do posicionador do Tropic, usa-se uma solução com um número de bandas igual ao da solução com a ferramenta do *Mango Parrot*.

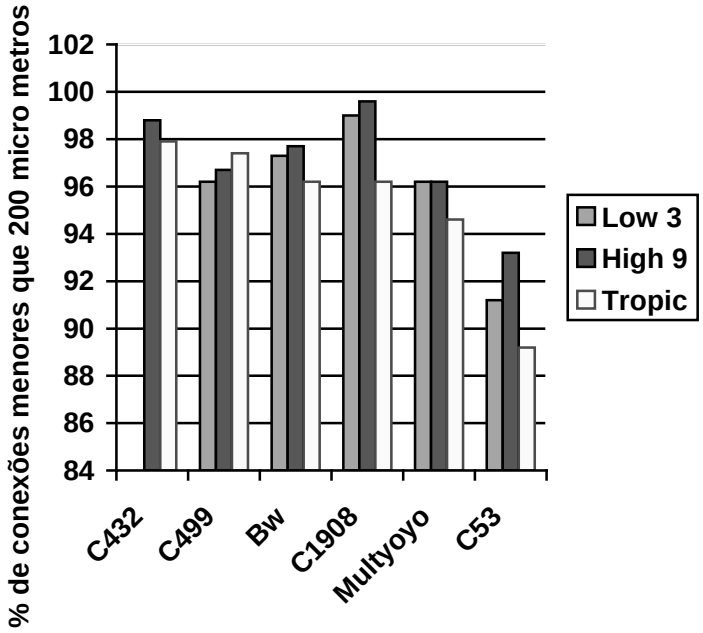

FIGURA 83 – Comparação da roteabilidade (% de conexões menores que 200μm) no Tropic de 3 algoritmos de posicionamento

Observando isoladamente o posicionamento do Tropic, observe que o crescimento da complexidade do circuito diminui a qualidade do posicionamento. Isto demonstra que o particionamento em quadratura implementado no Tropic começa a apresentar problemas conforme cresce o circuito. A quadratura é uma heurística incompleta, pois um número excessivo de particionamentos pode afastar demasiadamente elementos que devem se conectar. Assim, conforme cresce o tamanho do circuito, a quadratura começa a ter dificuldades, pois necessita de mais partições.

Por outro lado, o *Simulated Annealing* é mais regular, dado o tempo necessário, apresentando um forte declínio somente no circuito C53, que é bastante complexo, além de grande. Observe como o desempenho do Mult é comparável com o C499.

O gráfico da figura 84 foca em outra métrica de avaliação do posicionamento: a densidade os circuitos.

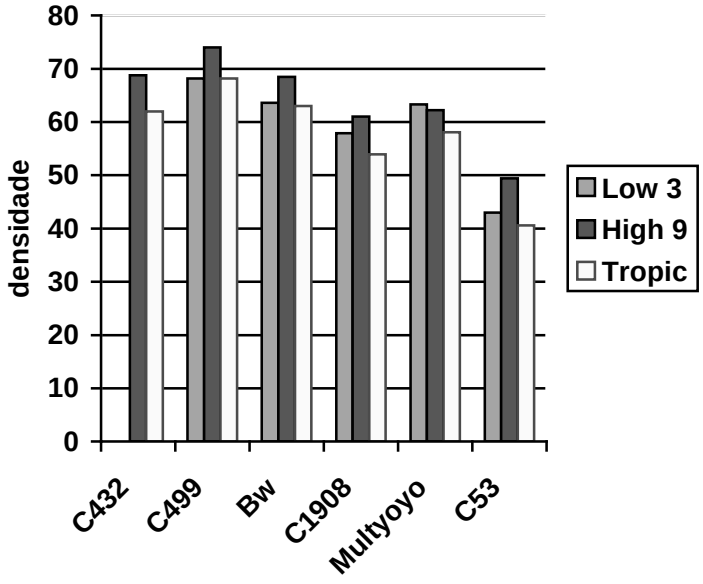

FIGURA 84 – Comparação da roteabilidade (densidade do leiaute) no Tropic de 3 algoritmos de posicionamento

Observa-se que o posicionador do Tropic é mais regular visto por esta métrica, mas no caso do maior circuito, o C53, há uma queda significativa de densidade. O gráfico deixa muito claro que o *Simulated Annealing* é sempre superior em *wirelength* ao algoritmo de posicionamento usado no Tropic, a não ser no caso do circuito C432 usando o SA Low3, que não pode ser roteado. Por outro lado, sabe-se que o algoritmo de posicionamento do Tropic é sempre superior ao *Simulated Annealing* em relação a tempo de CPU.

O gráfico da figura 85 reproduz o gráfico anterior, apresentando a taxa de ganho (em *wirelength*) do *Simulated Annealing* em relação ao Tropic.

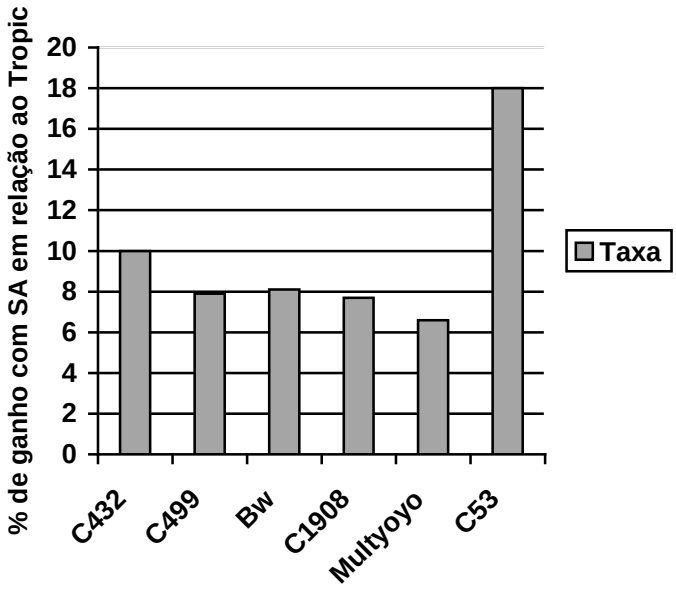

FIGURA 85 – Porcentagem de ganho do Simulated Annealing (usando high annealing) em relação ao posicionador do Tropic

Observa-se que o ganho do *Simulated Annealing* em relação do Posicionador do Tropic fica próximo de 10% (um pouco abaixo) para os *benchmarks* mais simples, enquanto que no C53 atinge 18%. A tendência, como foi visto, é que o ganho seja cada vez maior.

A tabela 52 compara o tempo do *Simulated Annealing* rodando em um Pentium IV 1,8GHz contra o Tropic rodando em uma SUN UltraSparc 10. A comparação é bastante imprecisa. É só para ter uma idéia das ordens de grandeza existente entre os algoritmos.

TABELA 51 – Comparação do tempo de execução do Tropic com o Simulated Annealing

Circuito	Low Annealing com 300 iterações e 3 repetições por célula	High Annealing com 400 iterações e 9 repetições por célula	Tropic (tempo total de toda a síntese)
C499	3 min (180s)	10 min (600s)	1 segundo
C53	2 horas e 10 min (7800s)	8 horas e 30 min (108000s)	11 segundos

Em uma análise geral dos gráficos, o *Simulated Annealing* mostra uma vantagem na qualidade do posicionamento em relação ao Tropic. Ou seja, a custa de maior tempo (ordens de grandeza – veja tabela 52) de processamento, obtém-se um resultado de posicionamento melhor. Observe que o *Simulated Annealing* propicia que o desempenho seja ainda melhor, bastando aumentar o número de repetições. Dentre as duas técnicas de *Simulated Annealing*, observa-se que, obviamente, o *High Annealing* possui um desempenho melhor as custas de mais iterações (400 iterações com 9 repetições por célula no *High Annealing*, 300 iterações e 3 repetições por célula para low). Porém, os dados do *Low Annealing* são interessantes por mostrar que é possível gastar menos tempo com *Simulated Annealing* com resultados ainda bons, melhores em média que o Posicionador em Quadratura do Tropic.

Outra observação relevante é em relação ao ótimo desempenho do posicionador do Tropic comparado ao posicionador em Quadratura implementado na ferramenta do *Mango Parrot*. Como ambos usam o mesmo algoritmo para particionamento, existem duas diferenças significativas: as errôneas estimativas de altura e o t*erminal propagation*.

Quanto à vantagem do *Simulated Annealing* com relação ao Posicionamento em Quadratura, faz-se a seguinte pergunta, a ser respondida em trabalhos futuros: A vantagem se deve por uma limitação do Algoritmo de Quadratura ou por uma limitação da Heurística de Particionamento utilizada (Fidduccia-Mateyeses)? Será que melhores algoritmos para particionamento não iriam propiciar um melhor desempenho da Quadratura?

9 Conclusões

Este trabalho fez uma análise ampla sobre os algoritmos de todos os algoritmos existentes na literatura clássica (Sherwani, Sait) foram abordados. Muitos artigos foram considerados, com propostas de novos modelos e algoritmos. Cabe uma pesquisa ainda mais profunda para uma comparação completa de algoritmos de posicionamento. Porém, este texto apresenta como importante contribuição a comparação, em um mesmo sistema de síntese, dos algoritmos mais conhecidos e utilizados. A análise é bastante precisa e baseada em dados reais.

O texto é voltado para o desenvolvedor de CAD. Assim, a maior parte dos algoritmos, ao contrário do que costuma encontrar-se na literatura, é explicada com detalhes de implementação, de forma que não fiquem questões em aberto. Isto só foi possível pela forte base de implementação por trás deste texto. O algoritmo de Fiduccia-Matteyeses, por exemplo, é um algoritmo complexo e por isto foi explicado em detalhe, para que o leitor seja capaz de implementar sem a necessidade de leituras adicionais.

Assim como uma revisão de técnicas conhecidas e publicadas, este trabalho oferece algumas inovações no fluxo de posicionamento. Propõe-se um novo algoritmo para posicionamento inicial (Plic-Plac), que tem uma ótima relação de custo de roteabilidade por tempo gasto. Também é proposta uma variação inédita do *Cluster Growth* que mostra ótimos resultados. O algoritmo de *Simulated Annealing* é bastante trabalhado, oferecendo uma série de inovações: cálculo automático da temperatura inicial, funções de perturbação gulosas (direcionadas a força), combinação de funções de perturbação atingindo melhores resultados, otimização no cálculo de tamanho dos fios (avaliação somente das redes modificadas e aproveitamento de cálculos anteriores). Todas estas modificações propiciam uma maior velocidade e convergência do método de Simulated Annealing.

O capítulo 8 mostra que os algoritmos construtivos abordados (incluindo o posicionador do Tropic, baseado em quadratura com t*erminal propagation*) apresentam um resultado pior que o *Simulated Annealing* em termos de qualidade de posicionamento, as custas de um longo tempo de CPU. Porém, o uso de técnicas propostas neste trabalho, em conjunto com outras técnicas propostas em outros trabalhos (Su 2001, por exemplo) pode acelerar o SA, de forma que a relação qualidade/tempo aumente.

Assim, depois de todas as implementações realizadas, diversas outras conclusões puderam ser tiradas a partir dos experimentos realizados:

Passagem de Posicionamento Relativo para Absoluto:

No posicionamento relativo, trabalha-se somente com a ordem das células. Assim, cada banda tem uma lista ordenada de células. Passar para o posicionamento absoluto significa escolher uma posição (X,Y) para cada célula. Três estratégias foram estudadas neste trabalho: Alinhar a Esquerda, Alinhar a Direita, Centralizar, Justificar. Foi verificado que Centralizar é a técnica com menor soma total de fios, o que implica em maior roteabilidade, menor consumo de potência e maior freqüência de operação.

Estimativas de tamanhos dos fios

Foram avaliadas algumas técnicas para estimar o tamanho dos fios. Foi verificado que o semiperímetro e a Origem-Para-Destinos são os dois métodos mais rápidos e com boa aproximação. Nas meta-heurísticas, é preciso que o tamanho dos fios seja medido por um método rápido, já que este cálculo é repetido milhares ou até milhões de vezes.

Plic-Plac

É um algoritmo desenvolvido neste trabalho. Ele está em sua versão inicial, sendo que há muitas etapas a aprimorar. Primeiro, deve ser estudada qual a melhor técnica de combinação das

pesquisas contrárias. Depois, deve ser estudada uma maneira equilibrada de seleção de bandas, de forma que as células com ordens horizontais maiores não fiquem prejudicadas por erros acumulados. De qualquer forma, o algoritmo já apresentou alguns resultados interessantes, especialmente em tempo de execução. A qualidade do posicionamento é bastante interessante para circuitos pequenos (comparado com clássicos algoritmos, como Quadratura), e para os circuitos maiores é bastante eficiente comparado com o posicionamento aleatório (perdendo para os demais). Porém, observa-se a importância da descoberta deste algoritmo na comparação com o posicionamento aleatório, que é muito utilizado ainda hoje para posicionamento inicial devido à sua rápida resposta. O uso do algoritmo Plic-Plac é meramente para posicionamento de entrada de uma meta-heurística qualquer. Assim, o tempo de execução e memória gasta são mais importantes que a qualidade em si.

Algoritmo de Crescimento de Aglomerados

É um algoritmo bastante conhecido, e muito estudado. Este trabalho apresenta diversas variações sobre o mesmo algoritmo, com algumas idéias novas, como o cálculo da banda média de inserção das células. Mostra-se, por algumas simulações, que o Cluster Growth proposto (CGC) é a melhor técnica comparada com outros CGs pois gera circuitos com bandas bem equalizadas e de qualidade significativa.

Algoritmos Construtivos Baseados em Particionamento

Um algoritmo de posicionamento baseado em particionamento baseia-se em dois eixos: estratégia de corte e algoritmo de particionamento.

Entre as estratégias de corte, as melhores são aquelas que mantêm partições de tamanhos iguais. Duas foram selecionadas: a Quadratura e a Bisecção em Bandas. A primeira alivia o congestionamento no meio e se aplica bem para os casos onde não há canal de roteamento ou é fácil estimar seu tamanho. A Bisseção em Bandas é mais adequada para os casos onde os espaços entre as bandas são imprevisíveis. Ela minimiza o roteamento vertical, mas aumenta o roteamento horizontal.

O problema de particionamento é np-completo. As heurísticas para o tratar dividem-se em construtivas e iterativas. São comparadas duas heurísticas construtivas: Aleatória e Crescimento de Aglomerados. É visto que a heurística aleatória é ruim em termos de qualidade, mas deixa as partições equilibradas. O crescimento de aglomerados gera particionamentos melhores, porém as partições ficam desbalanceadas. Dos algoritmos iterativos estudados, o FM apresenta o melhor resultado, pois é o mais rápido e gera posicionamentos com menor WL. O FM caracteriza-se por fazer migração de grupos de uma célula somente, e tem um mecanismo de atualização de ganhos bastante eficiente. Por esta razão, ganha em eficiência dos algoritmos baseados em KL. Foi visto que a melhor combinação de particionamento construtivo com iterativo é o particionamento inicial aleatório seguido de FM. Neste trabalho, também, foi proposta uma pequena modificação no FM para a inclusão de *pads* fixos na periferia do circuito.

Foi proposta uma alteração para considerar os pinos de E/S, dando um ganho de 2% em média.

Simulated Annealing

Neste trabalho foi descrita, com detalhe, a implementação do *Simulated Annealing* na ferramenta do *Mango Parrot*. Esta implementação usa uma série de técnicas conhecidas na literatura, juntando com técnicas propostas neste trabalho.

Primeiro discutiram-se funções de perturbação. Foi mostrado que a função de perturbação dupla é mais rápida, porém está atrelada ao posicionamento inicial. Foram apresentadas duas novas funções de perturbação baseadas em Posicionamento Guiado A Forças. Elas são gulosas e aumentam a convergência do algoritmo. Podem, porém, implicar em mínimos locais para circuitos grandes (mais de 1000 células). Mostra-se, no entanto, que o uso combinado

das quatro funções de perturbação apresentadas é bastante eficaz e pode combinar as vantagens das quatro funções. Também, mostra-se que a simples combinação de perturbações simples FD e duplas FD melhoraram os resultados em 18%, apontando suas limitações quando usadas isoladamente, mas uma vantagem significativa na combinação delas.

A otimização da função de custo (apresentada na sessão 6.2.6.1) é outra otimização proposta sobre o *Simulated Annealing*, melhorando o seu desempenho em até 50% nos casos testados. Observa-se, no entanto, que a tendência é que o ganho aumente ainda mais para circuitos maiores, pois a otimização evita que haja uma varredura em todas as redes do circuito, fazendo com que avalie-se somente as redes modificadas pela perturbação.

Foi demonstrado um método para o cálculo da temperatura inicial para o *Simulated Annealing*. Ele é muito importante por tornar o algoritmo independente do circuito a posicionar. Foi visto, porém, que este método deve ainda ser aprimorado para ser independente do posicionamento inicial.

Foram vistas duas técnicas para escalonamento da temperatura: *high annealing* e *low annealing*. O *High Annealing* é a melhor estratégia se considerarmos recursos infinitos de tempo de CPU. Ele é praticamente imune a mínimos locais, pois inicia com uma temperatura suficientemente alta para quebrar qualquer estrutura inicial. Porém pode ser considerado um método burro, por não aproveitar um bom resultado inicial. O *low annealing* aproveita o resultado do posicionamento inicial, porém está menos apto a escapar de mínimos locais. Há, portanto, um compromisso de desempenho com tempo de CPU na escolha da temperatura (probabilidade) inicial.

Mostrou-se que para *high-annealing* o ideal é usar uma variação de temperatura de rápida convergência, de modo que não se perca tempo na etapa de "desmanche" da solução inicial. Porém, para *low-annealing*, é melhor que a variação da temperatura seja mais lenta, assemelhando-se a uma reta.

Comparação entre algoritmos construtivos

Foi visto no capítulo 8 que os algoritmos construtivos apresentados tem resultados bastante variados em termos de roteabilidade, tempo de processamento e consumo de memória. Os algoritmos Plic-Plac e Cluster Growth apresentam um tempo de execução baixíssimo, praticamente igual ao algoritmo aleatório. Porém, conforme cresce a complexidade dos circuitos, pior é o seu desempenho comparado a um algoritmo de Quadratura, por exemplo. Os algoritmos baseados em particionamento são os melhores em qualidade da solução (dado este reforçado pelo posicionador do Tropic, que é baseado em Quadratura), mas apresentam duas desvantagens: tempo de execução e principalmente consumo de memória. Em máquinas limitadas, como a máquina usada para as execuções desta dissertação (Pentium IV 1,8 GHz e 512Mb de RAM), os algoritmos baseados em particionamento necessitaram de uso intensivo de memória virtual, de forma que o tempo de CPU necessário aumentou drasticamente.

Pós-Processamentos

Este texto propõe uma série de possíveis pós-processamentos para os algoritmos de posicionamento. O primeiro método discutido é para melhorar a equalização do tamanho das bandas. Foi verificado que o desbalanceamento das bandas leva a problemas graves de rotealibidade. A técnica baseia-se em um cálculo de equilíbrio das bandas. Duas possibilidades são apontadas: variância do tamanho das bandas e desvio padrão do tamanho das bandas Foi visto que a variância força uma convergência brutal para um circuito totalmente equalizado, comprometendo um pouco o tamanho dos fios. Por outro lado, o desvio padrão melhora significativamente a equalização sem deixar de lado o tamanho dos fios. Foi visto, na sessão 8, que ambos os métodos melhoram a roteabilidade do circuito.

Outro método de pós-processamento visto é o uso do algoritmo Guloso depois de terminado o *Simulated Annealing*. Ele faz uma modificação fina no resultado final obtido. A sessão 6.2.7 mostra que ele é capaz de melhorar a solução final em todos os casos.

Congestionamento é fundamental.

Os resultados de roteamento usando o Fluxo de Síntese Automática, considerando que todas as conexões passarão sobre a área dos transistores, são insatisfatórios, já que não conseguiu-se rotear nenhum circuito por completo. O circuito trabalhado tem em torno de 700 células, o que não é considerado um circuito de grande complexidade. Porém, analisando o roteamento final, observou-se uma série de espaços vazios, enquanto que há áreas excessivamente congestionadas. Este estudo mostra a importância no controle do congestionamento para este tipo de leiaute. Para a síntese do Tropic não é tão relevante o estudo do congestionamento, pois pode-se alocar quanto espaço for desejado para canais de roteamento. É muito importante que haja uma distribuição equilibrada da demanda por conexões (principalmente quando a área por roteamento for fixa).

Seguindo-se no mesmo fluxo, observou-se que o aumento do número de níveis de metal aumenta a roteabilidade dos circuitos congestionados, pois propicia que algumas conexões consigam escapar das áreas muito concorridas por metais superiores. Porém, a roteabilidade aumenta pouco e atinge um limite de saturação em 4 ou 5 níveis livres para roteamento.

Desta forma, este trabalho reúne uma série de observações e comentários de técnicas de posicionamento, auxiliando um projetista de CAD no desenvolvimento de ferramentas de posicionamento e roteamento. A análise dos algoritmos de posicionamento permitiu pequenas modificações em praticamente todos eles, que melhoram seu desempenho de um modo geral. Como alterações mais importantes, aparecem as perturbações do Simulated Annealing, o novo algoritmo de Posicionamento (Plic-Plac) e os algoritmos de pós-processamento. Além disto, a análise de artigos permitiu o levantamento do estado da arte dos algoritmos, observando que os novos algoritmos de posicionamento devem preocupar-se com questões de tecnologia *deep-submicron*, como forte congestionamento, análise complexa de *timing*, *crosstalk* e dissipação de potência.

Os caminhos de continuidade deste trabalho são muito variados. Primeiro, ainda podem ser incluídos mais algoritmos nos dados comparativos. Os algoritmos baseados em Força não foram analisados a fundo, sendo que existe um universo de possíveis variações. Além da inclusão de outros algoritmos, os seguintes trabalhos são propostos:

- A técnica de *terminal propagation* é um método muito importante, e deve ser incluído nos algoritmos de posicionamento com base em particionamento.
- O *Simulated Annealing* implementado ainda pode ser otimizado. Uma série de trabalhos de possíveis implementações foram publicados e devem ser incluídos (por exemplo, Su 2001). Ainda, são tópicos novos de pesquisa: 1- Encontrar a probabilidade inicial do SA com base no WL da solução inicial. 2- Estudar a combinação das funções de perturbação de forma a encontrar as melhores probabilidades de execução delas.
- Fluxo de síntese física. O projeto FUCAS está em andamento, sendo que as ferramentas para *Floorplanning*, Particionamento do Problema, Posicionamento, Roteamento, Geração do Leiaute, *Timing*, Extração Elétrica, etc. estão em fase de implementação. São trabalhos futuros a integração de trabalhos, bem como distribuição na internet de um fluxo completo de síntese.

Referências

AARTS E.; LAARHOVEN P. A New Polynomial-Time Cooling Schedule. INTERNATIONAL CONFERENCE ON COMPUTER AIDED DESIGN, ICCAD, 1985. **Proceedings. . .** [S.l.: s.n.], 1985.

CALDWELL, A.; KAHNG, A.; MARKOV, I. Hypergraph Partitioning With Fixed Vertices. In: DESIGN AUTOMATION CONFERENCE, DAC, 36., 1999, New Orleans. **Procedings...** New York: ACM, 1999.

CALDWELL, A.; KAHNG, A.; KENNINGS, A.; MARKOV, I. Hypergraph Partitioning For VLSI CAD: Methodology for Heuristic Development Experimentation and Reporting. In: DESIGN AUTOMATION CONFERENCE, DAC, 36., 1999, New Orleans. **Procedings...** New York: ACM, 1999.

CALDWELL, A.; KAHNG, A.; MARKOV, I. Improved Algorithms for Hypergraph Bipartitioning. In: ASIAN-PACIFIC DESIGN AUTOMATION CONFERENCE, ASP-DAC, 2000. **Proceedings...** [S.l.: s.n.], 2000.

CALDWELL, A.; KAHNG, A.; MARKOV, I. Optimal Partitioners and End-Case Placers for Standard-Cell Layout. **IEEE Transactions on CAD of Integrated Circuits and Systems**, New York, v. 19, n. 11, Nov. 2000.

CALDWELL, A.; KAHNG, A.; MARKOV, I. Can Recursive Bisection Alone Produce Routable Placements?. In: DESIGN AUTOMATION CONFERENCE, DAC, 37., 2000, Los Angeles. **Proceedings...** New York: ACM, 2000.

CHEN, W; HSIEH, C; PEDRAM, M. Simultaneous Gate Sizing And Placement. **IEEE Transations on Computer Aided Design and Systems**, New York, v. 12, n. 12, p. 206-214, Feb. 2000.

CHOU, Yih-Chih and LIN, Yon-Long. Effective Enforcement of Path-Delay Constraints in Performance-Driven Placement. **IEEE Transactions on Computer Aided Design and Systems**, New York, v. 21, n. 1, p. 15-22, Jan. 2002.

GAJSKI, D. D. **Silicon Compilation**. Reading: Adison – Wesley, 1988.

GEREZ, S. H. **Algorithms for VLSI Design Automation.** Chichester: John Wiley, 1999.

GROVER, L. K. A New Simulated Anealing algorithm for Standard Cell Placement. In: DESIGN AUTOMATION CONFERENCE, DAC, 23., 1986. **Proceedings...** New York: IEEE, 1999.

GROVER, Lov K. Standard Cell Placement Using Simulated Sintering. In: DESIGN AUTOMATION CONFERENCE, DAC, 24., 1987. **Proceedings...** New York: IEEE, 1987.

GUNTZEL, J.; PINTO, A.C.; MORAES, F; REIS, R. An Improved Path Enumeration Method Considering Different Fall and Rise Gate Delays**.** In: BRAZILIAN SYMPOSIUM ON INTEGRATED CIRCUIT DESIGN, SBCCI, 1998. **Proceedings...** New York, IEEE, 1998.

GÜNTZEL, J.; REIS, R. **Functional Timing Analysis of VLSI Circuits Containing**

Complex Gates. 2000. Tese (Doutorado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

HART, P. E.; NILSSON, N. J.; RAPHAEL, B. A Formal Basis to the Heuristic Determination of Minimum Cost Paths. **IEEE Transactions on Systems, Science and Cybernetics**, New York, p. 100-107, 1968.

HENTSCHKE, R.; JOHANN, M.; REIS, R. GURIA – **A Global Router for LEGAL detailed routing**. In: UFRGS MICROELECTRONICS SEMINAR, SIM, 16., 2001. **Proceedings...** Porto Alegre: Instituto de Informática da UFRGS, 2001. p. 45-48.

HENTSCHKE, R.; REIS, R. Lemon Dragon Physical Synthesis. In: UFRGS MICROELECTRONICS SEMINAR, SIM, 17., 2002. **Proceedings...** Porto Alegre: Instituto de Informática da UFRGS, 2002.

HENTSCHKE, R.; REIS, R. **Roteador do Dragão Limão**. 2002. Disponível em <www.inf.ufrgs.br/~renato/download> .

JOHANN, M.; CALDWELL, A.; KAHNG, A.; REIS, R.; A New Bidirectional Heuristic Shortest Path Search Algorithm. In: INTERNATIONAL ICSC CONGRESS ON ARTIFICIAL INTELLIGENCE AND APLICATIONS, 2000, Wollongong, Australia. **Procedings...** [S.l.: s.n.], 2000.

JOHANN, M.; REIS, R.; Net by Net Routing with a New Path Search Algorithm. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 13., 2000, Manaus, AM. **Proceedings...** New York: IEEE Computer Society, 2000.

JOHANN, M.; REIS, R. **Novos Algoritmos para Roteamento de Circuitos Integrados**. 2001. Tese (Doutorado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

JOHANN, M.; REIS, R. LEGAL: An Algorithm for Simultaneous Net Routing. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 14., 2001, Pirenópolis, GO. **Proceedings...** New York: IEEE Computer Society, 2001.

JOHANN, M.; SANTOS, G.; REIS, R. A LEGAL algorithm following global routing. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 15., 2002, Porto Alegre, RS. **Proceedings...** New York: IEEE Computer Society, 2002.

KARYPIS, G.; KUMAR, V. Multilevel k-way Hipergraph Partitioning. In: DESIGN AUTOMATION CONFERENCE, DAC, 36., 1999, New Orleans **Proceedings...** New York: ACM, 1999.

KLING, R.; BANERJEE, P. ESP : A New Standard Cell Placement Package Using Simulated Evolution. In: DESIGN AUTOMATION CONFERENCE, DAC, 24., 1987. **Proceedings...** New York: IEEE, 1987.

LAM, J.; DELOSME, J. Performance of a New Annealing Schedule. In DESIGN AUTOMATION CONFERENCE, DAC, 25., 1988. **Proceedings...** New York: IEEE, 1988.

LANDMAN, B.; RUSSO, R. On a pin versus block relationship for partitions of logic graphs. **IEEE Transactions on Computers**, New York, v. C-20, p.1469-1479, Dec. 1971.

LAZZARI, C.; REIS, R. **Gerador de Layouts**. Disponível em

<http://www16.brinkster.com/clazz/layouts/>. Acesso em: 19 dez 2002.

LIU, Y.; XIANLONG, H.; CAI, Y.; WU, W. CEP: A Clock-Driven ECO Placement Algorithm for Standard-Cell Layout. In: INTERNATIONAL CONFERENCE ON ASIC, 2001. **Proceedings...** [S.l.: s.n.], 2001.

LUBASZEWSKI, M.; REIS, R. **Geração automática de lógica aleatória utilizando a metodologia tranca.** 1990. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

MADDEN, P. Reporting of Standard Cell Placement Results. **IEEE Transations on CAD of Integrated Circuits and Systems**, New York, v. 21, n.2, p. 240-247, Feb. 2002.

MALLETA, S.; GROVER, L. Clustering based Simulated Annealing for Standard Cell Placement. In: DESIGN AUTOMATION CONFERENCE, DAC, 25., 1988. **Proceedings...** New York: IEEE, 1988.

MARQUARDT, A.; BETZ, V.; ROSE, J. Speed and Area Tradeoffs in Cluster-Based FPGA Architectures. **IEEE Transactions on Very Large Scale Integration Systems**, New York, v. 8, n.1, p. 84–93, Feb. 2000

MO, F.; TABARA, A.; BRYTON, K. A Timing-Driven Macro-Cell Placement Algorithm. In: INTERNATIONAL CONFERENCE ON COMPUTER DESIGN, ICCD, 2001. **Proceedings...** [S.l.: s.n.], 2001.

MORAES, F.; ROBERT, M.; AUVERGNE, D. A Virtual CMOS Library Approach for Fast Layout Synthesis. In: IFIP TC 10.5 INTERNATIONAL CONFERENCE ON VERY LARGE SCALE INTEGRATION, VLSI, 10., 1999, Lisboa. **VLSI**: Systems on a Chip. Boston: Kluwer, 2000.

PARAKN, P.; BROWN, R.; SAKALLAH, K. Congestion Driven Quadratic Placement. In: DESIGN AUTOMATION CONFERENCE, DAC, 35., 1998, San Francisco. **Proceedings...** New York: ACM, 1998.

PREAS, B.; LORENZETTI, M. **Physical Design Automation of VLSI Circuits**. Menlo Park: The Benjamin/Cummings, 1988.

OU, S.; PEDRAN, M. Timing-driven Placement Based On Partitioning With Dynamic Cut-Net Control. In: DESIGN AUTOMATION CONFERENCE, DAC, 37., 2000, Los Angeles. **Proceedings...** New York: ACM, 2000.

SAIT, S.; YOUSSEF, H. **VLSI Physical Design Automation – Theory and Practice**. New York: IEEE, 1995.

SAIT, S.; YOUSSEF, H.; KHAN, J.; MALEH, A. Fuzzy Simulated Evolution for Power and Performance Optimization of VLSI Placement. In: INTERNATIONAL JOINT CONFERENCE ON NEURAL NETWORKS, IJCNN, 2001, Washington. **Proceedings...** Piscataway: IEEE, 2001.

SECHEN, C.; VINCENTELLI, A. TimberWolf3.2: A new standard cell placement and global routing package. In: DESIGN AUTOMATION CONFERENCE, DAC, 23., 1986. **Proceedings...** New York: IEEE, 1986.

SHERWANI, Naveed A. **Algorithms for VLSI Physical Design Automation**. 3rd ed. [S.l.]: Kluwer Academic, 1998.

STARKWEATHER, T. et al. A Comparison of Genetic Sequencing Operators. In: INTERNATIONAL CONFERENCE ON GENETIC ALGORITHMS, 4., 1991. **Proceedings...** [S.l.: s.n.], 1991.

SU, L.; BUNTINE, W.; NEWTON, R. Learning as Applied To Stochastic Optimization for Standard-Cell Placement. **IEEE Transactions on CAD of Integrated Circuits And Systems**, New York, v. 20, n. 4, Apr. 2001.

TSAI, C.; KANG, S. Cell-level Placement for Improving Substrate Thermal Distribution. **IEEE Transactions on CAD**, New York, v. 18, n.2, p. 253-266, Feb. 2000.

WANG, M.; LIM, S.; CONG, J.; SARRAFZADEH, M. Multi-way Partitioning Using Bi-partition Heuristics. In: ASIA AND SOUTH PACIFIC DESIGN AUTOMATION CONFERENCE, ASP-DAC, 2000. **Proceedings...** New York: ACM SIGDA, 2000.

YANG, X.; KASTNER, R.; SARRAFZADEH, M. Congestion Estimation During Top-Down Placement. **IEEE Transations on CAD of Integrated Circuits and Systems**, New York, v. 21, n.1, Jan. 2002.

YILDIZ, M.; MADDEN, P. Improved Cut Sequences for Partitioning Based Placement. In: DESIGN AUTOMATION CONFERENCE, DAC, 38., 2001, Las Vegas. **Proceedings...** New York: ACM, 2001.

Anexo 1

Benchmarks

Ao longo do texto, foram feitas diversas experiências a fim de verificar propriedades de algoritmos, comparar desempenho, memória, etc. Todos eles usaram circuitos reais como exemplo, retirados do pacote de *benchmarks* ISCAS 87 e ISCAS 89. Basicamente dois grandes grupos de *benchmarks* foram estudados: circuitos seqüenciais e circuitos combinacionais. Esta propriedade pode ser observada a partir do nome do circuito. Todos os que começam pelo caracter 's' são seqüenciais, enquanto que os circuitos combinacionais começam por 'c'.

Todos os circuitos foram submetidos a uma etapa de síntese lógica, com mapeamento tecnológico. Este mapeamento visa transformar a lógica do circuito em um conjunto de portas lógicas que possam ser utilizadas pela ferramenta do *Mango Parrot*. Teoricamente, a ferramenta é capaz de sintetizar qualquer porta, sem limitação. Porém, o mapeamento tecnológico visa restringir o número de transistores em série para um melhor desempenho elétrico.

A tabela 23 mostra todos os circuitos utilizados neste trabalho.

Benchmarks utilizados ao longo do texto

Nome do Circuito	Número de Células	Número de Redes	Número de Pads	Máximo de Transistores em Série
C1908_3x3	783	816	58	3
Alu2 4x4	317	327	16	4
Alu4_4x4	2523	2537	22	4
Alu4_2x2	4844	4858	22	2
C53	2307	2485	301	5
Misex3	5477	5491	28	2
C5315_4x4	1612	1434	0	4
Mult	1468	1408	0	¿
C432_4x4	134	170	0	4
C499	405	364	0	?
Bw	473	468	0	2