Capítulo 3

Algoritmos de Travessia

Apesar dos algoritmos de travessia para posicionamento e roteamento (P&R) não serem novidades na literatura (Ferreira et al., 2007), essa dissertação introduz contribuições na área. Primeiro, o algoritmo YOTO apresenta uma nova abordagem para manipulação de grafos com múltiplas saídas. Cada instância de execução do algoritmo pode começar por nodo distinto, resultando em uma solução diferente. Para reduzir o tempo de execução no processo de exploração do espaço de soluções, uma versão utilizando GPUs foi implementada. Além disso, uma busca sistemática no espaço de solução foi criada com o objetivo de melhorar a exploração. Foram utilizados algoritmos de busca profundidade e largura. O YOTO também introduziu um novo percurso de travessia, o ZigZag. Diferente dos algoritmos tradicionais, o ZigZag percorre as arestas nas duas direções com o objetivo de capturar as correlações das múltiplas saídas.

Contudo, o YOTO é um algoritmo que se baseia em decisões aleatórias e gulosas, não resolvendo o problema da reconvergência dos grafos. Deste modo, um novo algoritmo foi desenvolvido, denominado pelo nome de YOTT, com o propósito de explorar as propriedades dos grafos. O YOTT cria anotações na lista de travessia para tratar os caminhos com reconvergência e evitar decisões aleatórias durante o posicionamento. Portanto, o YOTT insere uma metodologia sistemática para transferir a localidade do grafo de entrada para saída considerando as reconvergências e o posicionamento das entradas e saídas nas bordas. O resultado gerado melhorou a qualidade quando comparado com a abordagem YOTO.

3.1 Trabalhos Relacionados

Esta seção apresenta trabalhos relacionados ao problema de P&R com foco em abordagens que exploram teoria de grafos e mapeamento para CGRA.

Em (Ferreira et al., 2005) é apresentado um algoritmo de mapeamento para CGRA assíncrono que percorre grafo por níveis, seguindo a proposta apresentada em (Koren et al., 1988) para posicionamento de grafos em arquitetura de malha. Além de não considerar o escalonamento, pois os operadores são assíncronos sacrificando a vazão

para garantir o correto funcionamento, esta proposta apresenta uma baixa taxa de ocupação da arquitetura alvo e congestionamento de interconexões entre os níveis. A Figura 3.1 ilustra um exemplo de mapeamento por níveis com baixa ocupação e roteamento longos considerando um grafo pequeno e uma arquitetura hexagonal. A Figura 3.1(a) apresenta o grafo de entrada considerando dois operadores dentro de cada nó. A Figura 3.1(b) mostra que os 18 nodos foram mapeados em 29 células hexadecimais usando o algoritmo de (Koren et al., 1988). A ocupação da arquitetura foi de 23% com ALUs, 61% com roteamento e 16% vazias. Sendo 32 células roteadas somente por fios.

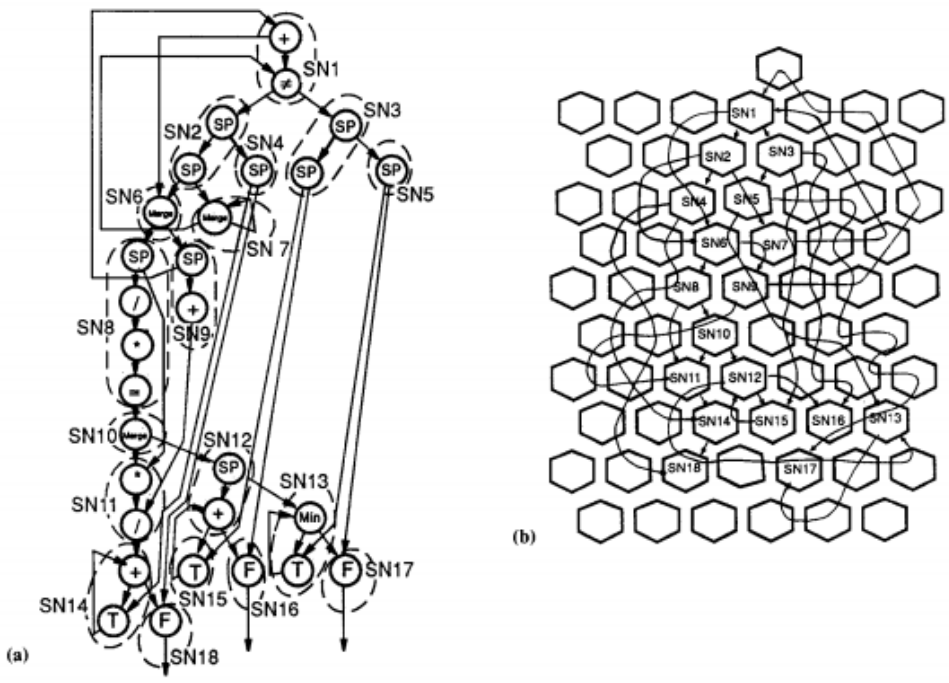

Figura 3.1: Exemplo de mapeamento de grafo por níveis em uma arquitetura hexadecimal. Retirada de (Koren et al., 1988).

Para melhorar a ocupação, (Ferreira et al., 2005) propuseram uma abordagem gulosa para fusão de níveis consecutivos como ilustrado na Figura 3.2. A Figura 3.2(a) mostra um grafo com 13 nodos e 18 arestas que resulta em um primeiro posicionamento por níveis ilustrado na 3.2(b). A Figura 3.2(c) apresenta a fusão da primeira linha com a segunda linha, gerando uma compactação e um reposicionamento do nodo A entre os nodos B e C para diminuir a ocupação no grid. No último nível, uma mudança do nodo M para a posição mais próximo do nodo H também foi realizada. A Figura 3.2(c) ilustra o roteamento final, onde o roteamento que sai dos nodos B, C, D, F, G faz o uso da técnica *multicast*.

Uma abordagem semelhante foi apresentado em (Robič and Šilc, 1994), porém explorando SA para fazer a compactação dos níveis como ilustrado na Figura 3.3. Comparando um grid inicialmente já posicionado e roteado, como ilustrado na Figura

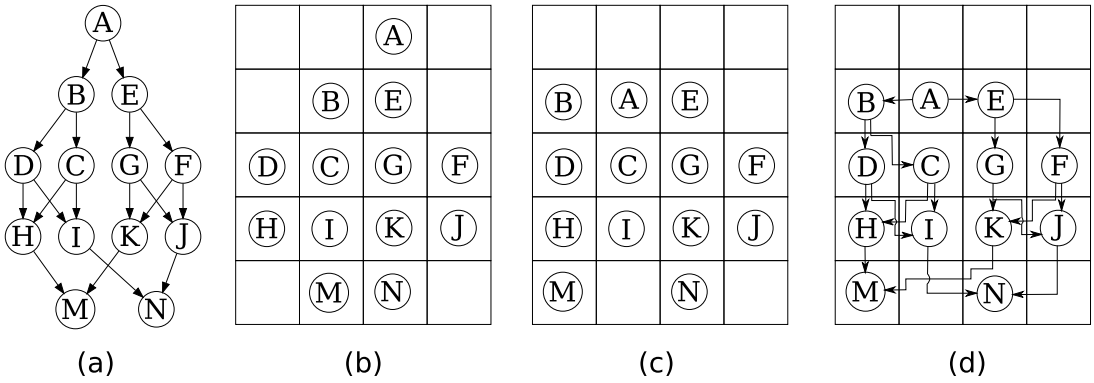

Figura 3.2: Mapeamento utilizando abordagem gulosa. (a) Grafo de entrada; (b) Mapeamento em um grafo por nível e profundidade; (c) Ajustes no mapeamento; (d) Roteamento do grafo. Adaptado de (Silva et al., 2006).

3.3(a), com a aplicação do SA para a diminuição do espaço no grid, pode-se observar uma diminuição de área ocupada passando de um grid de 7x18 para um grid 5x12, ou seja, uma redução de 47% em área, porém a ocupação ainda é baixa.

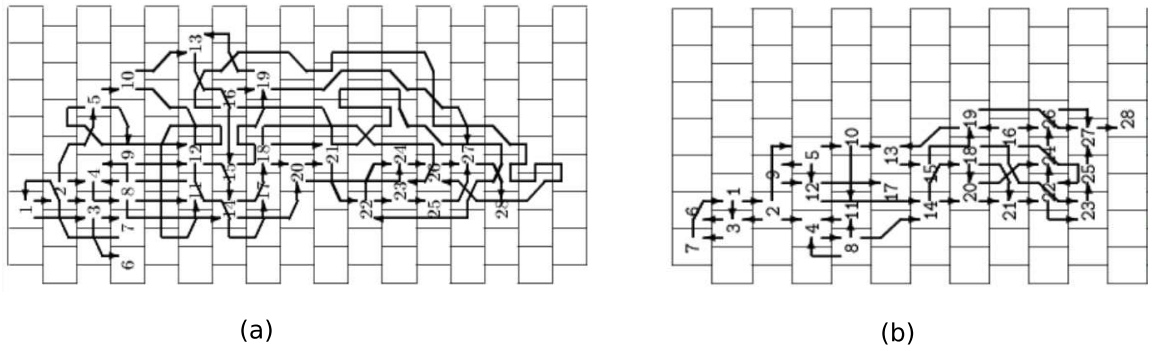

Figura 3.3: Compactação de níveis utilizando *Simulated Annealing*. (a) Grafo antes da compactação; (b) Grafo após a compactação. Adaptado de (Silva et al., 2006).

Considerando o P&R com balanceamento, mas para mapeamento de circuitos em QCA, um algoritmo guloso baseado em níveis foi apresentado em (Trindade et al., 2016). Devido as restrições de recursos de roteamento dos QCA em comparação com os CGRA e a solução gulosa apresentada, a abordagem foi avaliada em grafos com menos de 10 nodos (Trindade et al., 2016) com escalabilidade limitada. Uma solução exata para mapeamento em QCA é proposta em (Fontes Junior et al., 2018), que explora a partição do grafo em subgrafos para depois aplicar o mapeamento por níveis. Grafos com até 30 nodos foram mapeados porém o tempo de execução é da ordem de minutos a horas. As restrições de roteamento nas arquiteturas de QCA reduzem significativamente o número de soluções válidas e a taxa de ocupação das arquiteturas.

Uma abordagem utilizando algoritmo genético para o mapeamento em CGRA é apresentada em (Da Silva et al., 2006). A solução maximiza a ocupação porém

considerada arquiteturas com muitos recursos de roteamento. Para gerar a população inicial, algoritmos de travessia em grafos foram utilizados juntamente com percursos no grafos da arquitetura como ilustrado na Figura 3.4. A Figura 3.4(a) apresenta o grafo de entrada onde foi realizado um percurso em profundidade A, C, E, G, J e I, depois retorna em A para seguir pelo caminho B, D, M, volta em D e percorre F e H. Na arquitetura diversos percursos foram propostos. As Figuras 3.4(b) e (c) ilustram dois exemplos. O primeiro faz um zigzag em linha na arquitetura e segue sequencialmente a lista de profundidade do grafo de entrada. O segundo faz um zigzag em diagonal. Muitas outras variações são feitas gerando a população inicial de P&R para o algoritmo genético otimizar.

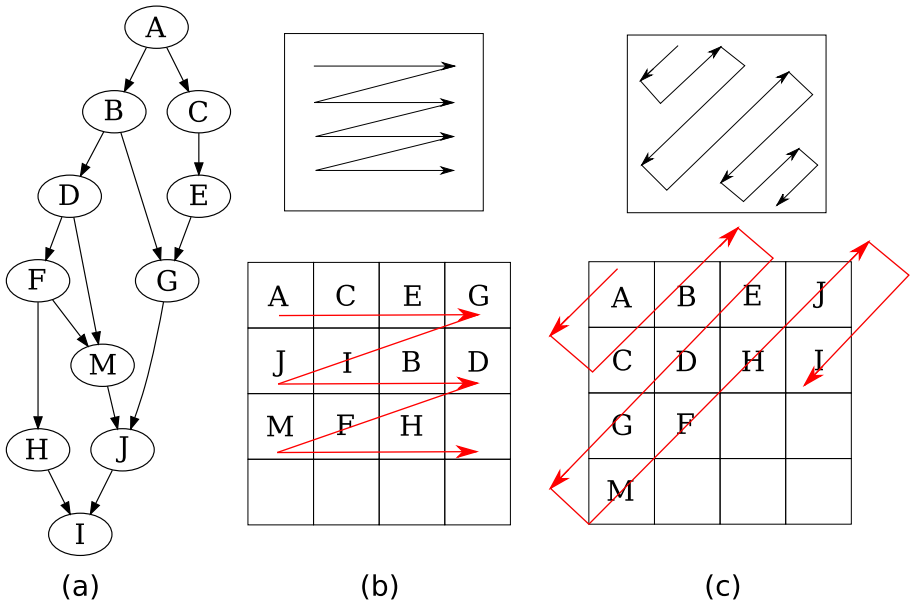

Figura 3.4: Percusso dos grafos. (a) Grafo de fluxo de dados; (b) Busca em profundidade; (c) Busca em amplitude. Adaptado de (Silva et al., 2006).

Um algoritmo de P&R em tempo polinomial $O(|V|^2)$ é proposto em (Lai et al., 2005), onde V é o conjunto de vértices do grafo da aplicação. A proposta é baseada na construção de uma árvore geradora mínima para posteriormente percorrer o grafo e realizar o posicionamento. Entretanto as soluções geradas apresentam baixa taxa de ocupação. Uma solução baseada em algoritmos para desenhos de grafos com a abordagem *split&push* é apresentado em (Yoon et al., 2009a). Por onde a cada passo o grafo vai sendo dividido em partições menores e uma solução usando programação inteira é aplicada, assim como mostrado em trabalhos recentes (Walker and Anderson, 2019b), tem problemas de escalabilidade e tempo de execução da ordem de segundos a minutos para grafos com 20 à 30 nodos.

Duas versões são propostas em (Ferreira et al., 2007) que resolvem o problema P&R em tempo polinomial utilizando a estratégia da travessia em profundidade. A versão do algoritmo mais simples tem uma complexidade $O(|E|)$, onde $|E|$ é a quantidade

de arestas. O algoritmo percorre utilizando busca em profundidade de forma gulosa. A versão do algoritmo mais complexo tem uma complexidade $O(n + |E|^3)$. Nesse algoritmo utiliza-se Dijkstra para guiar o percusso do grafo. A solução desse problema buscou reduzir a ocupação na arquitetura, fazendo com que utiliza-se o mínimo possível de recurso. Entretanto a arquitetura alvo considera um CGRA assíncrono. Caso algum caminho fique desbalanceado, a vazão será reduzida, pois alguns PEs irão aguardar mais tempo até todos os dados chegarem para disparar a computação local.

Uma solução para reduzir o tempo de execução do P&R é adicionar mais recursos de roteamento como, por exemplo, a inclusão de uma rede multiestágio global de interconexão (Ferreira et al., 2009, 2011b). Uma rede multiestágio tem o custo $O(n \log n)$ e algoritmos polinomiais de roteamento. A proposta apresentada em (Ferreira et al., 2009, 2011b) é uma solução híbrida com uma malha e uma rede multiestágio. Cada PE tem o recurso de rotear uma conexão através da rede.

Em (Oliveira et al., 2020; Carvalho et al., 2020) apresentam o uso da heurística *Simulated Annealing* utilizando uma abordagem de troca de posições locais. Em (Oliveira et al., 2020), além de uma de arquitetura homogênea, três arquiteturas heterogêneas (xadrez, coluna e borda) são avaliadas. 1000 soluções são geradas e avaliadas. Os resultados mostram que o tempo de execução 52% mais rápido que a ferramenta VPR 8.1 (Murray et al., 2020) com uma redução de custo sem perda de otimização de 76%. Comparando com o algoritmo YOTT, proposto nesta dissertação, considerando a exploração de 1000 soluções, o trabalho de Oliveira *et al* (Oliveira et al., 2020) reduz em 7% o comprimento de fios e em 15% o tamanho das filas. Entretanto, o tempo de execução é 1000x maior.

Considerando FPGA como arquitetura alvo, (Ferreira et al., 2013b, 2015) aplicam os algoritmos de travessias de grafos propostos em (Ferreira et al., 2007) priorizando o caminho crítico. Os resultados mostram uma redução de três ordens de grandeza no tempo de execução com uma perda de 30% no caminho crítico em comparação com a ferramenta VPR (Luu et al., 2011). Grafos com até 3000 nodos foram avaliados. Entretanto, o mapeamento em FPGA não considera balanceamento dos caminhos.

Outros trabalhos procuram utilizar inteligência artificial para resolver problemas de P&R. Em (Liu et al., 2018) é proposto o algoritmo RLMap para resolver o problema de mapeamentos de grafos em CGRAs como um agente na aprendizagem de reforço, no qual unifica o posicionamento, o roteamento e a inserção de PE por ações do agente. A Figura 3.5(a) apresenta o posicionamento inicial do *benchmark fdtd-apml* de tamanho de 21 nodos e com 22 arestas, mapeados em um *grid* 6×6 com três rotas inválidas $h - k, d - c$ e $u - v$ (arestas em vermelho no posicionamento do grafo). Utilizando a rede de aprendizado *Q-network*, chega-se ao resultado apresentado na Figura 3.5(b). Onde é gerada uma solução ocupando um espaço maior no *grid*. A Figura 3.5(c) apresenta um treinamento da rede após 10 mil passos, chegando a um

resultado ótimo sem problema de conflito de rotas e reduzindo a ocupação. Seus resultados mostraram um desempenho comparável na qualidade com as heurísticas DFGNet (Yin et al., 2017), Pattern (Mehta et al., 2013) e SPKM (Yoon et al., 2008). Além disso, se adapta a diferentes arquiteturas com uma conversão rápida. Contudo, o tempo de execução é elevado. Por exemplo, para o grafo da Figura 3.5 com apenas 21 nodos foram necessários 16 minutos para encontrar a solução.

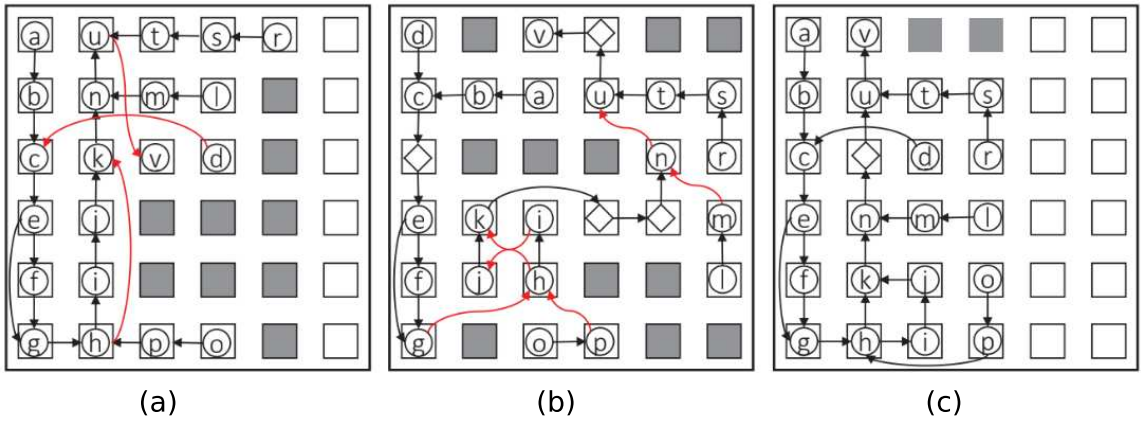

Figura 3.5: Estudo de caso do *Benchmark fdtd-apml*. (a) Posicionamento inicial; (b) Resultado do mapeamento após o primeiro passo; (c) Resultado final do mapeamento. Retirado de (Liu et al., 2018).

Em (Zhao et al., 2020) utiliza o modulo scheduling no mapeamento de grafos de CGRAs. Eles decompõem o problema do mapeamento temporal e espacial em duas etapas. O algoritmo de mapeamento está integrado ao ambiente de compilação LLVM (Lattner and Adve, 2004). Seus resultados mostraram melhora no desempenho de 5.4% a 14.2% dentre os *benchmarks* compilados com sucessos em relação ao algoritmo de P&R RAMP (Dave et al., 2018). Além disso, eles chegaram a utilizar benchmarks com até 154 nodos, mas os seus resultados chegaram a demorar 14 segundos para mapear, inviabilizando em aplicações de tempo real.

Em (Kojima et al., 2020) um algoritmo genético para o mapeamento dinâmico com modulo scheduling em CGRA (GenMap). Os 7 benchmarks utilizados foram de 12 à 45 nodos de tamanho e baseados em aplicações de processamento de sinal e criptografia. Seus resultados mostraram uma redução de até 15.7% em comprimento de fio sem alteração na qualidade da solução comparado com as abordagens SPKM (Yoon et al., 2008), RLMap (Liu et al., 2018), e CGRA-ME (Chin et al., 2017a). GenMap apresenta uma aceleração de 2x, 1.55x e 1.5x em relação as abordagens CGRA-ME, SPKM e RLMap, respectivamente.

3.2 Algoritmo baseado em travessia de grafos

3.2.1 Algoritmos baseados em profundidade

Ferreira et al. (2007) propôs duas versões de algoritmos utilizando a travessia em profundidade. Ambas versões obtém uma solução em tempo polinomial. A versão mais simples percorre o grafo em profundidade de forma gulosa enquanto que a versão mais complexa utiliza Dijkstra para guiar no percurso do grafo. Entretanto a arquitetura alvo considera um CGRA assíncrono. Caso algum caminho fique desbalanceado, a vazão será reduzida, pois alguns PEs irão aguardar mais tempo até todos os dados chegarem para disparar a computação local. A proposta do algoritmo é realizar uma travessia no grafo de fluxo de dados e no grafo que representa a arquitetura alvo. Existem muitas possibilidades para a travessia do grafo de fluxo de dados e do grafo da arquitetura. Ferreira et al. (2007) avalia 1, 10 e 50 travessias aleatórias. Esta dissertação apresenta o algoritmo Traversal que também explora várias travessias de forma eficiente durante o mapeamento. A solução proposta parte de um nó inicial, percorre o grafo utilizando alguma estratégia de travessia (largura, profundidade, ...). Durante o percurso, o algoritmo tem que tomar decisões. Por exemplo, a Figura 5.3(a) apresenta um exemplo de travessia. Primeiro, poderíamos começar pelo nodo A ou pelo nodo E. No exemplo começamos por A. Em A existem duas opções a seguir. Tomamos a decisão de seguir por B. Depois em B temos novamente duas opções, onde seguimos por D. Estas decisões foram realizadas no escopo do grafo de entrada.

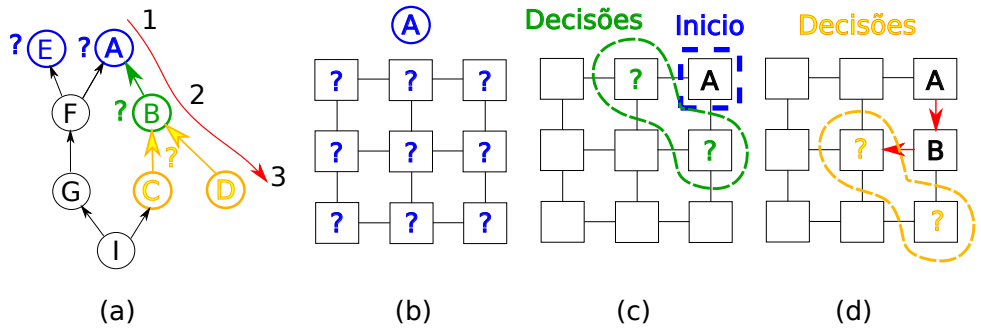

Figura 3.6: (a) Grafo de fluxo de dados; Tomadas de decisão: (b) Nodo inicial; (c) Posicionamento de um novo nodo; (d) Escolha de um novo nodo.

O mapeamento além de percorrer o grafo de entrada, percorre o grafo da arquitetura simultaneamente. Portanto, mais decisões devem ser tomadas ao percorrer o grafo da arquitetura. A Figura 5.3(b) ilustra 9 possibilidades para posicionar o nodo A que é o primeiro nodo percorrido do grafo de entrada. Uma vez posicionado A, a Figura 5.3(c) mostra a próxima decisão a ser tomada. Neste caso, o algoritmo explora a localidade do grafo de entrada e saída. Como B é vizinho de A no grafo de entrada, o algoritmo irá posicionar B em um nodo vizinho de A no grafo da arquitetura. Im-

portante destacar que esta operação é a base para o sucesso do algoritmo de travessia. O nodo A já se encontra posicionado e estamos visitando B através da aresta $A \leftarrow B$. Portanto, A irá determinar a posição de B buscando uma aresta no grafo da arquitetura onde A e B sejam vizinhos, transferindo assim a localidade de um grafo para o outro. Neste caso temos duas opções. Suponha que B seja posicionado abaixo de A como ilustrado na Figura 5.3(d). Agora o próximo passo irá buscar uma posição para o nodo D como vizinho de B com duas opções. A qualidade da solução dependente diretamente das decisões tomadas durante a travessia do grafo de entrada e do grafo da arquitetura.

3.2.2 Algoritmo YOTO

No artigo apresentado no Capítulo 2 é abordado o algoritmo para percurso Zigzag (Seção 4.3.3). Diferente dos algoritmos tradicionais, Zigzag percorre tanto na direção saída para entrada (S->E) tanto no sentido entrada para saída (E->S) buscando capturar as correlações em grafos com múltiplas saídas. A figura 3.7 compara uma travessia em profundidade com uma travessia com Zigzag. A Figura 3.7(a) apresenta o primeiro caminho, partindo da saída A, em profundidade na seguinte ordem: A, C, G, E, H, onde todos os descendentes de A foram percorridos. Porém em um grafo com múltiplas saídas, temos que recomeçar a travessia no nodo B para percorrer todo o grafo. Neste exemplo, o caminho percorrido foi B, D, F. Note que o algoritmo não observa correlações entre as saídas A e B do grafo. Neste exemplo, as arestas $D \leftarrow E$ e $F \leftarrow H$ podem requerer vários segmentos, pois o caminho que parte de B não tem nenhuma informação do posicionamento de E e H que foram realizados durante o caminho que começou na saída A. A Figura 3.7(b) mostra uma solução de P&R para uma travessia no grafo da arquitetura, seguindo a ordem em profundidade destacada com a linha pontilhada no grafo de entrada apresentado na Figura 3.7(a). Podemos observar que as arestas que foram visitadas tiveram sua localidade transferidas para o grafo de saída. Porém as arestas $D \leftarrow E$ e $F \leftarrow H$ tiveram um custo maior pois não foram visitadas, já que os nodos E e H já estavam posicionados na arquitetura quando D e F foram visitados. Além das múltiplas saídas, temos as reconvergências internas. Neste exemplo, a aresta $E \leftarrow G$ também não foi considerada, pois G já tinha sido posicionado quando E foi visitado. Resumindo, todas as arestas que não foram consideradas podem gerar um custo maior em segmentos pois a sua informação de localidade não foi avaliada no mapeamento. A Figura 5.3 mostra com uma linha cinza pontilhada o percurso de travessia no grafo de entrada e da arquitetura.

O percurso Zigzag é apresentado na Figura 3.7(c). A mudança de sentido ocorre quando o número de arestas de um nó é maior ou igual a 2 para as entradas (fanin) ou

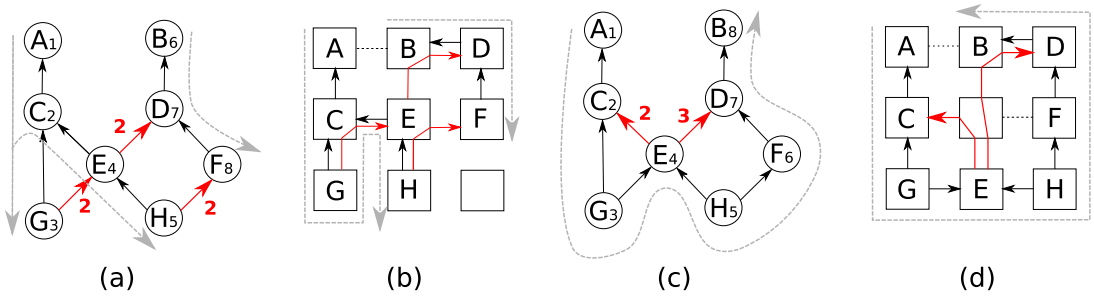

Figura 3.7: Comparação do Percurso em Profundidade com o Percurso ZigZag: (a) Grafo de Entrada com Travessia em Profundidade; (b) Grafo da Arquitetura em Profundidade; (c) Travessia em ZigZag do Grafo de Entrada; (d) Grafo da Arquitetura percorrido em ZigZag.

para as saídas (fanout). Desta maneira, quando iniciamos pelo caminho A, C, G não ocorre inversão de sentido no percurso, pois o nodo C possui um fanout 1 e a direção do percurso é saída para entrada. A direção é alterada quando o fanout é maior do que 1, o que acontece quando o nodo G é visitado. O sentido é invertido, agora indo das entradas para as saídas. O sentido será novamente alterado quando chegar no nodo E que possui fanin igual a 2. Esse processo continua até todos os nodos serem visitados. O percurso final para o exemplo foi: $A_1, C_2, G_3, E_4, H_5, F_6, D_7, B_8$. A Figura 3.7(d) apresenta o grafo da arquitetura que também foi percorrido nesta sequência. O resultado final depende também das decisões que foram tomadas durante a travessia da arquitetura. Em comparação com a travessia em profundidade para este exemplo, a travessia Zigzag visitou uma aresta a mais e explorou sua localidade. Porém duas arestas não foram exploradas: $C \leftarrow E$ e $D \leftarrow E$. Apesar da travessia Zigzag resolver as correlações das múltiplas saídas, o problema de reconvergências ainda continua presente. As informações de localidade destas arestas não são transferidas para o percurso do grafo da arquitetura, resultando em arestas com vários segmentos. Na próxima seção apresentamos o algoritmo YOTT que realiza duas travessias no grafo de entrada para capturar e transferir esta localidade.