UNIVERSIDADE FEDERAL DE PELOTAS

Programa de Pós Graduação em Computação

Dissertação de Mestrado

Um Novo Algoritmo de Busca de Caminhos em Grade

Stèphano Machado Moreira Gonçalves

Pelotas, 2016

STÈPHANO MACHADO MOREIRA GONÇALVES

Um Novo Algoritmo de Busca de Caminhos em Grade

Dissertação apresentada ao Programa de Pós-Graduação em Computação do Centro de Desenvolvimento Tecnológico da Universidade Federal de Pelotas, como requisito parcial à obtenção do título de Mestre em Computação.

Orientador: Felipe de Souza Marques

Coorientador: Leomar Soares da Rosa Jr.

Pelotas, 2016

Universidade Federal de Pelotas / Sistema de Bibliotecas
Catalogação na Publicação

G635n Gonçalves, Stéphano Machado Moreira

Um novo algoritmo de busca de caminhos em grade / Stéphano Machado Moreira Gonçalves ; Felipe de Souza Marques, orientador ; Leomar Soares da Rosa Júnior, coorientador. — Pelotas, 2016.

117 f. : il.

Dissertação (Mestrado) — Programa de Pós-Graduação em Computação, Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas, 2016.

1. Busca de caminhos. 2. Roteamento detalhado. 3. Algoritmo. I. Marques, Felipe de Souza, orient. II. Rosa Júnior, Leomar Soares da, coorient. III. Título.

CDD : 005

Elaborada por Aline Herbstrith Batista CRB: 10/1737

Resumo

GONÇALVES, Stèphano Machado Moreira. **Um Novo Algoritmo de Busca de Caminhos em Grade**. 2016. 118f. Dissertação (Mestrado em Computação) – Programa de Pós-Graduação em Computação, Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas, Pelotas, 2016.

O processo de síntese de circuitos possui uma enorme complexidade envolvida, exigindo o uso de algoritmos para automatizar os procedimentos. Uma das etapas desse grande processo é o roteamento, que visa determinar as rotas dos fios que conectam os componentes do circuito. O roteamento é subdividido em roteamento global e detalhado. No roteamento detalhado, são utilizados algoritmos de busca de caminhos em grade para definir as rotas dos fios. Contudo, é esperado que tais algoritmos possam lidar com pelo menos as regras de projeto mais básicas. Assim, considerando que o roteamento é responsável por grande parte do tempo envolvido na síntese de circuitos, este trabalho propõe um novo algoritmo de busca de caminhos genérico em grades tridimensionais, chamado SG-Router, com a capacidade de lidar com algumas das regras de projeto mais simples. O objetivo da proposta é realizar uma comparação de tempo de execução e qualidade de caminho com o algoritmo de Hetzel, estado da arte dos algoritmos de busca genéricos em grade utilizados no roteamento detalhado. O trabalho também apresenta uma série de propostas de otimizações de tempo e de qualidade de busca para a versão preexistente do algoritmo, que funciona apenas no escopo bidimensional. Grande parte dessas otimizações foram reaproveitadas no SG-Router. Os experimentos realizados na versão bidimensional melhorada mostraram que o algoritmo obteve o caminho ótimo em todas as buscas. O algoritmo se mostrou mais rápido que o algoritmo de Hetzel, adaptado ao espaço 2D, com um ganho em tempo de execução entre 2,68 a 7522 vezes mais rápido. Os experimentos com o SG-Router em cenários de busca com obstáculos aleatórios mostraram um ganho em desempenho de pelo menos 11, para os cenários com mais obstáculos, e de até 1897, para os cenários médios. O algoritmo apresentou uma deficiência para lidar com cenários semelhantes a labirintos, pois nesses casos o algoritmo apresenta uma facilidade para ocasionar estouros de memória. Esse problema impediu sua aplicação no roteamento detalhado. Contudo, o empecilho não é definitivo e pode ser contornado. O trabalho também sugere futuras melhorias para o SG-Router, tornando-o um algoritmo promissor para o roteamento detalhado e para cenários de busca mais genéricos.

Palavras-chave: busca de caminhos; roteamento detalhado; algoritmo.

Abstract

GONÇALVES, Stèphano Machado Moreira. **A New Path-Search Algorithm on Grids.** 2016. 118f. Dissertation (Master's Degree in Computing) – Postgraduate Program in Computer Science, Technology Development Center, Federal University of Pelotas, Pelotas, 2016.

The process of circuit synthesis has an enormous complexity involved, requiring the use of algorithms to automate the procedures. One of the stages of this long process is the routing, which aims to determine the wiring routes connecting the circuit components. The routing step is divided in global and detailed routing. In detailed routing, path-search algorithms on grids are used to determine the wiring routes. However, it is expected that these algorithms are able to handle at least the most basic design rules. Thus, considering that routing is very time consuming, this paper proposes a new generic path-search algorithm on three-dimensional grids, called SG-Router, able to handle some of the simpler design rules. The goal of the proposal is to perform a comparison, regarding runtime and path quality, with Hetzel's algorithm, which is the state of the art of the generic path-search algorithms on grid, used in detailed routing. The paper also presents some proposals of optimizations, regarding time and search quality of the already existing version of the algorithm, which works only in two-dimensional scope. Most of these optimizations were reused in the SG-Router. The experiments performed on the improved two-dimensional version of the algorithm showed that the algorithm obtained the optimal path in all searches. The algorithm was faster than Hetzel's algorithm, adapted to 2D space, presenting a speedup between 2,68 and 7522. The experiments with the SG-Router in random search scenarios showed a speedup of at least 11, for scenarios with more obstacles, and up to 1857, for average scenarios. The algorithm presented a deficiency to handle scenarios similar to labyrinths, since in these cases the algorithm can easily cause memory overflow. This problem prevented the use of the algorithm in detailed routing. However, the drawback is not final and can be bypassed. This work also suggests future improvements to the SG-Router, making it a promising algorithm for the detailed routing and more generic search scenarios.

Key-words: path-search; detailed routing; algorithm.

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

CAD	Computer Aided Design
HDL	Hardware Description Language
ISPD	Inernation Symposium on Physical Design
MRST	Minimal Rectlinear Steiner Tree
MST	Minimal Spanning Tree
RNR	Ripup-and-Reroute
WL	Wire-Length

Sumário

1 INTRODUÇÃO

Com o desenvolvimento da tecnologia, os circuitos integrados passaram a se tornar cada vez mais complexos. A redução no tamanho dos transistores, e consequentemente de outros componentes, possibilitou a concepção de circuitos com um número extremamente grande de portas lógicas, limitados a espaços físicos muito pequenos. Além disso, o mercado aumentou a demanda da tecnologia, exigindo maior agilidade na produção de circuitos integrados. Sendo assim, devido a alta complexidade, a concepção manual de circuitos integrados tornou-se tarefa inviável, acarretando no desenvolvimento de ferramentas CAD (*Computer Aided Design*) para automatizar o processo de síntese de circuitos.

O processo de síntese *standard cell* começa na descrição do comportamento lógico do circuito utilizando linguagens de descrição de *hardware* (HDL). Em seguida, uma estrutura de dados chamada *netlist* é gerada com base na descrição do circuito. Essa estrutura armazena todas as informações referentes aos componentes do circuito. Depois disso, inicia-se a síntese lógica, que tem por objetivo analisar o conjunto de funções lógicas que descrevem o circuito para aplicar otimizações, simplificando a posterior implementação física do mesmo e mantendo ao mesmo tempo seu comportamento. Uma vez que a síntese lógica tenha acabado, é iniciada a etapa de mapeamento tecnológico. Esta etapa consiste em definir quais portas lógicas serão utilizadas para implementar as funções lógicas que definem o circuito. Essas portas lógicas pertencem a uma biblioteca previamente definida, contendo informações sobre cada porta. Após o mapeamento tecnológico, é iniciada a síntese física. Esta etapa tem como objetivo definir as disposições físicas dos componentes do circuito. Ela é subdividida em outras fases, sendo o posicionamento e o roteamento as principais. A etapa de posicionamento visa determinar as posições dos componentes de forma a minimizar uma função custo e facilitar o

esforço das próxima etapa do fluxo de síntese. A função custo, geralmente refere-se a uma estimativa do *wire-length* (WL), que é o comprimento total de fios do circuito, dado um determinado posicionamento de células lógicas. Após o posicionamento, vem a fase de roteamento, que visa determinar as rotas dos fios necessários para realizarem as conexões entre os componentes, minimizando também o WL.

Devido a enorme complexidade, o roteamento é subdividido em duas etapas: roteamento global e detalhado. No roteamento global, são descobertas as áreas pelas quais os fios irão passar. O congestionamento de fios é levado em consideração nesta fase, para que o roteamento detalhado tenha seu esforço reduzido. Assim, o objetivo do roteamento global é fornecer instruções para o roteamento detalhado, além de reduzir seu esforço. O roteamento detalhado, por sua vez, visa determinar a localização exata das rotas dos fios, respeitando, com certa flexibilidade, as áreas delimitadas pelo roteamento global. O roteamento detalhado deve levar em conta uma série de restrições de manufatura, normalmente designadas como regras de projeto. Como o objetivo do roteamento detalhado é definir as rotas dos fios, o núcleo do roteador detalhado é implementado por um algoritmo de busca de caminhos. No entanto, vale ressaltar que o roteador detalhado não se restringe apenas a uma busca de caminhos, como será visto na sessão 2.2. O núcleo mencionado, pode tanto ser um algoritmo bem específico para o problema, levando em conta as peculiaridades da tecnologia utilizada na concepção do circuito, como pode ser um algoritmo mais genérico, ou ainda, um híbrido de ambos. No caso de um algoritmo mais genérico de busca de caminhos, este não precisa (nem deve) considerar todas as regras de projeto em sua lógica, pois isso acarreta em uma grande perda em desempenho. Boa parte das regras de projeto podem ser consideradas por módulos separados, os quais se comunicam com o algoritmo de busca. No entanto, o algoritmo deve considerar as regras mais triviais.

A busca de caminhos pode ser modelada em um espaço representado por uma grade (*grid routing*), considerando um conjunto de coordenadas discretas, assim como pode não haver grade (*gridless routing*), onde o conjunto de coordenadas é contínuo. Neste trabalho, serão tratados apenas algoritmos que atuam sobre uma grade (ou *grid*). O *grid* é uma estrutura de dados representada formalmente por um grafo de grade (WEISSTEIN, 2013). Contudo, a implementação do *grid* não precisa armazenar o grafo de forma explicita, como será visto

posteriormente. O objetivo da busca de caminhos é descobrir o caminho de menor custo (preferentemente) entre dois pontos (ou dois conjuntos de pontos) contidos no espaço de busca.

A primeira solução proposta para o problema do menor caminho entre dois pontos em um grafo com arestas positivas foi o algoritmo de Dijkstra (1959 apud CORMEN, 2002). O algoritmo utiliza uma abordagem por amplitude, pois parte de um ponto inicial e vai expandindo a busca para todos os nodos adjacentes, e assim sucessivamente até chegar no destino desejado. O algoritmo de Lee (1961), também conhecido como *Maze Router*, é a implementação do algoritmo de Dijkstra, no contexto específico de grafos de grade.

Posteriormente, a meta-heuristica do ramo da inteligência artificial, chamada A* (HART, 1968), foi trazida ao roteamento de circuitos integrados, por Rubin (RUBIN, 1974). O A* no roteamento atua de forma muito semelhante ao algoritmo de Lee, com a diferença de utilizar uma heurística para reduzir o esforço computacional, fazendo a busca convergir para a solução de forma muito mais rápida. Da mesma forma que o algoritmo de Lee, o A* é conhecido por garantir o caminho ótimo.

Outra abordagem para o problema é a pesquisa bidirecional. Neste caso são feitas duas pesquisas, uma partindo da origem e outra partindo do destino. Qualquer algoritmo de pesquisa pode ser utilizado. Contudo, existem problemas nesta abordagem, como a condição de término, a intersecção de pesquisas e o problema das frentes desencontradas. Neste contexto, surgiu o algoritmo LCS* (JOHANN, 2000).

Os algoritmos mencionados acima pertencem a uma classe denominada *maze*, visto que são todos baseados no *Maze Router* de Lee. No entanto, existe também a classe dos algoritmos de busca por linha (*Line Search*). Estes algoritmos utilizam outra estratégia para realizar o roteamento. Ao invés de expandir a busca de ponto a ponto, segmentos ortogonais são traçados para determinar o caminho. Como um segmento pode conter muitos pontos, a criação de um único segmento pode poupar muito esforço computacional em comparação aos algoritmos de busca por *maze*. Com isso, os algoritmos de busca em linha são conhecidos por apresentar um tempo de processamento e consumo de memória muito menores. Porém, estes algoritmos, em geral, não são capazes de garantir que o menor caminho seja encontrado, o que não ocorre com os algoritmos de busca por *maze*.

A primeira solução de busca por linha foi de Mikami e Tabuchi (1968). O algoritmo traça segmentos nos pontos de origem e de destino, e a partir destes segmentos, novos segmentos perpendiculares são traçados até que haja uma intersecção entre um segmento proveniente do ponto de origem com outro segmento do ponto de destino. O algoritmo garante que um caminho seja encontrado, mas não garante que o caminho seja ótimo. Mais tarde, Hightower (1969) propôs uma modificação para o algoritmo anterior, aumentando o desempenho e reduzindo o consumo de memória. No entanto, o algoritmo não consegue garantir que um caminho seja encontrado. Para superar esse problema, é necessária a utilização de técnicas de *backtracking*. Porém, isto reduz o desempenho do algoritmo, podendo se tornar comparável ao *Maze Router*. Posteriormente, foi proposto o algoritmo de Hetzel (HETZEL, 1998), visando unir os benefícios das buscas por linha e por *maze*. O algoritmo é semelhante ao A*, mas realiza a expansão através de intervalos. O desempenho é semelhante aos algoritmos de busca por linha e o menor caminho é sempre encontrado. No entanto, o algoritmo possuía algumas restrições quanto a algumas regras de projeto do roteamento em circuitos integrados. Com isso o algoritmo foi generalizado em (PEYER, 2009) e em (HUMPOLA, 2009), aumentando assim a amplitude de seu uso no roteamento detalhado. Experimentos mostraram que utilizar intervalos garante um *speedup* de pelo menos 6 vezes em relação a busca pontual (GESTER, 2012). Com isso, esse algoritmo é o estado da arte dos algoritmos genéricos de busca de caminhos utilizados no roteamento detalhado de circuitos integrados. Este algoritmo constitui o núcleo do roteador detalhado da ferramenta BonRoute (GESTER, 2013). No entanto, para áreas congestionadas, o algoritmo atua de forma idêntica ao algoritmo A*, tornando-o muito lento para estes casos. Levando em conta esse problema, em um trabalho anterior (GONÇALVES 2014), foi proposto um novo algoritmo de busca de caminhos, visando lidar com áreas congestionadas de maneira mais eficiente. O algoritmo, chamado ST-Router, utilizava uma abordagem de busca em linha, utilizando heurísticas de custos semelhantes ao A*, e estratégias para se contornar obstáculos. Os experimentos mostraram um imenso ganho de desempenho ao custo de um mínimo aumento no comprimento dos fios. Contudo, o algoritmo contemplava apenas o espaço bidimensional, enquanto o algoritmo comparado lidava com um espaço de busca tridimensional. Assim, para se aplicar o ST-Router na busca de caminhos em um roteador detalhado é necessário realizar

uma extensão do algoritmo, considerando o espaço tridimensional, e levando em conta algumas regras de projeto. Além disso, como o comprimento de fios é algo muito importante no roteamento, é desejável que o algoritmo possa garantir o caminho ótimo.

Sendo assim, este trabalho tem como objetivo apresentar duas contribuições para o algoritmo ST-Router. A primeira, e mais simples, é uma série de propostas de otimizações no algoritmo fazendo com que ele garanta o caminho ótimo mantendo um grande desempenho. A segunda, de maior relevância, propõe uma nova versão do algoritmo, agora chamado de SG-Router, levando em conta o escopo tridimensional e algumas regras de projeto. Apesar do algoritmo ainda não considerar algumas regras, as que ele considera já o aproximam bastante de uma aplicação no roteamento detalhado. Além disso, é discutido como algumas dessas regras podem ser consideradas e resolvidas. Vale ressaltar que a proposta do SG-Router refere-se a um algoritmo de busca de caminhos, não a um roteador detalhado. Pretende-se que o algoritmo, assim como outros algoritmos genéricos de busca de caminhos, possa ser aplicado (assim que tratar com as regras de projeto mínimas necessárias) dentro do processo de roteamento detalhado. O SG-Router foi comparado com o algoritmo de Hetzel, levando-se em conta sua versão mais recente em (GESTER, 2013). Os resultados mostram que o algoritmo ainda tem alguns aspectos a serem melhorados, mas apresenta resultados muito promissores.

O capítulo 2 apresenta uma revisão bibliográfica sobre o roteamento. São discutidos os problemas do roteamento global e do roteamento detalhado. O capítulo 3 apresenta alguns algoritmos de busca de caminhos, com aplicação no roteamento. O algoritmo ST-Router também está incluído, apesar de não ser utilizado no roteamento de circuitos, devido sua limitação à busca em duas dimensões. O capítulo 4 apresenta as modificações propostas no ST-Router, e os experimentos realizados para avaliar as modificações. O capítulo 5 apresenta a versão tridimensional do algoritmo, chamado SG-Router. O capítulo 6 apresenta os experimentos realizados para avaliar o desempenho e a optimalidade do SG-Router, bem como sua aplicabilidade no roteamento detalhado. O capítulo 7 apresenta as considerações finais da dissertação.

2 ROTEAMENTO DE CIRCUITOS INTEGRADOS

Um circuito integrado é constituído de portas lógicas, as quais são interligadas por fios. Uma porta lógica é um componente que implementa uma função lógica, possuindo assim entradas e saídas, as quais são chamadas de pinos ou terminais. Um pino de saída carrega o sinal elétrico por meio de fios, que se conectam a pinos de entradas de outras portas lógicas. Esse conjunto de pinos, conectados, por fios, é chamado de rede. Toda rede possui um pino chamado de fonte (em inglês: *source* ou *driver*), o qual é a origem do sinal elétrico, e pelo menos um pino chamado de dreno (*sink*).

O objetivo do roteamento é definir fisicamente o conjunto de fios que implementará cada rede do circuito. A principal preocupação do roteamento é otimizar o comprimento total de fios. Porém, existem outros fatores que também se deseja otimizar e que podem entrar em conflito uns com os outros (SCHULTE, 2012), como *delay* e consumo de potência. Além disso, a disposição dos fios deve obedecer uma série de regras de projeto dependentes da tecnologia utilizada.

O espaço de roteamento é modelado por um grafo de grade tridimensional. Normalmente, o roteamento realiza em torno de milhões de conexões em um grafo com bilhões de vértices (SCHULTE, 2012). Com isso, resolver o problema, mesmo que em tempo linear, é impraticável. Assim, o roteamento utiliza a estratégia de dividir para conquistar, resolvendo o problema em duas etapas: roteamento global e detalhado.

Após o roteamento, é verificado se o circuito atende a certas restrições, como *timing* (tempo de propagação do sinal, também chamado de *delay*), consumo de potência e regras de projeto. Dependendo de peculiaridades do roteamento, como a presença de longos fios em redes criticas, o *timing* pode ser insuficiente, “desperdiçando” assim o roteamento inteiro (SCHULTE, 2012). Geralmente, obter

um *timing* adequado consiste em um processo iterativo onde várias etapas da síntese física são executadas, incluindo o roteamento (SCHULTE, 2012). É por esse motivo que o tempo de execução constitui um fator de suma importância para o roteamento.

2.1 Roteamento Global

O objetivo do roteamento global é reduzir a complexidade do problema, fornecendo uma pré-solução ao roteamento detalhado, que define o roteamento final. Essa redução de complexidade se dá pela utilização de *grid* simplificado (Figura 1a), onde cada ponto engloba vários pontos do grid real. Assim, cada ponto representa uma área, chamada de célula global (*g-cell*). O *grid* pode ser tanto bidimensional como tridimensional. Cada aresta do grafo (que representa o *grid*) representa a adjacência entre duas áreas (*g-cells*). Cada aresta possui uma capacidade, que representa o número máximo de fios que podem passar por ambas regiões. A medida que o roteamento ocorre, o nível de congestionamento de cada aresta é calculado com base na capacidade da aresta e na quantidade de fios que passam por ela. Os resultados do roteamento global se dão com base na minimização de uma função custo, que envolve WL, *delay* e congestionamento. A saída do roteamento global consiste em uma série de redes roteadas. Cada uma dessas redes é definida por um conjunto de *g-cells*, que constituem regiões delimitadas, chamadas de túneis ou corredores, as quais o roteamento detalhado deve obedecer com certa flexibilidade (Figura 1b, c, d).

O roteamento global possui várias abordagens. Estas, dependem do tamanho da rede, isto é, do número de pinos. Pode-se dividir as abordagens em duas categorias: para redes de dois pinos e para redes com mais de dois pinos. Considerando redes de dois pinos, existem duas formas de se resolver o problema: utilizando roteamento por *maze* ou utilizando roteamento por padrões. No roteamento por *maze* é utilizado o algoritmo A* (sessão 3.2), ou alguma variação. No roteamento por padrões, a busca se restringe especificamente a considerar padrões de caminhos que seguem o formato L (uma dobra) ou Z (duas dobras, Figura 1d). Esta abordagem foi proposta por Chen (1999) e Kastner (2002). Ambas

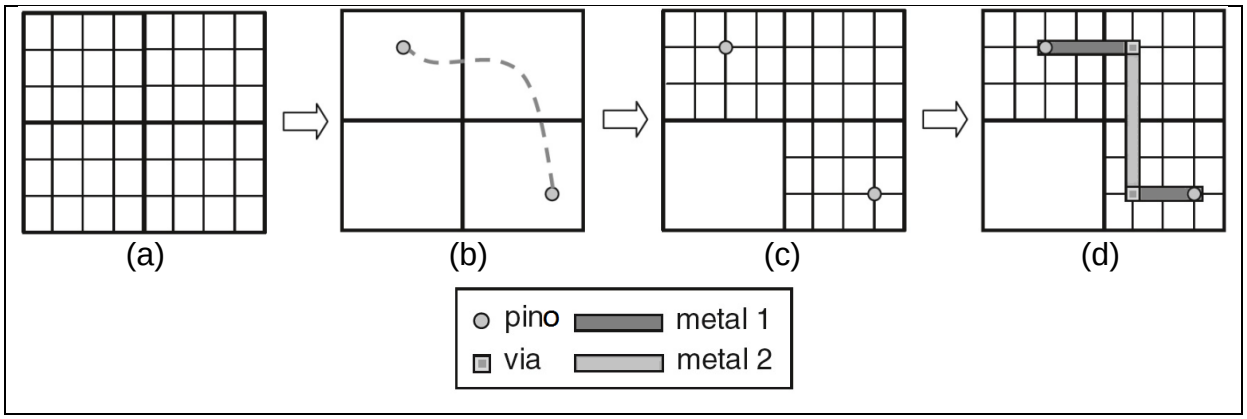

Figura 1 – Ilustração do roteamento global. (a) *Grid* detalhado (linhas finas) e *grid* simplificado (linhas grossas). O *grid* simplificado contém quatro *g-cells*. (b) O roteamento global determina a rota de conexão entre dois pinos, com relação as *g-cells.* (c) A área do roteamento detalhado é restrita de acordo com o caminho resultante do roteamento global. (d) A conexão é de fato realizada no roteamento detalhado.

as formas de busca (*maze* e padrões) garantem o caminho ótimo, mas o roteamento por padrões se restringe a um espaço de busca bem menor, apresentando um desempenho muito superior. Com isso, o roteamento por padrões é amplamente utilizado no roteamento global de redes de dois pinos (CHEN, 2016).

A segunda categoria de algoritmos utilizados para o roteamento global refere-se a redes de múltiplos pinos. Dentro desse escopo, existem duas principais abordagens para se resolver o problema. A primeira é decompor a rede em múltiplas redes de dois pinos e resolver cada uma com roteamento por padrões. A segunda é resolver o problema das árvores de Steiner (SHERWANI, 1998).

A primeira abordagem pode ser resolvida encontrando uma MST (*minimal spanning tree*) dos pinos da rede. Uma MST é uma árvore que conecta todos os vértices de um grafo, possuindo o menor custo possível, o qual é dado pela soma dos custos das arestas. No caso em questão, os vértices do grafo são os pinos da rede e as arestas representam as possíveis conexões de cada pino com os demais. O custo das arestas é o menor custo possível da conexão. A MST pode ser encontrada em tempo polinomial pelo algoritmo de Kruskal (1956).

A resolução do problema por esta abordagem pode apresentar resultados sub-ótimos. A Figura 2a mostra a rede resultante de uma MST com quatro pinos. Note que a rede não apresenta WL ótimo, o que é o caso da Figura 2b. Neste caso, a MST resultou em um grafo com arestas (p_3, p_2), (p_2, p_4) e (p_4, p_1). Assim, o roteador decompôs a rede em três redes de dois pinos, as quais foram roteadas

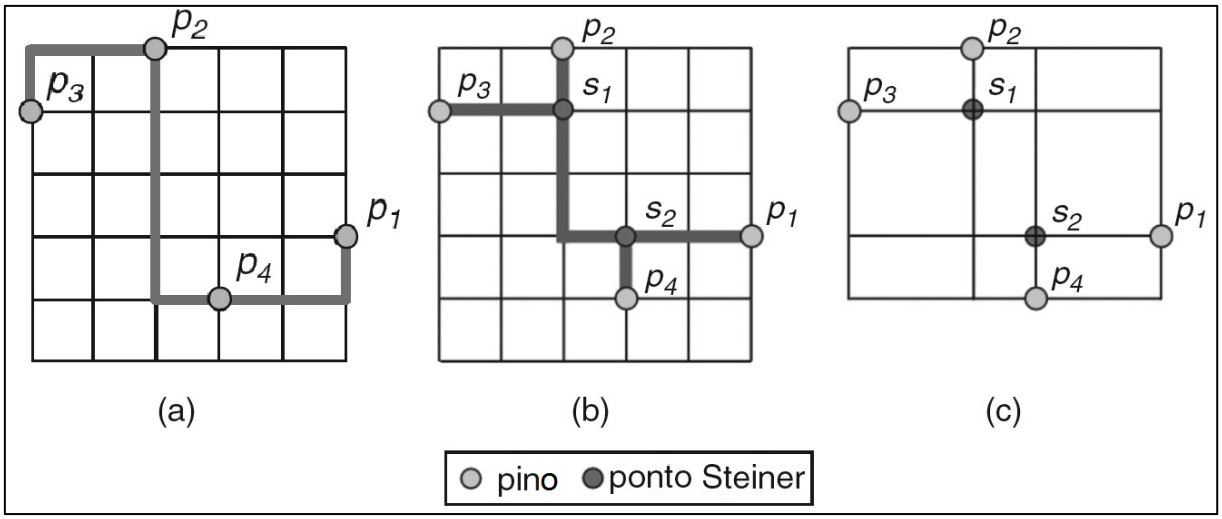

Figura 2 – Ilustração de uma MST (a), uma MRST (b) e um *grid* de Hanan, para uma rede de 4 pinos. Os *grids* e os caminhos são do roteamento global, não do detalhado.

separadamente. Considerando o caminho resultante de cada par de pinos, percebe-se que o WL é ótimo, o que não ocorre com o WL global. Isto se dá pelo fato de que o roteamento local (entre dois pinos) não prevê o compartilhamento de fios com outros roteamentos, pois eles ainda não aconteceram. Com isso, para considerar esse compartilhamento, é preciso deter uma visão global do problema, o que ocorre no problema das árvores de Steiner.

A segunda abordagem para se rotear uma rede de múltiplos pinos é encontrar a MRST (*Minimal Rectilinear Steiner Tree*) da rede. Uma MRST é uma MST com vértices adicionais chamados de pontos Steiner. Esses pontos são inseridos estrategicamente em locais para forçar o compartilhamento dos fios. O problema de se encontrar uma MRST se resume em determinar o número adequado de pontos Steiner e suas localizações, de forma a maximizar o compartilhamento. A princípio, existe um número infinito de pontos Steiner que precisam ser considerados para a construção de uma MRST. Contudo, Hanan (1966), provou que uma MRST pode ser encontrada com um número finito de pontos Steiner, pertencentes ao *grid* de Hanan, o qual é obtido pela projeção de linhas verticais e horizontais, sobre cada pino da rede (Figura 2c). Os vértices desse grid são os pontos de intersecção dessas retas. Apesar da contribuição de Hanan proporcionar uma grande redução no espaço de busca, o problema de se encontrar uma MRST é NP-Completo. Com isso, foram criadas heurísticas para resolver o problema de forma rápida, com pequena perda de WL em relação ao ótimo. O algoritmo FLUTE (CHU, 2008) é um bom exemplo do uso de tais heurísticas, apresentando resultado ótimo para redes de 9 pinos ou

menos. Contudo, esses métodos não levam em consideração fatores importantes para o roteamento, como obstáculos, áreas congestionadas e *delay*. Existem vários métodos que consideram a presença de obstáculos (LIN; WU; LI; HUANG; 2007, 2007, 2008, 2010). Quanto a consideração do congestionamento, o roteador FastRoute (PAN; PAN; ZHANG; XU; 2006, 2007, 2008, 2009) é uma das principais referências. Um outro método de se calcular uma MRST é utilizando o roteamento por *maze*. A vantagem dessa abordagem é que ela lida facilmente com todos os fatores envolvidos no roteamento, possibilitando uma solução de melhor qualidade, com a desvantagem do tempo de execução. Esta abordagem é utilizada pela ferramenta AMAZE (HENTSCHKE; 2007, 2009).

2.2 Roteamento Detalhado

Antigamente, quando o número de camadas disponíveis para roteamento (*layers*) era dois ou três, o roteamento possuía uma abordagem diferente da atualidade. Como a grande maioria das redes era conectada a células pertencentes ao mesmo canal (espaço entre duas fileiras de células lógicas), a abordagem era chamada de roteamento de canais. Nesse escopo surgiram os algoritmos *left-edge* (HASHIMOTO, 1971) e *dog-leg* (DEUTSCH, 1976). O roteamento global, mencionado na sessão anterior, não era utilizado, considerando a relativa proximidade entre os pinos da rede.

Atualmente, como o número de *layers* é muito maior, e a complexidade dos circuitos aumentou muito, devido ao maior número de células e redes, dificultando a vizinhança de células nos mesmos canais, o roteamento de canais perdeu utilidade, sendo substituído por um roteamento que lida com redes com pinos localizados em qualquer lugar (sempre sobre uma célula lógica) do circuito. Assim o roteamento necessita de uma abordagem de dividir para conquistar, executando um passo de roteamento global, gerando túneis que restringem o caminhos das redes, seguido do roteamento detalhado, que define exatamente o caminho dentro dos túneis (Figura 3), levando em consideração as regras de projeto.

O roteamento é realizado em um *grid* tridimensional. Cada plano (ou *layer*) possui uma direção (vertical ou horizontal), chamada de direção preferida, que restringe a direção dos fios. Uma conexão que desrespeita a direção preferida é chamada de *jog*. Planos adjacentes possuem direções preferidas diferentes, para

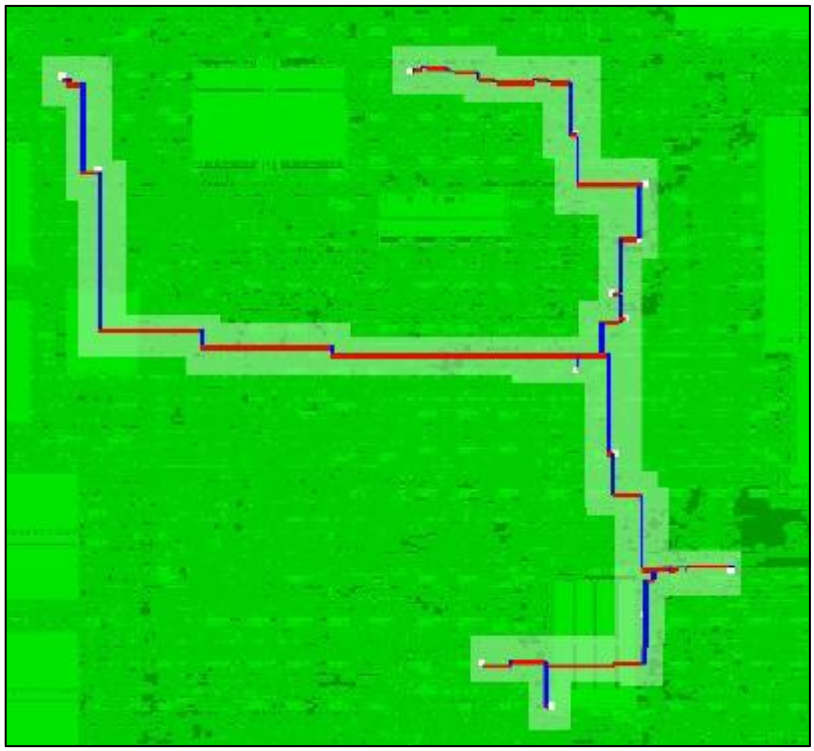

Figura 3 – Túneis de uma rede (verde claro) e o caminho da rede encontrado pelo roteamento detalhado (azul e vermelho). Fonte: (VYGEN, 2009)

evitar efeitos capacitância entre fios de mesma direção e posição em *layers* adjacentes, além de aumentar a roteabilidade. O “componente” que conecta dois planos é chamado de via.

No roteamento detalhado, o caminho resultante da rede é obtido de forma incremental, realizando buscas de caminhos entre porções da rede. Primeiramente, dois pinos são escolhidos e a rota entre ambos é obtida. A partir desse ponto, um novo pino é escolhido como alvo e o roteamento se dá partindo da rota preexistente até o novo alvo. Dependendo da abordagem, a origem da nova rota pode ser um dos pinos das rotas previamente encontradas, bem como a rota inteira. O roteador detalhado da ferramenta BonnRoute (GESTER, 2013) utiliza a primeira abordagem.

O roteamento detalhado possui duas etapas. A primeira consiste em conectar os pinos das redes, procurando seguir ao máximo as regras de projeto. A segunda, chamada *ripup and reroute* (RNR) consiste em analisar se algum caminho não pôde ser encontrado, removendo rotas que obstruem o caminho e roteando novamente a rota bloqueada e as removidas. O processo termina quando todas as redes foram roteadas ou quando um tempo limite é atingido.

A primeira etapa pode ser resolvida sob duas abordagens. A primeira, consiste em utilizar algoritmos genéricos de busca de caminhos em grafos (para

roteamento com *grid*) para encontrar as rotas. Neste caso, o algoritmo utilizado deve considerar algumas restrições e regras de projeto básicas. Quanto mais regras o algoritmo considera, mais a solução se aproxima dos requisitos de fabricação, diminuindo o esforço posterior da correção da solução para atender as regras. No entanto, quanto mais regras o algoritmo comtempla, menor é seu desempenho, e este é um fator crucial para o roteamento. É por este motivo que não é adequado levar em conta as regras mais complexas no algoritmo de busca de caminhos.

A segunda abordagem consiste em guiar a busca para atender ao máximo as regras de projeto, de forma a manter um bom desempenho. Nesses casos, pode-se utilizar métodos de busca especializados para lidar com as regras de projeto, podendo chamar algum algoritmo de busca mais genérico no meio do procedimento, ou pode-se utilizar abordagens que realizam uma espécie de pré-processamento antes da busca em si, objetivando guiar a busca para maximizar o número de regras de projeto respeitadas, que é o caso do roteador RegularRoute (ZHANG, 2013). Este roteador tenta explorar o uso de padrões de rotas que garantem o atendimento a muitas regras de projeto, evitando assim uma posterior análise para a verificação dessas regras. Mesmo nessa abordagem, a etapa de busca não considera regras muito complexas pelos mesmos motivos mencionados anteriormente. As regras não consideradas pela busca em si, são resolvidas em um pós-processamento, na etapa de RNR. No entanto, cabe ressaltar que é muito comum não se conseguir atender a absolutamente todas as regras.

Segundo Zhang (2013), o número de regras de projeto para uma tecnologia de 32 nm encontra-se na casa dos milhares. Muitas dessas regras são derivações de regras mais genéricas. A seguir, são apresentadas, de forma sucinta, algumas regras de projeto. A Figura 4 ilustra regras de espaçamento de fios e vias. O *pitch* é a distância entre os pontos do *grid*. Essa distância deve ser, no mínimo, a largura do fio mais a distância mínima entre fios. As regras de distância existem para impedir que o campo magnético gerado pela corrente de um fio interfira na corrente de um fio próximo. A Figura 5 ilustra exemplos de regras de fim de linha. Estas, restringem a distancia da extremidade de um fio em relação a outros fios da proximidade.

Cada *layer* possui fios com diferentes dimensões (Figura 6), que tendem a aumentar com o *layer*. Isso acarreta em um aumento no *pitch* de *layers* mais altos. Consequentemente, o *layer* com um *pitch* maior é desalinhado em relação aos *layers* inferiores de menor *pitch*. Com isso, não é possível pular de *layers* com *pitchs*

diferentes em qualquer lugar, apenas quando os pontos estão alinhados. Além disso, como os fios dos *layers* mais altos são mais espessos, é desejável que o algoritmo de roteamento considere que esses fios possuam custos maiores. As vias seguem o mesmo princípio, sendo maiores e mais custosas em *layers* mais altos.

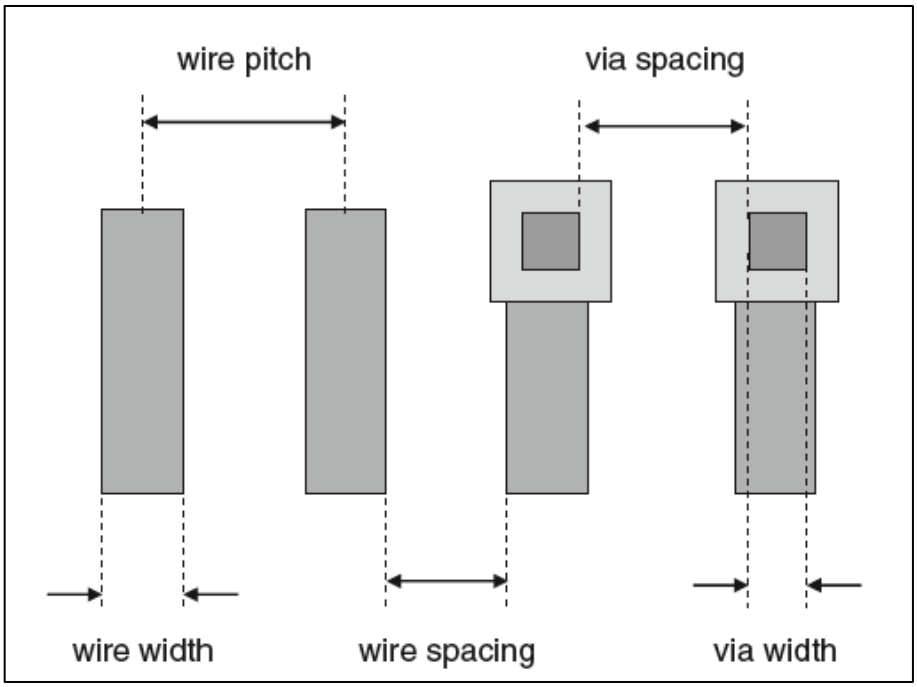

Figura 4 – Ilustração de regras de distancia de fios e vias. Fonte: (CHEN, 2016)

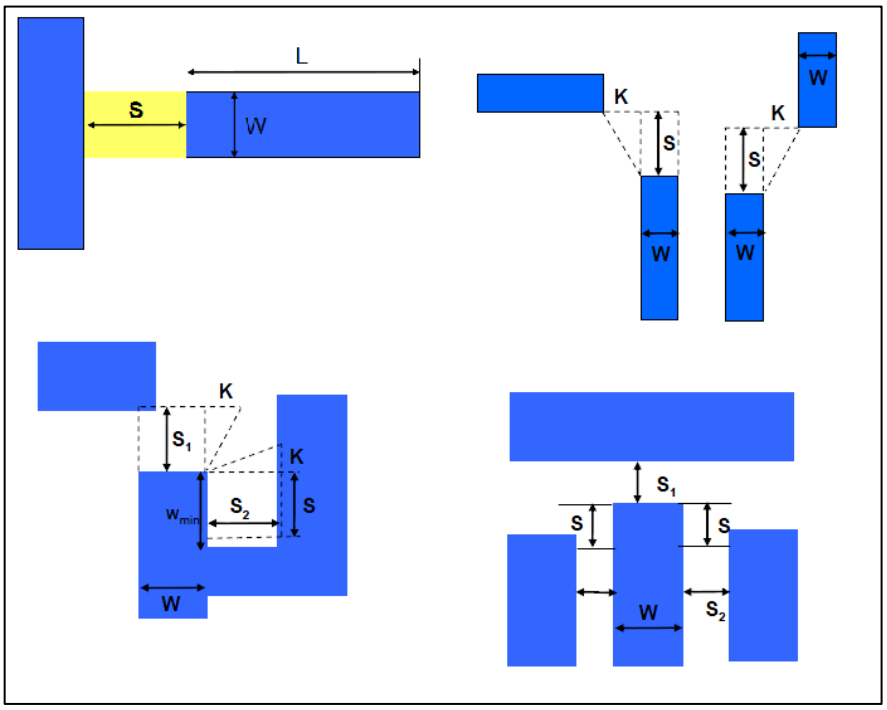

Figura 5 – Ilustração de regras de fim de linha.

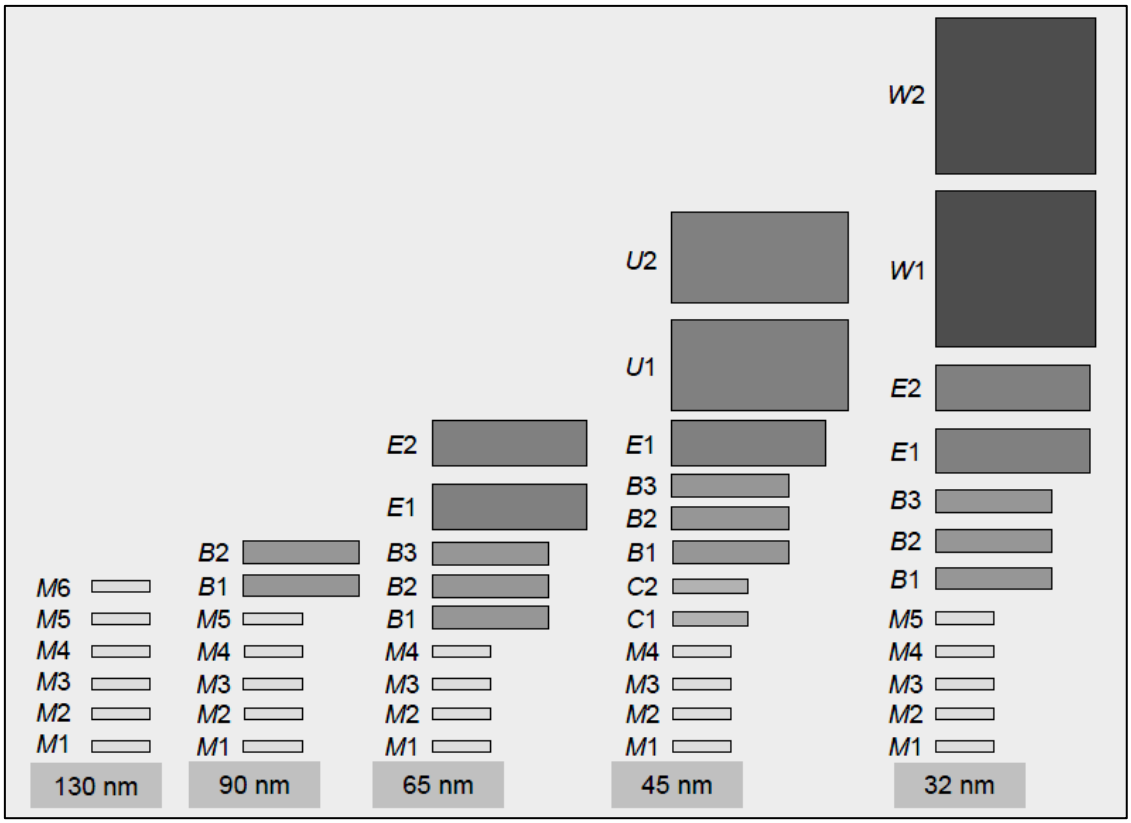

Figura 6 – Ilustração das espessuras de fios para cada *layer*, em diferentes tecnologias. (Fonte KAHNG, 2016)

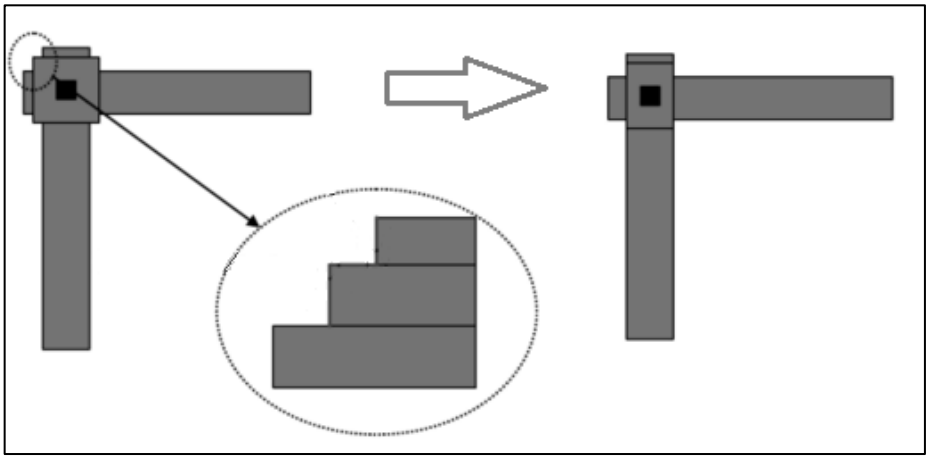

Figura 7 – Ilustração da regra de mínimo de pontas. A esquerda, tem-se a regra violada, com três pontas, e a direita, tem-se a correção.

O tamanho dos pinos também é determinado pelas regras. Normalmente, o tamanho é grande o suficiente para englobar vários pontos do *grid*. Assim, qualquer ponto do pino pode ser utilizado para realizar uma conexão com um fio. Com isso, é desejável que o algoritmo de roteamento seja capaz de lidar com múltiplos pontos de origem e destinos.

A Figura 7 mostra um exemplo da regra de mínimo de pontas. Essa regra proíbe a formação de pontas consecutivas de comprimento menor que um limiar predeterminado. A regra é aplicada a fios da mesma rede.

A regra de duplicação de vias consiste em utilizar duas vias, ao invés de uma, quando possível, para prevenir possíveis falhas no circuito, caso uma via venha a apresentar problemas, durante o ciclo de vida do circuito.

As regras de efeito antena são utilizadas para prevenir danos nos transistores durante a fabricação do circuito. O efeito antena ocorre quando cargas elétricas armazenadas em fios, que estão sendo fabricados, são descarregadas para os transistores, danificando-os ainda na fabricação do circuito.

Considerando que a fabricação do circuito envolve um grande número de regras de projeto, é desejável que o algoritmo de busca a ser usado no roteamento detalhado contemple pelo menos as regras mais simples. Com isso, é desejável que o algoritmo considere custos próprios para vias, que podem variar em cada *layer*, assim como diferentes custos de fios, e custos próprios para *jogs*. Boa parte das regras de espaçamento podem ser modeladas dentro da estrutura do *grid*, isentando assim o algoritmo de busca de verificar essas regras durante a execução. Igualmente, o desalinhamento dos pontos do *grid* de diferentes *layers* também diz mais respeito ao *grid*. Para tentar utilizar uma via, basta consultar o *grid* para verificar se isso é possível, no determinado ponto. Além disso, é desejável que o algoritmo seja capaz de lidar com múltiplos pontos de origem e destino.

3 ALGORITMOS DE BUSCA DE CAMINHOS

A busca de caminhos é um problema clássico da computação. Dado um grafo de arestas com custos positivos, deseja-se encontrar o menor caminho entre dois vértices do grafo. O algoritmo clássico utilizado para resolver esse problema é o algoritmo de Dijkstra. No roteamento de circuitos integrados, o grafo está associado a uma geometria, representando uma grade. Tais grafos são denominados grafos de grade.

Esta sessão apresenta algoritmos de busca de caminho em grafos de grade, que podem ser utilizados no roteamento detalhado. O algoritmo A*, por ser bastante genérico, pode ser utilizado também no roteamento global.

3.1 Algoritmo de Lee

O algoritmo de Lee (1961) é uma implementação específica do algoritmo de Dijkstra para a busca em um grafo de grade. O algoritmo é apresentado na Figura 8. O algoritmo inicia a busca no ponto de origem *s*. A partir deste ponto, os pontos adjacentes (com exceção das diagonais) são visitados e marcados com valores (Figura 9a). Para cada um desses pontos o processo se repete, gerando sucessivas ondas de expansão de busca, até que o alvo *t* seja encontrado (Figura 9b e c). Em seguida, o algoritmo inicia a fase de recuperação de caminho (Figura 9d). O ponto de destino é adicionado no caminho, e qualquer ponto adjacente com valor menor que o ponto de destino em uma unidade é adicionado no caminho. Este ponto é selecionado e o processo continua até que o ponto de origem seja alcançado. Depois, todos os pontos marcados são desmarcados. O algoritmo de Lee garante

que o caminho seja encontrado, caso exista algum, além de garantir que o caminho seja o menor.

A Figura 8 apresenta o algoritmo em sua forma “clássica”. Contudo, na prática, não é adequado utilizar uma implementação que siga fielmente esse pseudocódigo. Em primeiro lugar, os pontos do *grid* podem ter custos variáveis. Assim, ao marcar pontos adjacentes, deve-se somar o custo dos pontos em questão. Em segundo lugar, desmarcar todos pontos marcados pode ser custoso, e isso não é necessário. Basta utilizar uma estrutura básica chamada “nó”, que representa um ponto, mas possui outras informações associadas, como o custo e o nó pai. Assim, ao invés de marcar um ponto do *grid*, se cria um nó que o representa, e se atribui o custo apropriado. No final da busca, quando o alvo é encontrado, o caminho também já foi encontrado, visto que cada nó possui uma conexão com seu nó pai. Como os custos estão marcados nos nós, o *grid* não precisa ser avaliado para reinicializar os custos. Quanto ao *grid*, este poderia a princípio ser implementado de forma simples por uma matriz. No entanto, esta abordagem implica em um enorme consumo de memória no caso do roteamento de circuitos. Assim, é mais adequado utilizar uma estrutura (como um mapa *hash*) que armazene apenas as posições bloqueadas, além de posições com custos diferentes do padrão, se for o caso. Com isso, fica evidente que, apesar de se denominar este algoritmo (e os demais apresentados adiante) como algoritmo de busca em grafos, o grafo serve mais de modelagem teórica, pois não precisa ser explicitamente implementado.

```
Lee (Ponto s, Ponto t)
1  Marque s com valor 0;  i ← 0
2  repeat
3      Marque com i+1 todos vizinhos não marcados de cada ponto marcado com i
4      i ← i + 1
5  until t encontrado or impossível marcar mais pontos
6  if t não encontrado : return falha
7  Adiciona t ao caminho; p ← t
8  repeat
9      Adiciona ao caminho o vizinho p' de p de menor valor
10     p ← p'
11 until p = s
12 Desmarca todos os pontos marcados
```

Figura 8 – Pseudocódigo do algoritmo de Lee.

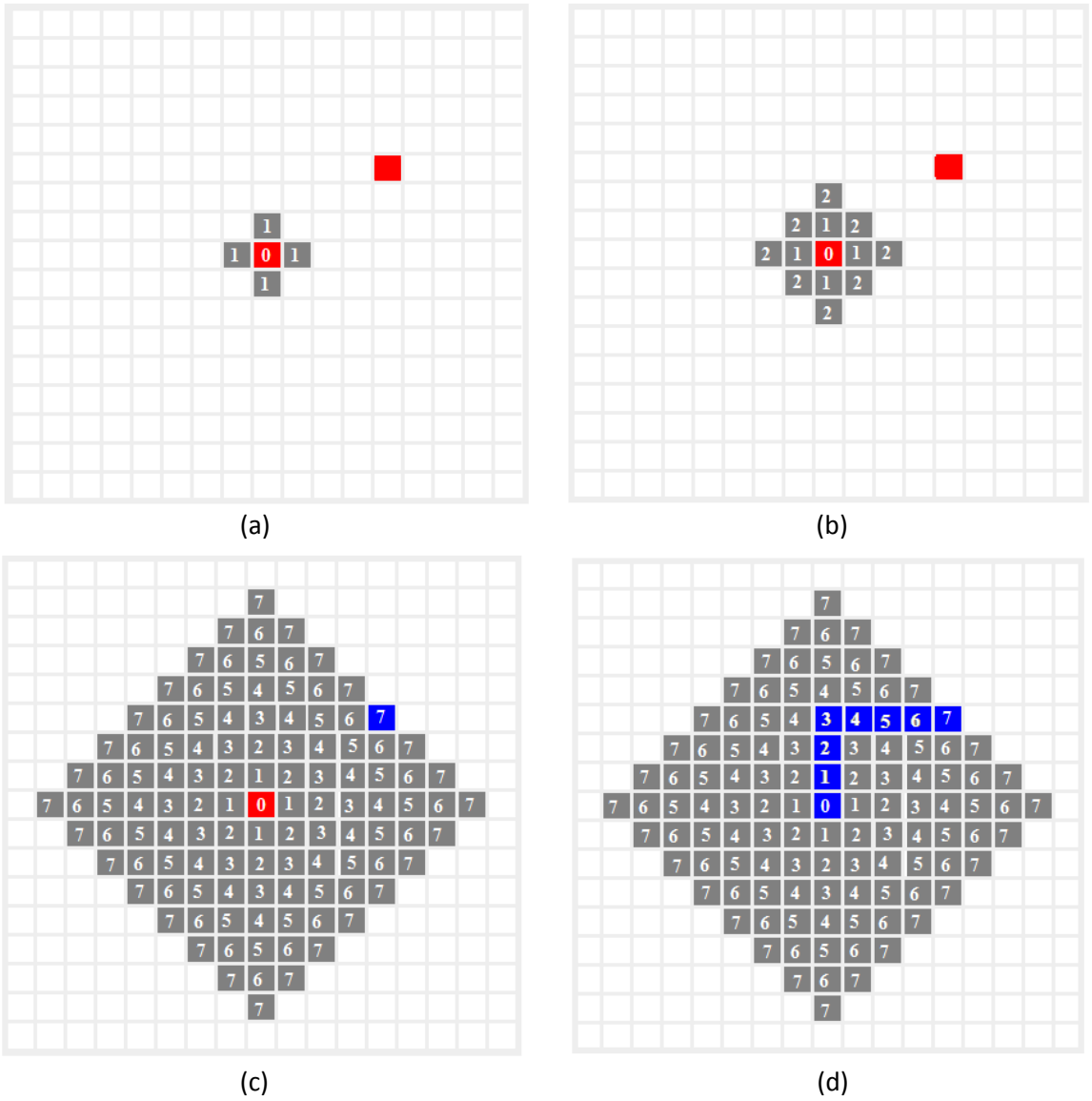

Figura 9 – Alguns passos da execução do algoritmo de Lee. Em vermelho estão os pontos de origem e destino. (a) Os quatro vizinhos de *s* são marcados. (b) Os vizinhos dos vizinhos de *s* são marcados. (c) O destino é alcançado. (d) O caminho é encontrado.

3.2 Algoritmo A*

O algoritmo de Lee expande sua busca em todas as direções, incluindo as direções contrárias ao destino. Em situações onde a área roteada não está congestionada, a expansão da busca em todas as direções torna-se desnecessária. Esse comportamento faz com que o algoritmo de Lee possua um elevado tempo de execução e consumo de memória. Para melhorar isso, Rubin (1974) aplicou a meta-heurística A* (HART, 1968) no algoritmo de Lee. Apesar de o A* ser uma meta-heurística da inteligência artificial, ele é comumente denominado, no roteamento de circuitos, como um algoritmo propriamente dito, que consiste um uma

implementação do algoritmo de Dijkstra utilizando uma heurística para atribuir custos mais apropriados para os nós, fazendo com que a busca convirja mais rapidamente para o destino. No algoritmo de Lee, o custo de um nó era o custo do caminho entre o nó e o ponto de origem. No algoritmo A*, o custo de um nó é dado pela função $f(n) = g(n) + h(n)$, onde $g(n)$ retorna o custo conhecido do nó n até o ponto de origem e $h(n)$ é uma função heurística que retorna a distância estimada do nó n até o ponto de destino. Sendo assim, a interpretação do custo de um nó representa o custo do caminho que parte da origem, passa pelo nó em questão, e chega ao destino. Como a função h estima o custo do nó até o destino, o custo total, dado por f, é denominado custo potencial de n. A função h é comumente implementada pela distância de *Manhattan* (em um espaço 2D), dada por $manh((x_1, y_1), (x_2, y_2)) = | x_1 - x_2 | + | y_1 - y_2 |$. A distância de *Manhattan* é a menor distância possível entre dois pontos em um grafo de grade. No caso do roteamento de circuitos, considerando que o *grid* é tridimensional, e que pode possuir pontos de custos variáveis (especialmente no roteamento global), além de *layers* com custos diferentes, vias com custos próprios e desalinhamento de *layers*, a distância de *Manhattan* se torna uma estimativa muito pobre, podendo até falhar (isto é fazer com que o algoritmo não encontre o melhor caminho), no caso do desalinhamento de *layers*, exigindo um mapeamento do *grid* para coordenadas reais, para correção. Com isso, são utilizadas estimativas que representam melhor a realidade. Cabe ressaltar que a heurística está na redução do espaço de busca do algoritmo, não influenciando em sua optimalidade, desde que a função h seja otimista, isto é, retorne o menor custo possível entre dois pontos.

O pseudocódigo do algoritmo A* é apresentado na Figura 10. Inicialmente o nó de origem, gerado sobre *s*, é adicionado no conjunto *O*. Os nós contidos nesse conjunto são chamados de nós abertos. Esse conjunto representa a fronteira do espaço de busca, isto é, os nós disponíveis a serem avaliados. O conjunto *O* é implementado por uma fila de prioridade (como um *heap*), visto que a cada iteração do algoritmo o nó de menor custo é obtido (linha 4). Se o destino foi alcançado, o caminho é retornado. Caso contrário, *n* é marcado como um nó visitado (ou fechado). Essa marcação tem como objetivo impedir que mais de um nó referente ao mesmo ponto seja adicionado em *O*. Em seguida ocorre o processo de expansão do nó *n* (linhas 7 - 9). Para cada ponto adjacente ao ponto de *n*, é gerado um nó *v* e se esse nó não for um nó já visitado nem possuir um correspondente *v'*, em *O*, de custo

menor ou igual ao custo de *v*, o nó *v* é adicionado em *O*, e *v'* é removido, caso exista. Uma boa maneira de lidar com a condição da linha 8 é utilizar um a estrutura que armazena todos os nós envolvidos na busca (abertos, isto é, em *O*, e fechados). Assim com apenas uma consulta nessa estrutura, que pode ser implementada por um mapa (de pontos para nós) *hash*, é possível avaliar a condição da linha 8.

A Figura 11 mostra uma comparação do espaço de busca do algoritmo A* (a) e do algoritmo de Lee (b). Note que se a função *h* do A* retornar 0, o algoritmo se comporta exatamente igual ao algoritmo de Lee. É evidente a grade diferença do espaço de busca entre os algoritmos. Obtendo os nós de custo potencial igual a 7, o algoritmo rapidamente convergiu para o destino, enquanto que o algoritmo de Lee perde tempo de processamento realizando ondes de expansão de busca em todas as direções. A heurística do A* possibilita que o algoritmo procure primeiro onde for mais promissor, implicando em um grande ganho em desempenho.

```
1  A* (Ponto s, Ponto t)
2      O ← {s}
3      while O ≠ ∅
4          n ← Nó de menor custo em O; O ← O – {n}
5          if destino alcançado: return path(n)
6          Marca n como fechado
7          for each vizinho v de n :
8              if v não está fechado and (v' ∉ O or c(v) < c(v')) :
9                  O ← O ∪ {v}; O ← O – {v'}, caso v' exista
10     return "Não existe caminho s-t"
```

Figura 10 – Pseudocódigo do algoritmo A*. A função *c*(*v*) representa o custo do nó *v*. A variável *v'* representa um nó de ponto igual a o nó *v*.

Outra consideração importante a respeito do algoritmo A* é o critério de desempate de nós com o mesmo custo. A maneira que o algoritmo decide a prioridade desses nós pode afetar drasticamente no desempenho do algoritmo. Note que na Figura 11a existem nós de custo 7 não visitados. Dependendo do critério de desempate, esses nós poderiam ser expandidos, assim como todos seus sucessores, formando um retângulo de pontos visitados, o que aproximaria o espaço de busca do espaço do algoritmo de Lee. Uma boa escolha é priorizar nós com o valor de *g* maior, ou nós de valor *h* menor. Dessa forma, nós que estão mais distantes do ponto de origem, e consequentemente mais próximos do destino, serão

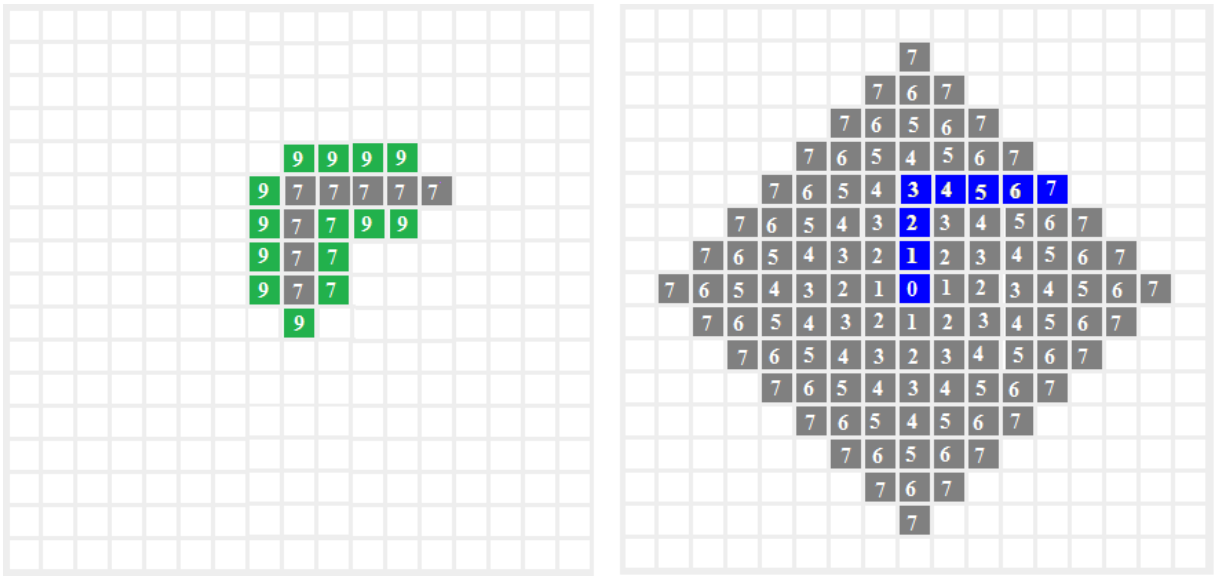

Figura 11 – Comparação do espaço de busca do algoritmo A* (a) e do algoritmo de Lee (b). Em (a) os pontos em cinza são os nós fechados, e os verdes são os abertos (contidos em *O*). Os números representam os custos dos nós. Em (a), a função *h* é implementada pela distância de *Manhattan*.

escolhidos primeiro. Isso dá ao algoritmo um aspecto semelhante à busca por profundidade em casos de desempate, pois a busca converge para o destino seguindo sempre o mesmo caminho, como ilustrado na Figura 11a. A medida que os nós foram adicionados no conjunto de abertos, o algoritmo escolheu os nós de menor custo que se encontravam mais próximos do destino, isto é, os últimos nós adicionados. No entanto, existem nós com o mesmo custo e com o mesmo valor de *g* e *h*. Neste caso, é necessário manter algum padrão em relação a ordem desses nós. Por exemplo, a prioridade pode ser dos nós adicionados por último na fronteira. Na Figura 11a, o nó acima de um nó expandido é o último dos quatro nós vizinhos a ser adicionado em *O*. Dessa forma, a cada iteração, é escolhido um nó situado acima do último nó expandido, até que o nó possua um custo maior. Em seguida, nós situados a direita do último nó expandido são selecionados até o caminho ser encontrado. Se o algoritmo não definir um critério de desempate de nós com o mesmo custo nem para nós com o mesmo valor de *g*, a busca pode tomar um aspecto de amplitude de forma desnecessária. Neste caso, todos os nós contidos no retângulo imaginário que une os pontos de origem e destino poderão ser visitados. Supondo que as dimensões desse retângulo sejam *m* x *n*, a complexidade do algoritmo no melhor caso seria *O*(*mn*). Utilizando as técnicas para definir prioridades

discutidas acima, a complexidade no melhor caso é de $O(m+n)$. Cabe ressaltar que, no pior caso, de qualquer forma a complexidade do algoritmo é de $O(mn)$.

Como mencionado anteriormente, é desejável que o algoritmo de busca a ser utilizado no roteamento detalhado possa lidar com múltiplos pontos de origem e destino. O algoritmo A* suporta esse "requisito" de forma natural, sem grandes modificações no algoritmo original. O algoritmo recebe dois conjuntos de pontos *S* e *T*, representando os pontos de origem e destino, respectivamente. Agora, cada nó possui uma referência para o seu ponto de destino atual. Na criação do nó, e durante a execução do algoritmo, essa referência é atualizada. O critério para se escolher um alvo é o custo estimado do nó ao alvo, utilizando a função *h*. Se o custo de um nó recém criado, utilizando ao alvo do nó pai é igual ao custo do pai, não é necessário atualizar o alvo. Caso contrário, se o custo aumentou, a atualização se faz necessária. No início da execução (linha 2), todos os nós de origem são adicionados em *O*. O resto da execução se dá normalmente, considerando apenas que todos os testes que envolviam o alvo da busca agora se referem ao alvo de um nó específico, como o cálculo da função *h* e o critério de parada do algoritmo.

3.3 Busca por Linha

A busca por *maze* possui a vantagem de garantir que o caminho ótimo seja encontrado, porém, o alto tempo de processamento e consumo de memória podem ter um grande peso durante a escolha de um algoritmo de roteamento. Mesmo o algoritmo A*, com sua grande redução de nós expandidos, enfrenta o mesmo problema, pois sua complexidade depende da distância entre os pontos de origem e destino. Com isso, os algoritmos de busca por *maze* podem se tornar inviáveis para *grids* enormes. Para superar esse problema, Mikami e Tabuchi (1968) desenvolveram o primeiro algoritmo de roteamento que utiliza intervalos (ou segmentos de reta) ao invés de pontos. Ao invés de percorrer todos os pontos em uma linha, o algoritmo cria uma linha que se estende até encontrar um obstáculo ou até encontrar o final do espaço de roteamento. Com isso, a complexidade desta classe de algoritmos não depende da distância entre os pontos de origem e destino, e sim do número de linhas criadas. Em contrapartida, de forma geral, os algoritmos desta classe não garantem que o melhor caminho seja encontrado.

O procedimento básico do algoritmo de Mikami e Tabuchi consiste em, primeiramente, adicionar na busca os pontos de origem e destino. Esses pontos são chamados de pontos base. Inicialmente, são criadas duas linhas perpendiculares que interceptam os pontos de origem e destino. Essas linhas são chamadas de linhas do nível 0. As linhas são estendidas até encontrar um obstáculo ou até alcançar o limite do espaço de roteamento. Se uma linha proveniente do ponto de destino interceptar uma linha do ponto de origem, o caminho é encontrado. Se isto não ocorrer, são criados novos pontos base sobre as linhas geradas. A cada iteração, para cada ponto base, uma linha do nível *i* (perpendicular as linhas do nível *i* -1) é criada sobre o ponto. Quando todos os pontos das quatro linhas iniciais forem ocupados por pontos base, alguma linha gerada é selecionada para que novos pontos base sejam criados sobre a linha. A Figura 12 mostra um exemplo da execução do algoritmo de Mikami e Tabuchi. Os pontos marcados com um “x” representam os pontos base. Os números são referentes ao nível das linhas. O algoritmo encontrou o caminho ao detectar a intersecção (denotada pelo “x” dentro de um círculo) de uma linha de *s* com uma linha de *t*.

Outro algoritmo que trabalha de forma muito similar ao algoritmo de Mikami e Tabuchi é o algoritmo de Hightower (1969). O diferencial do algoritmo é que ele considera apenas as linhas que não são bloqueadas por obstáculos. Além disso, cada linha pode possuir no máximo dois pontos base. Devido a essa restrição, o algoritmo de Hightower não garante que algum caminho seja encontrado, caso ele exista. Para lidar com isso, é necessário utilizar técnicas de *backtracking* para poder escolher os pontos base apropriados para que o caminho seja encontrado. No entanto isto acarreta em um aumento no tempo de processamento, que pode ser quase tão alto quanto o tempo do algoritmo de Lee (CHEN, 2016).

Uma última consideração sobre os algoritmos de busca por linha é o *grid* em que eles operam. Para usufruir do desempenho que o uso de linhas viabiliza, é necessário usar um *grid* que armazene segmentos, não pontos. De nada adiantaria o algoritmo gerar um segmento de tamanho *n*, consultando *n* posições do grid, para saber se o segmento intercepta algum obstáculo. Sendo assim, no caso do roteamento em circuitos, como cada *layer* possui uma direção preferida, o *grid*, em um dado *layer*, pode ser implementado por um vetor que armazena em cada posição uma lista ordenada de segmentos (seguindo a direção preferida do *layer*), cada uma referente a uma linha ou coluna. Assim, uma consulta no *grid* é resolvida

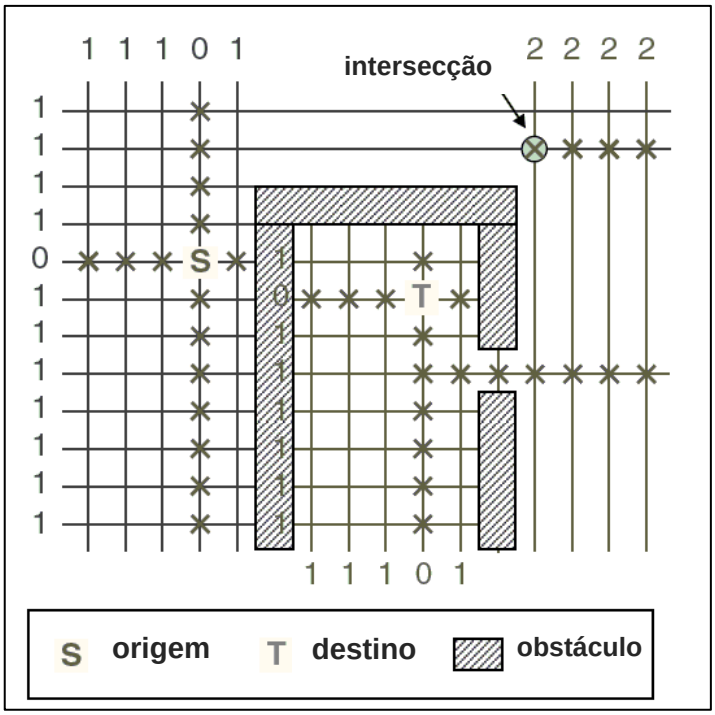

Figura 12 – Exemplo do uso do algoritmo de Mikami.

eficientemente em tempo O(*log n*), onde *n* é o tamanho da lista consultada. Em um problema de busca mais genérico (não se tratando do roteamento em circuitos especificamente), sem direções preferidas, pode se utilizar duas estruturas de vetores para cada plano, uma para cada direção. Isso aumenta o custo de se adicionar um segmento ao *grid*, pois um segmento de tamanho *n* vertical representa *n* segmentos de tamanho 0 horizontais. Alternativamente, pode-se manter apenas um vetor e pagar *n* consultas para um segmento de tamanho *n* com direção oposta a direção preferida. O primeiro caso otimiza as consultas durante a execução do algoritmo, aumentando o custo de construção do *grid*, enquanto o segundo caso, mantem um custo de construção baixo e aumenta o custo de consulta.

3.4 Algoritmo de Hetzel

Considere a Figura 13. Em (a), é mostrado o caminho *s-t* resultante da aplicação do algoritmo A*. Inicialmente a busca optou por seguir o caminho de cima, expandindo 4 nós para cima, seguido de 5 expansões para a direita, resultando em um caminho de 2 segmentos. Note que o custo dos nós em cada segmento jamais variou. Com isso, Hetzel (1998) percebeu que não era necessário marcar cada nó

pertencente a uma sequência com custos iguais. Se uma sequência de nós, pertencentes a mesma reta possui o mesmo custo (Figura 13b), esses nós são redundantes para a busca, podendo serem fundidos em um conjunto que os contém, o qual é marcado com o valor dos nós (Figura 13c). Esse conjunto representa um segmento, ou intervalo. No melhor caso, a abordagem proposta por Hetzel pode apresentar um aumento de desempenho extremo, pois independente da distância entre *s* e *t*, o algoritmo encontra o caminho em tempo constante, utilizando apenas dois segmentos, enquanto que o A* é afetado pela distância. No pior caso, o algoritmo se comporta como o A*. O caminho ótimo é garantido. Com isso, o algoritmo de Hetzel pode ser visto como um híbrido das buscas por *maze* e por linha.

A Figura 13 teve como objetivo apresentar a estratégia de unir conjuntos de pontos em segmentos, os quais são marcados com os valores dos nós contidos, e para isso usou um exemplo em um cenário bidimensional, sem grandes comprometimentos com a mecânica do algoritmo em si. Como o algoritmo foi proposto para um escopo 3D, levando em consideração as direções preferidas do *grid*, é relevante apresentar o algoritmo com suas peculiaridades. O pseudocódigo do algoritmo é o mesmo do A*, com algumas interpretações diferentes. Igualmente, sua implementação pode ser realizada de forma semelhante, mantendo-se a estrutura dos nós. Cada nó está associado a um segmento que parte do nó e se estende até se alinhar ao destino, ou até encontrar um obstáculo, seguindo a direção preferida do *layer* o qual o nó pertence.

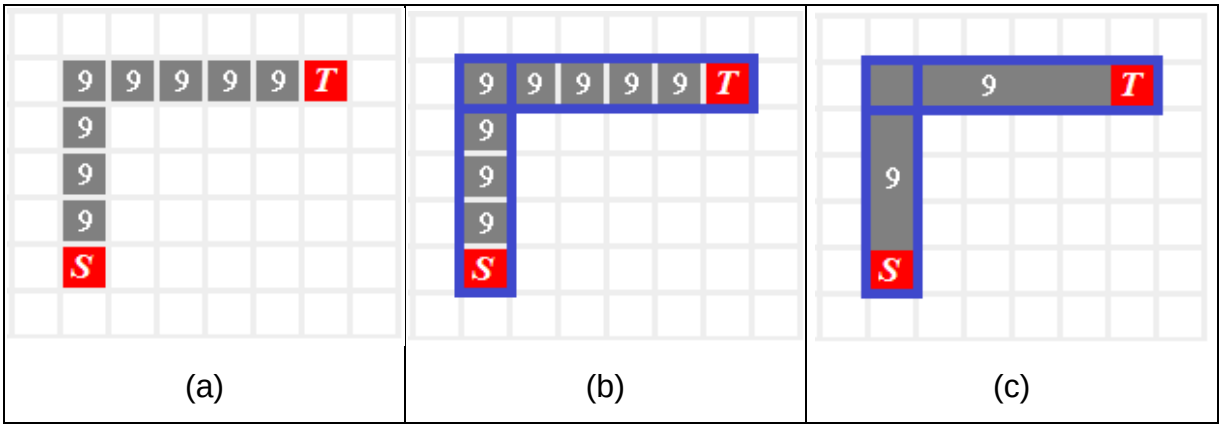

Figura 13 – Ilustração do princípio de fundir um grupo de nós redundantes em intervalos. Os pontos *s* e *t* representam a origem e o destino, respectivamente. Os custos se dão como na Figura 11a. (a) Caminho resultante da aplicação do algoritmo A*. Dois grupos de nós (b) redundantes são fundidos em dois intervalos (c), marcados pelo custo dos nós.

A Figura 14 mostra um exemplo da expansão de segmentos. Nesse cenário, o segmento correspondente ao nó localizado sobre *s* é facilmente obtido pelos pontos (x_s, y_s) e (x_t, y_s). Em seguida deve haver uma consulta no *grid* para se averiguar se não há algum obstáculo, exigindo a correção do segmento, o que é o caso da figura. Quando um segmento é expandido, ele gera nós em pontos vizinhos ao segmento. Caso os vizinhos se encontrem no mesmo *layer* do segmento expandido, somente um nó para cada segmento vizinho é gerado. Contudo, gerar todos nós adjacentes em *layers* diferentes pode ser desnecessário e custoso, como no caso da Figura 14a. Para evitar isso, o segmento expandido pode se limitar em gerar apenas o nó não aberto mais próximo da extremidade do segmento, como ilustrado na Figura 14b. Seguindo esta abordagem, é necessário manter o segmento expandido em *O* até que todos os nós vizinhos tenham sido gerados (note que o segmento permanece verde na figura).

Da mesma forma que no A*, é necessário verificar se os novos segmentos gerados podem de fato ser abertos (linhas 8, Figura 10). Para isso, é necessário uma estrutura que armazene todos os segmentos que participaram da busca (abertos e fechados). Essa estrutura pode ser implementada pela mesma estrutura que implementa o *grid*, pois para definir se um segmento pode ser aberto, é preciso verificar se ele intercepta outro segmento que já participou da busca. Se o ponto de origem (ponto do nó) do segmento *v*, candidato a aberto, estiver contido em um segmento *v',* fechado ou aberto de menor ou igual custo, *v* não é aberto. Caso *v'* tiver custo maior, ele é substituído por *v*, que passa a ser aberto. Caso *v* intercepte algum *v'*, fechado ou aberto de menor ou igual custo, sem que seu ponto de origem esteja contido em *v'*, *v* é corrigido para não haver intersecção, e é aberto normalmente. Se *v'* possuir custo maior que *v*, ele é substituído por *v*, que passa a ser aberto.

Quando proposto em (HETZEL, 1998), o algoritmo possuía algumas restrições, trabalhando apenas com a distância de *Manhattan* como função *h*, pois não considerava o desalinhamento de *layers* nem os diferentes custos de fios. Mais tarde Humpola (2009) fez o algoritmo considerar o desalinhamento de *layers*, e Peyer (2009) o generalizou ainda mais, permitindo custos de fios variáveis em diferentes *layers*, além de generalizar o algoritmo para utilizar quaisquer conjuntos de pontos (não apenas intervalos), sendo chamado a partir de então de *GeneralizedDijkstra*. A generalização para conjuntos arbitrários de pontos permitiu o

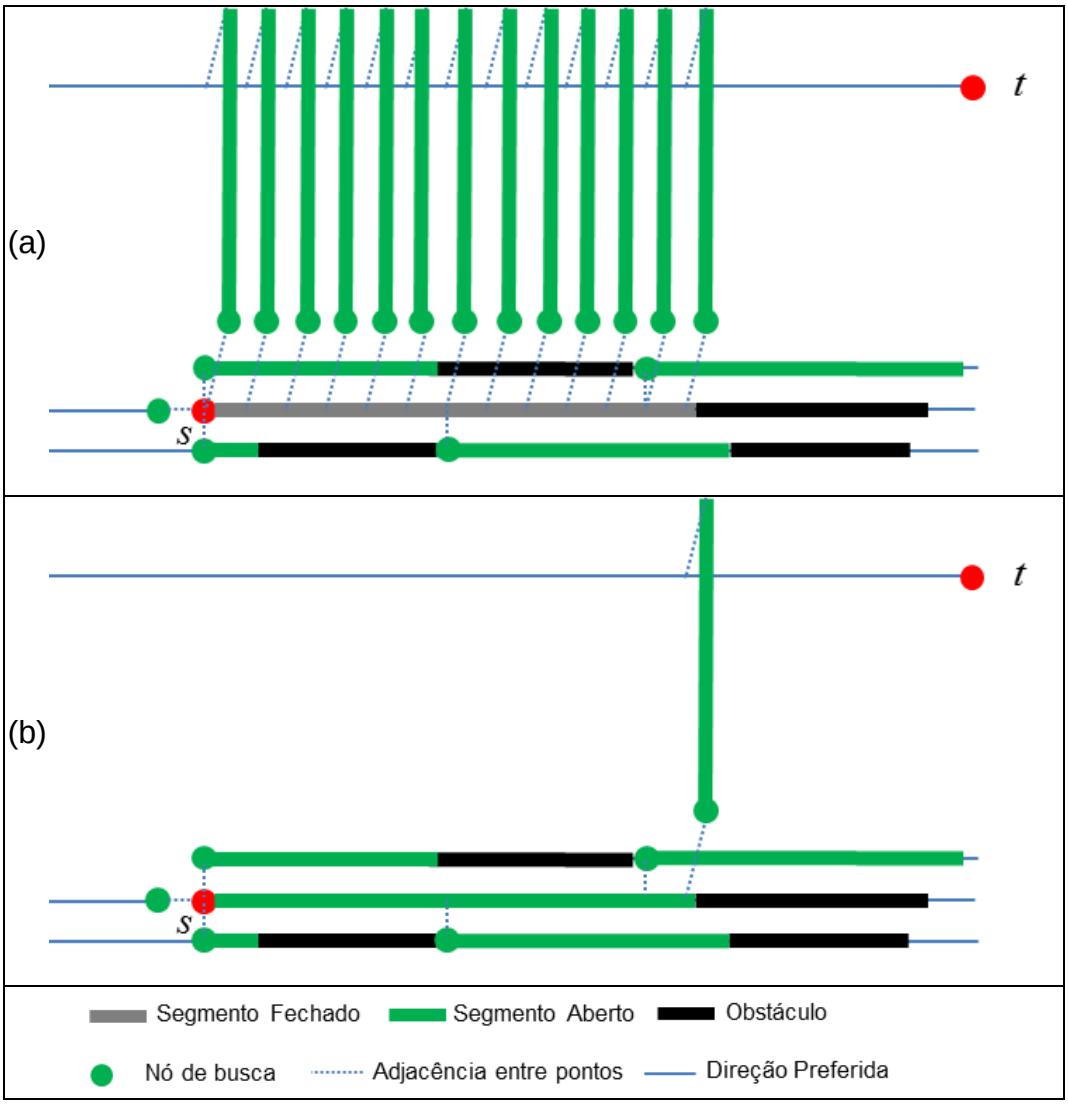

Figura 14 – Ilustração da expansão de segmentos. Os pontos *s* e *t* representam os pontos de origem e destino, respectivamente. Em (a), o segmento cinza é expandido gerando/abrindo todos os vizinhos. Em (b), apenas o vizinho (do *layer* adjacente) não aberto mais próximo da extremidade é gerado.

algoritmo ser executado utilizando retângulos. Assim, o algoritmo pôde ser utilizado para realizar um pré-processamento, no túnel do roteamento global, afim de analisar a área, calculando custos estimados de regiões para serem utilizados na função *h*, no roteamento propriamente dito. O roteamento em si era executado pelo *GeneralizedDijstra* utilizando intervalos, que vem a ser praticamente o algoritmo proposto por Hetzel (1998), com pouquíssimas modificações, mantendo a mecânica geral intacta. Por este motivo, o algoritmo mencionado neste trabalho como "algoritmo de Hetzel" representa a versão generalizada utilizando intervalos. Considerando os algoritmos de busca de caminho mais genéricos, mas que são capazes de lidar com as regras de projeto básicas, podendo ser utilizados dentro do

roteamento detalhado, o algoritmo de Hetzel pode ser considerado o estado da arte, visto que é ótimo e é muito mais rápido que o algoritmo A*. Segundo Gester (2012), a técnica de utilizar segmentos resulta em um aumento de velocidade, que é no mínimo 6 vezes mais rápido. Em um escopo mais abrangente, o algoritmo A* continua com grandes aplicações, como por exemplo no roteamento global, visto que lida facilmente com *grids* de custos variáveis por *layer* (no algoritmo de Hetzel, em cada *layer* o custo de uma conexão seguindo a direção preferida deve ser constante).

3.5 Algoritmo ST-Router

O algoritmo funciona em um espaço bidimensional, permitindo o cruzamento de fios ortogonais. O algoritmo pode ser considerado como de busca por linha guiado por custos. O *grid* utilizado é o mesmo dos algoritmos de busca por linha. A diferença é que são utilizados apenas dois planos (com direções preferidas ortogonais), e a distância em ambos é 0. O ST-Router ainda se distancia de uma aplicação prática no roteamento em circuitos integrados. Contudo, é relevante apresentar este algoritmo, visto que o algoritmo proposto neste trabalho (SG-Router) é uma adaptação do ST-Router para o espaço tridimensional.

A motivação da criação do algoritmo ST-Router parte de uma deficiência que o algoritmo de Hetzel possui ao lidar com áreas congestionadas. Como mencionado anteriormente, no pior caso, o algoritmo de Hetzel se comporta como o A*. A Figura 15 ilustra esse caso (para um escopo 2D), onde existe um obstáculo que bloqueia completamente qualquer rota com o menor caminho possível, exigindo que o algoritmo realize expansões em todas as direções para superar o obstáculo. Como os segmentos provenientes de expansões que se afastam do destino são pontos, a busca se comporta como a do A*, para estes casos. Em bege, é ilustrada a área de busca do algoritmo de Hetzel. Contudo, para que o obstáculo seja superado, não é necessário que a busca avance em todas as direções. Se o algoritmo, ao encontrar o obstáculo, tentar contorna-lo por ambos os lados (setas vermelhas na Figura 15), verificando qual lado é o mais promissor, com base no mesmo cálculo de custo, o menor caminho será encontrado. É evidente a grande diferença entre o espaço de busca do algoritmo de Hetzel e da estratégia mencionada.

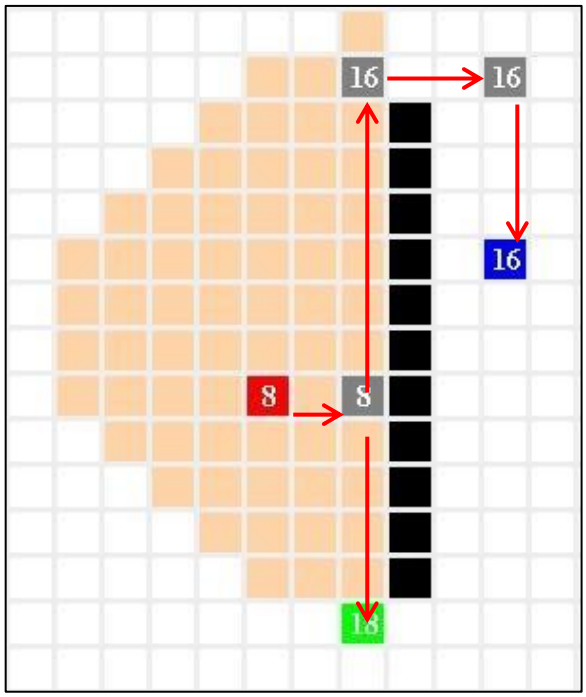

Figura 15 – Ilustração do exemplo motivador da criação do ST-Router. Espaço de busca do algoritmo de Hetzel (em bege). Em vermelho e azul, estão localizados os pontos de origem e destino, respectivamente. As setas vermelhas representam o espaço de busca utilizando a estratégia de se contornar o obstáculo. Os custos e cores seguem o padrão das figuras anteriores.

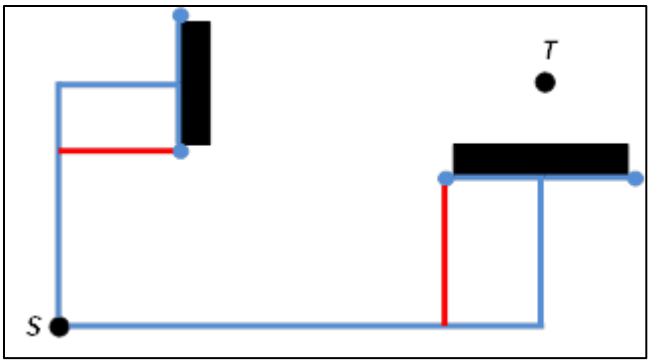

Figura 16 – Ilustração mostrando um exemplo de como a estratégia de se contornar obstáculos pode produzir caminhos de comprimento menor que o ótimo. Em azul, estão possíveis caminhos, até encontrarem obstáculos e os contornarem. Em vermelho os atalhos para os caminhos.

Apesar da estratégia de contornar obstáculos reduzir drasticamente o número de nós expandidos, ela pode gerar caminhos maiores que o ótimo, como ilustrado na Figura 16. Contudo, as linhas vermelhas mostram possíveis atalhos que, se utilizados, garantem que o menor caminho seja encontrado. Se toda vez que o algoritmo encontrar um obstáculo for realizada uma verificação para detectar atalhos e se a heurística conseguir realizar cada atalho, ela garante que o menor caminho seja encontrado, pois um atalho nada mais é do que um caminho alternativo de custo mínimo. Se o contorno de obstáculos gerar um caminho não ótimo que não

possa ser atalhado com um segmento (pela existência de obstáculos no caminho), o algoritmo pode ser chamado recursivamente para resolver o atalho. Assim, baseando-se no contorno de obstáculos e na realização de detecções de atalhos, o algoritmo ST-Router foi proposto em (GONÇALVES, 2014).

O fluxo geral do algoritmo (Figura 17) também é muito semelhante ao do A*. O algoritmo também trabalha com nós de busca, que utilizam o mesmo cálculo de custo do A*. Contudo, os nós possuem direções associadas (cima, baixo, direita e esquerda), que influenciam a maneira a qual são expandidos. Igualmente, existe um conjunto de nós abertos (*O*), e um nó é considerado fechado, ou visitado, quando não está contido em *O*. Como o algoritmo pode utilizar recursão para resolver atalhos, é necessário manter um histórico dos pontos de destino das buscas recursivas, para evitar laços recursivos. O conjunto de rotas *R* armazena essa informação.

```
STRouter(Ponto source, Ponto target, Conjuntos R, O)
1  if target ∈ R, return nil
2  R ← R ∪ {target}
3  initialize(O)
4  while O ≠ ∅
5      n ← Nó de menor custo em O
6      if n.point = target : break
7      visit(n)
8      expand(n)
9  if n.point = target : return n
10 else: return nil
```

Figura 17 – Fluxo geral do algoritmo ST-Router.

Existem dois tipos de nós: nós comuns e nós de contorno. Nós comuns podem ser definidos pela *n*-upla (*dir, point, parent*), onde *dir* é sua direção de expansão, *point* é o seu ponto e *parent* uma referência para o nó pai. Nós comuns têm como objetivo realizar um “salto” na busca, para alcançar o ponto mais próximo do destino (semelhantemente a criação de segmentos do algoritmo de Hetzel). Nós de contorno servem para contornar obstáculos. Podem ser definidos por (*dir, point, parent, detourDir*), onde *detourDir* é a direção do obstáculo. Com isso, um nó de contorno obrigatoriamente deve estar localizado em uma posição adjacente a um obstáculo.

O primeiro passo do algoritmo é verificar se o ponto de destino já não foi utilizado em alguma busca (linha 1). Caso esteja vazia, a lista de abertos é inicializada (linha 3) com nós comuns, apontando para *target* (Figura 19a). O laço principal segue o mesmo modelo do algoritmo A*, obtendo o melhor nó de *O*, verificando se é o alvo, marcando como visitado e o expandindo. As maiores diferenças começam com a função *expand* (Figura 18), que expande o nó de acordo com o seu tipo. A variável *nodeList* é uma lista temporária que armazena os futuros nós abertos, gerados pela expansão de *n*. Essa lista é necessária pois o custo dos nós será definido apenas após a verificação de atalhos, e o custo é uma das informações necessárias para se saber se um nó pode ser aberto. Após a expansão, é realizado um procedimento de atalhos em um dos nós (linha 4), enquanto os demais nós têm seu caminho atualizado de acordo com o caminho do nó que sofreu o procedimento (linha 5). Isto é possível apenas pelo fato de que todos os nós gerados pertencem ao mesmo ponto. Após o procedimento de atalhos, todos os nós são adicionados em *O*.

A expansão de nós comuns é definida na Figura 20. Primeiramente, é criado um segmento que parte do ponto do nó *n* e se estender até se alinhar ao ponto de destino. Como o objetivo é realizar um salto para o ponto alinhado ao destino, é necessário verificar se há algum obstáculo no meio do caminho. Para isso se realiza um consulta no *grid* passando o segmento criado. O *grid* retorna o ponto de intersecção *pt* do segmento com um obstáculo, caso haja algum. Se não há obstáculo, um nó comum é gerado, apontando para o destino (Figura 19b). Caso contrário, dois nós de contorno com direções de expansão ortogonais a *n* são gerados (passos 5 e 6, Figura 19c). Note que todo nó adicionado em *nodeList* é verificado anteriormente se não é um nó visitado. Assim como na condição do algoritmo A*, se um nó já foi visitado ele não pode ser aberto.

```
expand(Nó n)
1  if n é comum, expandCommonNode(n)
2  else expandDetourNode(n)
3  if nodeList está vazia, return
4  verifyShortcut(nodeList[0])
5  for each n ∈ nodeList: n.setParent(nodeList[0].parent)
6  for each node n' ϵ nodeList : O ← O ∪ {n'}
```

Figura 18 – Pseudocódigo da expansão de nós.

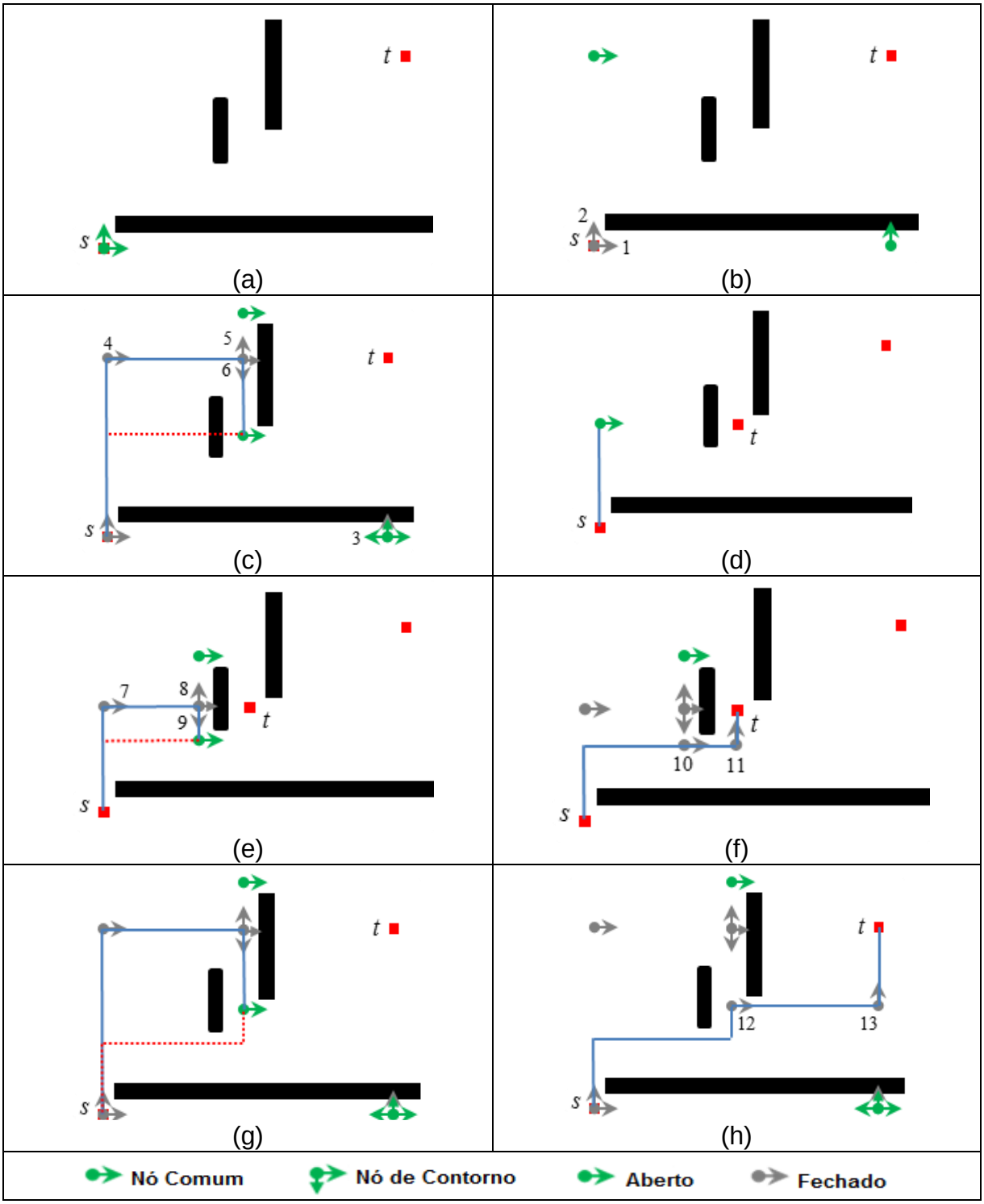

Figura 19 – Exemplo do funcionamento do algoritmo ST-Router. O nó de contorno é ilustrado por duas setas, sendo a maior a direção de expansão *dir* e a menor a direção do obstáculo *detourDir*. (a) *O* é inicializado adicionando-se dois nós comuns, apontando para *t*. (b) Ambos nós são expandidos (passos 1 e 2). (c) Ambos nós comuns são expandidos gerando nós de contorno. Os nós de contorno expandem (passos 5 e 6) gerando apenas nós comuns (condição da linha 3 da Figura 22 é falsa). O passo 6 cria um caminho comum atalho potencial. O segmento vermelho pontilhado é o segmento de atalho. (d) Uma chamada recursiva ocorre, após se ter adicionado o nó de atalho em *O*. (e) O passo 9 cria um caminho que possui um atalho. (f) O atalho é aplicado e o caminho é encontrado, retornando a busca para o nível anterior (g), e atualizando o pai do nó para este refletir o caminho otimizado. (h) As expansões de nós comuns finalmente chegam ao alvo *t*, retornando o menor caminho.

```
expandCommonNode(Nó n)
1  s ← Segmento que parte de n.point e se prolonga até se alinhar a target.
2  pt ← grid.intersectionPoint(s)
3  if pt ≠ nil :
4      Adiciona em nodeList dois nós de contorno localizados em pt, com direções
       dir ortogonais a n.dir, e direções detourDir iguais a n.dir
5  else Adiciona em nodeList um nó comum, localizado em s.p2, com direção dir
       apontando para target, onde s.p2 é o ponto de destino do segmento s.
```

Figura 20 – Pseudocódigo da expansão de nós comuns. O método *intersectionPoint* retorna o ponto livre adjacente ao primeiro obstáculo encontrado pelo segmento, ou *nil* caso não haja intersecção de *s* com algum obstáculo. $s.p_2$ representa o ponto de destino do segmento *s*.

A expansão de nós de contorno é definida na Figura 21. A cada iteração é verificado se a posição posterior (seguindo a direção de expansão) ao nó está ocupada. Se não está ocupada, é verificada a posição adjacente ortogonal à posição verificada. Esta posição está localizada na direção do obstáculo. Se a posição está ocupada, o algoritmo continua o procedimento. O algoritmo para quando o obstáculo contornado chega ao fim, ou quando encontra um novo obstáculo em sua direção de expansão. O procedimento realizado em ambos os casos é definido pela função *endDetour* (Figura 22). No primeiro caso, é gerado um nó de contorno apontando para a antiga direção do obstáculo. No segundo, é gerado um nó de contorno apontando para a direção contrária à direção do obstáculo. Se o nó de contorno aponta para o destino, também é criado um nó comum. Este passo é necessário pois sem ele a busca ficaria restrita às adjacências dos obstáculos. Idealmente, sempre deveria ser criado um novo nó de contorno, resultante da expansão de um nó de contorno. No entanto, como mostra a linha 3 da Figura 22, nem sempre um novo nó de contorno é criado. Essa condição existe pois o ato de criar sempre um novo nó de contorno pode prejudicar seriamente o desempenho, em alguns casos.

Com isso, foi adotada uma abordagem que utiliza uma heurística simples para decidir quando se deve adicionar um novo nó de contorno. Cabe ressaltar que esta abordagem evita uma grande queda de desempenho ao custo de perder a optimalidade no comprimento do caminho. Contudo, o aumento no comprimento do caminho é quase insignificante, enquanto que o aumento em desempenho é enorme. O critério de decisão de adição do novo nó de contorno funciona da seguinte maneira. Primeiramente é verificado se o nó comum pode ser criado. Caso negativo, o novo nó de contorno é gerado. Caso afirmativo, o nó de contorno é adicionado apenas se sua direção de obstáculo não apontar para o destino. A

```
expandDetourNode(Nó n)
1  s ← Segmento que parte de n.point e termina na posição
   adjacente, seguindo n.dir
2  do
3      pt ← grid.intersectionPoint(s)
4      if pt ≠ nil
5          endDetour(n, pt, true); return
6      s' ← Segmento que parte de s.p2 e termina na posição
       adjacente, seguindo n.detourDir
7      if not grid.intersects(s')
8          endDetour(n, s.p2, false); return
9      Prolonga s em uma unidade, atualizando s.p2
10 while true
```

Figura 21 – Pseudocódigo da expansão de nós de contorno. O método *intersects* retorna true se o segmento intercepta algum obstáculo.

```
endDetour(Nó n, Ponto p, Booleano interceptedFront)
1  if interceptedFront : n' ← new Nó de contorno (inverse(n.detourDir), p, n, n.dir)
2  else n' ← new Nó de contorno (n.detourDir, p, n, inverse(n.dir))
3  if not createCommonNode(n') or
   not pointsTo(n'.point, n'.detourDir, t) :  addNodeList(n')
```

Figura 22 – Pseudocódigo da finalização da expansão de nós de contorno. Note que o nó de contorno está definido conforma a *n*-upla definida no texto. A função *pointsTo*(p_1, *dir,* p_2) retorna true se o ponto p_1, utilizando a direção *dir*, aponta para o ponto p_2. A função *addNodeList* verifica se o nó foi visitado, e o adiciona em *nodeList* caso não tenha sido.

função *createCommonNode* verifica se um nó comum deve ser criado (e o cria, se for o caso), retornando verdadeiro caso afirmativo. O nó comum possui a mesma direção de expansão do novo nó de contorno. O nó comum pode ser criado somente quando a direção do nó de contorno aponta para o destino.

A estratégia de contornar obstáculos possibilita a geração de caminhos mais longos. Para corrigir isso, é necessário aplicar um método para corrigir o caminho. Após a expansão de um nó, o primeiro nó de *nodeList* é submetido a um procedimento de detecção e aplicação de atalhos. O procedimento, definido na Figura 23, e ilustrado com um exemplo naFigura 25, inicia percorrendo o caminho até que um segmento paralelo ao segmento *base* (*verifyShortcut*, linha 1) seja atingido. Em seguida, a direção do segmento paralelo deve ser comparada com a direção do segmento *base*. A direção de um segmento pode ser definida por uma seta que parte do seu ponto de origem p_1 e se estende ao ponto de destino p_2.

Lembrando que o ponto de origem de um segmento é o ponto de destino do segmento anterior e assim sucessivamente até se chegar na origem do caminho. Se as direções são opostas, o algoritmo procede sua execução. Se são iguais, o algoritmo para. Em raros casos, é possível encontrar um atalho quando as direções são iguais. No entanto, devido aos mesmos motivos da otimização de geração de um novo nó de contorno na função *endDetour*, a abordagem adotada pelo algoritmo permite a obtenção de um caminho não ótimo em troca de um grande aumento de desempenho. Para cada segmento paralelo à *base*, o algoritmo cria um nó de atalho (nós verdes na Figura 25). No final do processo, o nó de atalho mais promissor é escolhido para efetuar a atalho (linha pontilhada vermelha, Figura 25 e Figura 19c, e). Segmentos ortogonais à base, pertencentes ao caminho, também são verificados para a criação de um nó de atalho. Se um segmento perpendicular intercepta o segmento base, ou se seu sentido não aponta para *base*, um atalho deve ser criado. O primeiro caso implica em um segmento de atalho comprimento zero.

Após o nó de atalho mais promissor ser escolhido, o algoritmo tenta realizar uma conexão direta entre o segmento *base* e o nó de atalho. A função *doShortcut* (Figura 24) implementa essa funcionalidade. No primeiro passo, é gerado um segmento de atalho, utilizado em uma consulta no *grid*, afim de se saber se é possível realizar um atalho direto (linhas 2 e 3). Se não há obstáculos o atalho é realizado diretamente (linhas 8-10). Caso contrário, o mesmo procedimento que ocorre durante a expansão de um nó comum acontece, isto é, geram-se dois novos nós de contorno os quais são adicionados na fronteira, caso não visitados. Ignorar este passo pode permitir que o algoritmo encontre um caminho pior. Em seguida, é realizada uma chamada recursiva, cujo ponto de destino é o ponto de *n*. O ponto de origem sempre é o mesmo. AFigura 25c e d mostra o que pode ocorrer se o ponto de origem se localizar sobre o nó de atalho. O algoritmo pode encontrar um caminho ótimo local, mas no momento de anexar o caminho encontrado ao caminho preexistente, o caminho global não é ótimo. Para evitar isso, apenas se designa o ponto de destino e é passado para o algoritmo uma lista de abertos contendo o nó de atalho (linha 5, último parâmetro). Esse nó, por meio das referências dos nós antecessores (pais), guarda a informação do caminho. Assim, a busca recursiva tem acesso ao caminho global, podendo então detectar atalhos não previstos segundo a abordagem anterior (Figura 25e e f). Dentro da nova chamada do algoritmo *STRouter*, como *O* já contém um nó, ele não é inicializado.

```
verifyShortcut(Nó n)
1  base ← n.parentSegment()
2  n' ← n.parent.parent; s ← n'.parentSegment()
3  while dir(base) ≠ dir(s)
4      shortcut ← shortcutNode(base, s, n'); Atualiza s e n'
5      if s = nil : break
6      if s intercepta base : shortcut ← shortcutNode(base, s, n')
7      if n'.parent = nil and not pointsTo(s.p2, dir(s), base.p2) :
8          shortcut ← shortcutNode(base, s, n')
9      Atualiza s e n'; if s = nil : break
10 doShortcut(n, shortcut)
```

Figura 23 – Pseudocódigo da verificação de atalhos. O método *parentSegment* retorna um segmento que parte do ponto do nó pai ao ponto do nó chamador do método. A função *dir* retorna a direção de um segmento. A função *shortcutNode*(*base*, *s, n*) retorna um nó localizado sobre o segmento *s*, apontando para o segmento *base*, filho do nó *n*. A posição do nó retornado é a mais próxima de $s.p_1$ (inclusive). A atualização de *n'* refere-se a *n' ← n'.parent* e a atualização de *s* equivale a *s ← n'.parentSegment*().

```
doShortcut(Nó n, Nó shortcut)
1  base ← n.parentSegment()
2  s ← Segmento partindo de shortcut.point, que se estende até base
3  pt ← grid.intersectionPoint(s)
4  if pt ≠ nil :
5  Cria nós de contorno, como no caso de intersecção da expansão de nós comuns
6      n' = STRouter(source, n.point, R, {shortcut})
7      if n' ≠ nil : n.setParent(n'.parent)
8  else :
9      if n.point = s.p2: n' ← shortcut
10     else : n' ← Nó comum em s.p2, apontando para n.point, filho de shortcut
11     n.setParent(n')
```

Figura 24 – Pseudocódigo do procedimento de aplicação de atalho. O método *setParent* atribui um novo pai ao nó.

Como o algoritmo possui um passo recursivo, ele deve possuir uma base para a recursão. Na verdade, pode-se dizer que a recursão possui duas bases. A primeira é quando o algoritmo não detecta nenhum atalho em nenhum caminho. A segunda é quando são detectados somente atalhos que podem ser realizados em linha reta. Esses são os casos em que o algoritmo não prossegue na recursão. Ainda assim, existe a possibilidade de o algoritmo entrar em um laço recursivo quando existe uma chamada que visa encontrar um caminho cujo ponto de destino é o mesmo de uma chamada anterior. Para corrigir isso é utilizado o conjunto de rotas *R*, mencionado

no início da explanação do algoritmo, que contém todos os alvos das buscas realizadas.

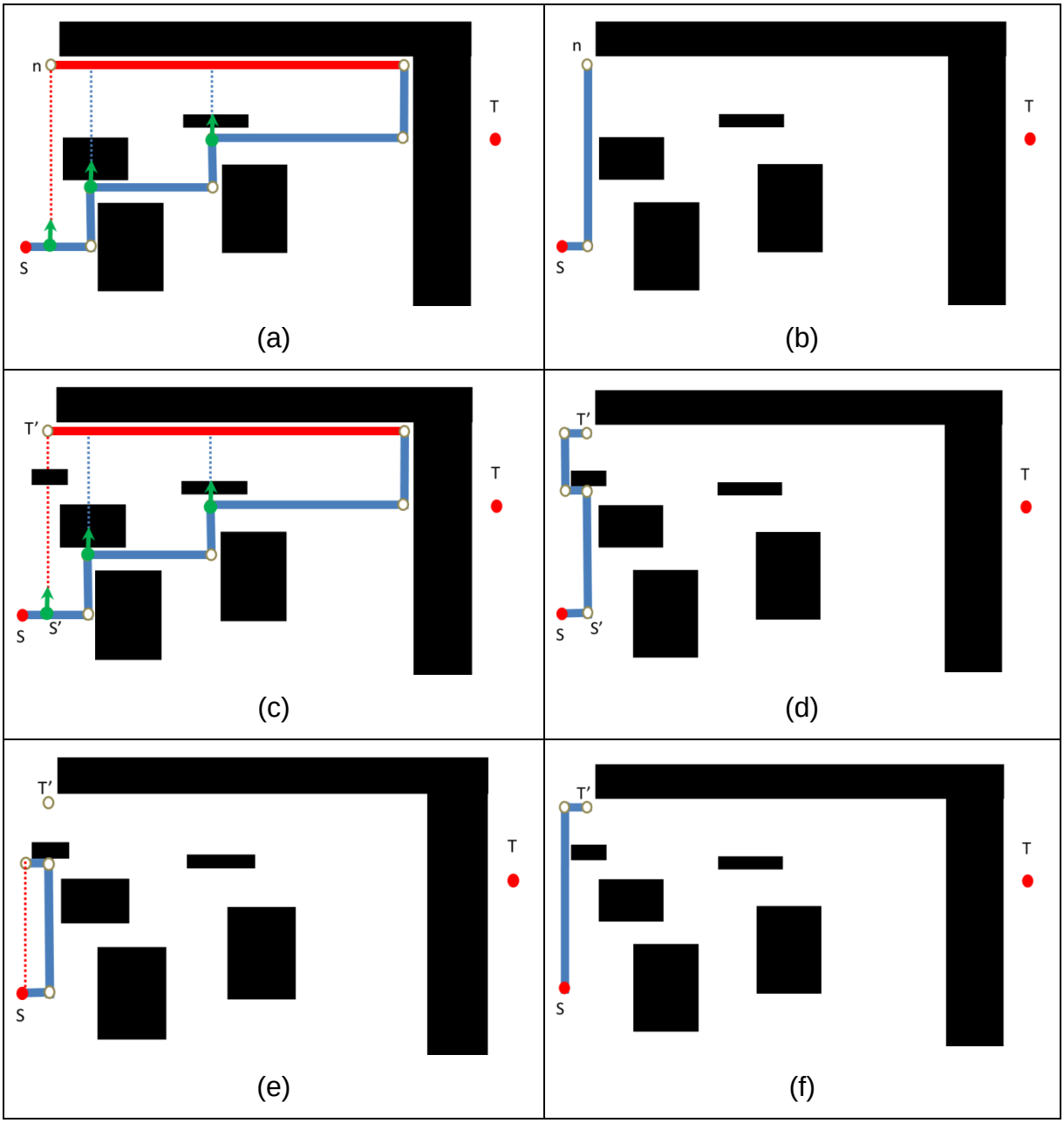

Figura 25 – Ilustração do procedimento de detecção de atalhos. Em vermelho (não pontilhado), o segmento *base*. As linhas azuis pontilhadas representam os possíveis segmentos de atalho considerados durante a busca. Os nós verdes representam os nós de atalho. Note que apesar de se mostrar três nós, em um dado momento da busca, existe apenas um nó de atalho, que é o último considerado. A linha pontilhada vermelha é o segmento de atalho mais promissor. (a) Melhor nó de atalho é escolhido e (b) o atalho é executado. (c) Existe um obstáculo bloqueando o segmento de atalho, demandando o passo recursivo. (d) Caminho não ótimo é encontrado, caso se limite a busca a um novo ponto de origem. (e) Passando o nó de atalho, inserido em *O*, para a chamada recursiva permite o algoritmo ter uma visão global do caminho dentro da chamada, encontrando o melhor caminho (f).

4 MELHORIAS PROPOSTAS SOBRE O ST-ROUTER

O algoritmo ST-Router, apresenta algumas características que fazem com que a optimalidade do resultado seja perdida. Essas características são apenas heurísticas, as quais podem ser removidas facilmente. Sem as heurísticas, em alguns casos, o algoritmo sofre uma grande queda de desempenho, podendo ser mais lento que o algoritmo de Hetzel. Com as heurísticas, o ST-Router possui um desempenho imensamente maior que o algoritmo de Hetzel, ao passo que o comprimento dos caminhos gerados é praticamente ótimo. Apesar do custo/benefício ser muito favorável, o comprimento de fios, para o roteamento em circuitos integrados, é um fator muito importante, pois reduz o custo do circuito, reduz o consumo de potência e o tempo de atraso do sinal. Com isso, esse capítulo tem como objetivo apresentar modificações no algoritmo, removendo as heurísticas mencionadas, fazendo com que o algoritmo deixe de ignorar caminhos melhores, além de apresentar propostas de otimizações de desempenho, para que a perda de desempenho ocasionada pela remoção das heurísticas seja atenuada. As modificações fizeram com que o algoritmo garantisse o caminho ótimo para todas as buscas realizadas e continuasse com um desempenho superior ao algoritmo de Hetzel. A respeito da garantia do caminho ótimo, não existe prova formal para essa afirmativa, porém, os resultados desse trabalho mostram que para absolutamente todos os roteamentos realizados, o comprimento de cada caminho obtido pelo ST-Router foi igual ao comprimento do caminho obtido pelo algoritmo de Hetzel, que é conhecido por ser ótimo.

4.1 Otimizações de Wire-Length

A primeira otimização de WL foi a remoção da condição de criação de nós de contorno, na função *endDetour*. Agora, sempre é criado um nó de contorno. A segunda otimização de WL foi a remoção da condição de interrupção do procedimento de detecção de atalhos. Antes, se o primeiro segmento paralelo à base possuía a mesma direção do segmento base, o procedimento terminava. Agora, o procedimento continua, possibilitando o encontro de algum atalho que antes não era encontrado. A terceira otimização de WL foi no conjunto *R*, necessário para evitar laços recursivos infinitos. Foi observado que cada elemento de *R* deve ser não apenas o destino de uma busca recursiva, mas o caminho encontrado. Na primeira versão, quando ocorria uma chamada recursiva com o destino igual a uma chamada anterior, o algoritmo simplesmente retornava falha. Porém, isso faz com que esse caminho seja abstraído, podendo levar o algoritmo a encontrar uma rota mais longa ao destino. Com isso, quando ocorre esta situação, a chamada recursiva retorna o caminho já conhecido e armazenado em *R*.

4.2 Modificação da Condição de Criação de Nós Comuns

A primeira proposta de otimização de desempenho diz respeito a condição de criação de um nó comum, na função *endDetour.* Na versão anterior, quando era possível criar um nó comum, este era gerado sobre o nó de contorno *n* e possuía a mesma direção. Agora, o nó comum é gerado apenas quando o nó *n* expande, gerando um novo nó de contorno *n'*, e a localização do nó comum é sobre o novo nó *n'* (Figura 26a). No entanto, se *n*' foi gerado de uma intersecção com um novo obstáculo (Figura 26c), não há motivos para se criar o nó comum, pois ele estará apontando para o obstáculo. Da forma anterior, o nó comum interceptaria o obstáculo logo a frente, gerando novos nós de contorno. Entretanto, isso é desnecessário, pois o nó *n* é suficiente para detectar essa intersecção, além de gerar apenas um novo nó de contorno, o que é menos custoso. Esse custo não é referente apenas ao custo de criação do nó, mas ao custo resultante de toda uma árvore de expansão proveniente de um nó. Um nó de contorno gera outro nó de contorno e possivelmente um nó comum, o qual frequentemente gera dois nós de contorno, e assim sucessivamente. Outro caso que poderia acontecer na primeira versão do algoritmo é quando o nó comum expande gerando outro nó comum que aponta para o obstáculo que *n* contorna. Este caso geraria uma intersecção e a

geração de dois nós de contorno, os quais contornariam o mesmo obstáculo contornado por *n*. Com a estratégia da nova versão esses problemas não

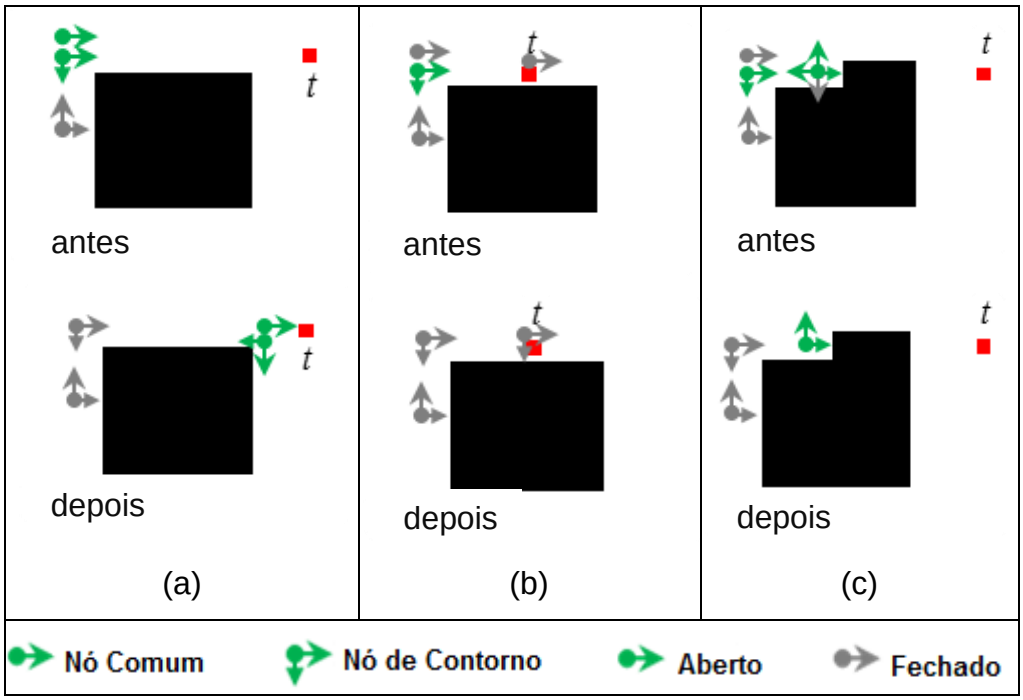

Figura 26 – Comparação da condição de criação de nós comuns na expansão de nós de contorno, entre a versão melhorada do ST-Router (depois) e a versão original (antes). Em (c) (antes), o nó comum expande e encontra um obstáculo logo à frente, criando nós de contorno. Em seguida, o nó criado, com direção para baixo, expande, gerando um nó de contorno, apontando para a esquerda. Em "depois" é ilustrado que isso não ocorre.

acontecem. Contudo, a nova estratégia demanda que haja um teste para verificar se o ponto de destino não intercepta o segmento entre n e n'. Da forma anterior, se o ponto de destino estivesse localizado sobre esse segmento (Figura 26b), o nó comum, localizado sobre *n* geraria outro nó comum sobre o destino. Ou ainda, se o destino estivesse no intervalo do segmento [n, n'], mas não estivesse sobre o segmento em si, o nó comum geraria outro nó comum apontando para o destino. Com a nova estratégia, não existe nó comum sobre *n* e, se o destino se encontra sobre [n, n'], ou em seu intervalo, a expansão de *n* passa direto pelo destino sem percebê-lo. Com isso, é necessário verificar se o destino encontra-se no intervalo durante a expansão e um nó de contorno. A modificação da condição de criação de nós comuns pode ser prejudicial em alguns casos. Se não houver um obstáculo à frente da direção de expansão do nó de contorno, como no caso (b) da figura, e se o obstáculo contornado for muito grande, esta modificação pode ser prejudicial, pois na abordagem original, o alvo seria alcançado mais rapidamente.

4.3 Padrões de Caminhos

A segunda otimização de desempenho é a introdução de padrões de caminhos. Os caminhos muitas vezes possuem padrões que podem ser explorados, evitando esforço computacional desnecessário. O primeiro padrão é ilustrado pela Figura 27. Para qualquer caminho, que se enquadre neste padrão em forma de escada, não é necessário aplicar a detecção de atalhos. Esta premissa é verdadeira devido ao fato de que o caminho não apresentar nenhuma volta, como ilustrado na Figura 27. Este padrão pode ser definido da seguinte forma. Após o primeiro segmento, tem-se *n* duplas de segmentos (*n* >= 0), sendo que o segundo segmento de cada dupla tem sempre sua direção igual a do primeiro segmento do caminho. Este padrão foi chamado de *ort*.

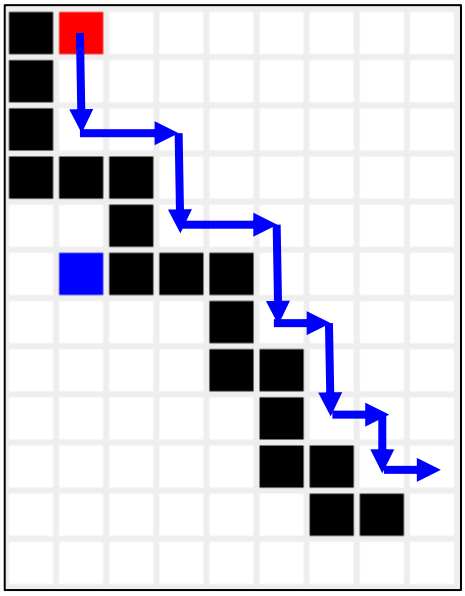

Figura 27 – Ilustração do padrão de caminhos em forma de escada. Os pontos em vermelho e azul são os pontos de origem e destino, respectivamente.

O segundo padrão é ilustrado na Figura 28. Este padrão pode ser definido por um caminho onde a primeira parte é definida pelo padrão *ort*, e a segunda parte também, porém, seguindo uma direção diferente da primeira, formando uma espécie de “arco” entre as duas partes. Este padrão foi chamado de *arc*. Na Figura 28, o caminho representa um caso específico do padrão *arc*, que pode ser explorado. Os

dois primeiros segmentos do caminho estão de acordo com o primeiro padrão. Em seguida, o próximo segmento tem a direção oposta a direção do primeiro segmento. A partir deste segmento, o caminho continua de acordo com o primeiro padrão, com formato de escada, sendo que cada segmento ortogonal ao primeiro possui sua direção apontando para a direção contrária do primeiro segmento. O último segmento do caminho deve ser paralelo ao primeiro, e estes devem estar desalinhados, de forma que não seja possível criar nenhum segmento de atalho. Sempre que um caminho com este padrão for submetido ao procedimento de detecção de atalhos, não é necessário percorrer o caminho inteiro, pois já é conhecido que o caminho será percorrido até a origem sem nenhum segmento de atalho, e será realizada uma chamada recursiva. Com isso, pode-se partir diretamente para a busca recursiva a partir da origem. Um caminho de padrão *arc* que foge deste caso específico não recebe nenhum tratamento especial. Um caminho que não é *ort* nem *arc* é chamado de *circ*.

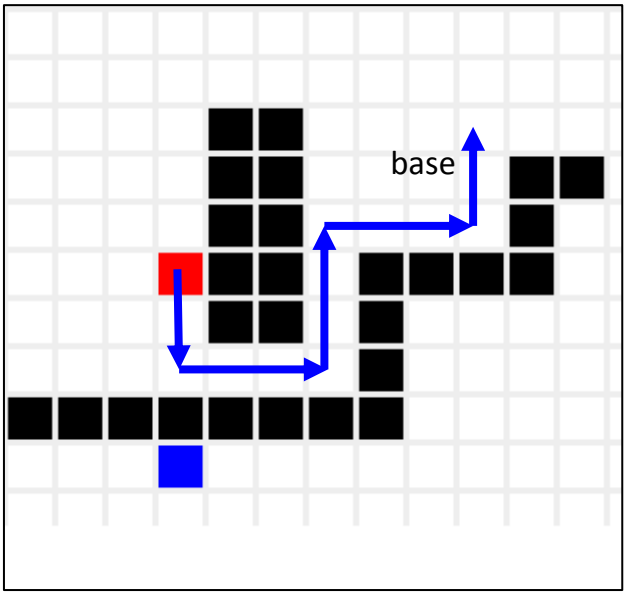

Figura 28 – Ilustração do segundo padrão, onde não é possível criar segmento de atalho.

A Figura 29 ilustra um caso específico do padrão *circ*, mas que também poderia ocorrer com o padrão *arc*. Este é um caso o qual não é esperado pela primeira versão do algoritmo. Neste cenário, a detecção de atalhos original seria interrompida no terceiro segmento. Como a segunda versão possibilita que o caminho seja percorrido até a origem, é necessário definir uma estratégia para

resolver esse caso. Na segunda versão da detecção de atalhos, seria gerado um segmento de atalho partindo do segmento em vermelho até a *base*. Como o

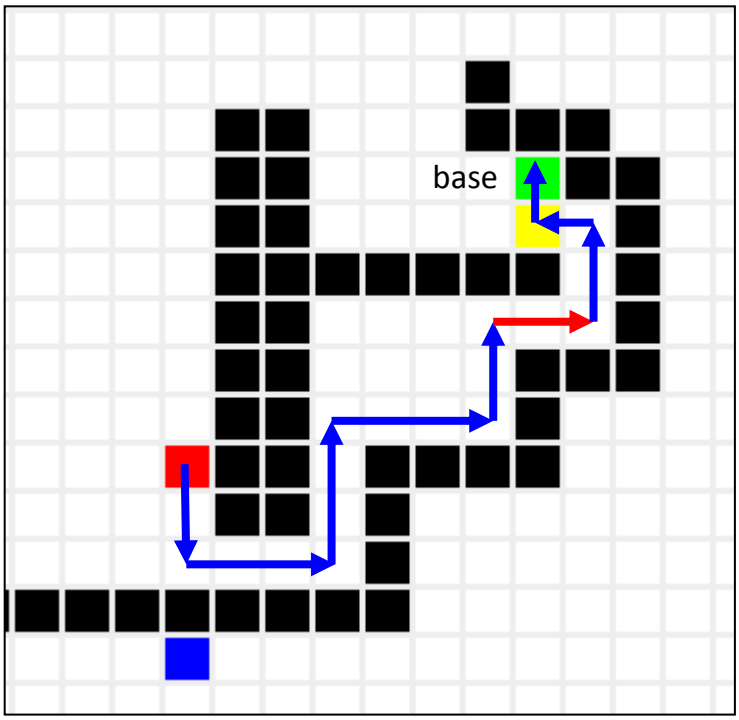

Figura 29 – Ilustração de um caso específico do padrão *circ*, que pode ser explorado partindo diretamente para a chamada recursiva.

segmento intercepta um obstáculo, uma chamada recursiva seria iniciada. No entanto, apesar de não ser o caso, o segmento em vermelho poderia ser menor que o segmento paralelo azul, tornando impossível a criação de um segmento de atalho, visto que ele não estaria alinhado à base. Com isso, com já é conhecido que o cenário não escapa de uma chamada recursiva, ao se encontrar o segmento vermelho é realizada uma chamada recursiva, passando *O* com um nó de origem criado sobre o segmento vermelho. Dessa forma, não é necessário existir alinhamento entre o segmento vermelho e o base.

Os padrões mencionados contribuem para reduzir o esforço computacional no procedimento de atalhos, evitando que o caminho seja percorrido desnecessariamente. Contudo, para implementar técnicas que levem em consideração esses padrões, é necessário saber o padrão de cada caminho submetido à detecção de atalhos. Para isso, cada nó deve conter uma informação adicional, referente ao padrão do seu caminho. A atualização da informação do padrão é realizada de forma incremental. A cada expansão, um novo nó é gerado, e é realizada uma conexão com seu nó pai, o qual contém uma informação de padrão.

Com base nessa informação e na posição do novo nó, seu padrão é calculado. Cabe ressaltar que os nós armazenam os padrões *ort, arc* e *circ* apenas, não fazendo discernimento dos casos específicos mencionados. Estes, são verificados na detecção de atalhos. A Figura 30 mostra o pseudocódigo da atualização de padrões de caminho, realizada a cada vez que o caminho é aumentado, isto é, a cada chamada do método *setParent* de um nó.

```
updatePattern(Nó n)
1   if n.parent = nill : n.pattern ← ort; return
2   if n.pattern = ort:
3       if orientation(n.fs) = orientation(n.parent.dir):
4           if dir(n.fs) = n.parent.dir : n.pattern ← ort
5           else  n.pattern ← arc
6       else if n.parent.parent.parent.dir = n.parent.dir :
7           n.pattern ← ort
8       else n.pattern ← arc
9   else if n.pattern = arc:
10      if orientation(n.fs) ≠ orientation(n.parent.dir) :
11          if pointsTo(n.parent.point, n.parent.dir, source) :
12              n.pattern ← circ
13          else n.pattern ← arc
14      else if  n.parent.dir = inverse(dir(n.fs)): n.pattern ← arc
15      else n.pattern ← circ
16  else n.pattern ← circ
```

Figura 30 – Pseudocódigo do procedimento de atualização do padrão do caminho de um nó *n*. A função *orientation* retorna a orientação (vertical ou horizontal) de um segmento (com base em sua direção), ou de uma direção propriamente dita. O campo *fs* representa o primeiro segmento do caminho. O campo *pattern* representa o padrão de caminho. A função *dir* retorna a direção de um segmento. A variável *source* é a origem da busca.

A Figura 31 apresenta o novo algoritmo de detecção de atalhos. É importante frisar que o pseudocódigo não apresenta detalhes de implementação, e em alguns casos pode omitir alguns detalhes, como avaliar se o próximo segmento obtido é nulo antes de dar continuidade no procedimento. A linha 3 verifica o padrão da Figura 28. As linhas 6 a 10 refletem a otimização de WL mencionada anteriormente, permitindo o algoritmo prosseguir mesmo quando o primeiro segmento paralelo a *base* possui a mesma direção da *base*. A linha 8 verifica o padrão da Figura 29. O teste da linha 12 agora é necessário pois o laço anterior permite que o segmento atual *s* e *base* não estejam alinhados.

```
verifyShortcut(Nó n)
1  if n.pattern = ort : return
2  base ← n.parentSegment()
3  if n.pattern = arc and base é paralelo a n.fs and not
   aligned(base, n.fs) and f(base, n.fs):
4      recursiveCall(source, n); return
5  n' ← n.parent.parent; s ← n'.parentSegment()
6  while dir(base) = dir(s)
7      Atualiza s, n'
8      if not pointsTo(s.p2, dir(s), base.p1) :
9          recursiveCall(n'.point, n); return
10     Atualiza s, n'
11 while dir(base) ≠ dir(s)
12     if aligned(base, s):
13         shortcut ← shortcutNode(base, s, n')
14     Atualiza s, n'
15     if s = nil : endShortcut(shortcut, n); return
16     if n'.parent = nil and not pointsTo(s.p2, dir(s), base.p2) :
17         shortcut ← shortcutNode(base, s, n')
18     Atualiza s, n'
19     if s = nil or  dir(base) = dir(s):
20         endShortcut(shortcut, n); return
21 doShortcut(n, shortcut)
```

Figura 31 – Pseudocódigo da nova versão do algoritmo de detecção de atalhos. A função *aligned* verifica se ambos segmentos estão alinhados de forma a poder ser traçar um segmento ortogonal, que liga ambos. A função *f* verifica se *n.fs* está do lado da *base* seguindo a direção da base (caso em que retorna *true*, ilustrado na Figura 28), ou a direção oposta. A função *recursiveCall*(p_1, *n*) realiza uma chamada recursiva inicializando *O* com nós localizados em p_1, e com alvo *n.point*. A função *endShortcut*(*shortcut, n*) chama a função *doShortcut*, caso *shortcut* seja *nil*, ou chama *recursiveCall*(*source*, *n*) caso contrário.

4.4 Nós de Caminho Inverso e Avaliação de Nós Visitados

A terceira otimização de desempenho refere-se ao critério para definir se um nó de contorno já foi visitado. Na primeira versão, um nó de contorno era considerado como visitado quando existia um nó fechado que possuía a mesma direção de expansão, mesma direção de obstáculo, e o mesmo ponto. Contudo, essa abordagem permite que haja uma repetição de contorno de um obstáculo, como ilustrado na Figura 32. Considere o nó 7. Como esse nó possui direção de expansão e de obstáculo diferentes do nó 5, o nó 7 não é considerado como visitado. Contudo, a expansão do nó 7 é direcionada para um perímetro já explorado

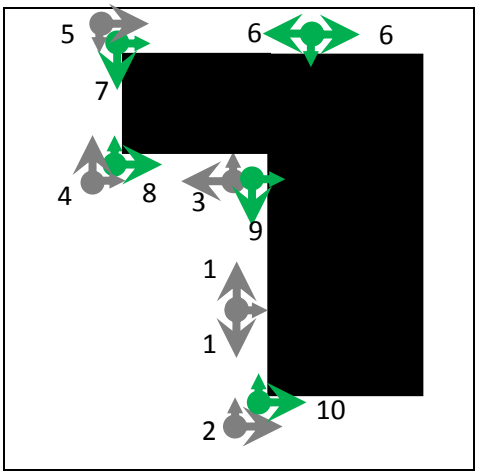

Figura 32 – Ilustração da repetição de contorno. Todos os nós são de contorno, sendo representados de acordo com o padrão adotado. O diferencial é que os nós verdes representam uma cadeia de expansão que visita perímetros já contornados pelos nós cinzas. Os números representam a ordem de criação dos nós.

pelo nó 5 e seus antecessores, em prévias expansões. A cadeia de expansão (nós em verde) do nó 7 para apenas ao se criar o nó 10 e se verificar que o nó 2 é igual, categorizando o nó 10 como visitado. Este cenário mostra que se forem considerados iguais apenas nós com exatamente as mesmas informações, existirá muito esforço computacional proveniente da repetição da busca nos mesmos perímetros. Com isso, foi adicionada outra condição para deixar o critério de decisão de igualdade de nós de contorno mais completo e apropriado. Agora, quando o nó verificado não possui um nó fechado com as mesmas informações, deve ser realizada outra consulta na estrutura que armazena os nós abertos e fechados, mas agora buscando um nó que tenha as informações que caracterizam que ele veio de um perímetro o qual o nó verificado realizará sua futura expansão. Esse nó é chamado de nó do caminho inverso. Ele é gerado com base no nó verificado e é realizada uma nova consulta, decidindo, finalmente, se o nó verificado foi visitado ou não. Para se conhecer as informações do nó de caminho inverso n_2 em relação a um nó n_1, é necessário se considerar todos os casos em que nós de contorno são criados. O primeiro caso é quando um nó comum gera dois nós de contorno no mesmo ponto, como os nós 1 e 6 (note que os nós comuns que os geraram estão abstraídos). Contudo, só existirá uma consulta no conjunto de visitados neste ponto, apenas se outro nó comum realizar uma expansão a qual é interceptada no mesmo ponto. Neste caso, o critério antigo é suficiente para definir que os novos nós já foram visitados. O segundo caso ocorre quando um nó de contorno expande

gerando outro, sem interceptar um obstáculo, como o nó 7 gerando o 8. Se a direção do nó pai de n_1 é oposta a direção de obstáculo de n_1, isto categoriza o segundo caso. O terceiro caso ocorre quando o nó de contorno expande e intercepta um obstáculo, gerando outro nó de contorno, como o nó 8 gerando o 9. Esse caso é caracterizado quando a direção do pai de n_1 *é* igual a direção do obstáculo de n_1. Considerando esses casos, o nó de caminho inverso n_2 do nó n_1, no segundo caso, possui direção *dir* igual a n_1.*detourDir* e direção *detourDir* igual a n_1.*dir*. No terceiro caso n_2.*dir* = *inverse*(n_1.*detourDir*) e n_2.*detourDir* = *inverse*(n_1.*dir*). Na Figura 32, os pares de nós 3 e 9, 4 e 8, 5 e 7, são nós de caminho inversos uns dos outros.

4.5 Mapeamento de Obstáculos

Um dos problemas do algoritmo ST-Router é a sobreposição de buscas das chamadas recursivas. Quando ocorre uma chamada, uma busca é realizada, onde obstáculos são contornados. Em seguida, quando ocorre outra chamada, tanto dentro da anterior quanto fora, é possível que os mesmos obstáculos, ou parte deles, sejam interceptados e contornados. A medida que o numero de chamadas cresce o problema da sobreposição de buscas aumenta. Este é um problema que compensa o grande ganho de desempenho proporcionado pela estratégia de direcionamento de expansão e contorno de obstáculos. O objetivo da última proposta de otimização é reduzir, ou remover, a sobreposição de buscas.

Considerando que o problema consiste em contornar obstáculos já contornados, a estratégia utilizada visa fazer com que o algoritmo se lembre dos contornos realizados. Para isso, cada vez que um obstáculo é contornado, a área contornada é mapeada para uma estrutura de dados global, a qual pode ser acessada por qualquer nível da recursão. Quando um nó de contorno expande, essa estrutura é consultada. Se existe algum mapeamento de contorno, a expansão simplesmente segue o perímetro já mapeado, sem a necessidade de realizar o procedimento de expansão de nós de contorno, o qual consulta a grade a cada passo até encontrar um novo obstáculo à frente ou até chegar à extremidade do obstáculo contornado.

A estrutura mencionada, chamada de mapa de obstáculos, é formada por nós de mapeamento. Esses nós possuem uma posição e dois ponteiros de conexão com outros nós. Para cada ponteiro, existe uma direção associada, e a direção do

obstáculo. Esses nós possuem grande semelhança com os nós de contorno, diferindo por não apresentarem custo, e principalmente por apresentarem 4 direções, ao invés de duas. Partindo de um desses nós, é possível navegar pelo perímetro do obstáculo, tanto para um lado, quanto para o outro.

Quando um nó comum gera dois nós de contorno, é inserido no mapa de obstáculos um nó de mapeamento com as direções de ambos nós de contorno. Na verdade, são inseridas duas entradas no mapa, cada uma com um nó de contorno como chave e o nó de mapeamento como valor. Quando um dos nós de contorno expande, é realizada uma consulta no mapa passando o nó como chave, e o retorno é o nó de mapeamento criado anteriormente. Em seguida, é verificado se o nó possui conectores. Se possui, é iniciado o procedimento de percorrer o perímetro do obstáculo, o qual será discutido posteriormente. Senão, a expansão é realizada normalmente, gerando um novo nó de contorno. Mais uma vez, é realizada uma consulta no mapa, agora com o novo nó de contorno gerado pela expansão. Se já existe um nó mapeando essa extremidade do obstáculo, é realizada sua conexão com o nó de mapeamento anterior. Senão, é criado um novo nó de mapeamento com um dos pares de direção igual ao par de direções do nó de contorno. O outro par é dado pelas direções do nó de caminho inverso do novo nó de contorno. Assim, uma direção aponta para o nó de mapeamento anterior, e a outra aponta para a mesma direção do nó de contorno gerado. O novo nó de mapeamento é ligado ao nó anterior.

A Figura 33 apresenta as novas funções de expansão de nós de contorno e de finalização de expansão. Primeiramente o nó de mapeamento correspondente a *n* é obtido (linha 1). Em seguida, se o nó possui um conector, seguindo as direções de *n*, é iniciado o processor de percorrer o perímetro mapeado pelo nó *o* (linhas 2 e 3), implementado pela função *traversePerimeter*. As linhas 12 e 13 apresentam a modificação necessária para se detectar o ponto de destino sobre a linha de expansão do nó de contorno. Se o destino é encontrado, a expansão termina, também gerando um novo nó de contorno, com as mesmas direções de *n* (função *endDetour*, linha 3). A função *endDetour* agora recebe como argumento o nó de mapeamento correspondente a *n*. Na linha 5, o nó de mapeamento correspondente ao nó de contorno criado é obtido. Na linha 6 é realizada a conexão entre esse nó e o nó correspondente a *n*. As linhas 7 e 8 tratam o caso de o alvo da busca se encontrar entre *n* e *n'*, mas sem estar contido no segmento que liga ambos nós. Se o

alvo é se localiza entre ambos os nós, mas na direção oposta ao obstáculo, o nó comum é criado em *checkDest*. Caso contrário, e se a expansão de *n* não encontrou um obstáculo à frente, é criado o nó comum, sobre *n'* (linha 8).

A função *traversePerimeter* é apresentada na Figura 34. O procedimento, de certa forma, imita o processo de expansão, porém, sem realizar as consultas no *grid* para contornar o obstáculo. O nó *last* representa o nó atual a ser "expandido".

```
expandDetourNode(Nó n)
1  o ← getMapNode(n)
2  if o.hasConnection(n) :
3      traversePerimeter(o, n); return
4  s ← Segmento que parte de n.point e termina na posição
   adjacente, seguindo n.dir
5  do
6      pt ← grid.intersectionPoint(s)
7      if pt ≠ nil :
8          endDetour(n, pt, 1, o); return
9      s' ← Segmento que parte de s.p2 e termina na posição
10     if not grid.intersects(s') :
11         endDetour(n, s.p2, 2, o); return
12     if targeAligned() and s.intersects(target) :
13         endDetour(n, s.p2, 3, o); return
14     Prolonga s em uma unidade, atualizando s.p2
15 while true


endDetour(Nó n, Ponto p,Código code, Nó de Mapeamento o)
1  if code = 1 : n' ← new Nó de Contorno(inverse(n.detourDir), p, n, n.dir)
2  else if code = 2: n' ← new Nó de Contorno(n.detourDir, p, n, inverse(n.dir))
3  else n' ← new Nó de Contorno (n.dir, p, n, n.detourDir)
4  if not addNodeList(n') : return
5  o' = getMapNode(n')
6  connect(o, o', n.detourDir)
7  if not checkDest(n') and n.dir ≠ n'.detourDir and pointsTo(n.point, n.dir, t) :
8      addNodeList((n.dir, n'.point, n, ∅))
```

Figura 33 – Pseudocódigo da nova versão das funções de expansão de nós de contorno e de finalização do procedimento de contorno. A função *getMapNode*(*n*) obtém o nó de mapeamento correspondente a *n*, caso ele exista no mapa de obstáculos, ou cria esse nó e o insere no mapa, caso contrário. O método *o.hasConnection*(*n*) verifica se *o* possui uma conexão a outro nó de mapeamento na direção *n.dir*, contornando um obstáculo na direção *n.detourDir*. A função *targetAligned* verifica se o alvo da busca se encontra na linha de expansão de *n*. Em *endDetour*, a função *addNodeList* realiza o mesmo procedimento de sempre, mas agora retornando se o nó foi adicionado em *nodeList* ou não. A função *connect*(*o, o', d*) realiza a conexão entre *o* e *o'*. As direções dos ponteiros de conexão são obtidas com base nas posições de ambos os nós, mas a direção do obstáculo é informada explicitamente pelo parâmetro *d.*

```
traversePerimeter(Nó de Mapeamento o, Nó n)
1  while true
2      last ← n
3      o ← o.getConnection(last)
4      if o = nil : return
5      visit(last)
6      n ← nextNode(o, last)
7      if n = nil return
```

Figura 34 – Pseudocódigo do procedimento para percorrer o perímetro mapeado. O método *o.getConnection*(*n*) retorna o conector de *o* referente as direções *n.dir* e *n.detourDir*. A função *visit* marca o nó como fechado. A função *nextNode*(*o, n*) analisa as direções de *o* e *n* e gera um nó , que seria o resultado da expansão de *n*, caso fosse realizada a expansão tradicional.

A cada iteração é obtido um nó de mapeamento conectado ao nó anterior, com base nas direções de *last* (linha 3). Com base nessas direções e nas direções do nó de mapeamento, é possível deduzir o resultado da expansão, isto é, se existe um obstáculo à frente ou não. Assim, é gerado um novo nó, que representa o resultado da expansão (linha 6). Dentro de *nextNode*, deve ser verificado se é possível realizar algum atalho no caminho do nó a ser retornado. Isso é feito analisando o seu padrão de caminho. Se o padrão não for *ort*, pode existir algum atalho, e nesse caso, o nó é adicionado em *nodeList* (caso não tenha sido visitado) e a função retorna *nil*, pois se prosseguir com as “expansões” o caminho pode não encontrar um atalho posteriormente, visto que o procedimento de detecção de atalhos é incremental. Note que cada nó *last* também é marcado como visitado. Essa função também realiza os testes para criação de nós comuns assim como em *endDetour*, permitindo a criação de nós comuns, que são imediatamente adicionados em *O* (apenas se o padrão do caminho for *ort*).

4.6 Experimentos e Resultados

Os experimentos foram realizados em um computador com sistema Linux com 130Gb RAM, CPU AMD *Opteron* de 1,4 GHz. O primeiro conjunto de experimentos tem como objetivo avaliar as modificações do ST-Router. O segundo conjunto visa avaliar o desempenho e a optimalidade do ST-Router com as melhorias em relação ao algoritmo de Hetzel, implementado no escopo 2D. Ambos algoritmos foram implementados na linguagem Java.

No primeiro conjunto de experimentos, o algoritmo com as melhorias foi comparado com a versão antiga. O algoritmo também foi testado com cada optimização ativada separadamente. O roteamento foi executado em um *grid* de 10^4 x 10^4 pontos, com obstáculos aleatórios. Os obstáculos consistem em segmentos de tamanho, posição e orientação aleatórios. Os pares de pontos origem e destino foram escolhidos aleatoriamente. O mesmo conjunto de pares origem-destino foi utilizado para ambos algoritmos. Assim, ambos executaram as mesmas buscas nos mesmos *grids*. Os caminhos resultantes não foram adicionados ao *grid*, pois isto iria rapidamente tornar o *grid* muito congestionado, tornando o progresso roteamento impossível (por questões de tempo de execução e de consumo de memória). Assim, para executar um grande número de buscas, foi utilizado um *grid* estático, representando um nível médio de congestionamento. Foram realizados experimentos com vários *grids*, variando a densidade do *grid*. A densidade é a proporção (área ocupada / área total) de área ocupada por obstáculos em relação à área livre. A densidade **não** é um percentual. Assim uma densidade igual a 1 representa 100% de área ocupada. A Tabela 1 mostra os resultados. A coluna "D" representa a densidade do *grid*. "#" Mostra o número de buscas realizadas; “1º” mostra os resultados da versão original do ST-Router; "WL_ON" representa o algoritmo em sua versão original com as três otimizações de WL; “Padrões” representa a coluna “WL_ON” com a otimização de padrões de caminhos ativada; “Comum” representa a coluna “WL_ON” com a modificação da condição de criação de nós comuns na expansão de nós de contorno; “Map” representa a coluna “WL_ON” com a proposta de otimização de mapeamento de obstáculos; “Inv” representa a coluna “WL_ON” com a otimização de nós de caminho inverso; “1º++” representa a versão original do algoritmo (coluna “1º”) com as propostas de otimizações de desempenho que se mostraram benéficas, que são nós de caminho inverso (coluna “Inv”) e padrões de caminhos (coluna “Padrões”); “Padrões + Inv” representa o algoritmo com as otimizações de WL (coluna “WL_ON”) e com as duas modificações benéficas; “speedup” representa o desempenho de “Padrões + Inv” em relação a “WL_ON” (“WL_ON” / “Padrões + Inv”). Note que cada linha da tabela apresenta os resultados obtidos a partir do mesmo *grid*, com os mesmos pares de origem-destino das buscas. Não são apresentados resultados referentes a WL, visto que as otimizações de desempenho não influenciam no WL, e que o ST-Router obteve o caminho ótimo em todos os casos, como será visto mais adiante, no

Tabela 1 – Tempos de execução (em segundos, ou em minutos, quando seguido pela letra “m”) das modificações do ST-Router.

D	***#***	***1º***	***WL_ON***	***Padrões***	***Comum***	***Map***	***Inv***	***1º++***	***Padrões + Inv***	***speedup***
0,1	10^5	1,3	1,4	1,4	139	2	1,5	1,2	1,1	1,23
0,3	10^5	1,7	5,2	3,8	460	53	4	1,8	4,5	1,15
0,5	10^5	4,1	25,2	20,0	51m	29,9m	22,7	4,2	22	1,14
0,7	10^5	14,8	50m	34m	1100m	547m	13,4m	13,8	13,1m	3,8
0,9	2.10^4	12,4m	8530m	8596m	-	-	4149m	10m	4038m	2,1

segundo conjunto de experimentos.

Surpreendentemente, o mapeamento de obstáculos mostrou um péssimo desempenho. Para 0,9 de densidade não foi possível realizar o roteamento pelos constantes estouros de memória que ocorreram. Existe um custo envolvido na construção do mapa de obstáculos e se o desempenho não foi bom é porque o custo prevaleceu o benefício. Porém, não se pode generalizar afirmando que o mapeamento de obstáculos é algo ruim. Em outro cenário experimental, essa modificação apresentou bons resultados. Este cenário é referente a um visualizador de circuitos digitais, onde circuitos mapeados com portas lógicas são carregados e apresentados na tela. Para isso são utilizados retângulos para representar as portas lógicas, que são posicionadas, sem muito compromisso com o problema de otimização de WL do posicionamento na síntese física. Uma vez que as portas lógicas são posicionadas o roteamento (2D) é executado, de forma simples, apenas decompondo-se as redes em conjuntos de pares de pontos. Nesse cenário, o ST-Router foi executado com as otimizações e o mapeamento de obstáculos apresentou uma melhora de desempenho de cerca de 30%, em relação ao tempo do algoritmo com as otimizações de WL apenas, no mesmo cenário. Além disso, como será apresentado posteriormente, essa modificação apresentou bons resultados para a versão tridimensional do algoritmo. Contudo, o fato de os resultados apresentarem grande diferença nos dois cenários mostra que essa modificação deve ser utilizada com cautela, pois dependendo das características gerais dos obstáculos ela pode ser prejudicial. A diferença nas características dos obstáculos entre os dois cenários é que, no primeiro, os obstáculos são segmentos apenas e, no segundo, existem muitos retângulos e também segmentos, que se aglomeram nos espaços entre linhas e colunas de retângulos.

A nova condição de criação de nós comuns na expansão de nós de contorno também não mostrou bons resultados. Um dos fatores de desempenho que esta modificação tem o potencial de melhorar é o caso da Figura 26c, comentado na sessão anterior. No entanto, esta modificação pode ser prejudicial se o obstáculo é longo e se não existir outro obstáculo na direção de expansão. Neste cenário, em alguns casos pode ser mais vantajoso realizar a expansão do nó comum da versão original do algoritmo, pois isto fará a busca convergir mais rapidamente ao destino, ao passo que, com a modificação, o algoritmo perde tempo contornando um obstáculo que não precisava ser contornado.

As otimizações de nós de caminho inverso e de padrões de caminhos apresentaram bons resultados. Para todas as densidades (exceto 0,1 que apresenta pouco desafio de roteamento) os nós de caminho inversos reduziram o tempo em uma boa quantia. Os padrões de caminhos melhoraram o tempo para as densidades 0,3, 0,5 e 0,7. Para 0,9 o tempo foi praticamente o mesmo da execução do algoritmo sem padrões de caminhos. Isso mostra que essa otimização tem um bom potencial, mas ainda deve ser melhor investigada. Para casos mais complexos os padrões detectados não fizeram a mínima diferença. De fato, existem muitos padrões não detectados. Os apresentados são apenas alguns que podem melhorar o desempenho em casos específicos. Quando o cenário foge daqueles casos, o uso dos padrões se torna irrelevante.

As otimizações de WL apresentaram uma enorme queda de desempenho. Para uma densidade de 0,7 o ST-Router original completou o roteamento em 14 segundos, contra 50 minutos de roteamento com as otimizações de WL. O uso de padrões de caminhos e nós de caminho inverso atenuou bastante a queda de desempenho, apresentando 13 minutos de roteamento para o mesmo cenário. Ainda assim, o algoritmo na versão original é extremamente mais rápido. A desvantagem é que ele não garante o caminho ótimo, enquanto os experimentos a serem mostrados a seguir apontam que as otimizações de WL garantem. Em um cenário onde se deseja o caminho ótimo, as otimizações de WL, nós de caminho inverso e padrões de caminhos são favoráveis. Se o caminho ótimo é desejável, mas não é uma necessidade, pode-se utilizar a versão original com essas duas otimizações de tempo (coluna “1°++”).

O segundo conjunto de experimentos visa comparar a versão melhorada do ST-Router, com as otimizações de WL e as duas otimizações de tempo que se

mostraram benéficas, com o algoritmo de Hetzel, adaptado ao escopo 2D. Essa adaptação consiste em considerar que o *grid* possui 2 *layers* e que o custo das vias é 0, além de não ser permitido o uso de *jogs*. Os resultados obtidos foram o tempo de execução e WL. Os experimentos seguiram a mesma metodologia do anterior. Para cada par origem-destino, ambos os algoritmos foram executados na mesma grelha estática. A Tabela 2 mostra os resultados. As colunas “ST” referem-se às informações do ST-Router. A coluna "Abertos" mostra o número de nós adicionados no conjunto de nós abertos *O*. "Média Abertos" representa o tamanho médio de *O* em cada inserção em *O*. A coluna “speedup” representa quantas vezes o ST-Router é mais rápido (tempo_Hetzel/tempo_ST). Os valores de tempo estão arredondados, omitindo as casas decimais (ou algumas delas). Para cada roteamento, o comprimento do caminho do ST-Router foi exatamente o mesmo do algoritmo de Hetzel, que é conhecido por ser ótimo. O *speedup* do ST-Router apresentou uma enorme variação, atingindo o pico de 7522 com 0,3 de densidade, caindo para 2,6 com 0,9 de densidade. ST-Router foi projetado para lidar com bloqueios de uma forma muito eficiente. Para a densidade mais elevada, o algoritmo ainda é muito mais rápido. O custo computacional de ST-Router depende muito do perímetro obstáculos, enquanto o algoritmo de Hetzel é afetado pela área livre de roteamento. A medida que a densidade aumenta, a área livre diminui, ao passo que o perímetro dos obstáculos se torna maior. Note-se que o algoritmo de Hetzel expandiu menos nós que o ST-Router na maior densidade. Mesmo assim o desempenho do ST-Router foi superior. Isto se deve ao fato de que o tamanho médio de *O* é bem menor no ST-Router. O custo envolvido na manipulação de O é “Abertos” * *log* (“Média Abertos”) (considerando que *O* é implementado com um *heap*). Uma vez que o tamanho médio de *O* no ST-Router é muito inferior ao algoritmo de Hetzel, isto compensa o grande número de nós abertos. Além disso, este cálculo é pessimista para o ST-Router, uma vez que "Abertos" representa a soma de todos os nós abertos em todos os níveis de recursão e cada nível tem seu próprio *O*.

Tabela 2 – Comparação do ST-Router com as melhorias com o algoritmo de Hetzel adaptado a 2D.

D	**#**	**Tempo**		**Abertos**		**Média Abertos**		***speedup***
		ST	***Hetzel***	***ST***	***Hetzel***	***ST***	***Hetzel***	
0,1	2.10^5	1	343	$5{,}8.10^5$	$3{,}9.10^7$	2,1	130	312
0,3	2.10^5	4,5	562m	$1{,}7.10^6$	$1{,}6.10^8$	3,8	332	7522
0,5	2.10^5	22	1041m	$9{,}1.10^6$	$5{,}3.10^8$	7,9	448	2835
0,7	2.10^5	13,1m	1474m	$2{,}8.10^8$	2.10^9	25,5	722	112
0,9	2.10^4	4038m	10826m	2.10^{10}	$1{,}5.10^{10}$	143	1,030	2,68

5 ALGORITMO SG-ROUTER

O SG-Router é um algoritmo de busca de caminhos que visa determinar a menor rota entre dois conjuntos de pontos *S* e *T*, pertencentes a um espaço de busca tridimensional. Esse espaço é representado pela estrutura de dados mencionada anteriormente, chamada *grid*. O *grid* é composto por vários planos (ou *layers*), de tamanhos limitados. Cada plano possui uma direção preferida de roteamento. O custo de uma aresta *e* do grafo que modela o *grid*, respeitando a direção preferida é sempre 1. Se *e* for um *jog*, *custo*(*e*) = *j*, sendo que $j \geq 1$. Caso *e* seja uma via que conecta um vértice do plano *i* ao plano *i+1*, $c(e) = v_i$, tal que $v_i \geq v_{i-1}$ e $v_0 \geq 1$. Recapitulando, um *jog* é uma conexão entre dois vértices adjacentes que pertencem ao mesmo *layer* e que desrespeitam a direção preferida de roteamento. Uma via é uma conexão entre dois vértices adjacentes que pertencem a *layers* diferentes. No que diz respeito à optimalidade, a implementação adotada, da mesma forma que na primeira versão do ST-Router, omite a criação de nós em certos casos, abrindo a possibilidade do caminho ótimo não ser encontrado, para que o desempenho não seja prejudicado. No entanto, quanto à mecânica do algoritmo, não há prova demonstrando se o algoritmo garante ou não o caminho ótimo. Contudo, pode-se afirmar, no mínimo, que o algoritmo apresenta uma enorme convergência para o ótimo.

A sessão 5.1 apresenta uma visão geral do algoritmo, contemplando de forma sucinta os principais novos conceitos, como a inserção de novos tipos de nós, tratamento de múltiplas origens e destinos, uso de funções de custo potencial melhoradas, entre outros. A sessão 5.2 apresenta o algoritmo completo, com todas as definições e pseudocódigos. Finalmente, a sessão 5.3 apresenta uma série de possíveis melhorias no algoritmo, além de discutir sobre outras regras de projeto ainda não consideradas.

5.1 Visão Geral do Algoritmo

Da mesma maneira que na versão 2D, o SG-Router utiliza estruturas de dados chamadas nós, com propriedades específicas, para realizar expansões. Considerando-se a mudança de escopo, do espaço 2D para o 3D, é prevista a necessidade de adição de novos tipos de nós, assim como a extensão do conjunto de direções utilizadas por nós. Agora, esse conjunto contempla as direções "*above*" e "below", além das preexistentes (*up, down, left, right*). O fluxo geral do algoritmo permanece o mesmo. Primeiramente, é realizada uma inicialização. Em seguida, os melhores nós são removidos de *O* e são expandidos, gerando outros nós. Estes nós são submetidos a um procedimento de verificação de atalhos, e em seguida, são adicionados em *O*.

O algoritmo utiliza 7 tipos de nós: nós comuns, nós de via, e 5 nós de contorno, sendo estes chamados de *detour1, detour2, detour3, detourVia* e *viaDetour.* Nós comuns funcionam de forma análoga à versão 2D. Expandem gerando nós comuns e possivelmente nós de via, como ilustrado na Figura 35a. Nós de via são semelhantes a nós comuns, mas expandem ao longo do eixo *z*, gerando nós comuns nos planos adjacentes a *t* (Figura 35b). As cores verde e verde escuro serão utilizadas para designar nós comuns e nós e via, nas figuras ao decorrer do trabalho. Os demais nós também terão cores designadas, facilitando sua identificação nas figuras.

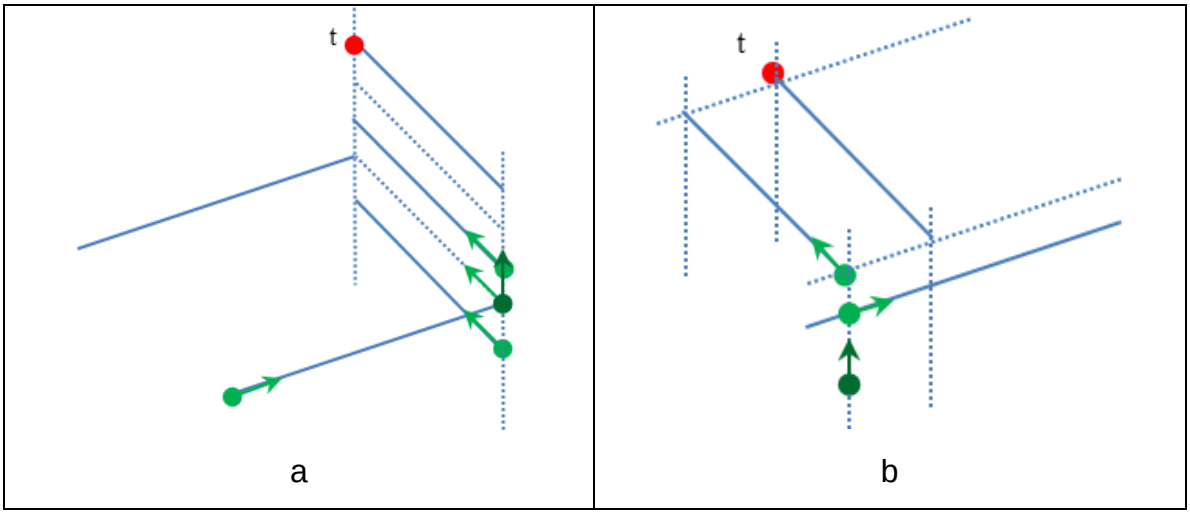

Figura 35 – Ilustração de casos básicos das expansões de um nó comum e de via. (a) Um nó comum (a esquerda) gera nós comuns (verde claro) e um nó de via (verde escuro), apontando para o ponto de destino (t). (b) Um nó de via (verde escuro) gera nós comuns (em verde). As linhas não pontilhadas representam a direção preferida (o mesmo padrão é utilizado posteriormente)

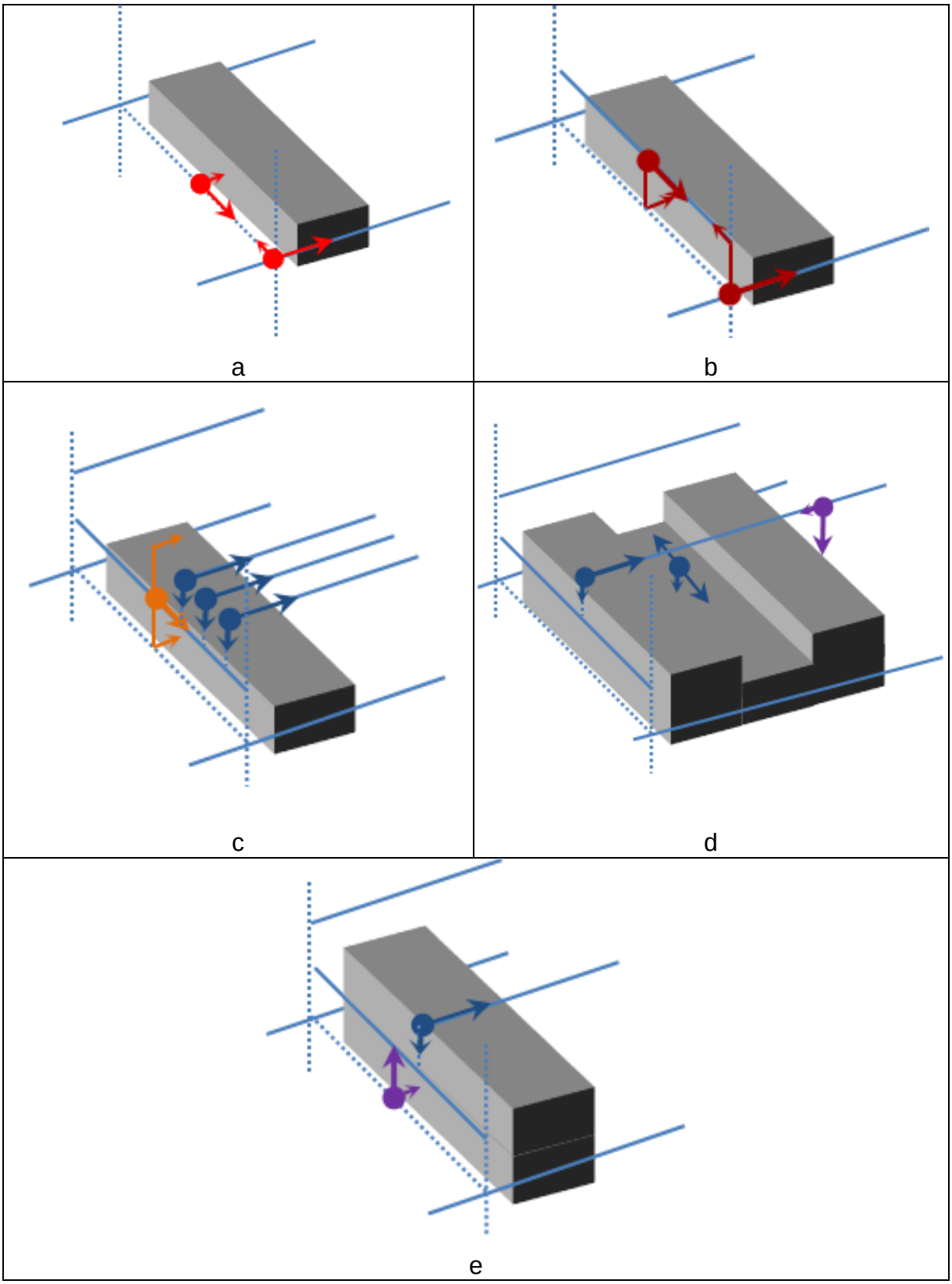

Figura 36 – Ilustração de casos simples das expansões dos novos nós de contorno. O bloco cinza representa um obstáculo (padrão utilizado posteriormente). (a) Um nó *detour1* (acima, a esquerda) gera um nó *detour1*. (b) Um nó *detour2* (acima, a esquerda) gera um nó *detour2*. (c) Nó *detour3* (laranja) gera nós *viaDetour* (azul). (d) Nó *viaDetour* gera nós *viaDetour* e *detourVia* (roxo). (e) Nó *detourVia* (roxo) gera nó *viaDetour* (azul).

As maiores diferenças, em relação à versão anterior do algoritmo, começam na adição dos novos nós de contorno. Os nós *detour1* (Figura 36a) representam os mesmos nós de contorno da versão 2D. Considerando que estes podem realizar a expansão ao longo de *jogs*, é necessário que existam nós que contornem o mesmo obstáculo, mas localizados nos planos adjacentes, permitindo uma expansão seguindo a direção preferida, com o objetivo de avaliar qual contorno é menos custoso. Estes são os nós *detour2*, ilustrados na Figura 36b. São definidos por (*dir, point, parent, detourDir1, detourDir2*). O campo *detourDir1* é a direção que aponta para o plano o qual está sendo contornado, e *detourDir2* é a direção do obstáculo. A primeira é sempre ao longo do eixo *z*, e a segunda sempre ao longo de *x* ou *y*.

Na versão anterior, ao realizar uma expansão de um nó de contorno, não havia a possibilidade de contornar os obstáculos por cima e por baixo, devido ao espaço 2D. Como o espaço 3D introduz essa possibilidade, a cada passo da expansão de um nó *detour2*, existe a possibilidade de se contornar o caminho por cima, com ilustrado na Figura 36c. Apesar dessa funcionalidade poder ser implementada junto à expansão de nós *detour2*, foi criado um outro tipo de nó (*detour3*) para realizar essa tarefa. Caso não seja assim, a expansão de um nó *detour2* pode causar esforço computacional desnecessário. Nós *detour3* são *detour2* com direções adicionais. No caso da Figura 36c, o nó “tateia” o obstáculo por baixo, mas verifica a parte de cima para tentar o contorno por essa direção. Caso isso seja possível, é gerado um nó *viaDetour* (nós em azul). São definidos por (*dir, point, parent, det2, detourDir1, detourDir2*). O campo *det2* é uma referência para um nó *detour2*. Este nó é usado para se saber onde se localiza o obstáculo. Além disso *det2* é usado para evitar esforço computacional, quando *det2* já foi expandido e não é necessário realizar testes de intersecção ao longo de *dir*. Os campos *detourDir1* e *detourDir2* representam, respectivamente, a direção do plano o qual será verificado se pode-se criar nós *viaDetour*, e a direção dos nós a serem criados, assim como a direção do teste de intersecção. Na Figura 36c, *detourDir1* é *above* e *detourDir2* é *right*.

Nós *viaDetour* são nós de “contorno de vias”. Expandem seguindo uma direção contida em um plano, mas verificam os planos inferiores (no caso da Figura 36d) ou superiores para tomar decisões de geração de novos nós. Se o plano adjacente possui um obstáculo, nenhum nó é gerado. Se não possui, mas o segundo plano o possui, dois nós *viaDetour* são gerados no plano adjacente. Se

ambos os planos não possuem obstáculos, um nó *detourVia* é gerado. Sua definição é a mesma de qualquer nó de contorno “tradicional” (que possui apenas uma direção de contorno). Contudo, *dir* sempre se estende ao longo do plano *x-y*, e *detourDir* sempre é ao longo do eixo *z*.

Nós *detourVia* são nós de via que contornam obstáculos contidos nos planos. Conforme ilustrado na Figura 36e, geram um nó *viaDetour* quando não encontram obstáculos nos planos. Sua definição é de um nó de contorno tradicional. O campo *dir* se prolonga em *z* e *detourDir* sempre está contido no plano *x-y*.

Evidentemente, os casos da Figura 36 não contemplam situações de bloqueios durante as expansões, nem considerações sobre criação de nós comuns, junto à nós de contorno. Como o objetivo dessa sessão é apresentar os conceitos gerais do algoritmo, tais detalhes serão discutidos na próxima sessão.

Em virtude da mudança de escopo, o procedimento de atalhos sofreu algumas modificações. O uso de padrões de caminhos foi reduzido, considerando-se apenas os padrões *“ort”* e *“circ”*. Isto se deve ao fato de se ter constatado que algumas considerações antigas não tinham impacto (ou apresentavam impacto negativo) no paradigma 3D. Com isso, é necessário realizar um estudo mais aprofundado sobre padrões de caminhos, no espaço tridimensional, para que se possa utilizá-los com mais eficácia. O mapeamento de obstáculos e os nós de caminho inverso são utilizados. A modificação da condição de criação de nós comuns sobre nós de contorno foi desconsiderada, para a versão 3D.

Considerando que a distância de *Manhattan* é um *lowerbound* muito pobre para o roteamento, devido às direções preferidas, vias e *jogs*, foi implementada uma função mais sofisticada para se calcular o custo mínimo entre dois pontos. Essa função, no entanto, não pode implementar a função *h*, pois pode fazer com que o algoritmo não encontre o melhor caminho. Isto será discutido na próxima sessão. Com isso, essa função é utilizada somente para calcular o *lowerbound* entre um *s-t* (*source-target*). Assim, o custo de um nó é dado por *max*(*g+h*, *lowerbound*(*s, t, grid*)). Note que a função *lowerbound* recebe o *grid* como parâmetro. Ela realiza uma pequena análise no *grid* para retornar um custo mínimo mais apropriado. A Figura 37 ilustra os principais conceitos utilizados para a implementação da função. Os casos *a*, *b* e *c* referem-se ao cenário onde *s* e *t* estão desalinhados e no mesmo *layer*. Neste caso, não há como realizar a conexão sem o uso de vias ou de *jogs*. O melhor caso é ilustrado por *a*. A conexão pode ser realizada com 2 vias (por cima,

ou por baixo) ou *n jogs* (em tracejado). Para definir se o cenário é o do caso *a*, são criados dois segmentos (em verde), partindo de *s* em direção a *t* e vice-versa, que são utilizados em uma consulta no *grid*. Com isso, é possível saber até que ponto o caminho está livre de obstáculos, tanto no lado de *s* quanto no lado de *t*. Se existe intersecção entre os intervalos de ambos segmentos (área delimitada pelas retas pontilhadas em vermelho), o algoritmo calcula o menor custo entre 2 vais por cima, 2 vias por baixo, ou *n jogs*. Note que apesar das vias de *layers* inferiores possuírem

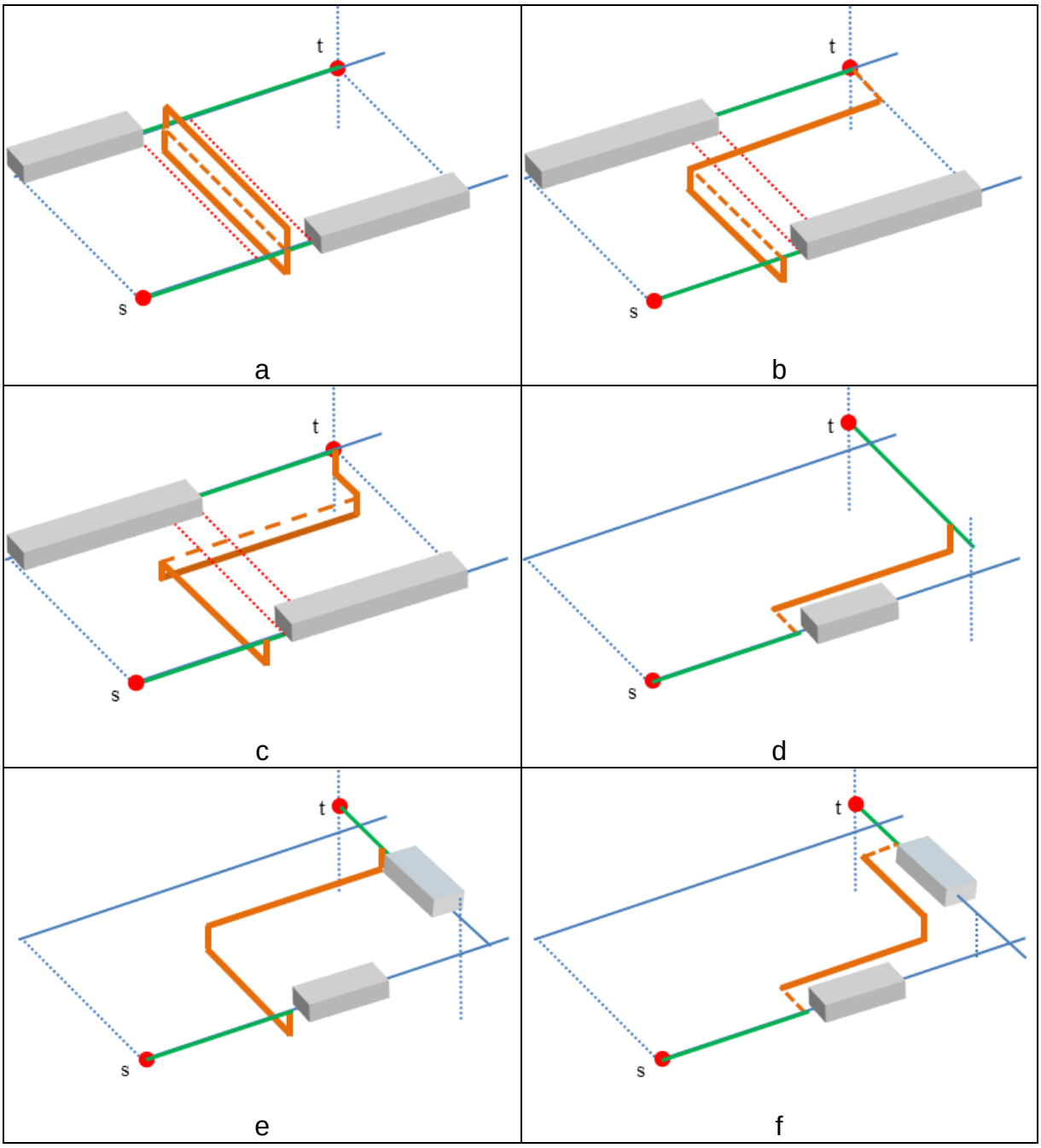

Figura 37 – Ilustração dos principais conceitos utilizados na função *lowerbound.*

menor custo (de acordo com a definição do início da sessão 4) a Figura 37a considera o uso de vias tanto por cima quanto por baixo. Essa verificação é necessária pois se o *layer* inferior não existir, as vias de cima são consideradas (nos outros cenários da Figura 37 as vias superiores são omitidas, pois está sendo considerado que o *layer* inferior existe). Caso não exista intersecção (casos *b* e *c*), além do custo mínimo já mencionado (2 vias ou *n jogs*), é calculado uma penalidade adicional. Em *b* considera-se o custo de mais um *jog*. Em *c*, considera-se 2 vias adicionais ou *n jogs*. O algoritmo avalia o menor custo das possibilidades de *b* e *c* e retorna o *lowerbound*. Note que foram realizadas apenas duas consultas no *grid*, e com base em seus resultados, o algoritmo deduz o menor custo possível. Obviamente, é possível que existam outros bloqueios capazes de impedir as conexões previstas, fazendo com que o custo mínimo seja maior do que o calculado, mas isso já é previsto para uma função que determine um limitante inferior. Os casos *d, e* e *f* consideram que *s* e *t* estejam em *layers* adjacentes. Mais uma vez, são criados dois segmentos (em verde) para consulta no *grid*. O melhor caminho possível utiliza apenas uma via e exige que ambos segmentos não sejam interceptados por nenhum obstáculo. Se um dos dois segmentos é bloqueado (cenário *d*), o custo mínimo é incrementado por um *jog*. Se ambos segmentos são bloqueados, o custo é incrementado por 2 vias (*e*) ou 2 *jogs* (*f*).

A função *lowerbound* também considera casos onde a diferença entre *layers* é igual a 2, ou quando *s* e *t* estão alinhados. Os conceitos utilizados são análogos aos apresentados. Caso a diferença seja maior que 2, a função retorna $|s.x - t.x| + |s.y - t.y| + nVias(s.z, t.z)$, onde *nVias* retorna o custo de todas as vias entre os dois *layers* dos parâmetros.

Considerando que o algoritmo lida com múltiplos *s-t*, foi necessário adicionar uma nova estrutura de dados (*L*) contendo os *lowerbounds* de cada possível par de pontos de origem e destino avaliados no início e durante a execução do algoritmo (a cada chamada recursiva novos *lowerbounds* são calculados).

Para lidar com múltiplas origens e destinos, o algoritmo utiliza conceitos semelhantes aos do A*. Cada nó possui o campo *source*, o qual é uma referência para o ponto de origem do caminho, e o campo *target*, que é uma referência para o destino atual. O critério para se escolher um destino é calcular o custo potencial do nó até cada alvo e selecionar o alvo de menor custo. Esse custo potencial é

calculado pela função *h*, não pela função *lowerbound*. A atualização de *targets* poderia ocorrer de acordo com a maneira tradicional, isto é, quando um nó filho tem seu custo maior que o custo do seu pai, referindo-se ao *target* do pai, o *target* do filho é atualizado. No entanto, devido ao uso da recursão, o *target* de um nó pai pode estar apontando para um destino de uma recursão que já acabou. Devido a isso, a implementação atual sempre calcula o *target* de cada nó gerado. Apesar de parecer simples e natural a forma com que o algoritmo lida com múltiplos *s-t*'s, existem complicações, as quais foram resolvidas, mas não de forma muito satisfatória, merecendo assim um estudo mais profundo futuramente sobre esta questão. As complicações resultam do fato de que, utilizando a abordagem clássica na lida de múltiplos *s-t*'s, o algoritmo pode não achar o par *s-t* responsável pelo melhor de todos os caminhos. Foi observado que isso se deve a dois fatos: 1) o comportamento de contornar obstáculos e de se atalhar caminhos, que permite que um nó possa gerar outros nós com custo menor, pode impedir o algoritmo de enxergar o melhor *s-t* ; 2) um nó partindo de um *s* (originário do melhor caminho) pode ser rapidamente "descartado" da busca ao expandir e gerar um nó já visitado, proveniente de outro *s*. Para corrigir o caso 1 deve se assegurar que para cada alvo exista algum nó que o persegue. No entanto, isto implica que alguns nós devem ter alvos fixos. Este é mais um cuidado a ser tomado na implementação. O caso 2 foi solucionado considerando os pontos de origem e destino de um nó no critério de igualdade de nós. Nós são comparados para se decidir se já foram visitados ou se já existe algum nó igual com custo menor ou equivalente. Em outras palavras, além de todos os critérios anteriores, dois nós são considerados iguais apenas se possuírem os mesmos *s-t*. Contudo, esta solução pode ser muito nociva para o desempenho, visto que o algoritmo pode, para cada *s*, percorrer toda área de busca investigada pelos *s*'s anteriores.

5.2 O Algoritmo

O primeiro passo do algoritmo é a inicialização do conjunto de associações de pares de pontos à *lowerbounds* (*L*), e do conjunto de abertos *O* (Figura 38, linha 1). Este passo não é realizado nas chamadas recursivas, visto que $O \neq \emptyset$. Para cada ponto $s \in S$ e para cada $t \in T$, $L[s][t] \leftarrow$ *lowerbound*(*s*, *t, grid*). A inicialização do conjunto de abertos é definida na Figura 39. Para cada *s* são criados alguns nós

comuns e/ou de via apontando para seus destinos. Nós comuns com direções não coincidentes com a direção preferida do plano são criados em planos adjacentes a *s*. Caso o ponto adjacente em questão esteja ocupado, nós *detour2* são criados para contornar o bloqueio. Os critérios para a criação desses nós estão definidos na função *addNodeList* (Figura 40), linha 11. Essa função visa realizar um controle sobre os nós candidatos a serem adicionados em *O*. Se os nós já foram visitados, eles não são adicionados em *nodeList*. Se as suas posições estão bloqueadas, são criados nós *detour2* para contornar o obstáculo. O último laço de *initOpenSet* assegura que cada alvo será buscado por algum nó de origem. Para evitar confusão, é importante ressaltar que o escopo das estruturas *grid*, *L* e *R* é global, incluindo todas as chamadas recursivas. Os conjuntos de *sources* e *targets S* e *T*, assim como *O*, têm seu escopo global, mas limitado dentro de cada chamada recursiva.

```
SG-Router(Origens S,  Destinos T, Conjunto de Abertos O)
1   if O = ∅ : inicializa L, O
2   Atualiza R
4   while O  ≠ ∅
5       n ← melhor nó ∈ O
6       if n.point = target: break
7       close(n)
8       expand(n)
9   if n.point = target :
10      R[n.source][n.target] ← n
11      return n
12 else: return nil
```

Figura 38 – Pseudocódigo do laço principal do algoritmo.

```
initOpenSet(Origens S, Destinos T, Nó parent)
1   for each s ∈ S :
2       t ← selectTarget(s)
3       Cria nós de via, caso |t.z-s.z| > 1, e nós comuns, apontando para t,
        e os adiciona em nodeList, por meio da função addNodeList
4       O ← O ∪ nodeList
5   for each t ∈ T :
6       Obtém s ∈ S de menor custo potencial em relação a t
7       Realiza os passos das linhas 3 e 4
8   return O
```

Figura 39 – Pseudocódigo da inicialização do conjunto de nós abertos. A função *selectTarget* (*p*) seleciona o alvo mais promissor do ponto *p* com base na função *h.*

```
addNodeList(Nó n)
1  if n.z está for a dos limites : return false
2  if not grid.isFree(n.point) :
3      if n é comum ou viaDetour :
4          dirs ← ortDirs(n.orientation())
5          for i ← 0 to 1 :
6              nd ← new Detour2(dirs[i], n.fparent.point, n.fparent,
               n.fparent.fparent.dir, n.dir)
7              if not visited(nd) :  nodeList.add(nd)
8      else if n é detour2 :
9          nd ← new Detour2(n.fparent.fparent.inverseDir(), n.fparent.point,
           n.fparent , n.fparent.fparent.dir, n.dir)
10         if not visited(nd) :  nodeList.add(nd)
11     return false
12 if n.isDetour() : getMapNode(n.point).setFutureConnection(n)
13 if not visited(n) :
14     nodeList.add(n);  return true
15 return false
```

Figura 40 – Pseudocódigo da função *addNodeList*. O método *isFree* verifica se o ponto está livre no *grid.* A função *ortDirs*(*o*) retorna as direções da orientação ortogonal a *o*. O campo *fparent* é o nó pai original. Esse campo é necessário visto que o pai de um nó pode ser atualizado como na detecção de atalhos e em possíveis otimizações de caminho em *setParent*.

```
expandCommon (Nó n)
1  s ← Cria segmento
2  p ← grid.intersectionPoint(s)
3  dirs ← ortDirs(n.orientation())
4  if n.isJog() :
5      if p ≠ nil :
6          addNodeList(new Detour1(dirs[0], p, n, n.dir))
7          addNodeList(new Detour1(dirs[1], p, n, n.dir))
8          jogIntersection(p, n)
9          createVia(p, n)
10     else :
11         if s.destPoint() = n.target : Cria nó qualquer em n.target
12         else Cria nó comum ou de via, apontando para n.target
13 else :
14     if p ≠ nil :
15         createDetourNodes(p, n, dirs[0],  dirs[1], above)
16         createDetourNodes(p, n, dirs[0],  dirs[1], below)
17         createVia(p, n) ;  Cria nós detourVia
18     else :
19         if s.destPoint() = n.target : Cria nó qualquer em n.target
20         else Cria nó comum ou de via, apontando para n.target
```

Figura 41 – Pseudocódigo da função *expandCommonNode*.

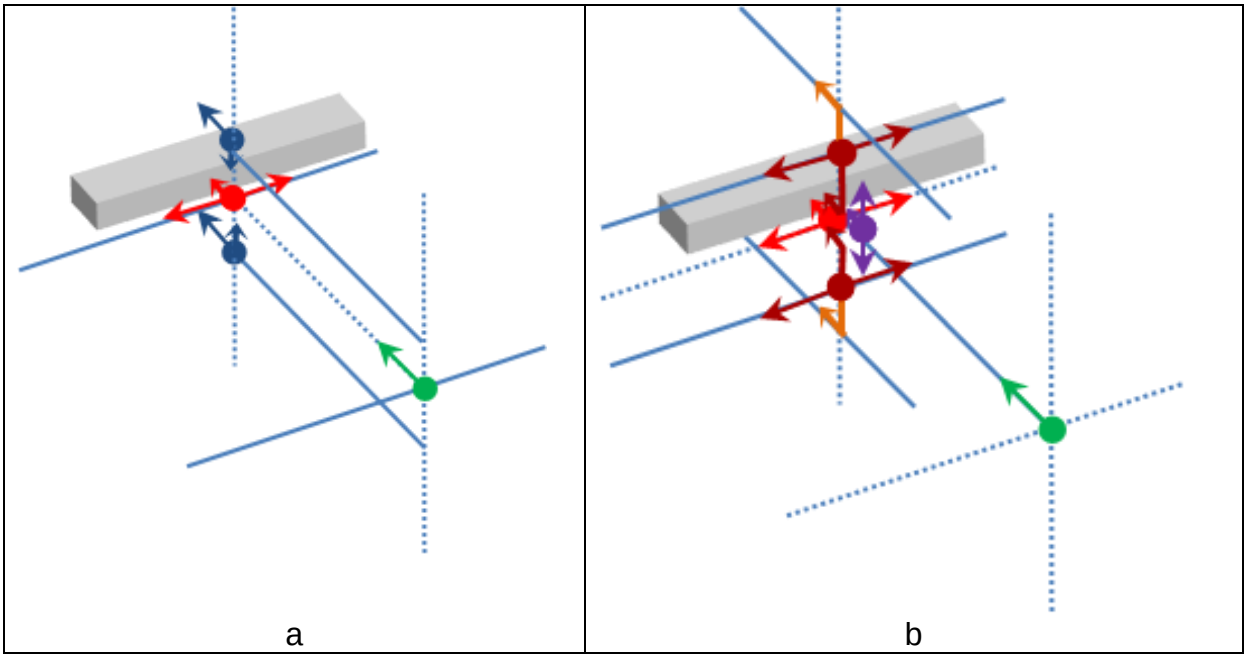

Figura 42 – Ilustração dos casos de intersecção da expansão de um nó comum. Em (b), alguns nós estão ligeiramente deslocados para facilitar a visualização.

O segundo passo é a atualização do conjunto de rotas *R*. Se for a primeira chamada do algoritmo, *R*[*s*][*t*] ← *nil*, para cada *s* ∈ *S* e *t* ∈ *T*. Caso contrário, *R*[*n.source*][*t*] ← *nil*, para cada *n* ∈ *O*. Em seguida, o laço clássico é executado.

A função *expand* realiza a expansão do nó, de acordo com seu tipo, e depois realiza a verificação de atalhos sobre todos os nós gerados, por meio da função *applyShortcuts*. A expansão de nós comuns é ilustrada pela Figura 41. Semelhantemente a expansão de nós comuns da versão 2D, um segmento é criado (linha 1), partindo de *n.point* até se alinhar a *n.target*. Se o nó é um *jog* (linha 4), isto é, se sua direção não está de acordo com a orientação preferida do *layer*, e se o segmento é bloqueado por algum obstáculo, dois nós *detour1* (linhas 6, 7) e dois nós *viaDetour* são criados. Os últimos, são criados na função *jogIntersection.* A função gera ambos os nós nos pontos adjacentes ao ponto de intersecção *p*, conforme mostrado na Figura 42a. Além disso, nós comuns são criados sobre ambos, caso apontem para o destino. Neste caso, o destino utilizado na avaliação da condição de criação de nós comuns, é o *target* do nó pai (intermediário entre o nó comum e o expandido, omitido na ilustração). Sendo assim, os nós comuns são forçados a terem como alvo o *target* do nó pai. Isto é diferente dos alvos fixos, mencionados na sessão anterior. Os nós subsequentes desses nós comuns podem possuir outros alvos. A função *jogIntersection* também realiza o controle necessário dos

OMapNodes (nós de mapeamento de obstáculos). Se a distância entre o *layer* do destino e *p* for maior que 1, um nó de *via* é criado (função *createVia*). Caso não haja intersecção de *s*, indiferentemente se *n* é ou não um *jog*, nós comuns e/ou de *via* são gerados, conforme já ilustrado na Figura 35a. Os nós comuns, gerados pela expansão de um nó comum, onde não houve intersecção, têm seu alvo designado para o alvo do nó expandido. Caso haja intersecção e *n* não for *jog* (linhas 14-17), são gerados 4 nós *detour2*, 4 nós *detour3* (um para cada *detour2*), 2 nós *detour1*, 2 nós *detourVia*, 1 nó de *via* e 0 ou 2 nós comuns, dependendo da posição do alvo. A Figura 42b ilustra essa situação (não considerando nós comuns e de *via*, assim como as demais figuras). A função *createDetourNodes* (Figura 43) cria 2 nós *detour2*, 2 nós *detour1*, 2 nós *detour3* e 0 ou 1 nós comuns, além de controlar o mapa de obstáculos. O parâmetro *d3* indica se a posição dos nós é acima ou abaixo de *p*. O *target* dos nós comuns é o mesmo dos nós *detour2* de direções correspondentes. Note que detalhes de implementação estão omitidos. A função é chamada duas vezes, mas apenas 2 nós *detour1* são gerados e adicionados em *nodeList*. Além disso, é importante lembrar que todos os nós adicionados nesta lista temporária passam pela função *addNodeList*, a qual pode detectar que um nó não pode ser criado em seu ponto, por este estar ocupado, tomando assim a atitude apropriada, conforme o tipo do nó. Vale esclarecer também que a ordem dos parâmetros nos construtores de nós segue a ordem dos campos na definição dos nós, apresentada na sessão anterior.

```
createDetourNodes(Ponto p, Nó n, Direção d1, Direção d2, Direção d3)
1  addNodeList(new Detour1(d1, p, n, n.dir))
2  addNodeList(new Detour1(d2, p, n, n.dir))
3  o ← getMapNode(p)
4  n2 ← new Node(d3, p, n)
5  p2 ← Ponto adjacente a p, de acordo com d3
6  if addNodeList(new Detour2(d1, p2, n2, inverse(d3), n.dir)):
7      Cria nó comum em p2, se aplicável
8      Cria nó detour3
9  if addNodeList(new Detour2(d2, p2, n2, inverse(d3), n.dir)):
10     Cria nó comum em p2, se aplicável
11     Cria nó detour3
12 if Algum nó foi adicionado em nodeList : connect(o, n2, p)
```

Figura 43 – Pseudocódigo da função *createDetourNodes*

```
expandVia (Nó n)
1   s ← Cria segmento de via, partindo de n.point.z até n.target.z ou o plano anterior
2   p ← grid.intersectionPoint(s)
3   if p ≠ nil :
4       Cria nós viaDetour em p e no plano anterior e nós comuns, se aplicável
5       Conecta o OMapNode  de n.point a o1 e o1 a o2
6   else :
7       if s.destPoint().z = n.target.z :
8           if n.point e n.target estão desalinhados :
9               Cria nó comum em n.target.z e no plano anterior
10          else : Cria nó comum em n.target.z
13      else : Cria nó comum em s.destPoint()
```

Figura 44 – Pseudocódigo da função *expandVia.*

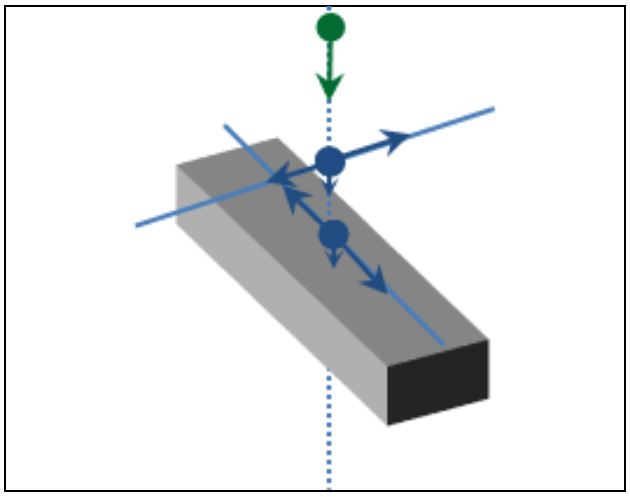

Figura 45 – Ilustração do caso de intersecção da expansão de um nó de *via*.

A Figura 44 mostra o pseudocódigo da expansão de nós de *via*. Quando não há bloqueio (linhas 7-13), é gerado um ou dois nós comuns (caso ilustrado na Figura 35b). Quando há bloqueio, quatro nós *viaDetour* são gerados, conforme a Figura 45. Os OMapNode's *o1* e *o2* se localizam sobre os pares de nós criados, sendo que *o1* está mais próximo a *n.point*.

O pseudocódigo da função *expandDetour2* (Figura 46) é muito semelhante ao da versão 2D. Além disso, o pseudocódigo da função *expandDetour1* não será apresentado, visto que difere de *expandDetour2* apenas nas chamadas de *endDetour2*, as quais são substituídas por *endDetour1*, e no uso de *n.detourDir2* (linha 9), que é substituído por *n.detourDir*. O caso de intersecção da expansão de nós *detour2* é praticamente igual ao da expansão de nós comuns, ilustrado na Figura 42b. A diferença é que um nó *detour2* não precisa ser criado, pois estaria

```
        expandDetour2 (Nó n)
1   o ← getMapNode(n.point)
2   if o.hasConnection(n) :
3       traversePerimeter(n, o); return
4   s ← Cria segmento de tamanho 1
5   do
6       p ← grid.intersectionPoint(s)
7       if p ≠ nil :
8           endDetour2(n, p, true, o); return
9       s' ← orthogonal(s, n.detourDir2)
10      if not grid.intersects(s')
11          endDetour2(n, s.destPoint(), false, o); return
12      inc(s)
13  while true
```

Figura 46 – Pseudocódigo da função *expandDetour2*.

apontando diretamente para o obstáculo. Este detalhe está considerado na linha 5 da Figura 47, onde o terceiro parâmetro de *createDetourNodes* é *nil.* Note que o tratamento deste caso, onde uma das direções é *nil* não está considerado no pseudocódigo da função *createDetourNodes*. Como mencionado anteriormente, os pseudocódigos apresentam os conceitos, omitindo detalhes de implementação e possíveis otimizações, facilitando assim o entendimento da lógica do algoritmo. A função *getMapNode* obtém o OMapNode correspondente ao ponto do parâmetro. Caso não exista tal OMapNode, um novo é criado, adicionado ao mapa de obstáculos, e retornado pela função. A função *connect*(OMapNode *o,* Nó *n,* Ponto *p*)

```
endDetour2(Nó n, Ponto p, Booleano intercepted, OMapNode o)
1   connect(o, n, p)
2   if intercepted :
3       d ← inverse(n.detourDir2)
4       createDetourNodes(p, n, d, n.detourDir2, inverse(n.detourDir1))
5       createDetourNodes(p, n, d, nil, n.detourDir1)
6       Cria nós detourVia e via, se aplicável
7   else :
8       p2 ← Ponto adjacente a p seguindo n.detourDir1
9       addNodeList(new Detour2(n.detourDir2, p2, n', inverse(n.detourDir1),
        inverse(n.dir)))
10      Cria nó comum, se aplicável
11      connect(getMapNode(p), n', p2)
```

Figura 47 – Pseudocódigo da função *endDetour2.*

```
endDetour1(Nó n, Ponto p, Booleano intercepted,  OMapNode o)
1   connect(o, n, p)
2   if intercepted :
3       if n.isJog() :
4           addNodeList(new Detour1(inverse(n.detourDir), p, n, n.dir))
5           jogIntersection(p, n)
6           Cria nó comum, se aplicável
7       else :
8           createDetourNodes(p, n, inverse(n.detourDir), n.detourDir,  above)
9           createDetourNodes(p, n, inverse(n.detourDir), n.detourDir,  below)
10          Cria nós detourVia e via, se aplicável
11  else :
12      addNodeList(new Detour1(n.detourDir, p, n, inverse(n.dir)))
13      Cria nó comum, se aplicável
```

Figura 48 – Pseudocódigo da função *endDetour1*.

conecta *o* ao OMapNode correspondente a *p*, utilizando as direções de *n*. Em *endDetour2*, o nó *n'* se localiza em *p* e é filho de *n*. O nó comum da linha 10 da função *endDetour2* tem seu *target* designado para o alvo do nó *detour2* da linha 9. De forma geral, todo nó comum criado sobre um nó de contorno, com a mesma direção deste, têm seu alvo igual ao alvo do nó de contorno.

A expansão de um nó *detour1* gera outro nó *detour1*, no mesmo *layer*, caso não haja intersecção. Isto implica no fato de que este tipo de nó pode ser ou não um *jog* (o que não ocorre com os nós *detour2*). A Figura 48 mostra a finalização da expansão de um nó *detour1*. Se houve intersecção e *n* é *jog* a geração é quase idêntica ao caso da expansão de um nó comum *jog* (Figura 42a), diferindo apenas na omissão da criação de um nó *detour1* (o que estaria apontando para o obstáculo). Se o nó não é *jog* a intersecção se dá exatamente igual a intersecção de um nó comum (Figura 42b).

O pseudocódigo da expansão de nós *viaDetour* é apresentado na Figura 49. A expansão contorna obstáculos do plano adjacente a *n.point.z* e do próximo plano, seguindo a direção de contorno *n.detourDir*, até que uma das três condições seja satisfeita: 1) a expansão é bloqueada pela frente; 2) a posição adjacente a *n* está vazia; 3) as posições de ambos os planos estão vazias. Os casos 2 e 3 foram ilustrados na Figura 36d. Contudo, a figura mostra o resultado de duas expansões consecutivas de um nó *viaDetour*. A primeira gera dois nós *viaDetour* no plano abaixo (caso 2), e a segunda gera um nó *detourVia* (caso 3). Os três casos são

```
expandViaDetour(Nó n)
1  o ← getMapNode(n.point)
2  if o.hasConnection(n) :
3      traversePerimeter(n, o); return
4  s ← Cria segmento de tamanho 1
5  do
6      p ← grid.intersectionPoint(s)
7      if p ≠ nil :
8          endViaDetour(n, p, true, dif, o); return
9      s' ← orthogonal(s, n.detourDir)
10     p ← grid.intersectionPoint(s')
11     dif ← |p.z-n.point.z| ou 2, caso p = nil
12     if dif  > 0 : endViaDetour(n, p, false, dif, o); return
13     inc(s)
13 while true
```

Figura 49 – Pseudocódigo da função *expandViaDetour*.

abrangidos pela função *endViaDetour* (Figura 50), de acordo com os parâmetros *intercepted* e *dif*. O primeiro, da mesma forma que em *endDetour1* e *endDetour2*, controla se houve bloqueio pela frente (caso 1). O segundo controla os casos 2 e 3. Independente se *n* é ou não *jog*, a resolução da expansão, em caso de intersecção, segue a mesma linha do caso de intersecção de um nó comum, evitando apenas a criação de nós no plano adjacente, conforme mostra a Figura 51.

A expansão de nós *detourVia* é apresentada na Figura 52. Este tipo de nó tenta pular para o próximo *layer* com a mesma orientação de *n.point.z* (por isso a criação do segmento de tamanho 2, na linha 4). Caso consiga, é verificado se é possível prosseguir verticalmente ou horizontalmente (de acordo com *n.detourDir*). A expansão pode resultar em 4 casos: 1) Nenhum bloqueio ocorre ao realizar o pulo e ao verificar se é possível prosseguir seguindo *n.detourDir* (Figura 36e); 2) o pulo é realizado mas ocorre um bloqueio na direção *n.detourDir* (Figura 53c); 3) o pulo é bloqueado na altura 2 (Figura 53b); 4) o pulo é bloqueado na altura 1 (Figura 53a). Os casos 1 e 2 são tratados pela função *endDetourVia* (Figura 54) e os casos 3 e 4 são tratados pela função *endDetourVia2* (Figura 55). Em *expandDetourVia*, a variável *inc* representa o incremento o qual somado a *n.point.z* resulta no *layer* objetivo do salto. Em *endDetourVia*, linhas 2 a 4, é verificado se é possível adicionar um nó *viaDetour* como um *jog*, no *layer* adjacente a *n*. Note que a função de intersecção do *grid*, diferentemente de outras situações, está recebendo outro parâmetro além do segmento, que é o *layer* o qual deve ser consultado (nos outros

```
endViaDetour(Nó n, Ponto p, Booleano intercepted, Int dif, OMapNode base)
1  dirs ← ortDirs(n.orientation())
2  if intercepted :
3      if n.isJog() :
4          addNodeList(new Detour1(dirs[0], p, n, n.dir))
5          addNodeList(new Detour1(dirs[1], p, n, n.dir))
6          jogIntersection(p, n)
7          Cria nó comum, se aplicável
8          o ← getMapNode(p)
9      else :
10         createDetourNodes(p, n, dirs[0], dirs[1], inverse(n.detourDir))
11         if dif ≥ 1: createDetourNodes(p, n, dirs[0], dirs[1], n.detourDir)
12         Cria nós detourVia e via, se aplicável
13 else :
14     if n.isJog() :
15         n2 ← new Node(n.detourDir, p, n)
16         o ← getMapNode(p)
17         addNodeList(new DetourVia(n2.dir, p, n2, inverse(n.dir)))
18         Cria nó de via, se aplicável
19         connect(base, n, o)
20         connect(o, n2, p) ; return
21     else if dif = 1 :
22         createViaDetourNodes(p, n, dirs[0], dirs[1], n.detourDir)
23         addNodeList(new ViaDetour(n.dir, p, n, n.detourDir))
24     else :
25         o ← getMapNode(p)
25         if dif = 2 : addNodeList(new DetourVia(n.detourDir, p, n, inverse(n.dir))
26         else addNodeList(new ViaDetour (n.dir, p, n, n.detourDir))
27 connect(base, n, o)
```

Figura 50 – Pseudocódigo da função *endViaDetour*.

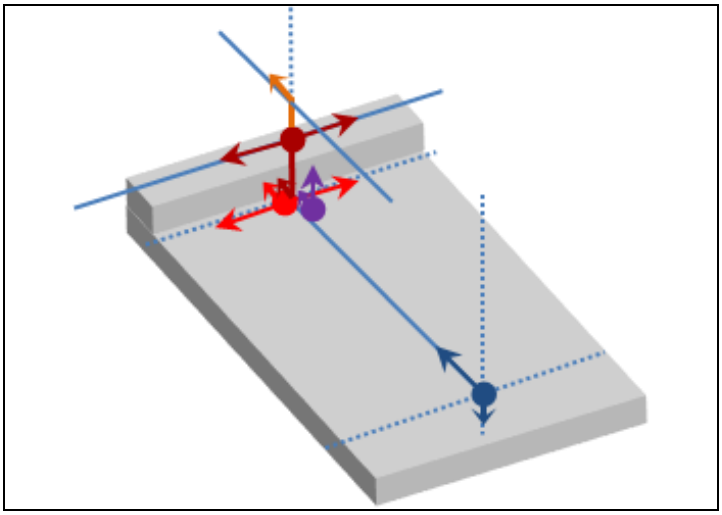

Figura 51 – Ilustração do caso 1 (intersecção pela frente) da expansão de um nó *viaDetour*.

```
expandDetourVia(Nó n)
1  base ← getMapNode(n.point)
2  if base.hasConnection(n) :
3      traversePerimeter(n, base); return
4  s ← Segmento de via de tamanho 2, seguindo n.dir
5  p ← grid.intersectionPoint(s)
6  if p = nil :
7      endDetourVia(n, n.dir, n.detourDir, inc, s')
8      connect(base, n, o)
9  else : endDetourVia2(n, p, inc, ortDirs(n.detourDir), base)
```

Figura 52 – Pseudocódigo da função *expandDetourVia*.

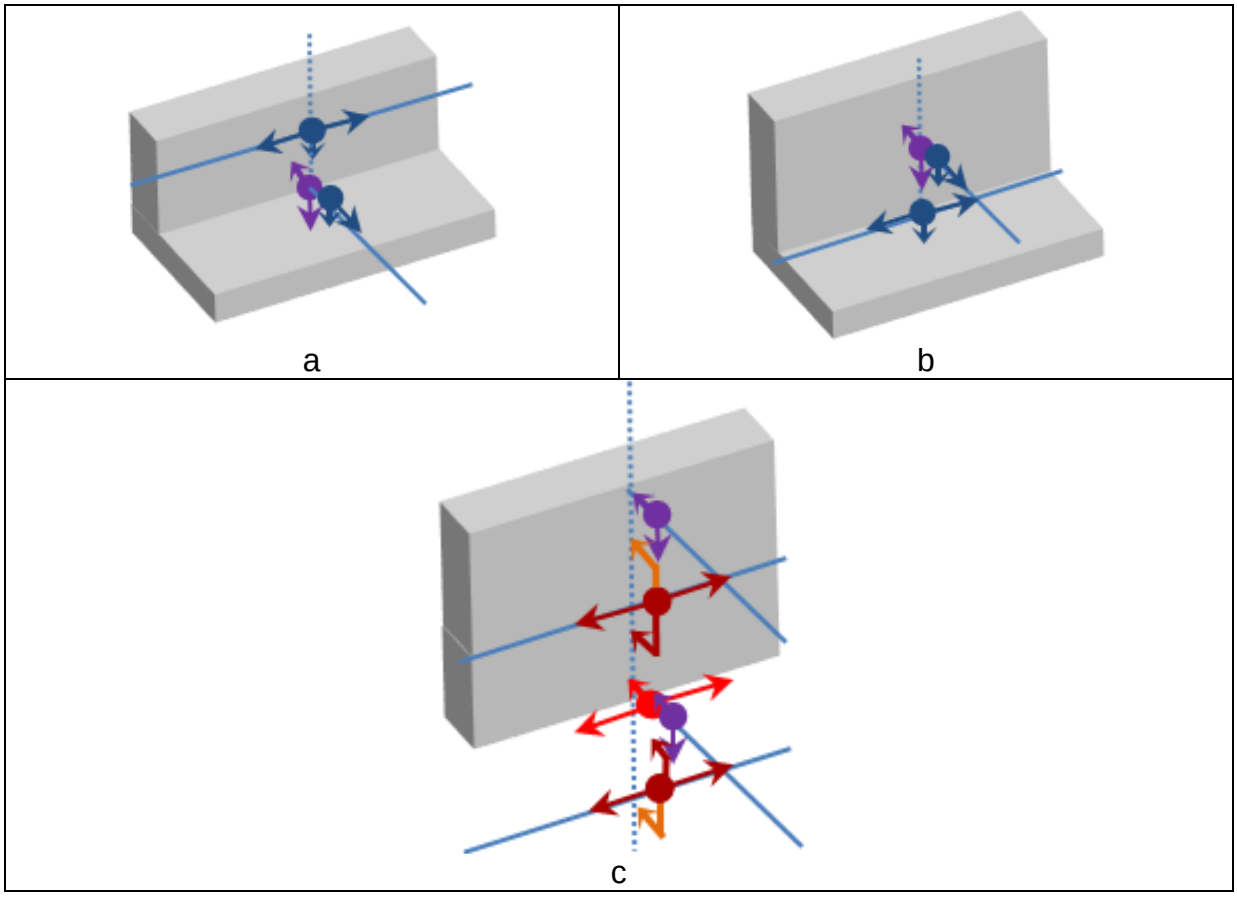

Figura 53 – Ilustração dos casos de intersecção da expansão de um nó *detourVia* (roxo). Em (c), o nó expandido é o que se encontra acima.

```
endDetourVia(Nó n, Direção dir, Direção detour, Int inc, Segmento s)
1  dirs ← ortDirs(detour)
2  if n.fparent.dir = n.detourDir and not grid.intersects(s, n.point.z+inc/2) :
3      addNodeList(new ViaDetour(detour, p', n, inverse(dir)))
4      Cria nó comum, se aplicável
5  if grid.intersects(s, n.point.z+inc) :
6      createDetourNodes(p, n, dirs[0], dirs[1], above)
7      createDetourNodes(p, n, dirs[0], dirs[1], below)
8      addNodeList(new DetourVia(dir, p, n, detour))
9  else :
10     addNodeList(new ViaDetour(detour, p, n, inverse(dir)))
11     Cria nó comum, se aplicável
12 o ← getMapNode(p)
```

Figura 54 – Pseudocódigo da função *endDetourVia*.

```
endDetourVia2(Nó n, Ponto p, Int inc, Direção[] dirs, OMapNode base)
1  addNodeList(new ViaDetour(inverse(n.detourDir), n.point, n, n.dir))
2  Cria nó comum, se aplicável
3  if p = n.point : createViaDetourNodes(p', n, dirs[0], dirs[1], n.dir)
4  else :
5      createViaDetourNodes(p, n, dirs[0], dirs[1], n.dir)
6      if grid.isFree(p'') :
7          addNodeList(new ViaDetour(n.detourDir, p'', n, inverse(n.dir)))
8      else :
9          addNodeList(new ViaDetour(inverse(n.detourDir), p'', n, n.dir))
10         addNodeList(new Detour1(dirs[0], p'', n, n.detourDir))
11         addNodeList(new Detour1(dirs[1], p'', n, n.detourDir))
12     Cria nó comum, se aplicável
```

Figura 55 – Pseudocódigo da função *endDetourVia2*.

casos supõem-se que o segmento tenha a informação do *layer*). Na linha 12, *o* é considerado uma variável global, podendo ser utilizada fora dessas funções, como ocorre no caso de *expandDetour3*, como será visto a seguir. Em *endDetourVia2*, o caso 3 ocorre na afirmação da condição da linha 3 . A variável *p'* refere-se ao ponto superior a *n*, na Figura 53a. A variável *p''* refere-se ao ponto adjacente a *p*, seguindo a direção *n.detourDir*. Os nós *viaDetour* criados nas linhas 7 e 9 são sempre *jogs*, visto que *p''.z* está entre *n.point.z* e *n.point.z* + *inc*, e que *n.detourDir* nunca é *jog* nestes dois últimos planos.

A Figura 56 mostra o pseudocódigo da expansão de nós *detour3*. A condição da linha 3 impede que seja realizado o teste de intersecção pela frente, caso o nó

detour2 associado (*n.det2*) já tenha sido expandido e, consequentemente, percorrido o caminho à frente. A expansão da um passo à frente e em seguida testa se a posição adjacente (*p2*, linha 11), seguindo *n.detourDir1* (*above*, no caso da Figura 57), está ocupada (linha 12). Se está ocupada, o algoritmo prossegue, dando mais um passo a frente, incrementando *s* (linha 19). Caso contrário, é realizado um teste de intersecção com a chamada da função já discutida, *endDetourVia*. Caso haja um bloqueio, nós são criados conforme o pseudocódigo de *endDetourVia*, cujo comportamento é ilustrado na Figura 57.

```
expandDetour3(Nó n)
1   s ← Cria segmento de tamanho 1
2   do
3       if s.destPoint() ainda não percorrido por n.det2
3           p ← grid.intersectionPoint(s)
4           if p ≠ nil :
8               endDetour2(n, p, true, omap.get(n.point)); return
9       s' ← orthogonal(s, n.det2.detourDir2)
10      if not grid.intersects(s') : return
11      p2 ← Ponto adjacente a s.destPoint(), seguindo n.detourDir1
12      if grid.isFree(p2) :
13          n2 ← new Node(n.detourDir1, s.destPoint(), n)
14          endDetourVia(n2,n.detourDir1,n.detourDir2,inc, s')
15          p3 ← Ponto adjacente a s.destPoint(), seguindo inverse(n.detourDir1)
16          if grid.isFree(p3) : connect(getMapNode(n2.point), n2, o)
17          else connect(getMapNode(p3), n2.dir, n.detourDir2, o)
18          addNodeList(new Detour3(n.dir, s.destPoint(), n, n.det2, n.detourDir1,
            n.detourDir2)) ; return
19      inc(s)
20  while true
```

Figura 56 – Pseudocódigo da função *expandDetour3*.

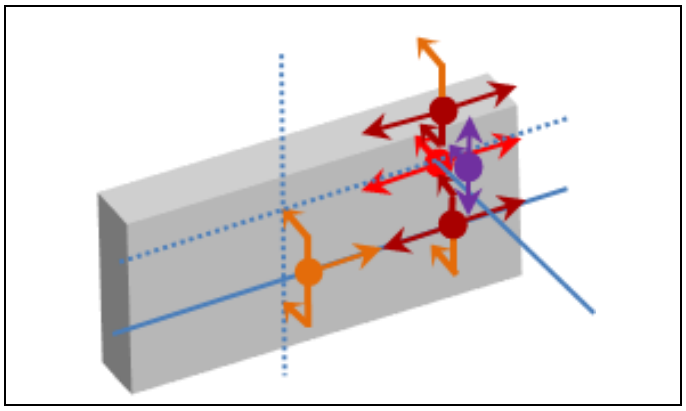

Figura 57 – Ilustração da expansão de um nó *detour3* (a esquerda) e da intersecção da direção *detourDir2*.

Após a expansão, os nós gerados são submetidos ao procedimento de atalhos. Isso acontece na função *applyShortcuts*. Considerando que a expansão de um nó *n* pode gerar nós que não provém diretamente de *n* (como nós *detour2* e *detour3*, e possivelmente *viaDetour*), é necessário aplicar a detecção de atalhos primeiramente no nó intermediário (filho direto de *n*) para depois no nó gerado em questão. Isto é necessário visto que a detecção de atalhos considera que a cada novo segmento gerado no caminho seja executado o procedimento de atalhos, e no caso dos nós mencionados, dois novos segmentos são gerados na mesma expansão. Além disso, o número de nós gerados em uma expansão pode ser relativamente grande (em torno de 15), e com isso, a aplicação do procedimento de atalhos em cada nó pode ser impraticável. Com isso, a função *applyShortcuts* tem como objetivo reduzir o número de detecções de atalhos executadas, cuidando também os casos dos nós mencionados. O primeiro passo da função é percorrer *nodeList*, selecionando o nó mais apropriado para se executar a detecção de atalhos. Se existe algum nó que não tenha sido gerado diretamente por *n*, o nó intermediário (chamado de *i*) e o nó filho (chamado de *f*) são obtidos para a execução de atalhos. Se existe mais de um nó nessa mesma condição, ou se não existem tais nós, o nó mais próximo de *n* é escolhido. Em seguida, é realizada a detecção de atalhos. Se existe algum nó intermediário, a detecção é aplicada primeiramente nele, para em seguida ser aplicada em *f*. Depois, para cada nó em *nodeList*, se o nó em questão coincidir com a posição de *i* ou *f*, ou com o nó verificado anteriormente pela detecção de atalhos, o campo do nó pai do nó em questão é atualizado para o nó pai do nó coincidente. Caso não haja tais coincidências, o procedimento de atalhos é executado.

O algoritmo de detecção de atalhos é apresentado na Figura 58. A ilustração do primeiro passo (linha 1) é apresentada na Figura 59. Quando o caminho possui duas vias conectadas ao mesmo plano, conforme ilustrado, o algoritmo avalia o atalho menos custoso e o aplica, caso não haja intersecção. Em seguida, se o padrão do caminho for *ort*, é verificado se o caminho possui algum *jog*. Se for o caso, o caminho é corrigido, caso esteja de acordo com as situações mostradas na Figura 60. Os demais passos do código são semelhantes à versão 2D, diferindo apenas no tratamento de casos referentes ao escopo tridimensional.

```
verifyShortcut(Nó n)
1  Corrige a "ponta" do caminho, se aplicável
2  if n.pattern = ort:
3      Corrige caminho minimizando o custo, caso haja jogs; return
4  base ← n.parentSegment()
5  n' ← n.parent.parent
6  s ← n'.parentSegment()
7  while dir(base) = dir(s)
8      update s, n'
9      if s.isVia() = base.isVia() and not pointsTo(s.destPoint() dir(s), n.point) :
10         recursiveCall(n'.point, n'.parent, n); return
11     update s, n'
12 while dir(base) ≠ dir(s)
13     if aligned(base, s):
14         shortcut ← shortcutPoint(base, s, n')
15         parent ← n'.parent
16     do :
17         update s, n'
18         if s = nil : endShortcut(shortcut, parent, n); return
19         if s é orthogonal a base :
20             if pointsTo(s.destPoint(), inverse(n'.dir), n.point) :
21                 if caminho intercepta a si mesmo :
22                     Corrige caminho; return
23                 shortcut ← shortcutPoint(base, s, n')
24                 parent ← n'.parent
25     while s orthogonal a base
26     if s = nil or  dir(base) = dir(s):
27         endShortcut(shortcut, parent, n); return
28 doShortcut(n, shortcut)
```

Figura 58 – Pseudocódigo da função *verifyShortcut*.

A função *recursiveCall* (Figura 61), realiza um preparatório para a chamada recursiva do algoritmo. O parâmetro *parent* é o nó pai dos nós gerados na inicialização de *O*, em *initOpenSet*. Na linha 3, é necessário verificar se existe alguma entrada de *dest.point* em *R*[d*est.source*], visto que n pode ser *nil* tanto por não existir tal entrada quanto por existir, mas ter o valor *nil* associado. O último caso caracteriza uma busca de caminhos iniciada e não finalizada. Neste caso, nada pode ser feito e o algoritmo retorna (linha 3).

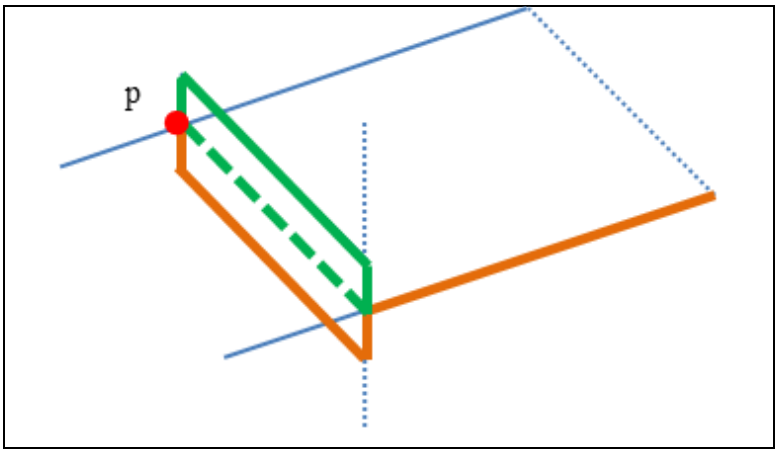

Figura 59 – Ilustração da correção da “ponta” do caminho. O ponto *p* representa a ponta. Em verde, estão as possibilidades de otimização de custo.

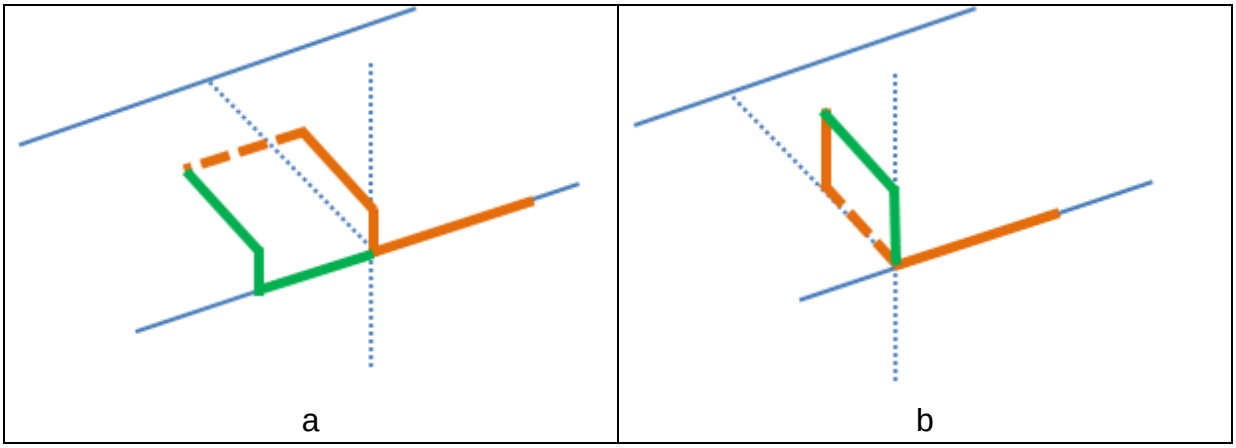

Figura 60 – Ilustração da correção de custo de caminhos *ort* com *jogs.* Em verde, a correção do caminho.

```
recursiveCall(Ponto source, Nó parent, Nó dest)
1  n ← R[dest.source][dest.point]
2  if n = nil:
3      if R[dest.source].contains(dest.point) : return
4      L[dest.source][dest.point] ← lowerbound(dest.source, dest.point, grid)
5      S' ← {source}
6      T' ← {dest.point}
7      O' ← initOpenSet(S', T', parent)
8      n ← SGRouter3D(S', T', O')
9  if n ≠ nil : dest.setParent(n.parent)
```

Figura 61 – Pseudocódigo da função *recursiveCall*.

As maiores diferenças da detecção de atalhos estão na função *doShortcut*. Da mesma forma que na versão anterior do algoritmo, esta função tem como objetivo realizar um atalho direto e, caso não consiga, realiza uma chamada

recursiva. A diferença é que na versão 2D, o atalho direto consiste em um segmento e, na nova versão, este pode ser formado por vários segmentos. A Figura 62 ilustra os casos de atalhos diretos. O caso *a* ocorre quando os pontos a serem atalhados (*p1* e *p2*) estão alinhados (em *x* ou em *y*) e no mesmo *layer*. Neste caso, existem três possibilidades de atalhos. As tentativas são realizadas seguindo a ordem de menor custo dos segmentos de atalho. O caso *b* ocorre quando os pontos estão alinhados, mas em *layers* diferentes. O algoritmo realiza duas tentativas, uma pelo extremo superior e outra pelo inferior. Entre ambos extremos, é possível que haja mais possibilidades de atalhos (como ilustrado na Figura 62b), mas a implementação atual ignora essas possíveis tentativas. Não foi avaliado se é mais vantajoso utilizar apenas duas tentativas ou mais. Em *c*, os pontos possuem os mesmos valores de *x* e *y*, diferindo apenas em *z*. Um único segmento é o suficiente para realizar o atalho. Em *d*, os pontos estão desalinhados e no mesmo *layer*. Como ilustrado na figura, existem seis possíveis caminhos. O algoritmo realiza as tentativas sempre seguindo a ordem de menor custo. Finalmente, em *e*, os pontos estão desalinhados e em *layers* diferentes. Este é outro caso em que podem existir inúmeras possibilidades de atalhos. No entanto, o algoritmo sempre tenta atalhar pelos extremos superior e inferior. No caso da ilustração, não existem outras possibilidades além do demonstrado (existem 5 possibilidades neste caso). Note que estes casos são definidos pela posição dos pontos *p1* e *p2*, não pelo formato do caminho. Para cada caso podem existir vários possíveis caminhos diferentes. Toda vez que um desses segmentos encontra um bloqueio, é necessário criar nós no ponto de intersecção, exatamente como funciona a expansão de nós comuns (ou de via, se o segmento for de via), no caso de intersecção. Da mesma forma que na versão 2D, evitar este passo pode fazer com que o algoritmo não encontre o menor caminho.

O uso de nós de caminho inverso, herdados das melhorias da versão 2D, merece certa atenção. A função *inversePathNode* continua a ser utilizada na função *visited*, em *addNodeList*. No entanto, como existem novos nós de contorno, é natural que existam novas possibilidades de nós de caminho inverso. A Figura 63 mostra alguns exemplos de criação de nós de caminho inverso. O nó *n* tem como pai o nó *p*, e *inv*(*n*) representa o nó de caminho inverso de *n*. Em *a* tem-se o mesmo caso existente na versão 2D, utilizando nós *detour1*. O caso *b* refere-se a nós *detour2*. Note que neste caso, *p* não é o pai imediato de *n*, visto que este possui um nó de *via*

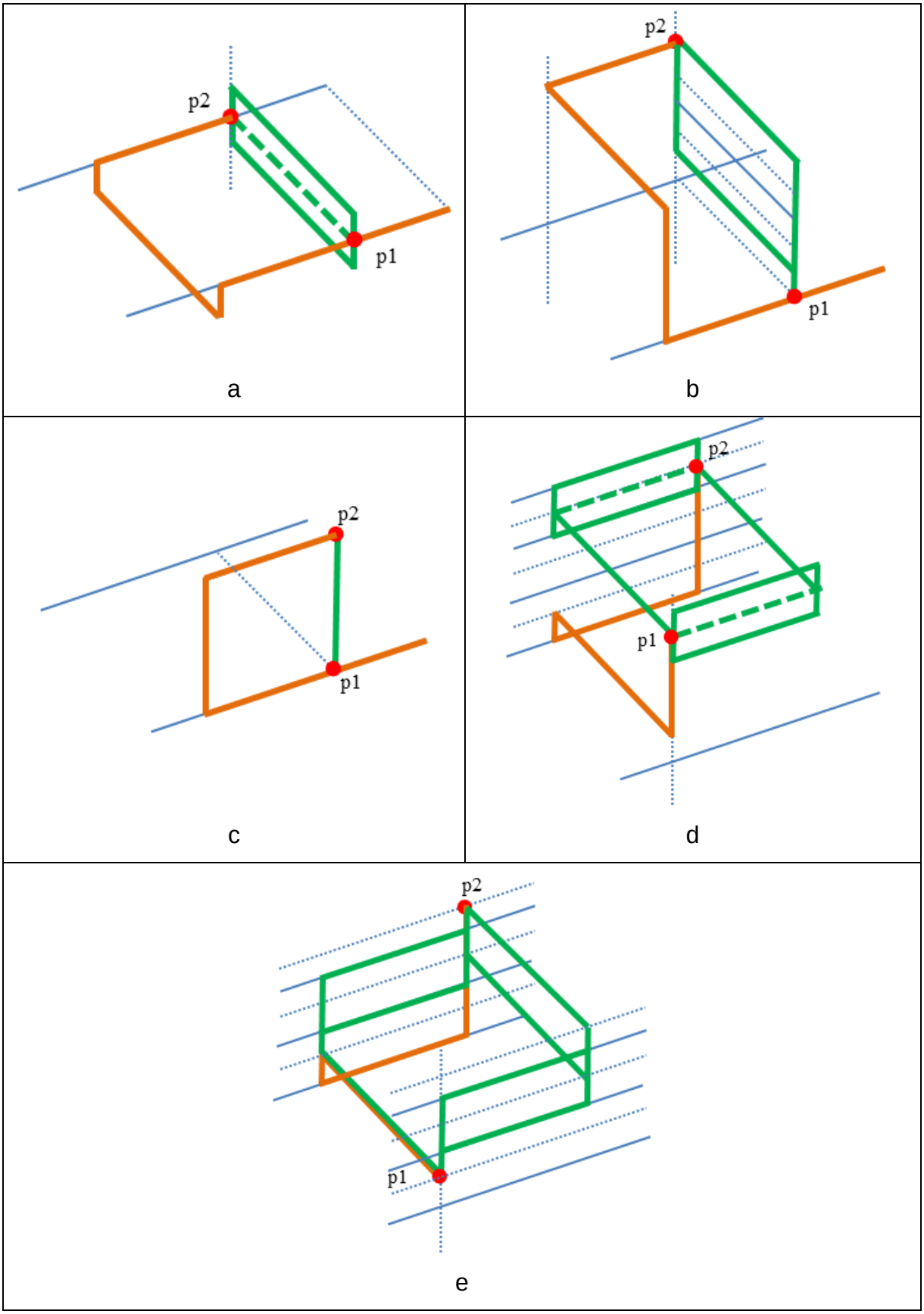

Figura 62 – Ilustração dos casos de atalhamento direto em *doShortcut*. Em laranja, o caminho original. Em verde, os possíveis atalhos. Deseja-se aplicar o atalho de *p1* a *p2*.

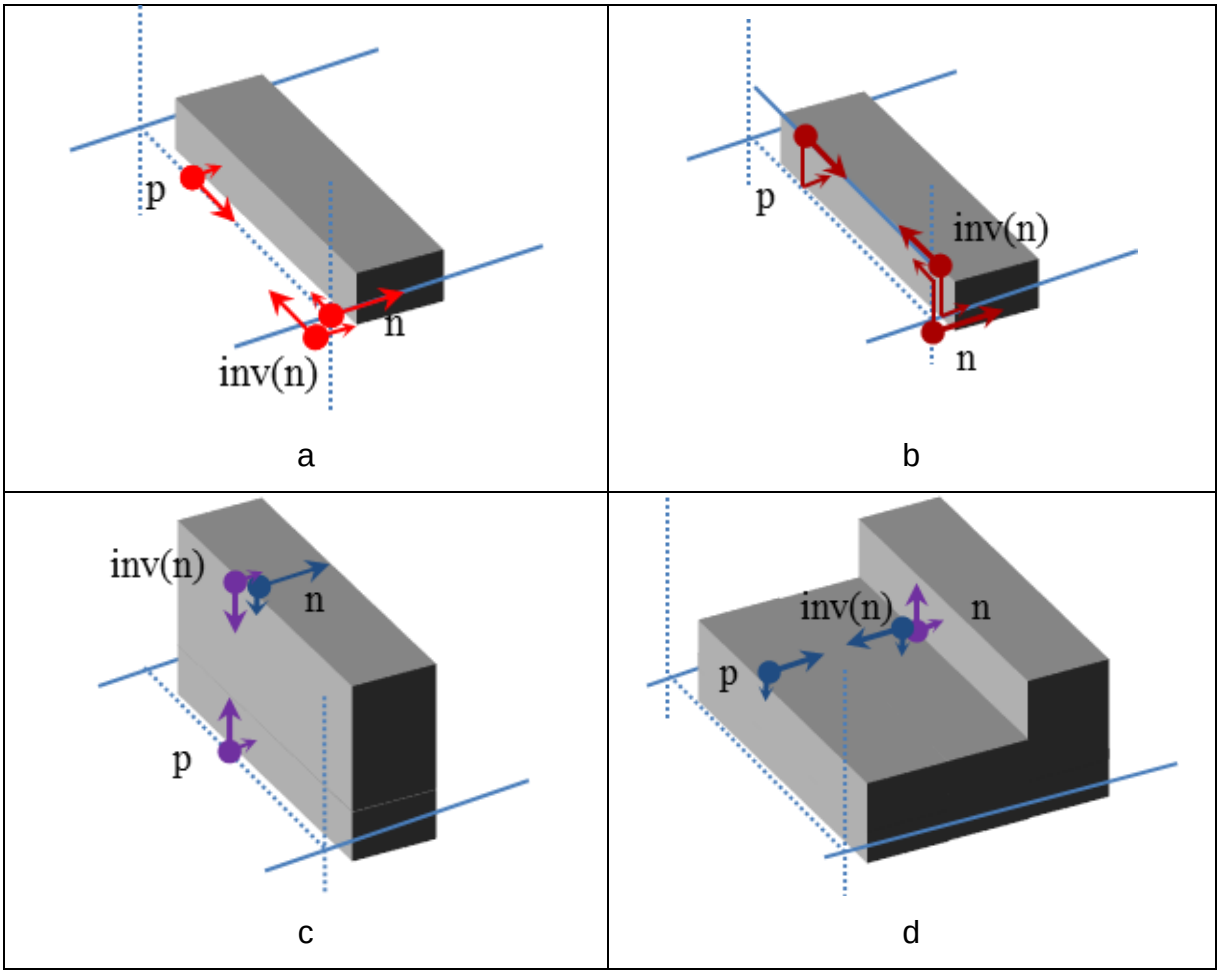

Figura 63 – Ilustração de exemplos básicos de criação de nós de caminho inverso.

auxiliar como pai, localizado no mesmo plano de *p*, apontando para baixo. Nós *detour3* não possuem inversos. No caso *c*, é mostrado um nó *detourVia* sendo gerado como inverso do nó *viaDetour* e em *d*, o oposto ocorre. Cabe relembrar que é a disposição de *n* e *p* que definem *inv*(*n*). O nó *inv*(*n*) sempre aponta (com a direção *dir*) para *p*, e sua direção de contorno é a mesma de *p*. Assim como na versão anterior do algoritmo, *p* deve ser um nó de contorno e, caso não seja, *inv*(*n*) representa o nó *n* com a direção *dir* invertida.

Para encerrar esta sessão, resta ainda dissertar sobre a implementação da função *h* e o motivo pelo qual ela não pode ser implementada pela função *lowerbound*. Na Figura 64a, se *lowerbound* for aplicada no ponto *n*, a função detectará o obstáculo à frente e isso aumentará o custo retornado. No entanto, o exemplo considera que o melhor caminho tenha um custo menor que o custo de *n* usando *lowerbound*. Além disso, para que o algoritmo possa enxergar o melhor caminho, ele precisa primeiro expandir *n*, para depois este ser atalhado (Figura 64b),

gerando um novo nó de custo menor (note que em *a* o caminho tem três vias e em *b* apenas duas). Com isso, se existe outro nó em *O*, com custo menor que o de *n*, mas com o custo maior que o do melhor caminho, o algoritmo pode selecionar esse nó e encontrar um caminho não ótimo. Pelo que foi observado, esta é uma peculiaridade do algoritmo, que ocorre com os nós *detour2*. Na verdade, esse comportamento, que exige em alguns casos a obtenção de um nó de maior custo para a geração de outro de custo menor, já existia na versão 2D. Contudo, isso era resolvido adicionando nós no ponto de intersecção do segmento de atalho (este caso sempre vem acompanhado de uma detecção de atalhos com intersecção). Isto também ocorre na versão 3D, dentro da função *doShortcut*, como já foi citado. No entanto, o caso da Figura 64 é diferente, pois ocorre devido a possível oscilação de *layers* dos nós mencionados. Sendo assim, é importante ressaltar que a função *lowerbound* pode mascarar o melhor caminho para qualquer tipo de nó. Porém, o caso dos nós *detour2* pode levar ao mesmo erro, mesmo sem o uso da função *lowerbound*. Sendo assim, os nós *detour2* têm seu custo calculado como se estivessem localizados no seu pai imediato (campo *fparent*), que é um nó de *via* auxiliar, a menos que o custo se torne pior. Assim, no caso da Figura 64, o custo de *n* vai ser o mesmo para ambos os casos, pois este irá contemplar apenas duas vias.

O cálculo padrão da função *h* é dado pela distância de Manhattan entre o ponto e o alvo, mais o custo das vias existentes entre os *layers* do ponto e do alvo. Se o *layer* é o mesmo, nenhum custo de via é adicionado. Contudo, existem algumas exceções para o cálculo padrão. A primeira consiste no caso mencionado dos nós *detour2*. A segunda é o caso dos nós *viaDetour*. Esses nós têm grande potencial de prejudicar o desempenho do algoritmo, visto que cada um pode adicionar mais dois do mesmo tipo, no *layer* adjacente, a cada expansão. Considerando isso, o seu custo, caso *detourDir* não aponte para o destino, prevê o deslocamento do caminho em uma unidade, seguindo *detourDir*, mas ao mesmo tempo, cuidando se o caminho até este ponto adjacente não pode ser menor que o *g* atual somado ao deslocamento. Mais precisamente, se $g+h < lowerBound(source, pt, grid) + h + dif$, $h = lowerBound(source, pt, grid) + h + dif - g$, onde *dif* é o custo de *point* a seu ponto adjacente (seguindo *detourDir*) *pt*, e *h* é o custo seguindo o cálculo padrão .

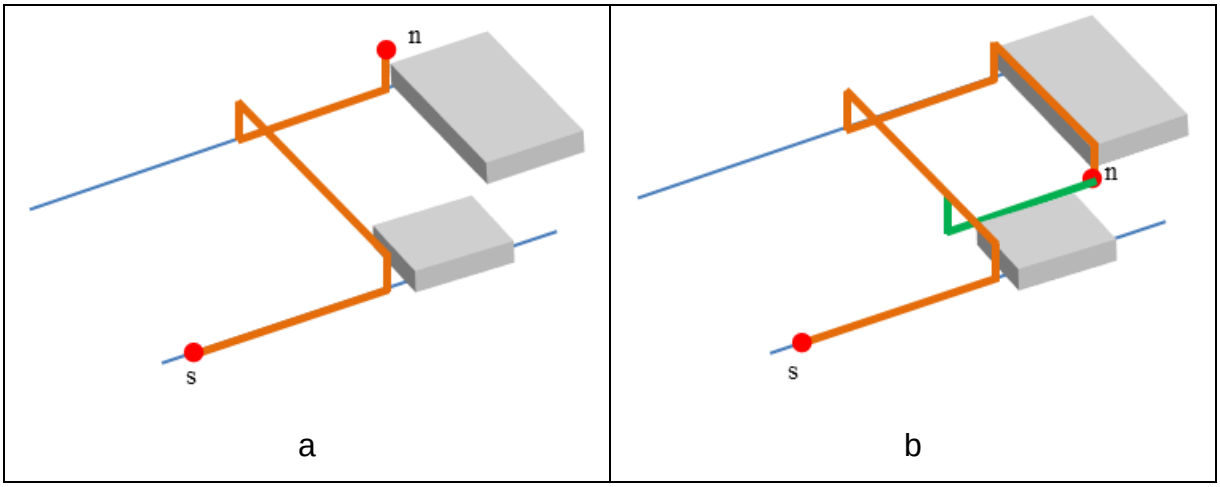

Figura 64 – Ilustração de um caso em que é necessário obter um nó de custo mais alto para gerar outro de custo menor. O ponto *s* representa o início do caminho e *n* é o último nó do caminho. Em verde tem-se o caminho atalhado.

5.3 Futuras Melhorias Propostas

O algoritmo SG-Router lida com as principais regras de projeto básicas, como o espaço tridimensional, múltiplos *sources* e *targets,* orientação dos *layers*, uso de *jogs* e de vias. No entanto, a implementação atual não prevê diferentes custos de fios, para cada *layer*, apenas diferentes custos de vias. Contudo, isso não retrata um problema que possa ameaçar a lógica do algoritmo, visto que isso pode ser tratado da mesma maneira como a relatividade do custo de vias foi tratada. Existem três situações a serem abordadas para se contemplar diferentes custos de fios. A primeira é na atualização do custo conhecido de um nó (função *g*). Isso é resolvido facilmente por multiplicar a distância entre o nó e seu pai pelo custo de cada ponto do grid, entre o nó e o pai. A segunda situação é no cálculo de custos potenciais, na função *h* e em *lowerbound*. Como estas funções têm como objetivo calcular um limitante inferior para o custo, não há problema se o custo calculado está abaixo do custo real (na verdade isso frequentemente ocorre). Logo, se estas funções ignorassem os custos dos fios, considerando o custo de cada ponto do *grid* como 1, isso não mudaria em nada a optimalidade do algoritmo. No entanto, teria um impacto prejudicial no desempenho, pois quanto mais próximo é o custo potencial do custo verdadeiro, mais rápido o algoritmo encontra o caminho. Independentemente dessas considerações, o algoritmo pode muito bem considerar os custos dos fios, com base no *layer* atual e no *layer* de destino. A terceira situação refere-se à ordem de tentativas de atalhos diretos, em *doShortcut*. Da maneira como apresentado, o

algoritmo, quando executando tentativas de atalhos diretos entre dois pontos em *layers* distintos (com diferença de 3 pelo menos), pode executar tentativas tanto por cima quanto por baixo, não importando a ordem, pois o custo será o mesmo. Contudo, ao considerar custos de fios para cada *layer*, é necessário priorizar as tentativas nos *layers* inferiores (considerando que o custo aumente conforme os *layers*).

Outra consideração sobre regras de projeto é a distância mínima entre fios e vias. Esta, pode ser facilmente resolvida, fora do algoritmo, pela estrutura do *grid*, a qual pode considerar as distâncias mínimas nas consultas de intersecção, com base nas especificações do fio, que estão de acordo com o *layer*.

Como discutido, a função *h* não pode prever, com maior precisão, o custo potencial, visto que isso pode mascarar a obtenção do melhor caminho. Sendo assim, pretende-se investigar sobre esta questão, afim de identificar casos em que seja possível utilizar um cálculo mais aprimorado.

Outra investigação que se faz necessária, é a avaliação do impacto de desempenho ocasionado por se realizar um maior número de tentativas de atalhos diretos. Essa possibilidade aparece nos casos onde a diferença entre os *layers,* dos dois pontos a serem atalhados, é de pelo menos três. Considerando que as conexões dos pinos (não os pinos propriamente ditos) se encontram no primeiro *layer*, e que o número de *layers* é imensamente pequeno em relação a largura e altura do *grid*, é esperado que valha a pena tentar mais atalhos diretos, pois a ausência de uma tentativa pode implicar em uma recursão custosa.

O tratamento de múltiplos *sources* e *targets* merece um estudo mais aprofundado. Como o algoritmo trabalha com nós seguindo a mesma metodologia de custos dos algoritmos clássicos, o tratamento da funcionalidade de múltiplos pontos foi realizado da maneira clássica. No entanto, é possível que o algoritmo encontre dificuldades para esta abordagem, requerendo tratamentos específicos para alguns casos. Estes são a garantia de que cada *target* deve estar sendo seguido por algum *source*, no início da execução do algoritmo; a utilização do *source* e *target* de cada nó, para o critério de igualdade de nós; a imposição de um *target* para um nó, em certas ocasiões. O segundo caso pode ser muito prejudicial para o desempenho. No terceiro, não se sabe se a imposição pode fazer com que algum *target* seja “perdido”. Mesmo que o primeiro caso garanta que cada *target* tenha algum *source* no início, é possível que o melhor *source* para um dado *target* não

seja o escolhido no inicio da execução. Todos esses casos devem ser estudados com mais cautela, afim de se propor uma solução mais adequada com o comportamento do algoritmo.

A atual maneira que o algoritmo lida com nós visitados é marcar um nó expandido como visitado e, caso futuramente outro nó com as mesmas características for gerado, ele é descartado. No entanto, é possível otimizar isso, pelo menos para nós comuns. Todos os pontos contidos no segmento gerado pela expansão de um nó comum possuem o mesmo custo. Sendo assim, é possível utilizar segmentos, ao invés de nós, para determinar se um nó é visitado. Quando um nó comum é expandido, seu segmento (assim como seu custo) é adicionado em um conjunto de visitados. Quando um nó comum é gerado, é verificado se este está contido em algum segmento mencionado, caracterizando assim, como um nó visitado. Quanto a nós de contorno, é requerida uma atenção especial, visto que a expansão pode percorrer pontos com custos crescentes.

O algoritmo de detecção de atalhos consiste numa adaptação da versão 2D. No entanto, seria mais adequado se o algoritmo fosse repensado para o escopo 3D. Isso poderia identificar certos padrões que não são previstos com a atual implementação. Além disso, deve-se investigar sobre padrões de caminhos 3D, afim de reduzir o número de recursões e/ou o número de procedimentos de atalhos. Esta é uma das principais necessidades de otimização, visto que a explosão recursiva pode ser muito prejudicial ao desempenho e a memória.

A última otimização prevista refere-se à implementação do algoritmo. Na maneira atual, quando a expansão de um nó é interceptada, são gerados vários nós no ponto de intersecção. Em alguns casos, são gerados em torno de 15 nós. Isso pode ser prejudicial ao desempenho. Sendo assim, se todos esses nós pudessem ser “comprimidos” em uma única estrutura de dados, a qual seria inserida em *O*, ao invés de cada nó individualmente, o número de elementos em *O* seria consideravelmente reduzido. É evidente que alguns desses nós encontram-se em pontos diferentes de outros. Ainda assim, parte desses nós, como os nós *detour2* e *detour3*, poderiam ser comprimidos igualmente e, no momento da expansão, serem deslocados para cima, ou para baixo, antes do procedimento de expansão apresentado. Essa otimização também aliviaria o pequeno esforço computacional da função *applyShortcuts*, que busca selecionar o(s) nó(s) mais adequado(s) para a realização do procedimento de atalhos, visto que o procedimento seria aplicado ao

conjunto de nós comprimidos, ao mesmo tempo. Esta otimização, somada a possíveis padrões de caminhos investigados, poderia evitar o duplo procedimento de atalhos, ocorrido com nós que não são filhos diretos do nó expandido.

6 EXPERIMENTOS E RESULTADOS COM O SG-ROUTER

O objetivo dos experimentos é avaliar o desempenho e a optimalidade do SG-Router em relação ao algoritmo de Hetzel. Primeiramente, será mostrado um conjunto de experimentos que visam avaliar ambos algoritmos em um cenário de busca mais genérico. O segundo conjunto de experimentos visa avaliar ambos algoritmos em um cenário mais próximo do roteamento detalhado da síntese física de circuitos. Os experimentos foram realizados em um computador com sistema Linux com 130Gb RAM, CPU AMD *Opteron* de 1,4 GHz. Ambos algoritmos foram implementados na linguagem Java.

O roteamento se deu em um *grid* tridimensional, com um *pitch* igual em todos os *layers*, visto que a implementação atual do SG-Router não está considerando o desalinhamento de *layers*. Do mesmo modo, como a implementação atual também não considera diferentes custos de fios em diferentes *layers*, o custo de uma conexão entre dois pontos adjacentes do mesmo *layer*, seguindo a direção preferida, é 1 em todos os *layers*. Foi utilizado o custo 3 para *jogs*. Para vias, as do primeiro *layer* possuem custo 1,5, e as demais possuem 0,5 a mais em relação ao *layer* anterior. Os pontos de origem e o destino de cada busca foram gerados aleatoriamente dentro das três dimensões do *grid*. Quando foram utilizadas múltiplas origens e destinos, estes foram localizados à esquerda dos pontos sorteados, formando um retângulo de 10 pontos de largura por 5 de altura. O mesmo conjunto de pares origem-destino foi utilizado para ambos algoritmos. Assim, ambos executaram as mesmas buscas nos mesmos *grids*. Para fazer uma comparação justa, a função de estimativa *h*, do algoritmo de Hetzel, foi implementada com a função *lowerbound*, apresentada na sessão 5.1, que possui uma estimativa mais aprimorada. Da mesma forma que nos experimentos da versão 2D, foram utilizadas várias densidades e, para cada uma delas, foram gerados obstáculos aleatórios. Os

obstáculos consistem em segmentos de tamanho, posição e orientação aleatórios. O *grid* permaneceu estático durante o roteamento, não sendo atualizado com os caminhos resultantes de cada busca. O objetivo disso é avaliar como ambos algoritmos lidam com cada nível de congestionamento. Além disso, se os caminhos fossem adicionados ao *grid*, o nível de congestionamento aumentaria muito rapidamente, impedindo que fosse executado um grande número de roteamentos por *grid*. O número de buscas e o tamanho do *grid* estão diretamente relacionados nesta questão. Se o número de buscas for muito grande e o *grid* for pequeno, logicamente haverá mais comprimento de fios que o *grid* possa comportar. Se o *grid* for grande demais, ocorre o problema que incentivou a existência do roteamento global, que é a ocorrência de um elevado tempo de execução e consumo de memória para buscas em uma área imensa, fazendo com que o roteamento possa se tornar impraticável. Como se deseja realizar um grande número de buscas, para aumentar a confiabilidade dos resultados, foi utilizado o *grid* estático. Além de variar o nível de congestionamento do *grid*, os experimentos também variaram o tamanho do *grid*. Foram utilizados *grids* de 100, 1000, 10000 e 25000 unidades de largura e altura. O número de *layers* foi 10 em todos os casos. Para todas as densidades foram executadas 100000 buscas.

A Tabela 3 apresenta os resultados para buscas realizadas sem múltiplas origens e destinos. “D” representa a densidade utilizada. Recapitulando, a densidade é a proporção de área ocupada por obstáculos em relação à área livre. A densidade **não** é um percentual propriamente dito. Assim uma densidade igual a 1 representa 100% de área ocupada, não 1%. A coluna “Speed Up” representa quantas vezes o SG-Router é mais rápido que o algoritmo de Hetzel; “*O*” é o número total de nós abertos; “Média *O*” é o tamanho médio de *O* a cada inserção em *O*; “WL” é o custo total de todos os caminhos obtidos; “Perdas SG” representa o número de vezes que o SG-Router obteve um caminho de maior WL que o ótimo (obtido pelo algoritmo de Hetzel); “Mem SG” representa o número de buscas que o SG-Router estourou a memória; “Falhas Hetzel” representa o número de vezes que o algoritmo de Hetzel se manteve na mesma busca por cerca de 2 dias, exigindo o cancelamento da busca, e partindo para a próxima. Para cada estouro de memória do SG-Router, não se sabe se o mesmo aconteceria com o algoritmo de Hetzel, ou se ele somaria mais um ponto em “Falhas Hetzel”, pois para cada par de pontos o SG-Router foi

executado primeiro, e o algoritmo de Hetzel não foi executado posteriormente, nesses casos.

A maior densidade utilizada foi de 0,1. Para valores maiores (como 0,2 e 0,3), o SG-Router apresentou muitos estouros de memória, dificultando a execução dos experimentos. Para os *grids* de tamanho 100 foram utilizadas densidades maiores, visto que o tamanho do espaço de busca é menor, implicando em um menor esforço computacional. Nesse caso, a densidade máxima foi de 0,3. Seria possível utilizar densidades maiores, porém, isso não se mostrou útil, visto que as densidades 0,2 e 0,3 já foram o suficiente para mostrarem a tendência do comportamento dos resultados ao se aumentar a densidade.

A Figura 65 apresenta um gráfico que mostra a influência do aumento do tamanho do *grid* no *speedup*. Quanto maior é o tamanho do *grid* maior é o ganho em desempenho do SG-Router. Isso é explicado pelo fato de que o algoritmo de Hetzel tem seu desempenho afetado pela área disponível (não ocupada por obstáculos), enquanto o SG-Router não é afetado pela área, mas pelo perímetro dos obstáculos contornados e pelo procedimento de atalhos. Quanto maior o *grid*, maior é a área disponível. O perímetro também aumenta, mas a área aumenta muito mais. Logo, o algoritmo de Hetzel é mais afetado. Obviamente isso é uma tendência genérica, com ênfase na variação do tamanho do *grid* apenas, considerando uma mesma densidade. A variação da densidade pode influenciar nesse comportamento, como será visto. Para os *grids* menores, o *speedup* é bem menor, podendo ser até desfavorável. Isso é natural, considerando que o espaço de busca é muito pequeno fazendo com que a perda em desempenho do algoritmo de Hetzel, seja desprezível, enquanto que o SG-Router possui a detecção de atalhos e a recursividade, que pode acabar complicando os casos simples.

A Figura 66, evidencia o comportamento do *speedup* ao se variar a densidade. A Figura 67 apresenta uma visão mais detalhada dos gráficos, para os *grids* menores. Para as menores densidades, o *speedup* é pequeno. A medida que a densidade aumenta, o *speedup* aumenta intensamente, até alcançar 0,01 de densidade. A partir desse ponto o *speedup* diminui drasticamente. Mais uma vez , a maneira que os algoritmos lidam com os obstáculos explica esse comportamento. Em um cenário com densidade baixa, existem poucos obstáculos e, consequentemente, poucos bloqueios de caminho. O cenário motivador da criação do ST-Router, apresentado na Figura 15, ou algum cenário diferente, mas

Tabela 3 – Resultados da comparação do SG-Router com o algoritmo de Hetzel, utilizando um ponto de origem e um ponto de destino. Por padrão o tempo é medido em segundos, mas pode ser em minutos, quando seguido da letra “m”.

***grid* 100x100**												
D	**Tempo**		**Speed up**	***O***		**Média *O***		**WL**		**Perdas SG**	**Mem SG**	**Falhas Hetzel**
	SG	**Hetzel**		**SG**	**Hetzel**	**SG**	**Hetzel**	**SG**	**Hetzel**			
0,001	2,2	3,1	1,43	$8,9.10^5$	$1,9.10^6$	4	10	7945621,5	7945621,5	0	0	0
0,005	1,4	1,7	1,18	$9,3.10^5$	2.10^6	4,1	10,3	7992516	7992516	0	0	0
0,01	1,4	2,1	1,47	$9,6.10^5$	$2,2.10^6$	4,2	10,6	7963902	7963900	1	0	0
0,05	2,6	2,6	0,99	$1,4.10^6$	$2,9.10^6$	5,5	13	7995183	7995183	0	0	0
0,1	7,6	5,6	0,73	$2,8.10^6$	$3,5.10^6$	7,6	16	7305393,5	7305392,5	1	0	0
0,2	190	11	0,06	$4,9.10^7$	6.10^6	15,2	22,4	8115201	8115196	8	0	0
0,3	55**m**	24	0,007	$7,0.10^8$	$8,7.10^6$	29	28	8209681,5	8209663,5	19	0	0
***grid* 1000x1000**												
D	**Tempo**		**Speed up**	***O***		**Média *O***		**WL**		**Perdas SG**	**Mem SG**	**Falhas Hetzel**
	SG	**Hetzel**		**SG**	**Hetzel**	**SG**	**Hetzel**	**SG**	**Hetzel**			
0,001	2,4	3,5	1,44	9.10^5	$2,2.10^6$	4	10,7	67991376	67991376	0	0	0
0,005	2,6	5,4	2,08	$9,5.10^5$	$2,8.10^6$	4,3	12,8	67969680	67969680	0	0	0
0,01	3	9,4	3,09	10^6	$4,0.10^6$	4,5	16,9	67937592	67937592	0	0	0
0,05	11,4	30	2,69	$3,3.10^6$	$1,2.10^7$	6,4	42	68300000	68300000	0	0	0
0,1	16**m**	187**m**	11,65	$7,3.10^7$	$2,3.10^7$	12,3	76	68669848	68669848	0	0	0
***grid* 10000x10000**												
D	**Tempo**		**Speed up**	***O***		**Média *O***		**WL**		**Perdas SG**	**Mem SG**	**Falhas Hetzel**
	SG	**Hetzel**		**SG**	**Hetzel**	**SG**	**Hetzel**	**SG**	**Hetzel**			
0,001	3,4	56	16,8	$9,3.10^5$	$3,8.10^6$	4,2	16,2	666757180	666757180	0	0	0
0,005	2,6	7,5**m**	168	$9,7.10^5$	$1,2.10^7$	4,4	43	668467200	668467200	0	0	0
0,01	3,4	21**m**	384	$1,1.10^6$	$2,2.10^7$	4,7	75	667194820	667194820	0	0	0
0,05	2,4**m**	119**m**	49	$6,6.10^7$	10^8	7,6	336	670168900	670168900	0	0	0
0,1	27**m**	493**m**	17,9	$3,9.10^8$	$1,8.10^8$	22,9	602	666890648	666890648	0	7	2
***grid* 25000x25000**												
D	**Tempo**		**speed up**	***O***		**Média O**		**WL**		**Perdas SG**	**Mem SG**	**Falhas Hetzel**
	SG	**Hetzel**		**SG**	**Hetzel**	**SG**	**Hetzel**	**SG**	**Hetzel**			
0,001	3,5	18,5**m**	315	$9,4.10^5$	$7,5.10^6$	4,2	28,3	1670574210	1670574210	0	0	0
0,005	2,9	56**m**	1.149	$9,8.10^5$	$2,5.10^7$	4,4	87	1671197950	1671197950	0	0	0
0,01	4,4	140**m**	1.897	$1,3.10^6$	$5,2.10^7$	5,1	173	1665469310	1665469310	0	0	0
0,05	29**m**	927**m**	32	$2,3.10^7$	$2,4.10^8$	16,1	784	1673912598	1673912598	0	3	0
0,1	145**m**	1.638**m**	11,2	$1,6.10^9$	$4,2.10^8$	47	1.383	1578103866	1578103866	0	15	2

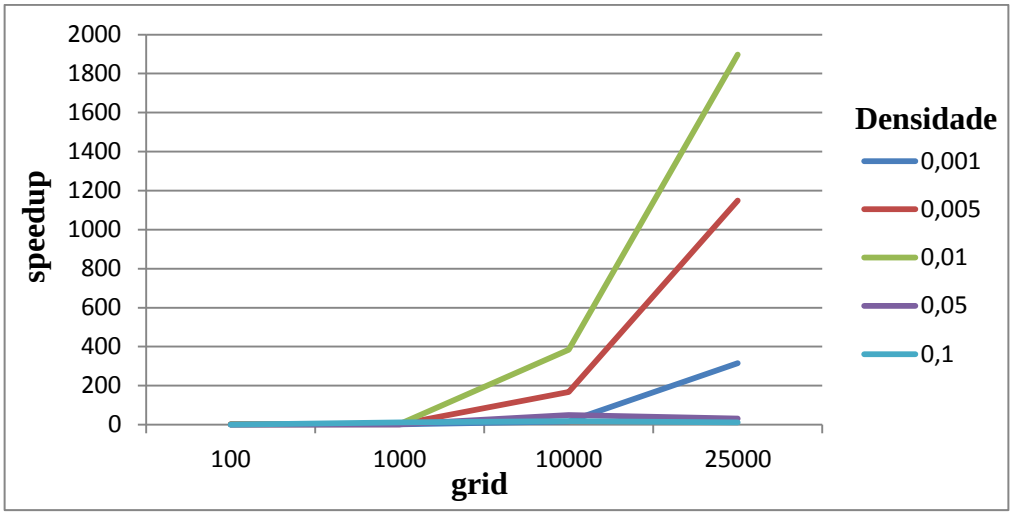

Figura 65 – Gráfico que mostra, com base nos dados da Tabela 3, o aumento do *speedup* (eixo vertical) do SG-Router a medida que o tamanho do *grid* (eixo horizontal) aumenta. Cada curva representa uma densidade.

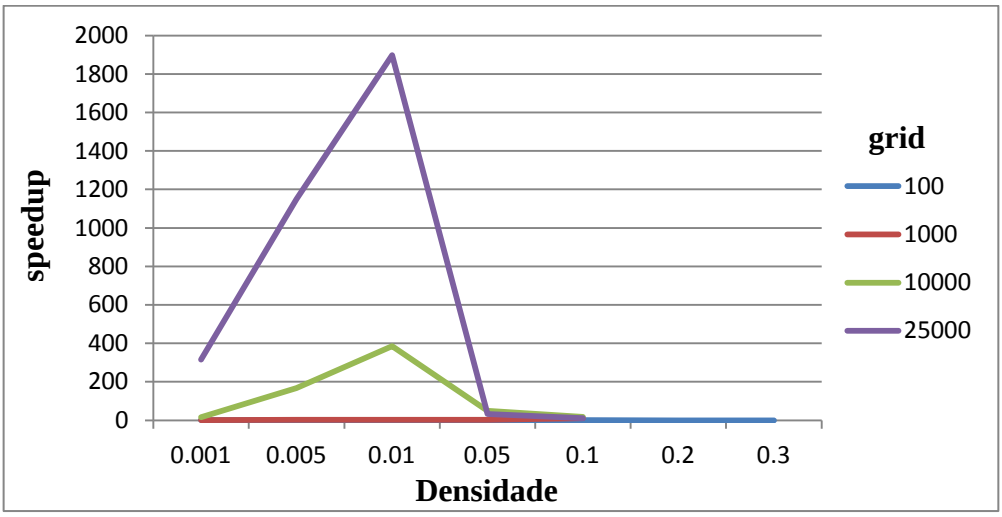

Figura 66 – Gráfico que mostra, com base nos dados da Tabela 3, o comportamento do *speedup* (eixo vertical) do SG-Router a medida que a densidade (eixo horizontal) aumenta. Cada curva representa um tamanho de *grid*.

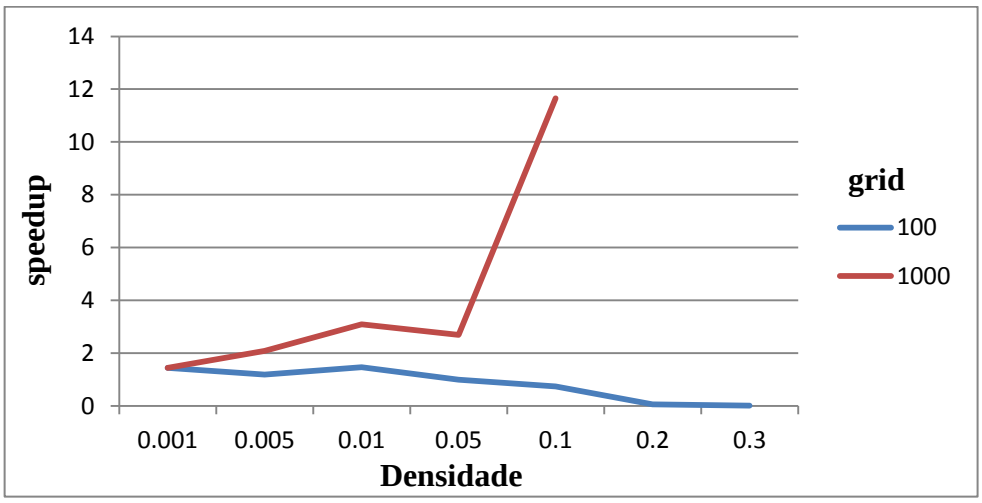

Figura 67 – Visão detalhada do gráfico anterior, para os tamanhos de *grid* 100 e 1000.

equivalente (no sentido de proporcionar o mesmo comportamento da busca), não costuma ocorrer. Assim, o ST/SG-Router possui pouca vantagem. A medida que o número de obstáculos aumenta, o cenário da Figura 15 se torna cada vez mais comum. É nesse ponto que o SG-Router aumenta sua vantagem. O *speedup* cresce drasticamente, alcançando o pico de 1897 no *grid* de tamanho 25000. Contudo, quanto mais obstáculos um *grid* possui, menor sua área livre, e quanto menor a área livre menor o espaço de busca do algoritmo de Hetzel. Com isso, a partir de certa densidade (0,01, no caso), a vantagem que o SG-Router apresenta naqueles cenários não compensa a vantagem que o algoritmo de Hetzel apresenta com um espaço de busca menor. Além disso, a medida que a densidade aumenta, o *grid* se torna cada vez mais semelhante a um labirinto, com corredores estreitos e cheios de voltas. Nesses casos, algoritmo de Hetzel começa a ganhar vantagem, ou pelo menos atenuar a vantagem do SG-Router, visto que a área é relativamente pequena, e o perímetro das “paredes dos corredores” é maior. Mesmo reduzindo drasticamente o *speedup* a partir de 0,01, ele permanece superior na maior densidade testada nos *grids* maiores, variando de 11 a 17. A partir de 0,1 o SG-Router apresenta estouros de memória muito frequentes. Este problema não é explicado pelo simples fato de o algoritmo contornar perímetros maiores de obstáculos. Se o problema fosse esse, o algoritmo simplesmente se tornaria um pouco mais lento. O grande problema são as buscas recursivas. Cada busca possui suas estruturas de dados, como a lista de abertos *O* e a estrutura que armazena o histórico de nós (*O* + fechados). A medida que vão ocorrendo buscas recursivas, novas estruturas vão sendo criadas e manipuladas. Quando as buscas ocorrem uma dentro da outra, as estruturas ficam todas armazenadas na memória e, se a pilha de recursão é suficientemente grande, ocorre o estouro de memória. Note que não se trata de estouro de pilha recursiva, mas estouro de memória. Quando uma recursão acaba, idealmente todo conteúdo armazenado na memória deve ser liberado, mas como o responsável por isso na linguagem Java é o *garbage collector* e, este atua de forma automática, não se sabe exatamente como e quando a memória está sendo liberada. Com isso, é possível que este seja um motivo contribuinte ao acúmulo de memória, mas isso ainda deve ser investigado. De qualquer maneira, se a pilha recursiva for muito grande e cada nível de recursão possuir uma grande quantidade de memória correspondente, ocorrerá um estouro de memória. Finalmente, considerando que a medida que o algoritmo constrói caminhos com

muitas voltas, as chamadas recursivas aumentam cada vez mais. Se o caminho possui muitas voltas devido aos obstáculos, como no caso de um labirinto, não haverão atalhos diretos. Ainda assim, o algoritmo realizará muitas chamadas recursivas em vão. Logo, fica explicado o motivo dos estouros de memória para densidades mais altas. Quanto maior a densidade, maior o número de obstáculos, maior é o número de voltas de um caminho e assim, maior é o número de recursões, que tendem a aumentar, por encontrarem frequentemente mais obstáculos ao se tentar realizar o atalho direto. O caso do *grid* de tamanho 100 é uma evidência para essa explicação. O pequeno tamanho permitiu a utilização de densidades maiores que 0,1, porém, para 0,3, o SG-Router apresentou um enorme tempo de execução e consumo de memória. Note que o consumo de memória ocasionado pela recursão está diretamente ligado ao tempo de execução. O tempo está ligado ao perímetro de obstáculos contornados e ao tamanho dos caminhos percorridos na detecção de atalhos, mas também é muito afetado pela manipulação de *O* e da estrutura relacionada. Para 0,3 de densidade o SG-Router apresentou um tempo de 55 minutos enquanto o algoritmo de Hetzel apresentou apenas 24 segundos. Considerando que um *grid* de tamanho 100 é muito pequeno, não é cabível explicar essa enorme discrepância pelo aumento de perímetro contornado apenas. Com isso, deve existir outro fator responsável por esse comportamento. A grande ocorrência de buscas recursivas explica perfeitamente esse fenômeno.

Em todos os *grids*, exceto o de tamanho 100, o SG-Router obteve o caminho ótimo em todas as buscas. No menor *grid*, para densidades 0,01 e 0,1 o algoritmo não encontrou o caminho ótimo em apenas uma busca em cada *grid*. A diferença em WL foi de 2 de um total de 7963900, e de 1 de um total de 7305392,5, para as densidades 0,01 e 0,1, respectivamente. Na maior densidade testada, houve 19 buscas não ótimas. O acréscimo de WL de todas essas buscas soma 18 unidades apenas, de um total de 8209663,5. O acréscimo de WL em todos os casos foi mínimo. O número de buscas não ótimas também foi muito pequeno, em relação ao total de buscas (100000). Essa perda de optimalidade não está relacionada a mecânica do algoritmo em si. O que ocorreu foi o mesmo da primeira versão do ST-Router. Em alguns casos, o algoritmo está deixando de criar nós. Mais precisamente, nós *viaDetour jogs* não estão sendo criados. Gerar esses nós compromete o desempenho da busca, ao passo que não gera-los não influencia na optimalidade, na grande maioria dos casos. Essas perdas de optimalidade

apareceram mais no *grid* pequeno pois a busca tem menos liberdade de escolha de caminhos, visto que o espaço é menor. Assim, se o melhor caminho deve passar por uma região muito específica, a qual não pode ser explorada por não existirem nós que façam isso, o melhor caminho não é encontrado. Outro fator relacionado é a densidade. Quanto maior a densidade, mais o *grid* se aproxima de um labirinto e mais complexos são os possíveis caminhos, ao passo que a área livre é menor, o que influi no que foi mencionado acima. Além disso, o *grid* pequeno permite a exploração de mais possibilidades de busca (origem-destinos), em proporção ao seu tamanho. Por exemplo, 100000 buscas (pares origem-destino) aleatórias em um *grid* de 100x100x10 se aproximam mais do número total de buscas possíveis nesse *grid*, do que em um *grid* de 10000x10000x10.

A análise de nós abertos e tamanho médio de *O* é igual à versão 2D. Normalmente, o número de nós abertos no algoritmo de Hetzel é maior, pois como já foi dito ele é mais sensível a área e, na maioria das densidades, a área disponível é bem maior que o perímetro. Para cenários mais congestionados, como 0,1 ou mais, isso acaba se invertendo. Ainda assim o SG-Router é mais rápido visto que o tamanho médio de *O* a cada inserção é bem menor que no algoritmo de Hetzel.

Os resultados do roteamento com múltiplas origens e destinos é apresentado na Tabela 4. O gráfico de tamanho de *grid* é apresentado na Figura 68. Os gráficos da densidade são apresentados nas figuras 69 e 70. Os resultados seguem a mesma linha dos anteriores. *Grids* maiores oferecem um desafio maior para o algoritmo de Hetzel. Densidades pequenas, não influenciam muito no *speedup* do SG-Router. Densidades maiores fazem o algoritmo de Hetzel atuar de forma pobre, enquanto o SG-Router atua de forma extremamente rápida. A partir de certo ponto, a medida que a densidade aumenta, o *trade-off* de área e perímetro/complexidade de caminho se torna cada vez mais desfavorável para o SG-Router. A questão do WL e dos estouros de memória permanece igual. Contudo, ocorreram menos estouros. Isto se deve ao fato de que existem mais opções de caminhos. Quando um par origem-destino se torna desfavorável no custo, existem vários outros pares que podem ser investidos pela busca.

Uma diferença em relação aos experimentos anteriores é que até a densidade 0,05 (com exceção do maior *grid*) o tempo do SG-Router variou pouco. Isto também se deve ao fato dos múltiplos pontos proporcionarem certa flexibilidade na busca. Se uma rota que parte de uma origem específica encontra uma área congestionada,

existem várias outras origens que podem fornecer rotas que não passem pela zona congestionada.

O *speedup*, apesar de ser bem alto, foi bem menor que nos experimentos anteriores. O SG-Router não lida com múltiplos pontos de forma tão natural como o algoritmo de Hetzel. Ainda assim, os resultados foram bons, pois a mecânica do

Tabela 4 – Resultados da comparação do SG-Router com o algoritmo de Hetzel, utilizando múltiplos pontos de origem e de destino. Por padrão o tempo é medido em segundos, mas pode ser em minutos, quando seguido da letra “m”.

***grid* 100x100**												
D	**Tempo**		**Speed up**	***O***		**Média *O***		**WL**		**Perdas SG**	**Mem SG**	**Falhas Hetzel**
	SG	**Hetzel**		**SG**	**Hetzel**	**SG**	**Hetzel**	**SG**	**Hetzel**			
0,001	38	10	0,3	$1,9.10^7$	4,3.106	95	21	67231352	67231352	0	0	0
0,005	39	14,5	0,4	$1,9.10^7$	5,1.106	95	24	67105912	67105912	0	0	0
0,01	40	20	0,5	$1,9.10^7$	6,2.106	94	27	67195680	67195680	0	0	0
0,05	47	53	1,1	$1,9.10^7$	1,3.107	92	48	67173752	67173752	0	0	0
0,1	80**m**	360**m**	4,5	$9,8.10^7$	2,3.107	91	78	67138928	67138928	0	0	0
0,2	3,3**m**	34	0,17	$3,6.10^7$	6,2.106	80	29	6171388,5	6171384	5	0	0
0,3	104**m**	47	0,008	$9,7.10^8$	9,5.106	94	34	7252478	7252458	22	0	0
***grid* 1000x1000**												
D	**Tempo**		**Speed up**	***O***		**Média *O***		**WL**		**Perdas SG**	**Mem SG**	**Falhas Hetzel**
	SG	**Hetzel**		**SG**	**Hetzel**	**SG**	**Hetzel**	**SG**	**Hetzel**			
0,001	42	12,5	0,3	$1,9.10^7$	$4,3.10^6$	95	21,7	67044336	67044336	0	0	0
0,005	40	16	0,4	$1,9.10^7$	$5,3.10^6$	95	24,8	67202784	67202784	0	0	0
0,01	40	20	0,5	$1,9.10^7$	$6,3.10^6$	95	28	67086708	67086708	0	0	0
0,05	50	2**m**	2,3	2.10^7	$1,3.10^7$	92	49	67034008	67034008	0	0	0
0,1	80**m**	360**m**	4,5	$9,8.10^7$	$2,3.10^7$	91	78	67138928	67138928	0	0	0
***grid* 10000x10000**												
D	**Tempo**		**Speed up**	***O***		**Média *O***		**WL**		**Perdas SG**	**Mem SG**	**Falhas Hetzel**
	SG	**Hetzel**		**SG**	**Hetzel**	**SG**	**Hetzel**	**SG**	**Hetzel**			
0,001	46	110	2,4	$1,9.10^7$	$5,9.10^6$	96	26	666903680	666903680	0	0	0
0,005	44	11**m**	15	$1,9.10^7$	$1,3.10^7$	95	49	668403580	668403580	0	0	0
0,01	44	19**m**	26	2.10^7	$2,3.10^7$	95	81	665975870	665975870	0	0	0
0,05	57	112**m**	118	$2,2.10^7$	$9,5.10^7$	94	311	666065020	666065020	0	0	0
0,1	44**m**	175**m**	3,9	$5,1.10^8$	$1,7.10^8$	101	538	663866536	663866536	0	4	0
***grid* 250000x250000**												
D	**Tempo**		**Speed up**	***O***		**Média *O***		**WL**		**Perdas SG**	**Mem SG**	**Falhas Hetzel**
	SG	**Hetzel**		**SG**	**Hetzel**	**SG**	**Hetzel**	**SG**	**Hetzel**			
0,001	46	10**m**	13,9	$1,9.10^7$	$8,0.10^6$	96	34	1667844990	1670574210	0	0	0
0,005	46	75**m**	96	$1,9.10^7$	$2,7.10^7$	96	95	1667450880	1671197950	0	0	0
0,01	41	259**m**	378	$1,9.10^7$	$5,6.10^7$	95	190	1664772990	1665469310	0	0	0
0,05	7,2**m**	1.341**m**	185	$1,3.10^8$	$2,4.10^8$	98	761	1693861794	1673912598	0	2	0
0,1	115**m**	2.643**m**	23	$1,4.10^9$	4.10^8	111	1.289	1670318590	1578103866	0	1	0

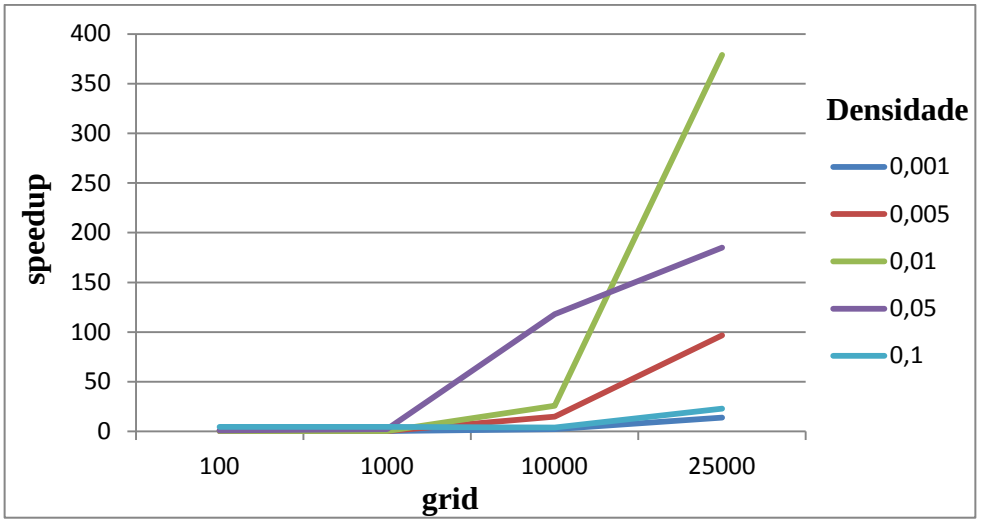

Figura 68 – Gráfico que mostra, com base nos dados da Tabela 4, o aumento do *speedup* (eixo vertical) do SG-Router a medida que o tamanho do *grid* (eixo horizontal) aumenta. Cada curva representa uma densidade.

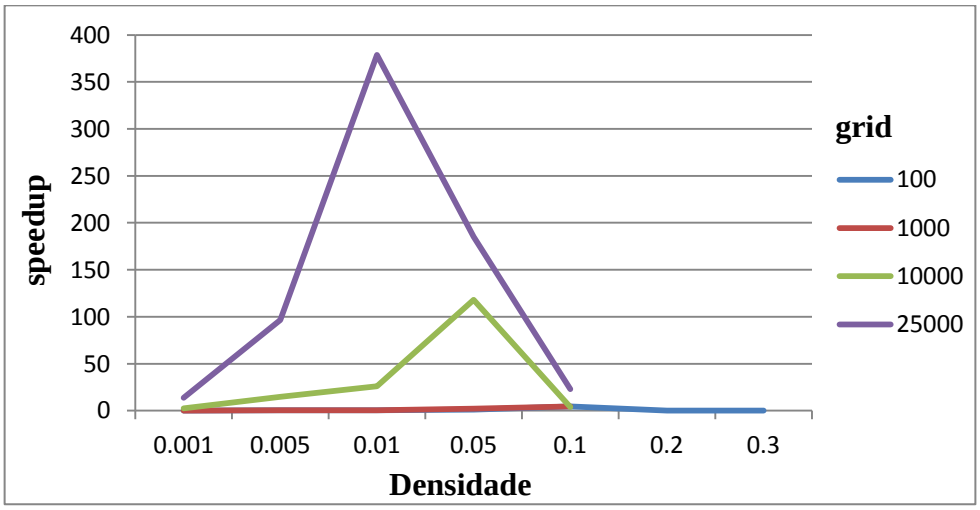

Figura 69 – Gráfico que mostra, com base nos dados da Tabela 4, o comportamento do *speedup* (eixo vertical) do SG-Router a medida que a densidade (eixo horizontal) aumenta. Cada curva representa um tamanho de *grid*.

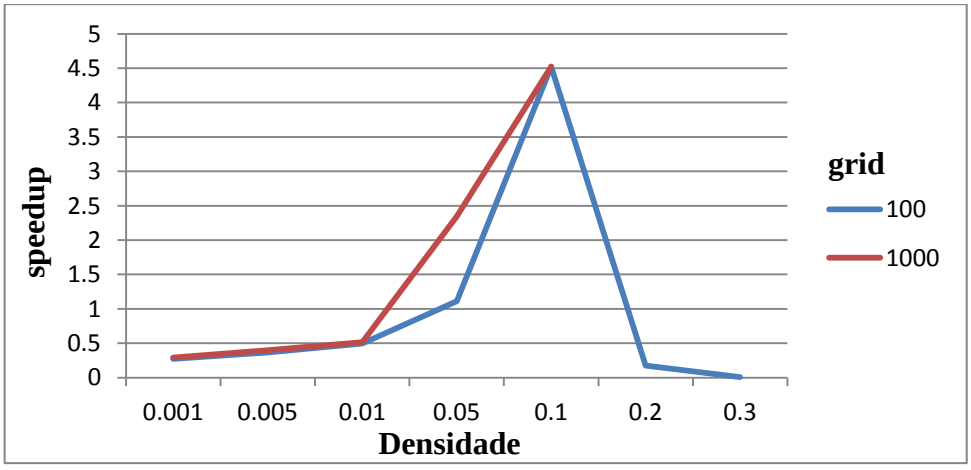

Figura 70 – Visão detalhada do gráfico anterior, para os tamanhos de *grid* 100 e 1000.

Tabela 5 – Resultados da comparação do SG-Router **com** mapeamento de obstáculos e **sem**.

D	Tempo		O		Média O		Mem SG	
	Sem	**Com**	**Sem**	**Com**	**Sem**	**Com**	**Sem**	**Com**
0,001	3,9	3,4	$9{,}3.10^5$	$9{,}3.10^5$	4,2	4,2	0	0
0,005	2,5	2,6	$9{,}9.10^5$	$9{,}7.10^5$	4,4	4,4	0	0
0,01	2,8	3,4	$1{,}1.10^6$	$1{,}1.10^6$	4,8	4,7	0	0
0,05	1,3**m**	2,4**m**	$1{,}9.10^7$	$6{,}6.10^7$	7,9	7,6	0	0
0,1	375**m**	27**m**	$1{,}4.10^9$	$3{,}9.10^8$	60	22,9	6	7

algoritmo, a respeito de múltiplos pontos, permite que caminhos de diferentes origens contornem os mesmos obstáculos, implicando em perda de desempenho.

O mapeamento de obstáculos também foi avaliado na versão tridimensional do algoritmo proposto. A Tabela 5 mostra os resultados da execução do SG-Router com mapeamento e sem mapeamento, no *grid* de tamanho 10000. Para 0,1 de densidade, o SG-Router com mapeamento de obstáculos foi 13,88 vezes mais rápido. O número de nós abertos e o tamanho médio de *O* foram menores, com mapeamento.

O primeiro conjunto de experimentos realizou uma avaliação de ambos algoritmos em um escopo relativamente genérico. Contudo ainda resta avaliar o SG-Router em um cenário mais próximo do roteamento detalhado. Para isso foram obtidos *benchmarks* (ISPD, 2008) da competição de roteamento global do *International Symposium on Physical Design* (ISPD), que é a conferência especialista na área de automação de síntese física de circuitos integrados. Também foi necessária a utilização de um roteador global. O roteador FastRoute4 (XU, 2009) foi escolhido. A metodologia experimental consistiu em carregar um *benchmark*, executar o roteamento global, gerando como saída as regiões para restringir o espaço de busca no roteamento detalhado. Em seguida, para cada rede, foram obtidos os túneis da rede, provenientes do roteador global e o roteamento detalhado se deu com a restrição dos mesmos. Cada rede foi decomposta em redes de dois pontos. Para cada rede de dois pontos, ambos algoritmos foram executados, e o caminho resultante do SG-Router foi adicionado ao *grid*. Contudo, no primeiro (e menor) *benchmark* utilizado (*adaptec1*), o SG-Router estourou a memória, antes da vigésima busca (rede de 2 pontos). Nas primeiras buscas o tempo de execução foi igual ao do algoritmo de Hetzel. Mais adiante, o SG-Router demorou cerca de 300 segundos em uma busca onde o algoritmo de Hetzel finalizou em cerca de 4

segundos. Na próxima busca houve o estouro de memória. Como isso aconteceu bem no início do roteamento, não se pode considerar que o que ocasionou esse comportamento foram obstáculos propriamente ditos, pois estes são fios previamente roteados e não haviam fios suficientes para promoverem enorme congestionamento no circuito. Com isso, os túneis das buscas que ocasionaram a grande queda em desempenho e o estouro de memória, foram analisados. Enfim, foi constatado que esses túneis eram muito complexos, com muitas voltas, ao longo das três dimensões. Considerando que um túnel atua exatamente como uma parede de obstáculos, exigindo que o caminho tome determinados rumos, o caso em questão está completamente de acordo com o que foi discutido na primeira sessão de experimentos. Quando a densidade é alta, o *grid* se assemelha a um labirinto, cheio de voltas, obrigando a busca a formar caminhos complexos, implicando em um maior número de intersecções na tentativa de atalho direto, gerando grande número de buscas recursivas, que repetem o mesmo problema, prejudicando muito o desempenho e/ou estourando a memória. No caso anterior, a alta densidade formava “tuneis de obstáculos” e, no caso em questão, os túneis já existem desde o início. Sendo assim, os experimentos não tiveram continuidade, visto que não é aceitável tolerar estouros de memória em um roteador. O algoritmo SG-Router, em seu estado atual, ainda não pode ser aplicado como algoritmo de busca no roteamento detalhado. Contudo, vale lembrar do enorme ganho em desempenho do algoritmo, nos casos mais favoráveis. Um *speedup* na escala de milhares é algo extremo. Isso demonstra um enorme potencial na aplicabilidade do SG-Router, promovido pela ideia de contornar obstáculos. Uma vez que o problema da recursão e dos estouros de memória estejam resolvidos, mesmo que ao custo de uma pequena perda de optimalidade, o algoritmo proposto terá boas chances de se tornar o estado da arte, no que diz respeito a algoritmos de busca genéricos, com aplicação no roteamento detalhado. Nos casos mais extremos de congestionamento e espaço de busca reduzido, ainda será esperado que o *speedup* não seja muito alto, podendo ser insignificante ou até mesmo menor, assim como nos casos mais simples. No entanto, entre os extremos, se localizam a grande maioria dos cenários, e, nesses casos, o *speedup* é extremo, como os gráficos demonstraram. Se o problema dos estouros de memória for resolvido, sem introduzir outras variáveis que possibilitam perda de desempenho em casos específicos, o *speedup* do SG-Router será como mostrado nos gráficos, com a diferença de que o ponto de declínio se

encontrará mais além do ponto atual (ou se encontrará no mesmo lugar, mas o declínio será mais lento), e o ponto em que o *speedup* se aproxima de 1 (podendo ser maior, menor ou igual) será bem mais próximo da densidade 1 do que atualmente. As propostas de futuras melhorias no SG-Router, apresentadas na sessão 5.3, são promissoras para resolver o problema, principalmente o uso de padrões de caminhos. Se o algoritmo tomar conhecimento de padrões que detectem que uma chamada recursiva é inútil, isto poderá fazer toda a diferença na questão dos estouros de memória.

7 CONCLUSÕES

O processo de síntese de circuitos possui uma enorme complexidade envolvida, exigindo o uso de algoritmos para automatizar os procedimentos. Uma das etapas desse grande processo é o roteamento, que visa determinar as rotas dos fios que conectam os componentes do circuito. Devido a grande complexidade inerente ao problema e o elevado número de componentes que requerem conexões, o roteamento é dividido em duas etapas, possibilitando assim sua solução. A primeira etapa (roteamento global) fornece instruções para a segunda (roteamento detalhado), facilitando sua execução. O roteamento detalhado possui regras de projeto que devem ser atendidas. Ele pode ser realizado utilizando-se algoritmos de busca de caminhos especializados para atender grande parte das regras, ou utilizando-se algoritmos mais genéricos, mas que consigam lidar com as regras mais simples. Dos últimos mencionados, o algoritmo de Hetzel é o estado da arte. Assim, considerando que o roteamento consome muito tempo e que tempo também é um fator importante para a síntese de circuitos, este trabalho propôs um novo algoritmo (SG-Router) de busca de caminhos genérico, mas com capacidade de lidar com algumas das regras de projeto mais simples. O trabalho também apresentou uma série de propostas de otimizações de tempo e WL para a versão preexistente do algoritmo (ST-Router), que funciona apenas no escopo bidimensional.

A versão melhorada do ST-Router foi testada, avaliando-se o impacto de cada proposta de otimização. O algoritmo também foi comparado ao algoritmo de Hetzel, reduzido ao escopo bidimensional. Algumas propostas de otimizações de tempo se mostraram boas enquanto outras foram prejudiciais. Utilizando o algoritmo de Hetzel para validar a optimalidade do ST-Router, foi observado que em todos os roteamentos o ST-Router obteve o caminho ótimo, o que aponta sua optimalidade. No entanto não se tem uma prova formal. As otimizações de WL causaram uma

grande perda de desempenho, mas as otimizações de tempo que se mostraram benéficas atenuaram a perda, garantindo que o ST-Router continue muito mais rápido que o algoritmo de Hetzel.

O SG-Router partiu da versão melhorada do ST-Router. Das propostas de otimizações de tempo, foram herdadas os nós de caminho inverso e o mapeamento de obstáculos, que se mostrou prejudicial na versão bidimensional, para um cenário de obstáculos aleatórios. A implementação atual do SG-Router não contempla algumas regras de projeto desejáveis para um algoritmo de busca, mas a mecânica do algoritmo não parece ter problemas com elas. O SG-Router foi testado em um espaço de busca tridimensional, com obstáculos aleatórios. Nesse cenário, a quantidade de obstáculos (densidade) e o tamanho do *grid* foram variados. O algoritmo de Hetzel também foi executado nos mesmos cenários. O WL obtido pelo SG-Router, em praticamente todas as buscas foi ótimo. Nas que não foi, o aumento foi mínimo. O aumento se deu porque a implementação impediu a criação de certos nós em alguns casos, pois caso contrário, o algoritmo sofre enorme perda em desempenho. Foi observado que quanto maior o tamanho do *grid* maior a vantagem do SG-Router, visto que o algoritmo de Hetzel é bastante sensível ao espaço de busca, enquanto o SG-Router é mais sensível ao perímetro dos obstáculos contornados. Para densidades pequenas a vantagem do SG-Router é pequena visto que o algoritmo de Hetzel não cai nos casos em que seu desempenho é drasticamente reduzido. A medida que a densidade aumenta, o *speedup* do SG-Router aumenta drasticamente, podendo alcançar a casa dos milhares. Contudo quanto mais obstáculos, menor é a área de busca disponível e isso favorece o algoritmo de Hetzel. Com isso, a partir de certa densidade, o *speedup* do SG-Router começa a diminuir na mesma proporção que aumentou até o ponto em que o algoritmo causa estouros de memória. Esses estouros são causados porque, para altas densidades, a distribuição aleatória dos obstáculos faz com que a busca exija rotas com muitas voltas, fazendo com que as tentativas de atalhos diretos falhem, gerando assim buscas recursivas, que caem no mesmo problema. Isso implica em um enorme número de recursões cada uma com seu histórico de nós, ocasionando assim o estouro de memória.

Foi realizada uma tentativa de experimentação que se aproxima mais do roteamento detalhado, afim de se avaliar a aplicabilidade do SG-Router nesse contexto. Devido ao mesmo problema da recursão, não foi possível completar o

roteamento. Com isso, o algoritmo SG-Router ainda é insuficiente para ser aplicado no roteamento detalhado. Contudo, o enorme potencial de *speedup* é motivador e promissor. Além disso, as melhorias sugeridas na sessão 5.3 apontam meios para o aumento do desempenho e para a resolução do problema de memória. Se o problema da recursão for resolvido, o algoritmo SG-Router terá grandes chances de se tornar o estado da arte dos algoritmos de busca genéricos em grade, com aplicação no roteamento detalhado. Apesar de ser esperado que o *speedup* seja pequeno nos casos extremos de congestionamento, a grande maioria dos cenários (caso médio) encontra-se entre os extremos e são nestes cenários que o SG-Router encontra o auge de sua vantagem em desempenho.

REFERÊNCIAS

CHEN, H. M.; ZHOU, H.; YOUNG, F. Y.; WONG, D. F.; YANG, H. H.; SHERWANI, N. Integrated floorplanning and interconnect planning. In: Proc. IEEE/ACM Int. Conf. on computer-Aided Design. 1999. p. 354–357.

CHEN, H. Y.; CHANG, Y. W. Global and Detailed Routing. Disponível em: <http://cc.ee.ntu.edu.tw/~ywchang/Courses/PD/EDA_routing.pdf> Acesso em: 14 fev. 2016.

CHU, C. ; WONG, Y. C. FLUTE: Fast Lookup Table Based Rectilinear Steiner Minimal Tree Algorithm for VLSI Design. **Competência**: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, v. 27, n. 1, p. 70–83, 2008.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Algoritmo de Dijkstra. In: CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Algoritmos: Teoria e Prática**. 2.ed. Rio de Janeiro: Campus. 2002. p. 470–474.

DEUTSCH, D. N. A “dogleg” channel router. In Proceedings of ACM/IEEE Design Automation Conference (DAC). 1976. p. 425–433.

GAREY, M. R.; JOHNSON, D. S. The rectilinear Steiner tree problem is NP-complete. In: SIAM Journal Applied Mathematics. 1977. p. 826–834.

GESTER, M., MÜLLER, D., NIEBERG, T., PANTEN, C., SCHULTE, C., VYGEN, J. Algorithms and Data Structures for Fast and Good VLSI Routing. In: Design Automation Conference (DAC). 2012. p. 459-464.

GESTER, M., MULLER, D., NIEBERG, T., PANTEN, C., SCHULTE, C., VYGEN, J. BonnRoute: Algorithms and data structures for fast and good VLSI routing. **Competência:** ACM Transactions on Design Automation of Electronics Systems v. 18 n. 2, p. 1-24, 2013.

GONÇALVES, S. M. M., DA ROSA JR, L. S., MARQUES, F. S. 2014. A New General Purpose Line Probe Routing Algorithm. In: International Conference on Electronics, Circuits and Systems. 2014. p. 658-661.

HANAN, M. On Steiner's problem with rectilinear distance. In: SIAM Journal on Applied Mathematics. 1966. p. 255–265.

HART, P. E.; NILSSON, N. J.; RAPHAEL, B. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. **Competência:** IEEE Transactions on Systems Science and Cybernetics, v. 4. n. 2, p. 100–107, 1968.

HASHIMOTO, A. STEVENS, J. Wire routing by optimizing channel assignment within large apertures. In Proceedings of ACM/IEEE Design Automation Conference. 1971. p. 155–169.

HENTSCHKE, R. F.; NARASIMHAM, J.; JOHANN, M. O.; REIS, R. L. Maze routing steiner trees with effective critical sink optimization. In: Proceedings of the 2007 international symposium on Physical design (ISPD), ACM New York, NY, USA, 2007. p. 135-142.

HENTSCHKE, R. F.; NARASIMHAM, J.; JOHANN, M. O.; REIS, R. L. Maze Routing Steiner Trees With Delay Versus Wire Length Tradeoff. **Competência:** IEEE Transactions on Very Large Scale Integration (VLSI) Systems, v. 17, n. 8, p. 1073 – 1086, 2009.

HETZEL, A. A sequential detailed router for huge grid graphs. In: Design, Automation and Test in Europe. Paris, 1998. p. 332–338.

HIGHTOWER, D. A solution to line routing problems on the continuous plane. In Proceedings of ACM/IEEE Design Automation Conference (DAC).1969. p. 1–24.

HUANG, T.; YOUNG, E. F. Y. Obstacle-avoiding rectilinear Steiner minimum tree construction: an optimal approach. In: Proceedings of the International Conference on Computer-Aided Design (ICCAD) , IEEE Press Piscataway, NJ, USA, 2010. p. 610-613.

HUMPOLA, J. **Schneller Algorithmus für kürzeste Wege in irregulären Gittergraphen**. 2009. Teese (Doutorado), Universidade de Bonn, Bonn, 2009.

ISPD GLOBAL ROUTING CONTEST 2008. Disponível em: <http://archive.sigda.org/ispd2008/contests/ispd08rc.html#head-benc>. Acesso: 14 fev. 2016.

JOHANN, M.; CALDWELL, A.; KAHNG, A.; REIS, R. Net by Net Routing with a New Path Search Algorithm. In: Symposium on Integrated Circuits and Systems Design. Manaus, 2000. p. 144-149.

KAHNG, A. B., LIENIG, J., MARKOV, I. L., HU, J. VLSI Physical Design: From Graph Partitioning to Timing Closure, Chapter 6: Detailed Routing. Disponível em: http://vlsicad.eecs.umich.edu/KLMH/downloads/book/chapter6/chap6-130304.pdf. Acesso: 14 fev. 2016.

KASTNER, R.; BOZORGZADEH, E.; SARRAFZADEH, M. Pattern routing: use and theory for increasing predictability and avoiding coupling. In: IEEE Trans. on Computer-Aided Design. 2002. p. 777–790.

KRUSKAL, J. B. On the shortest spanning subtree of a graph and the traveling salesman problem. In: Proc. the American Mathematical Society. 1956. p. 48–50.

LEE, C. Y. An Algorithm for Path Connections and Its Applications. **Competência:** IRE Transactions on Electronic Computers, v. EC-10, n. 3, p. 346–365, 1961.

LI, L.; YOUNG, E. F. Y. Obstacle-avoiding rectilinear Steiner tree construction. In: Proceeding International Conference on Computer-Aided Design (ICCAD) , IEEE Press Piscataway, NJ, USA, 2008. p. 523-528.

LIN, C. W. ; CHEN, S. Y.; LI, C. F.; CHANG, Y. W.; YANG, C. L. Efficient obstacle-avoiding rectilinear steiner tree construction. In: Proceeding of International Symposium on Physical Design (ISPD) . ACM New York, NY, USA, 2007. p. 127-134.

MIKAMI, K.; TABUCHI, K. A computer program for optimal routing of printed circuit connectors. In: Proceedings of International Federation for Information Processing. 1968. p. 1475–1478

PAN, M.; CHU, C. FastRoute: a step to integrate global routing into placement. In: Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design **(**ICCAD), ACM New York, NY, USA , 2006. p 464-471.

PAN, M.; CHU, C. FastRoute 2.0: A High-quality and Efficient Global Router. In: Proceedings of the 2007 Asia and South Pacific Design Automation Conference (ASP-DAC), IEEE Computer Society Washington, DC, USA, 2007. p. 250-255.

PEYER, S., RAUTENBACH, D., VYGEN, J. A generalization of Dijkstra's shortest path algorithm with applications to VLSI routing. **Competência:** Journal of Discrete Algorithms. v. 7, n. 4, p. 377-390, 2009.

RUBIN, F. The Lee Path Connection Algorithm. **Competência:** IEEE Transaction on Computers.v. C-23, n. 9, p. 907-914, 1974.

SHERWANI, N. Steiner Tree Algorithms. In: SHERWANI, N. Algorithms for VLSI Design Automation. 3.ed. New York, Bostom. 1998. p. 111–115.

SCHULTE, C. **Design Rules in VLSI Routing.** 2012. Tese (Doutorado em Computação). Faculdade de Ciências Matemática-Naturais, Universidade de Bonn, Bonn, 2012.

VYGEN, J. Global routing. 2009. Disponível em <http://www.or.uni-bonn.de/~vygen/files/hz3h.pdf>. Acesso em 16 fev. 2016.

WEISSTEIN, Eric W. Grid Graph. MathWorld - A Wolfram Web Resource. Disponível em: http://mathworld.wolfram.com/GridGraph.html. Acesso em: 14 fev. 2016.

WU**,** P. C.; GAO, J. R.; WANG, T. C. A Fast and Stable Algorithm for Obstacle-Avoiding Rectilinear Steiner Minimal Tree Construction. In: Proceeding Asia and South Pacific Design Automation Conference (ASP-DAC), IEEE Computer Society Washington, DC, USA, 2007. p. 262-267.

XU, Y.; ZHANG, Y.; CHU, C. FastRoute 4.0: global router with efficient via minimization. In: Proceedings of the 2009 Asia and South Pacific Design Automation Conference (ASP-DAC), IEEE Press Piscataway, NJ, USA, 2009. p. 576-581.

ZHANG, Y., CHU, C. RegularRoute: An Efficient Detailed Router Applying Regular Routing Patterns. **Competência**: IEEE Transactions on Very Large Scale Integration (VLSI) Systems, v. 21, n. 9, p. 1655-1668, 2013.

ZHANG, Y.; XU, Y.; CHU, C. FastRoute3.0: a fast and high quality global router based on virtual capacity. In: Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), IEEE Press Piscataway, NJ, USA, 2008. p. 344-349.