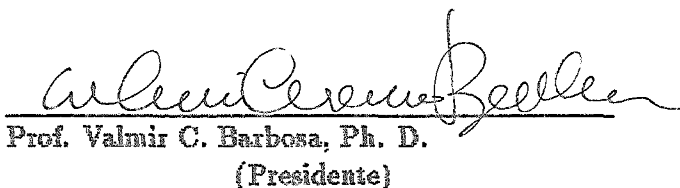
Métodos de Busca Heurística Paralela

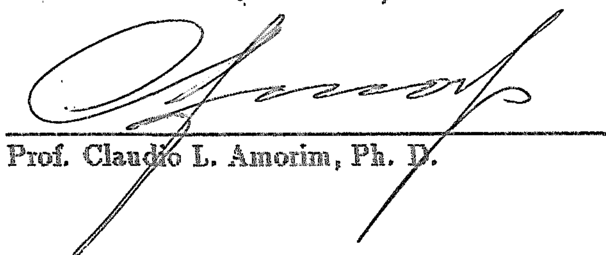
Lélio de Paiva Sá Freitas

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:



Prof. Valmir C. Barbosa, Ph. D.
(Presidente)



Prof. Claudio L. Amorim, Ph. D.

Prof. Antonio de Almeida Pinho, D. Sc.

RIO DE JANEIRO, RJ - BRASIL
SETEMBRO DE 1990

FREITAS, LÉLIO DE PAIVA SÁ

Métodos de Busca Heurística Paralela [Rio de Janeiro] 1990

ix, 54 p., 29.7 cm, (COPPE/UFRJ, M. Sc., ENGENHARIA DE SISTEMAS

E COMPUTACÃO, 1990)

TESE - Universidade Federal do Rio de Janeiro, COPPE

1 - Métodos de busca 2 - Paralelo

I. COPPE/UFRJ II. Título(Série).

Aos meus pais

Agradecimentos

A todas as pessoas que, diretamente ou indiretamente, participaram da minha jornada, que tem neste trabalho um dos seus frutos.

Aos meus pais, Élcio e Lélia, pelo seu total apoio e presença em todos os momentos.

Agradeço também ao meu orientador Valmir C. Barbosa, pela sua dedicação ao trabalho de pesquisador e educador.

Aos amigos Luciano, Marília, Victor, Sérgio e Maurício Maia, pelo incentivo para realizar este importante passo na minha vida profissional.

A todos os amigos da COPPE, pelo grande companheirismo.

Ao pessoal do laboratório e todos os funcionários da COPPE pelo apoio logístico e técnico.

Resumo da Tese apresentada à COPPE como parte dos requisitos necessários para a obtencão do grau de Mestre em Ciências (M. Sc.)

Métodos de Busca Heurística Paralela

Lélio de Paiva Sá Freitas

Setembro de 1990

Orientador: Valmir Carneiro Barbosa

Programa: Engenharia de Sistemas e Computação

Este trabalho consiste no desenvolvimento de versões paralelas de algoritmos de busca do tipo "best-first". São apresentados métodos paralelos baseados nos algoritmos seqüenciais A^* e BS^*. As principais diferenças entre os vários procedimentos desenvolvidos estão na forma de busca, unidirecional e bidirecional, e na distribuição do conjunto de dados, listas centralizadas e distribuídas. Os algoritmos paralelos foram aplicados sobre um problema de planejamento de rotas em uma rede de nós geograficamente dispersos. Foi observado que, quando usadas listas centralizadas, os métodos bidirecionais de busca paralela, baseados no BS^*, apresentaram um tempo de execução menor do que os unidirecionais, baseados no A^*. Mas esta situação se inverte quando utilizamos listas distribuídas, sendo que o algoritmo unidirecional apresenta o melhor desempenho entre todas as versões implementadas. Os testes foram desenvolvidos em uma rede de *transputers*, utilizando a linguagem *Occam* de programação paralela.

Abstract of Thesis presented to COPPE as partial fulfillment of the requirements for the degree of Master of Science (M. Sc.)

Parallel Heuristic Search Methods

Lélio de Paiva Sá Freitas

September, 1990

Thesis Supervisor: Valmir C. Barbosa

Department: Programa de Engenharia de Sistemas e Computação

This work describes the design and implementation of parallel versions of best-first search algorithms. The parallel methods were based on the A^* and the BS^* sequential algorithms. The main differences among the procedures investigated are the search strategies, unidirectional or bidirectional, and the data set distribution, using centralized or distributed lists. The parallel methods were applied to a route planner problem for a network of geographically dispersed nodes. When centralized lists were used, it was observed that the bidirectional search methods, based on BS^*, had a lower execution time than the unidirectional ones, based on A^*. In contrast, when distributed lists were used, the unidirectional methods performed better. Overall, the unidirectional algorithm with distributed list yielded the best performance among all the implemented versions. The tests have been performed on a transputer network, utilizing the Occam language for parallel programming.

Índice

I Introdução 1

II Métodos de Busca Seqüencial 4

II.1 Notações 4

II.2 Estratégias de Busca Não Informada 5

II.2.1 Depth-First : Uma Estratégia LIFO 5

II.2.2 Breadth-First : Uma Estratégia FIFO 6

II.3 Estratégias Direcionadas 6

II.3.1 Estratégias Best-First 7

II.3.2 Funções Peso Recursivas 8

II.3.3 Algoritmos Best-First Especializados 11

II.3.4 Propriedades dos Algoritmos de Busca 13

II.4 Algoritmos de Busca Bidirecional 18

II.4.1 Algoritmo BS^* 19

II.4.2 Propriedades do BS^* 22

III Métodos de Busca Paralela 25

III.1 Versões Paralelas do Algoritmo A^* 25

III.1.1 Lista Centralizada 25

III.1.2 Lista Distribuída 29

III.2 Versões Paralelas do Algoritmo BS^* 33

III.2.1 Lista Centralizada 33

III.2.2 Lista Distribuída 38

IV Implementação e Resultados 44

IV.1 OCCAM 44

IV.2 O Transputer 45

IV.3 Aplicação Implementada 46

IV.4 Distribuição de Processos 47

IV.5 Resultados 48

IV.6 Conclusões 53

Lista de Figuras

III.1 Processos do Algoritmo A^* Paralelo de Lista Centralizada 26

III.2 Processos do Algoritmo A^* Paralelo de Lista Distribuída 30

III.3 Processos do Algoritmo BS^* Paralelo de Lista Centralizada 34

III.4 Processos do Algoritmo BS^* Paralelo de Lista Distribuída 39

IV.1 Configuração da rede de processadores do tipo Transputer 45

IV.2 Acelerações do Algoritmo BS^* seqüencial 49

IV.3 Acelerações dos Algoritmos Paralelos, utilizando 2 processadores, para grafos com probabilidade de conexão igual a 0.1 49

IV.4 Acelerações dos Algoritmos Paralelos, utilizando 2 processadores, para grafos com probabilidade de conexão igual a 0.3 50

IV.5 Acelerações dos Algoritmos Paralelos, utilizando 2 processadores, para grafos com probabilidade de conexão igual a 0.5 50

IV.6 Acelerações dos Algoritmos Paralelos, utilizando 4 processadores, para grafos com probabilidade de conexão igual a 0.1 51

IV.7 Acelerações dos Algoritmos Paralelos, utilizando 4 processadores, para grafos com probabilidade de conexão igual a 0.3 52

IV.8 Acelerações dos Algoritmos Paralelos, utilizando 4 processadores, para grafos com probabilidade de conexão igual a 0.5 52

Capítulo I

Introdução

Métodos de busca são utilizados para resolver uma grande classe de problemas que, ou não possuem uma solução direta ou a solução é muito ineficiente.

Geralmente, tais problemas podem ser resolvidos em vários passos, até encontramos as suas soluções, que chamaremos de soluções completas. Para cada passo podem existir uma ou mais soluções alternativas, que chamaremos de soluções parciais do problema. A tarefa de encontar a solução completa do problema inicial a partir de uma de suas soluções parciais constitui um subproblema. Um subproblema também pode ser resolvido em passos e apresentar soluções parciais, que também serão soluções parciais do problema inicial. O conjunto de todas as soluções parciais possíveis do problema com as soluções completas do mesmo, pois um problema pode apresentar mais de uma solução, formam o espaço do problema.

Para resolver essa classe de problemas, muito freqüentes em algumas áreas do conhecimento, como a pesquisa operacional, procede-se a uma busca através de todo o espaço do problema.

Esse espaço pode ser representado por um grafo orientado, onde cada nó filho é uma solução parcial do nó pai e os arcos representam as operações necessárias para chegar àquela solução.

O grafo onde se realiza a busca geralmente possui grandes dimensões, tornado uma busca totalmente aleatória um processo muito ineficiente. Para guiar esse processo, utilizam-se procedimentos heurísticos. Heurísticas são estratégias que utilizam informações facilmente acessíveis e livremente aplicáveis para o controle de processos de solução de

problemas. Também devido às dimensões do grafo, torna-se inviável todo o seu armazenamento na memória de um computador, sendo necessário estabelecer um meio de gerar subproblemas a partir de um subproblema original, ou seja, gerar nós filhos a partir do nó pai. Essa operação é chamada de expansão de um nó.Um algoritmo de busca eficiente deve expandir o menor número de nós possível.

A técnica de busca mais geral é conhecida como "Branch & Bound" e consta de dois processos principais :

1. "Branching" : gera os subproblemas;
2. "Bounding " : reduz o número de subproblemas a serem considerados no processo de Branching, identificando aqueles nós que não conduzirão a uma solução completa do problema original.

O objetivo deste trabalho é adaptar esses métodos de busca para sistemas do tipo MIMD de processamento paralelo [2] [3] [5], a fim de acelerar o tempo de execução, mantendo a qualidade da solução.

Os principais problemas encontrados para uma implementação paralela dos métodos de busca acima descritos são :

1. identificar o grau de paralelismo existente no algoritmo;
2. distribuir o conjunto de dados, que é irregular e gerado dinamicamente;
3. manter o conhecimento adquirido ao longo da execução do algoritmo ,que é utilizado para melhorar a sua eficiência, em uma memória distribuída [4].

Este trabalho está organizado na seguinte forma : o segundo capítulo apresenta um resumo sobre procedimentos de busca em grafos do tipo OR e funções de avaliação heurística, o capítulo seguinte discute as várias possibilidades de paralelismo existente nestes procedimentos e o quarto capítulo descreve a implementação dos algoritmos paralelos e apresenta os resultados obtidos.

Na apresentação dos métodos de busca, é dado maior enfoque aos algoritmos do tipo "Best-First", por apresentarem um bom desempenho na maior parte dos casos.

Dentre os algoritmos "Best-First", destacamos o A^*, detalhando os passos do algoritmo e suas propriedades. Este algoritmo é considerado o melhor método seqüencial de busca e as conclusões sobre a aceleração do tempo de execução serão feitas tomando-o como referencial. Os procedimentos bidirecionais de busca, onde duas árvores de busca são expandidas, também são abordados neste trabalho. Intuitivamente os métodos bidirecionais possuem um maior grau de paralelismo, pois surge as possibilidades de expandir as árvores em paralelo e de reduzir o espaço de busca. Escolhemos o BS^*, pertencente à classe dos algoritmos bidirecionais admissíveis e que não utilizam a técnica de 'wave-shaping', por apresentar o melhor desempenho dentro desta classe.

No capítulo dos métodos de busca paralela, são apresentadas versões dos algoritmos A^* e BS^* para serem executadas por mais de um processo. Todos os algoritmos paralelos apresentados mantêm as mesmas propriedades dos algoritmos seqüenciais originais. Basicamente são divididos em duas classes :

- na primeira, todas as informações sobre os nós da árvores de busca são centralizadas em um único processo, sendo chamados de algoritmos de lista centralizada;
- no segundo, as informações sobre os nós da árvore permanecem distribuídas entre os vários processo que tomam parte da busca, sendo chamados de algoritmos de lista distribuída.

Os algoritmos foram implementados em uma rede de 'transputers', utilizando a linguagem de programação OCCAM. No último capítulo é feita uma rápida abordagem do tipo de processador e da linguagem de programação utilizada. Neste capítulo é descrita a aplicação implementada, um planejador de rotas, e apresentados os resultados obtidos, seguidos das conclusões. São apresentados como resultados, as acelerações obtidas em cada algoritmo implementado. Definimos como aceleração, a razão entre o tempo de execução do algoritmo A_* e o tempo de execução do algoritmo em questão para solucionar um mesmo problema.

Capítulo II

Métodos de Busca Seqüencial

II.1 Notações

Um grafo consiste de um conjunto de nós que representam a codificação de um subproblema. Alguns pares de nós são conectados por arcos orientados , que representam as operações realizadas para alcançar um nó filho a partir de seu nó pai.

Se um arco é direcionado de um nó n para um nó n', o nó n' é chamado de sucessor de n e o nó n é chamado de pai de n'.

Freqüentemente associam-se pesos aos arcos do grafo, representando o custo associado à operação necessária para passar de um subproblema ao seu sucessor. Chama-se de geração de nó o ato de computar o código de representação de um nó a partir do código do seu nó pai. O novo nó é dito um nó gerado e o nó pai um nó explorado.

A expansão de um nó consiste na geração de todos os seus sucessores. A seqüência de nós $n_1, n_2, n_3, \ldots, n_{k-1}, n_k$, onde todo nó n_i é sucessor de n_{i-1}, é chamado de caminho de comprimento k de n_1 até n_k. Se existe um caminho de n_1 a n_k, afirma-se que n_k é descendente de n_1 e que n_1 é ancestral do nó n_k .

Um procedimento de busca consiste em estabelecer uma regra para determinar a ordem na qual os nós serão expandidos. Distingue-se um procedimento de busca desinformado ou cego de um direcionado. O primeiro estabelece a ordem de expansão baseado apenas na informação adquirida durante a busca, não levando em conta características da parte inexplorada do grafo, nem da solução que se deseja encontrar. Uma busca direcionada utiliza informações sobre o domínio do problema e a natureza do objetivo a alcançar

para estabelecer a ordem de expansão dos nós, de maneira que a direção mais promissora seja seguida.

Em qualquer momento da busca sobre o grafo, podemos dividir os seus nós em quatro conjuntos distintos :

1. nós expandidos;
2. nós explorados, mas não expandidos;
3. nós gerados, mas não explorados e
4. nós não gerados.

Os nós expandidos são também chamados de fechados e os demais de abertos.

II.2 Estratégias de Busca Não Informada

Como já foi dito, as estratégias de busca cega só utilizam informações adquiridas durante a própria busca, sendo a sua ordem de expansão independente da localização do nó objetivo no grafo [6].

Esses procedimentos mostram-se ineficientes e, geralmente, impraticáveis para problemas que possuem um grande espaço de busca. Eles serão descritos sucintamente para permitir uma comparação com os procedimentos direcionados. As descrições limitam-se a buscas conduzidas em árvores, sendo necessárias algumas alterações para utilizá-las em grafos gerais. Estas alterações serão discutidas nas próximas seções.

II.2.1 Depth-First : Uma Estratégia LIFO

Na busca"depth-first", atribui-se uma prioridade maior para os nós localizados nos níveis mais baixos do grafo de busca. Cada nó escolhido para exploração tem todos os seus sucessores gerados, após o que um dos seus sucessores, que acabaram de ser gerados, é selecionado para expansão. Essa ordem de expansão é seguida até que ocorra um bloqueio. Um bloqueio ocorre quando não são gerados sucessores ou um nó objetivo é encontrado. Se não foram gerados nós sucessores, o processo reinicia do nó aberto localizado no nível mais baixo do grafo.

Essa estratégia funciona bem para problemas com várias soluções igualmente desejáveis, ou quando há sinais bem evidentes de qual nó sucessor é o mais promissor para expansão. Mas pode ocorrer situações em que longos e infrutíferos caminhos sejam percorridos antes que a solução do problema seja encontrada.

II.2.2 Breadth-First : Uma Estratégia FIFO

Na busca"breadth-first", atribui-se uma maior prioridade aos nós localizados nos níveis mais altos do grafo, explorando progressivamente cada nível. Este procedimento implementa uma política "first-in first-out" entre os nós abertos, selecionando aqueles que permanecem nesse conjunto há mais tempo. Essa estratégia garante terminar com uma solução, quando esta existir, e que esta será a de nível mais alto no grafo, desde que o grafo seja localmente finito, ou seja, toodos os nós possuam um número finito de nós vizinhos (pais e filhos). A grande desvantagem desse método é a necessidade de armazenar todos os nós de dois níveis do grafo, os do nível que está sendo explorado e o do nível seguinte, que são gerados na expansão.

II.3 Estratégias Direcionadas

Os procedimentos de busca direcionada utilizam informações sobre o domínio do problema e a natureza da solução para conduzir a busca na direção mais promissora. Estas informações são chamadas de conhecimentos heurísticos e são utilizadas no passo de decisão de qual o próximo nó a expandir. Dessa forma, torna-se possível implementar uma nova política de busca, chamada de "best-first"(BF), que seleciona para expansão o nó mais promissor entre todos os nós abertos, independentemente de sua localização no grafo. A estimativa do nó mais promissor pode ser baseada em vários fatores. Uma possibilidade é determinar o grau de dificuldade em resolver o problema representado em cada nó. Outro caminho é estimar a qualidade do conjunto de soluções possíveis para cada subproblema, ou seja, o comprimento até o nó objetivo. Uma terceira alternativa é antecipar a quantidade de informação que poderá ser gerada na expansão de cada nó e a importância dessas informações para a busca.

Em todos os casos, a expectativa de cada nó n é estimada numericamente por uma função de avaliação heurística $f(n)$, a qual pode depender da descrição do nó, do

objetivo ou solução, das informações adquiridas durante a busca e qualquer informação extra sobre o domínio do problema.

II.3.1 Estratégias Best-First

Uma estratégia de busca "best-first" sobre um grafo segue os seguintes passos:

Algoritmo Best-First Genérico

1 insira o nó inicial s em uma lista chamada OPEN de nós a expandir;

2 se a lista OPEN estiver vazia, então termine o procedimento, informando que não existe solução.

3 remova da lista OPEN o nó n que possui o menor valor da função f e o insira em uma lista chamada CLOSED de nós expandidos.

4 expanda o nó n, gerando todos os seus sucessores com ponteiros para n.

5 se qualquer um dos sucessores do nó n for um nó objetivo, então termine o procedimento, fornecendo como solução o caminho, ao longo dos ponteiros, até o nó inicial s.

6 para cada sucessor m de n faça

6.1 calcule $f(m)$.

6.2 se não estiver na lista OPEN nem na lista CLOSED um nó m' igual a m, então pule para o passo 6.8.

6.3 se $f(m')$ for menor ou igual a $f(m)$, então despreze o nó m e pule para o passo 6.

6.4 substitua o nó m' pelo nó m.

6.5 direcione o ponteiro de m para n.

6.6 se m estiver na lista CLOSED, então transfira m para a lista OPEN.

6.7 pule para o passo 6.

6.8 insira m na lista OPEN.

7 volte ao passo 2.

O procedimento acima mantém uma árvore de busca $\mathbb{T}$, que é uma subárvore do grafo que representa o espaço do problema. Para isso, sempre que existirem dois caminhos entre s e um mesmo nó m, aquele que atribui a m o maior valor da função f é descartado (passos 6.3 a 6.5). Uma outra possibilidade para o passo 6.6 seria manter duplicatas de nós idênticos nas listas OPEN e CLOSED, eliminando a necessidade da pesquisa das listas, que pode ser computacionalmente intensa.

Uma outra variação possível consiste em apenas redirecionar os ponteiros e propagar novos valores aos descendentes de um nó n, sempre que um caminho melhor entre n e s for encontrado. Esta modificação evita a reexpansão do nó e de muitos dos seus descendentes, já gerados anteriormente, ao custo do redirecionamento dos ponteiros e da propagação de novos valores. Poderão existir redirecionamentos e propagações supérfluas, já que muitos dos descendentes podem não requerer regenerações, pois a busca pode tomar outra direção.

Utilizando qualquer uma das variações citadas, a qualquer momento o procedimento mantém, para cada nó explorado m, um único e distinto caminho entre o nó inicial e m.

II.3.2 Funções Peso Recursivas

Para um dado caminho P, designamos seu peso por W_P, onde W_P é a propriedade de P escolhida como medida de otimização, representando tanto o seu mérito (Q) como o seu custo (C). Se removermos de P todos os seus nós, exceto os descendentes de um certo nó n, a parte do caminho que permanece é um caminho solução para n e seu peso é denotado por $W_P(n)$. Em geral, o peso de um caminho solução pode ser uma função complexa de vários parâmetros do caminho, como : peso dos nós, peso dos arcos e peso dos nós extremos.

Definição 1 *Uma função peso $W_P(n)$ é recursiva se para todo o nó n do caminho*

$$W_P(n) = F[E(n); W_P(n_1)]$$

onde :

n_1 é o sucessor imediato de n;

$E(n)$ representa uma função das propriedades locais que caracterizam n;

F é uma função combinação, monotônica em seus argumentos $W_P(.)$.

Se tal função combinação F existe, é possível determinar o mérito de qualquer caminho a partir do custo do nó final e subindo em direção ao nó inicial, até determinar o mérito de todo o caminho solução. Esse tipo de função é também conhecida com o nome de "rollback".

Se todo o grafo for explorado, pode-se determinar o caminho ótimo e sua qualidade ou mérito $Q^*(s)$, que será igual ao máximo $Q_P(s)$ sobre todos os caminhos P do grafo, cujos nós terminais sejam um nó objetivo e que iniciem em s. Essa maximização pode ser feita recursivamente, por um procedimento de rotulação de mérito, utilizando-se o seguinte algoritmo :

Procedimento de Rotulação de Mérito

1. se n é um nó terminal (sem sucessores), então

   $Q^*(n) = v(n)$, onde $v(n)$ é o mérito final da solução associada ao nó n.

2. se n não é um nó objetivo e não possui sucessores, então n representa um subproblema sem solução e seu custo associado é $-\infty$, $Q^*(n) = -\infty$.

3. se n não é um nó terminal e possui sucessores $n_1, n_2, \ldots, n_b$, então

$$Q^*(n) = \max_i F[E(n); Q^*(n_i)], i = 1, \ldots, b$$

A solução ótima R^* é também definida em termos de Q^*, com o a seguir :

1. o nó s inicial está em R^*.

2. se um nó n está em R^*, então existe um sucessor m de n, que também está em R^* e é tal que

$$F[E(n); Q(m)] = \max_i F[E(n); Q(n_i)], i = 1, \ldots, b$$

Mas, como foi dito anteriormente, o grafo do espaço do problema muitas vezes é muito grande para ser gerado totalmente. Nesse caso, para um grafo parcialmente expandido, se atribuirmos a cada nó n da fronteira de busca um valor estimado de $Q^*(n)$ igual a $h(n)$, podemos determinar o que parece ser a solução de potencial mais promissor. Esta solução é realizada em dois passos :

1. ao último nó de cada caminho partindo do nó inicial, atribui-se uma função de avaliação $h(n)$, que estima a qualidade da melhor solução encontrada do subproblema descrito por n.
2. utiliza-se a função combinação F e o procedimento de rotulação de mérito como se os valores estimados $h(n)$ fossem os verdadeiros méritos terminais.

Podemos agora definir uma função recursiva de avaliação $e(n)$, que fornece uma estimativa de $Q^*(n)$ para todos os nós do grafo explorado :

$$e(n) = \begin{cases} h(n) & \text{se } n \text{ está na lista OPEN} \\ \max_i F[E(n); e(n_i)] & \text{se } n \text{ está na lista CLOSED,} \\ & \text{onde } n_i \text{ são os sucessores de } n \end{cases}$$

A partir da função de avaliação $e(n)$ podemos definir a solução R_0 com potencial mais promissor, do topo para baixo, como segue :

1. o nó inicial s está em R_0.
2. se um nó n da lista CLOSED está em R_0, então existe um sucessor m de n, que também está em R_0 e é tal que $e(n) = F[E(n); e(m)]$.

Tradicionalmente a função denotada por e deve representar os méritos da solução e ser maximizada e a função denotada por f deve representar os custos e ser minimizada. Para o algoritmo "Best-First Genérico", o valor da função $f(n)$ é igual a $-e(n)$.

Sob circunstâncias normais (como quando o peso da solução é determinado pela soma do custo de seus arcos), não é necessário percorrer todo o caminho de n até s para calcular a estimativa do caminho mais promissor de s até um nó objetivo, passando por n ($= e_n(s)$). Alguns poucos parâmetros armazenados na descrição do nó pai permitirão calcular $e_n(s)$ localmente e transmiti-lo de pai para filho a cada expansão de nó.

II.3.3 Algoritmos Best-First Especializados

Os algoritmos "best-first" diferem entre si pelo tipo de função de avaliação f utilizada. Calcular a função f, obter as informações necessárias para decidir qual o nó mais promissor e propagá-las dentro do grafo constituem a maior parte do esforço de busca. A seguir apresentamos alguns algoritmos "best-first" que realizam estas tarefas de busca de modo particular.

Algoritmo Z^*

Quando os valores da função de seleção f de um algoritmo BF são determinados recursivamente pelo procedimento de "rollback", o algoritmo passa a ser chamado de algortimo Z. Se ainda modificarmos este algoritmo , retardando o seu teste de terminação, como a seguir, ele torna-se o algoritmo Z^* :

Algoritmo Z^*

1 insira o nó inicial s em uma lista chamada OPEN de nós a expandir;

2 se a lista OPEN estiver vazia, então termine o procedimento, informando que não existe solução.

3 remova da lista OPEN o nó n que possui o menor valor da função f e o insira na lista chamada CLOSED de nós expandidos.

4 se o nó removido de OPEN for um nó objetivo, então termine o procedimento, fornecendo como solução o caminho, ao longo dos ponteiros, até o nó inicial s.

5 expanda o nó n, gerando todos os seus sucessores com ponteiros para n.

6 para cada sucessor m de n faça

6.1 calcule $f(m) = -e(m)$, onde $e(m)$ é uma função recursiva de avaliação..

6.2 se existir na lista OPEN ou na lista CLOSED um nó m' igual a m, então pule para o passo 6.5.

6.3 insira m na lista OPEN.

6.4 pule para o passo 6.

6.5 se $f(m')$ for menor ou igual $f(m)$, então despreze o nó m e pule para o passo 6.

6.6 substitua o nó m' pelo nó m.

6.7 direcione o ponteiro de m para n.

6.8 se m estiver na lista CLOSED, então transfira m para a lista OPEN.

7 volte ao passo 2.

Neste algoritmo, o valor da função f é calculado através de uma função recursiva de avaliação, que, como já foi descrito na seção anterior, utiliza uma função de "rollback". Comparando com o algoritmo BF genérico, notamos uma modificação, de modo a fazer que um nó só seja testado como nó objetivo ao ser selecionado para expansão e não quando é gerado. O algoritmo BF com esta mesma modificação chama-se BF^*.

Algoritmo A^*

Uma especialização do algoritmo Z^*, onde o objetivo da busca é encontrar o caminho de menor custo aditivo, chama-se A^* . O algoritmo consiste dos seguintes passos :

Algoritmo A^*

1 insira o nó inicial s em uma lista chamada OPEN de nós a expandir.

2 se a lista OPEN estiver vazia, então termine o procedimento, informando que não existe solução.

3 remova da lista OPEN o nó n que possui o menor valor da função f e o insira na lista chamada CLOSED de nós expandidos.

4 se o nó removido de OPEN for um nó objetivo, então termine o procedimento, fornecendo como solução o caminho, ao longo dos ponteiros, até o nó inicial s.

5 expanda o nó n, gerando todos os seus sucessores com ponteiros para n.

6 para cada sucessor m de n faça

6.1 calcule $h(m)$.

6.2 $g(m) := g(n) + c(n, m)$; onde $c(n, m)$ é o custo do nó n até o nó m.

6.3 $f(m) := g(m) + h(m)$.

6.4 direcione o ponteiro de m para n.

6.5 se existir na lista OPEN ou na lista CLOSED um nó m' igual a m, então pule para o passo 6.8.

6.6 insira m na lista OPEN.

6.7 pule para o passo 6.

6.8 se $g(m')$ for menor ou igual a $g(m)$, então despreze o nó m e pule para o passo 6.

6.9 substitua o nó m' pelo nó m.

6.10 se m estiver na lista CLOSED, então transfira m para a lista OPEN.

7 volta ao passo 2.

A diferença entre os algoritmos Z^* e A^* está no cálculo da função f, que no último assume a forma geral $f(m) = F[E(n), f(n), h(m)]$, onde $E(n)$ é um conjunto de parâmetros locais, caracterizando m (por exemplo, o custo de n até m).

II.3.4 Propriedades dos Algoritmos de Busca

Definiremos duas funções para os nós de um grafo :

$g^*(n)$ = o custo do melhor caminho entre o nó inicial s e o nó n e

$h^*(n)$ = o custo do melhor caminho entre o nó n e um nó objetivo β.

A função f, utilizada no passo de seleção do algoritmo A^*, consiste de duas parcelas aditivas, $g(n)$ e $h(n)$, onde :

$g(n)$ é o custo do caminho atual do nó inicial s até o nó n e

$h(n)$ é uma estimativa de $h^*(n)$, tal que $h(\beta) = 0$.

Existem, portanto, as seguintes relações :

$g_p(n) \geq g^*(n)$ e $h_p(n) \geq h^*(n)$.

O índice P indica o valor de g e h para um caminho particular P. Chamando de $f^*(n)$ o custo ótimo sobre todos os caminhos que levam do nó inicial s até um nó objetivo β e que passam pelo nó n, temos :

$$f^*(s) = h^*(s) = g^*(\beta) = f^*(\beta).$$

Se n^* é um nó que faz parte do caminho ótimo entre s e β, temos :

$$f^*(n^*) = f^*(s) = C^*,$$

onde C^* é o custo do melhor caminho entre s e um nó objetivo β.

Se n pertence a P^*, que é o melhor caminho entre s e β, então há um caminho P que passa por n e custa C^*, o que implica que $g_P(n) + h_P(n) = C^*$.

Utilizando a otimalidade de g^* e h^*, temos que $g^*(n) + h^*(n) \leq C^*$. Como não há outro caminho P' tal que $g_{P'}(n) + h_{P'}(n) < C^*$, pois P é o caminho ótimo, temos que $g^*(n) + h^*(n) = C^*$.

Da mesma forma, supondo que n seja um nó que não pertence ao caminho ótimo, implica que $f^*(n) > C^*$. Se $g^*(n) + h^*(n) \leq C^*$, então existiria um caminho P, passando por n, tal que $g_P(n) + h_P(n) \leq C^*$, qualificando P como o caminho ótimo e contradizendo a hipótese de n ser um nó fora do caminho ótimo.

As duas propriedades da função f^* :

$$f^*(n) > C^*, \quad n \notin P^* \text{ e}$$
$$f^*(n) = C^*, \quad n \in P^*$$

são equivalentes ao princípio da otimalidade da programação dinâmica, que afirma que um caminho é ótimo se e somente se todos os seus segmentos são ótimos. Essa é uma característica única das propriedades de recursividade e preservação da ordem das medidas de custo aditivo.

Essas propriedades fazem da função f^* um discriminador perfeito que fornece perfeitos sinais prévios de que um nó pertence ou não ao caminho ótimo. Perseguindo sempre os nós com valor de f^* mínimo, chegaríamos rápidamente à solução ótima, nunca sendo desviados para outros caminhos.

O problema é que normalmente nem todas as informações necessárias para computar f^* são acessíveis de maneira imediata, restando a possibilidade de aproximar f^* por uma função f mais acessível. O algoritmo A^* faz esta tentativa através da combinação $f = g + h$.

Já demonstramos que a função h^* obedece à seguinte equação :

$$g^*(n) + h^*(n) \geq C^* \; \forall n \Rightarrow$$

$$g^*(n) + h^*(n) \geq h^*(s) \Rightarrow$$

$$k(s,n) + h^*(n) \geq h^*(s)$$

onde $k(s,n)$ é o custo do caminho ótimo de s até n.

A inequação acima é válida para qualquer par de nós, não envolvendo necessariamente o nó s, pois temos :

a) se n' for um descendente de n :

$$h^*(n) \leq k(n,n') + h^*(n') \; \forall n$$

b) se n' não for descendente de n, então $k(n,n') = \infty$, logo a inequação $h^*(n) \leq k(n,n') + h^*(n')$ também é satisfeita.

Definição 2 *Uma função heurística $h(n)$ é* **consistente**, *se a inequação $h(n) \leq k(n,n') + h(n')$ é satisfeita para todo o par de nós n e n'.*

Definição 3 *Uma função heurística é* **admissível** *se :*

$$h(n) \leq h^*(n) \; \forall n$$

Definição 4 *Uma função heurística é* **monotônica** *se satisfaz a inequação $h(n) \leq c(n,n') + h(n')$ para qualquer par de nós n e n', tal que n' seja sucessor de n.*

Toda a função heurística consistente é também admissível, pois se substituirmos na equação n' por um nó objetivo β, teremos :

$$h(n) \leq k(n,\beta) + h(\beta) \; \forall n,$$

como $h(\beta) = 0$ e $k(n, \beta) = h^*(n)$, temos a condição de admissibilidade, $h(n) \leq h^*(n)$.

A partir deste ponto do trabalho consideraremos o algoritmo A^* sempre utilizando uma função heurística monotônica.

Lema 1 *Em qualquer momento antes do algoritmo A^* terminar, existe um nó n' na lista OPEN de nós não expandidos, que pertence a P^* com $f(n') \leq C^*$.*

Prova : Supondo que n' seja o nó do caminho ótimo mais próximo ao nó inicial s e que está na lista OPEN, então todos os seus ancestrais estão na lista CLOSED e seus ponteiros devem coincidir com o caminho P^* até n', logo $g(n') = g^*(n')$. Utilizando a admissibilidade de h temos :

$$f(n') = g(n') + h(n') = g^*(n') + h(n') \leq g^*(n') + h^*(n') = f^*(n');$$

$$f(n') \leq f^*(n') = C^*;$$

$$f(n') \leq C^*.$$

■

Teorema 1 *O algoritmo A^* é admissível, isto é, sempre encontra a solução ótima quando esta existir.*

Prova : Suponha que o algoritmo A^* termina com um nó objetivo β e $f(\beta) > C^*$. Como o algoritmo sempre escolhe o nó da lista OPEN com o menor valor da função f, para todo o nó n pertencente a OPEN : $f(\beta) \leq f(n)$. Isso significa que imediatamente antes da terminação, todos os nós da lista OPEN possuíam valor da função $f > C^*$, o que contradiz o lema 1. Logo o nó β deve ter $g(\beta) = C^*$, significando que o caminho ótimo foi encontrado. ■

Teorema 2 *Um algoritmo A^*, guiado por uma heurística monotônica, encontra sempre caminhos ótimos para todos os nós expandidos, isto é :*

$g(n) = g^*(n) \; \forall n \in CLOSED$ *(lista de nós expandidos).*

Prova : Suponha que o algoritmo A^* seleciona para expansão um nó n. Se n é o único nó na lista OPEN que faz parte do caminho P^*, ótimo de s até n, então todos os seus ancestrais ao longo do caminho P^* já foram expandidos, logo $g(n) = g^*(n)$ [6].

Se $g(n) > g^*(n)$, então o caminho P^* possui, no mínimo um outro nó na lista OPEN. Seja n' diferente de n, o nó de P^* na lista OPEN mais próximo de s (isto é, o de menor nível na árvore). Da mesma maneira, todos os ancestrais de n', ao longo do caminho P^*, estarão na lista CLOSED e $g(n') = g^*(n')$. Utilizando a propriedade da consistência, temos :

$$f(n') = g^*(n') + h(n') \leq g^*(n') + k(n', n) + h(n)$$

Como n' é um ancestral de n, então :

$$g^*(n) = g^*(n') + k(n', n).$$

Temos então que :

$$f(n') = g^*(n') + h^*(n') \leq g^*(n) + h(n)$$

Se $g(n)$ fosse maior do que $g^*(n)$, então $f(n') < f(n)$, o que levaria o nó n' a ser escolhido para expansão, e não n. Daí concluimos que $g(n) = g^*(n)$. Como todos os nós em CLOSED obedecem à equação $g(n) = g^*(n)$, quando utilizamos uma função monotônica, podemos afirmar que eles nunca serão reexpandidos pelo algoritmo A^*, pois se tornará impossível encontrar um valor de $f(n)$ menor que $g^*(n) + h(n)$.■

Terminação

O algoritmo A^* sempre termina em grafos finitos. A razão dessa propriedade é que o número de caminhos acíclicos em tais grafos é finito e em toda a expansão de nó, o algoritmo A^* adiciona caminhos à sua àrvore de busca. Cada novo arco representa um novo caminho acíclico, logo eles serão eventualmente exauridos.

Completude

O algoritmo A^* é completo sobre grafos finitos, isto é, sempre acha uma solução, se esta existir. Se o algoritmo não encontra a solução, então a sua lista OPEN de nós a expandir

está vazia. Mas se uma solução existe, isto quer dizer que houve um nó $n^* \in P^*$ em OPEN (ao menos o nó inicial $s \in P^*$ entrou na lista OPEN) que foi expandido e não gerou sucessores, o que é absurdo pois todo o nó pertencente ao caminho solução (exceto o nó objetivo) possui ao menos um sucessor que também pertence a P^*.

II.4 Algoritmos de Busca Bidirecional

Os algoritmos de busca bidirecional expandem simultaneamente duas frentes de busca , uma do nó inicial s até o um nó objetivo β e outra de um nó objetivo β até o nó inicial s . Eles podem ser usados quando se conhece os estados inicial e objetivo do problema e se deseja encontrar o caminho entre os dois. O objetivo é fazer com que as duas frentes se encontrem no meio do caminho, reduzindo assim o espaço de busca e o tempo de execução. A potencialidade deste método reside nas possibilidade de reduzir o espaço de busca, já que a soma do espaço de duas árvores de busca de profundidade $p/2$ é menor do que o de uma árvore de profundidade p, pois o tamanho de uma árvore tende a crescer exponencialmente com a sua profundidade.

O tempo de execução também tende a ser reduzido, pois um espaço de busca menor implica na expansão de um menor número de nós, diminuindo os esforços de cálculo de funções de avaliação e procura do nó mais promissor para expansão.

As potencialidades, aqui citadas, dos algoritmos bidirecionais só são alcançadas quando as frentes de busca convergem. No caso de existirem vários caminhos solução diferentes e de tamanhos comparáveis, as frentes de busca podem diferir no caminho a seguir durante as expansões, causando, na pior das hipóteses, a geração de duas árvores de busca distintas, podendo gerar mais que o dobro de nós do que um algoritmo unidirecional.

Para convergir as frentes de busca, são utilizadas várias técnicas, como "wave-shaping" ou a postulação de nós intermediários, que obrigatoriamente devem fazer parte da solução. O BS^* [1] utiliza o princípio da cardinalidade de Pohl como uma estratégia de controle para a convergência das frentes de busca. Este princípio estabelece que o próximo nó a expandir deve ser o nó mais promissor da direção de busca que possuir o menor número de nós abertos. Podemos interpretar este princípio como uma tentativa de manter os dois conjuntos de nós a expandir do mesmo tamanho, que seria a situação ideal

se as duas frentes se encontrassem exatamante no meio do espaço de busca.

O único requisito para que este algoritmo seja admissível, é que a heurística utilizada seja monotônica, como demonstraremos a seguir.

II.4.1 Algoritmo BS^*

Basicamente é utilizado um algoritmo A^* em cada direção de busca e uma variável comum às duas, que registra o custo do melhor caminho completo encontrado entre o nó inicial s e o nó objetivo β. Este caminho passa por um nó comum às duas árvores de busca, denominado $MEET$.

O valor da variável comum, aqui chamada de $LMIN$, sempre é atualizado quando um novo caminho completo é encontrado. Um caminho completo pode ser encontrado quando um nó gerado numa direção existir na árvore de direção oposta.

O algoritmo termina quando o valor de $LMIN$ for menor ou igual a todos os valores da função f dos nós de ambas as árvores ou uma das árvores tiver sido totalmente expandida.

São definidas quatro operações sobre os nós da árvore, que permitem diminuir o número de nós a serem expandidos. Essas operações são chamadas de *NIPPING*, *PRUNING*, *TRIMMING* e *SCREENING*.

NIPPING

Existe uma modificação em relação ao algoritmo A^*, no passo de seleção do nó mais promissor. O nó mais promissor só será realmente expandido se :

- não foi encontrado o melhor caminho completo que passe por este e
- existe possibilidade dele pertencer ao caminho completo ótimo.

Se o nó mais promissor não obedecer a essas condições, ele será transferido para o conjunto de nós expandidos sem realmente sofrer expansão, numa operação chamada *NIPPING*. Os nós que devem sofrer esta operação são aqueles que estão fechados na árvore oposta, pois :

- ou já se conhece o melhor caminho completo que passa por estes nó (quando $g = g^*$),
- ou então ele sofreu uma operação de *NIPPING* anteriormente, significando que não oferecia possibilidade de pertencer a um caminho melhor do que o já encontrado até o momento.

PRUNING

Outros nós que devem ser descartados de expansão são os descendentes de um nó que sofreu *NIPPING* na árvore oposta, pois ou o caminho passando por este já foi determinado ou não há possibilidade dele fazer parte de um caminho melhor do que o já encontrado. Essa operação é denominada de *PRUNING*.

TRIMMING

Os nós dos conjuntos OPEN, que possuem os valores da função f maiores do que o valor de *LMIN*, também podem ser descartados, numa operação denominada *TRIMMING*, pois:

a) se n está em OPEN e $g(n) = g^*(n)$, o seu valor da função f não irá ser modificado no futuro, e como, por admissibilidade, $f(n) \leq f^*(n)$, o melhor caminho passando por n não será melhor do que o melhor caminho encontrado até o momento;

b) se $g(n) > g^*(n)$, como a heurística é monotônica nas duas direções de busca, se isto foi causado por uma operação de *NIPPING*, *PRUNING* ou *TRIMMING*, significa que já havia determinado que nem n nem seus descendentes fazem parte do caminho ótimo;

c) se $g(n) > g^*(n)$ e isto não foi causado por uma operação de *NIPPING*, *PRUNING* ou *TRIMMING*, significa que existe um nó ascendente de n no conjunto OPEN, que ao ser expandido irá gerar o nó n com um valor de $g(n) = g^*(n)$. Logo, descartá-lo no momento, não irá afetar a solução.

SCREENING

Durante o processo de expansão, os sucessores gerados que possuírem o valor da função f maiores ou iguais a $LMIN$ podem ser descartados, em uma operação denominada *SCREENING*. Isto equivale a uma antecipação da operação *TRIMMING*, a que estes nós seriam submetidos se fossem inseridos no conjunto OPEN.

A seguir descrevemos o algoritmo :

Algoritmo BS^*

dicionário de dados

$g_i(n)$ e $h_i(n)$:	estimativas de $g_i^*(n)$ e $h_i^*(n)$, respectivamente;
$árvore_1$:	árvore de busca do nó inicial s para o nó objetivo β ;
$árvore_2$:	árvore de busca do nó objetivo β para o nó inicial s ;
$\mid OPEN_i \mid$:	número de nós a expandir na $árvore_i$;
d :	direção de busca ;
d' :	direção oposta à atual direção de busca ;
$c(m,n)$:	custo do nó m até o seu nó sucessor n ;
$g_i^*(n)$:	custo ótimo do nó s até o nó n se $i = 1$, ou do nó n até o nó β se $i = 2$;
$h_i^*(n)$:	custo ótimo do nó s até o nó n se $i = 2$, ou do nó n até o nó β se $i = 1$;
$p_i(m)$:	pai do nó m na árvore i ;
$f_i(n)$:	$g_i(n) + h_i(n)$;
$CLOSED_i$:	conjunto dos nós expandidos da árvore i ;
$OPEN_i$:	conjunto dos nós a expandir da árvore i ;
$T_1(n)$:	conjunto dos sucessores do nó n no grafo ;
$T_2(n)$:	conjunto dos pais do nó n no grafo.

1 $LMIN := \infty$; $g_1(s) := 0$;$g_2(\beta) := 0$; $f_1(s) := h_1(s)$; $f_2(\beta) := h_1(s)$.

2 coloque s no conjunto $OPEN_1$ e β no conjunto $OPEN_2$.

3 se $OPEN_1$ ou $OPEN_2$ estiverem vazios, então pule para o passo 4.

3.1 se $\mid OPEN_1 \mid \leq \mid OPEN_2 \mid$, então $d := 1$, senão $d := 2$.

3.2 $d' := 3 - d$.

3.3 transfira o nó n de $OPEN_d$ com o menor valor da função f_d para $CLOSED_d$.

3.4 se $n \notin CLOSED_{d'}$, então pule para o passo 3.8.

3.5 identifique o conjunto Ω formado pelos descendentes de n na árvore d' que estão no conjunto $OPEN_{d'}$.

3.6 Remova os nós em Ω do conjunto $OPEN_{d'}$. (operação de $PRUNING$)

3.7 pule para o passo 3.(operação de $NIPPING$)

3.8 $trimflag := false$.

3.9 para cada nó $m \in T_d(m)$, tal que $m \notin CLOSED_d$ faça

3.9.1 $g := g_d(n) + c_d(n, m)$.

3.9.2 $f := g + h_d(m)$.

3.9.3 se $(f \geq LMIN)$, então pule para o passo 3.9. (operação de $SCREENING$)

3.9.4 se $((m \in OPEN_d)$ e $(g_d(m) \leq g))$, então pule para o passo 3.9.

3.9.5 $g_d(m) := g$.

3.9.6 $f_d(m) := f$.

3.9.7 $p_d(m) := n$;

3.9.8 se $[(m$ está na árvore oposta)e $((g_1(m) + g_2(m)) \geq LMIN)]$ ou (m não está na árvore oposta), então pule para o passo 3.9 .

3.9.9 $LMIN := g_1(m) + g_2(m)$.

3.9.10 $MEET := m$.

3.9.11 $trimflag := true$.

3.10 se $trimflag$, então remova de $OPEN_1$ e $OPEN_2$ todos os nós cujos valores da função f sejam maiores ou iguais a $LMIN$ e que não sejam os nós s e β.(operação de $TRIMMING$)

3.11 pule para o passo 3.

4 se $LMIN = \infty$, então não existe solução , senão o caminho ótimo custa $LMIN$ e é dado pelos ponteiros p de ambas as árvores a partir do nó $MEET$;

II.4.2 Propriedades do BS^*

O algoritmo BS^* sempre termina quando aplicado sobre grafos finitos, pois o número de caminhos acíclicos em tais grafos é finito e em toda expansão de nó do algoritmo um novo caminho é adicionado a uma das árvores. Logo, eventualmente, todos os caminhos serão exauridos.

Antes que o algoritmo BS^* encontre o caminho completo ótimo P^*, existirão nós n_i e n_j, pertencentes aos conjuntos $OPEN_1$ e $OPEN_2$ respectivamente, tais que n_i e n_j fazem parte do caminho P^*.

Suponha que os caminhos $P_1 = (s, n_2, n_3, ..., n_i)$ e $P_2 = (n_j, n_{j+1}, n_{j+2}, ..., \beta)$ sejam subcaminhos não nulos de P^* e que o nó n_i é ancestral de $n_j (i < j)$, pois de outra maneira o caminho ótimo já teria sido encontrado.

Os nós n_i e/ou n_j estariam no conjunto $CLOSED_d$ se :

1. os nós n_i e/ou n_j foram selecionados para expansão e sofreu *NIPPING*;
2. os nós n_i e/ou n_j tinham um ancestral que sofreu *NIPPING* na árvore oposta, logo eles sofreram *PRUNING*;
3. o nó n_i e/ou o nó n_j foi expandido e o seu sucessor, que também estava no caminho ótimo P^*, foi removido do conjunto $OPEN_d$ por uma operação de *TRIMMING*;
4. o nó n_i e/ou o nó n_j foi expandido e o seu sucessor, que também estava no caminho ótimo P^*, não foi inserido no conjunto $OPEN_d$ por causa da operação *SCREENING*.

Lembrando que o algoritmo A^* guiado por uma heurística monotônica sempre encontra o caminho ótimo para todos os nós expandidos, podemos afirmar que os casos 1 e 2 implicam que os caminhos P_1 e P_2 tinham um nó comum, logo o caminho ótimo já havia sido encontrado. O caso 3 implica que o nó n_{i+1} ou o nó n_{j+1} sofreu *TRIMMING*, logo : $f_1(n_{i+1}) \geq LMIN$. Como n_{i+1} e seus ancestrais estão no caminho ótimo :

$$g_1(n_{i+1}) = g_1^*(n_{i+1})$$

$$f_1(n_{i+1}) = g_1^*(n_{i+1}) + h_1(n_{i+1}) \rightarrow f_1(n_{i+1}) \leq g_1^*(n_{i+1}) + h_1^*(n_{i+1}) \rightarrow$$

$f_1(n_{i+1}) \leq$ custo do caminho ótimo.

Supondo que o algoritmo ainda não encontrou o caminho ótimo, o custo do mesmo será menor que o valor da variável $LMIN$, implicando em $f_1(n_{i+1}) \leq LMIN$, contradizendo a hipótese 3.

O evento 4 resulta de uma operação de *SCREENING*, que pode ser encarado como uma antecipação do *TRIMMING*, o qual acabamos de demonstrar que não pode ocorrer.

Sendo descartadas todas as hipóteses de n_i e n_j pertencerem ao conjunto $CLOSED_d$, eles tem que pertencer ao conjunto $OPEN_d$. Utilizando a mesma prova para refutar a hipótese 3 para os nós $(i-1)$ e $(j-1)$, podemos demonstrar que antes do algoritmo terminar, os nós n_i e n_j estarão respectivamente nos conjuntos $OPEN_1$ e $OPEN_2$ e $f_1(n_i) \leq LMIN$ e $f_2(n_j) \leq LMIN$.

Se existe um caminho de s até β, o BS^* não terminará antes de achar o caminho ótimo entre estes dois nós. Como acabamos de provar, enquanto o algoritmo não achar o caminho ótimo, existirão nós n_i e n_j em $OPEN_1$ e $OPEN_2$ respectivamente, logo nenhum dos conjuntos OPEN estarão vazios, que é a condição para o algoritmo terminar.

Os algoritmos bidirecionais possuem o potencial de trabalharem sobre espaços de busca menores, significando que poderá haver uma significativa redução no tempo de execução através de uma implementação paralela.

Capítulo III

Métodos de Busca Paralela

Neste capítulo apresentamos procedimentos de busca paralela baseados nos algoritmos A^* e BS^*. A idéia básica é repartir a tarefa de expansão de um nó por vários processos. Cada processo gera os sucessores do nó que pertençam a um subconjunto do conjunto de nós que formam o espaço do problema. Surgem dois tipos principais de algoritmos :

- os que mantem todos os nós abertos em um único processo, que são chamados de algoritmos de lista centralizada, e
- aqueles nos quais o conjunto de nós abertos permanece disperso entre os vários processos, que são denominados algoritmos de lista distribuída.

III.1 Versões Paralelas do Algoritmo A^*

III.1.1 Lista Centralizada

Nesta versão, um processo mestre M comanda k processos escravos E_i, $0 \leq i \leq k-1$. O processo mestre controla as duas listas de nós do algoritmo , $OPEN$ e $CLOSED$. A cada expansão de nó, o processo mestre seleciona o nó m mais promissor da lista $OPEN$ e o transfere para a lista $CLOSED$, depois o transmite aos processos escravos, que realizam a tarefa de gerar os sucessores de m e calcular os respectivos valores das funções de avaliação (g e f). O processo mestre recebe os nós gerados pelos processos escravos e os insere na lista $OPEN$. Após receber os nós de todos os escravos, o mestre seleciona um novo nó para expansão. O final do algoritmo é detectado pelo mestre, ao selecionar um nó objetivo para expansão ou quando a sua lista $OPEN$ estiver vazia. O final do algoritmo é então comunicado aos seus escravos. Os processos escravos E_i consistem básicamente de um

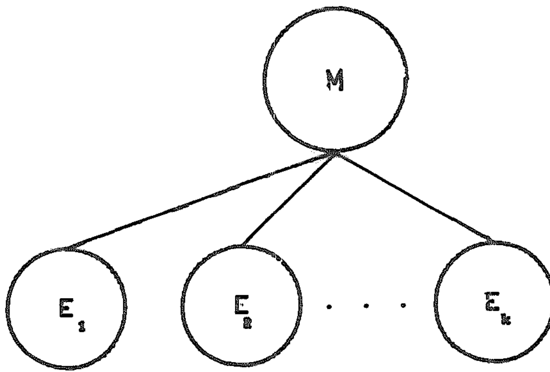


Figura III.1: Processos do Algoritmo A^* Paralelo de Lista Centralizada

"loop" de receber um nó m, gerar todos os sucessores de m que estejam contidos em um subconjunto Ψ_i do espaço do problema, calculando os seus respectivos valores das funções f e g, e ransmitir estas informações de volta ao mestre. Um processo escravo termina quando recebe uma comunicação do processo mestre sobre o final do algoritmo.

Um esquema dos processos é apresentado na figura III.1.

Processo Mestre do Algoritmo A^* Paralelo de Lista Centralizada

1 insira nó inicial s na lista $OPEN$.

2 se a lista $OPEN$ estiver vazia, então pule para o passo 11.

3 selecione o nó n da lista $OPEN$ com o menor valor da função f.

4 se nó n e' um nó objetivo, então pule para o passo 11.

5 envie n e $g(n)$ para todos os processos escravos E_i.

6 transfira o nó n para a lista $CLOSED$.

7 $recebidos := 0$.

8 recebe nós gerados de um dos processos $E_i \rightarrow$

8.1 para cada nó m gerado por E_i faça

8.1.1 se $(m \in OPEN$ ou $n \in CLOSED)$ e $(g(m) \leq g_i(m))$, então pule para o passo 8.1.1.

8.1.2 $f(m) := f_i(m); g(m) := g_i(m); p(m) := n$.

8.1.3 se m está na lista $CLOSED$, então tranfira o nó m da lista $CLOSED$ para a lista $OPEN$, senão insira o nó m na lista $OPEN$.

8.2 $recebidos := recebidos + 1$.

9 se $recebidos <$ número de processos escravos, então pule para o passo 8.

10 pule para o passo 2.

11 envie mensagem de fim de execução para todos os processos escravos E_i.

12 se lista $OPEN$ está vazia, então não há solução, senão a solução é o caminho de n até s através dos ponteiros p;

Se compararmos o algoritmo do processo mestre com o algoritmo A^* descrito no capítulo anterior, verificamos que a diferença encontra-se apenas nos passos correspondentes às tarefas de gerar os sucessores e calcular as suas estimativas de custo. Estas tarefas foram dividas entre os processos escravos. Notemos que os subconjuntos Ψ_i dos nós sucessores devem ser disjuntos, para que os escravos não realizem passos repetidos, isto é :

$$\Psi_i \cap \Psi_j = \emptyset \; \forall i \neq j.$$

Para o caso particular de utilização de uma função heurística monotônica demonstramos que um nó fechado nunca será reaberto. Como todos os processos escravos recebem todos os nós fechados, eles podem manter uma lista dos nós fechados, idêntica à que seria mantida pelo nó mestre.

Cada processo escravo é também capaz de manter uma parte da lista $OPEN$ do processo mestre, formada pelos nós gerados pelo processo e enviados ao processo mestre e ainda não recebidos de volta para expansão. Os valores das funções f e g dos nós da parte da lista $OPEN$ também são mantidos atualizados, já que os conjuntos Ψ_i são disjuntos, qualquer alteração no valor de uma funcão de um nó n pertencente a Ψ_i, será sempre calculada pelo mesmo processo E_i.

Notamos então, que utilizando uma função monotônica e sendo os subconjuntos Ψ_i disjuntos, podemos eliminar a lista $CLOSED$ do processo mestre e transferir os passos

correspondentes à manutenção desta lista do processo mestre para os processos escravos. Teremos então os seguintes algoritmos :

Processo Mestre do Algoritmo A^* Paralelo de Lista Centralizada

1 insira nó inicial s na lista $OPEN$.

2 se a lista $OPEN$ estiver vazia, então pule para o passo 11.

3 selecione o nó n da lista $OPEN$ com o menor valor da função f.

4 se nó n e' um nó objetivo, então pule para o passo 11.

5 envie n e $g(n)$ para todos os processos escravos E_i.

6 retire n da lista $OPEN$.

7 $recebidos := 0$.

8 recebe nós gerados de um dos processos E_i →

8.1 para cada nó m gerado por E_i faça

8.1.1 $f(m) := f_i(m); g(m) := g_i(m); p(m) := n$.

8.1.2 insira o nó m na lista $OPEN$.

8.2 $recebidos := recebidos + 1$.

9 se $recebidos <$ número de processos escravos, então pule para o passo 8.

10 pule para o passo 3.

11 envie mensagem de fim de processamento para todos os processos escravos E_i.

12 se lista $OPEN$ está vazia, então não há solução, senão a solução é o caminho de n até s através dos ponteiros p;

Processo Escravo i do Algoritmo A^* Paralelo de Lista Centralizada

1 recebe n e $g(n)$ do processo mestre →

1.1 insira n na lista $CLOSED_i$.

1.2 se n estiver na lista $OPEN_i$, então retire n de $OPEN_i$.

1.3 para cada nó m sucessor de $n \in \Psi_i$, tal que $\notin CLOSED_i$ faça

1.3.1 calcule $h(m)$.

1.3.2 $a := g(n) + c(n, m)$.

1.3.3 $b := a + h(m)$.

1.3.4 se $m \in OPEN$ e $g_i(m) \leq a$, então pule para o passo 1.3.

1.3.5 $g_i(m) := a; f_i(m) := b$.

1.3.6 insira m na lista GERADOS.

1.3.7 se $m \notin OPEN_i$, então insira m em $OPEN_i$.

1.4 envie para processo mestre os nós da lista GERADOS, com respectivos valores de f_i e g_i.

1.5 esvazie a lista GERADOS.

1.6 pule para o passo 1.

2 recebe mensagem de fim de processamento do processo mestre $\rightarrow$ finaliza o processo.

A solução encontrada por esta versão paralela possui as mesmas propriedades da solução encontrada pelo algoritmo A^* seqüencial, pois a ordem de seleção dos nós para expansão e o cálculo das estimativas são idênticos.

III.1.2 Lista Distribuída

Na versão de lista centralizada, a lista $OPEN$ dos nós abertos era centralizada no processo mestre, que a mantinha ordenada e selecionava o nó mais promissor para expansão. Dessa maneira a comunicação com o processo mestre torna-se um gargalo ("bottleneck").

A razão para manter a lista $OPEN$ centralizada era facilitar a seleção do nó mais promissor daquele conjunto. Porém se encontrarmos maneira simples e eficiente de selecionarmos o nó promissor, mesmo com a lista distribuída, esta necessidade desaparece.

Teorema 3 *Seja o conjunto $OPEN$ dividido em k subconjuntos disjuntos $OPEN_i$. Denominemos M o nó que obedece à relação $f(M) \leq f(n) \forall n \in OPEN$ e M_i ,$i = 1, 2, \ldots, k$, os nós pertencentes a $OPEN_i$, tal que $f(M_i) \leq f(m) \forall m \in OPENi$. Seja Ω o conjunto*

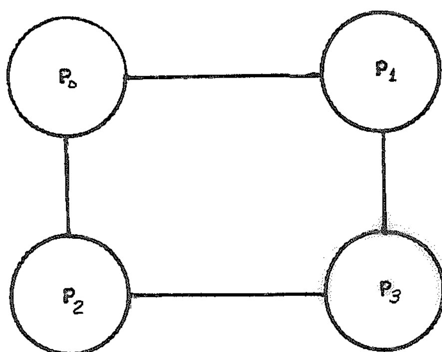


Figura III.2: Processos do Algoritmo A^* Paralelo de Lista Distribuída

formado pelos k nós M_i, i de 1 até k, e M_Ω o nó pertencente a Ω, tal que $f(M_\Omega) \leq f(n) \forall n \in \Omega$. Afirma-se que o nó M é igual a M_Ω.

Prova : Supondo que o nó M não pertence ao conjunto Ω, e que M pertença ao subconjunto $OPEN_j$. Se M não é igual a M_j, então $f(M) > f(M_j)$, o que contraria a hipótese de M ter o menor valor da função f. Logo o nó M pertence ao conjunto Ω.

Supondo que o nó M é diferente de M_Ω, teríamos que $f(M_\Omega) \leq f(M)$. Como M_Ω pertence ao conjunto $OPEN$, estaríamos contrariando a hipótese de M ser o nó do conjunto $OPEN$ com o menor valor da função f. Conclui-se que $M_\Omega = M$.■

No algoritmo de lista centralizada provamos que se utilizarmos uma heurística monotônica e os subconjuntos Ψ_i forem disjuntos, cada processo escravo E_i pode manter uma parte da lista $OPEN$ e essas partes serão disjuntas. Dessa maneira, a seleção do nó mais promissor resume-se à tarefa de escolher o melhor nó entre o conjunto formado pelo melhor nó da lista $OPEN$ de cada processo E_i. Isso pode ser feito de uma maneira simples, como mostraremos adiante.

Dividiremos o algoritmo A^* em k processos P_i, onde k é uma potência de dois. Cada processo P_i mantém uma lista de nós fechados $CLOSED_i$ e uma lista de nós abertos $OPEN_i$, com as mesmas características das listas mantidas pelos processos escravos do algoritmo de lista centralizada.

O esquema dos processos é apresentado na figura III.2.

Para iniciar a expansão de um novo nó, é necessário selecionar o nó N mais promissor da lista $OPEN$ global. Para isso, cada processo seleciona o nó mais promissor n_i de sua lista local $OPEN_i$.

Utilizando-se $k = 2^d$ e cada processo executando os passos do algoritmo abaixo, todos os processos determinam qual o nó N mais promissor :

Algoritmo de Seleção do Nó Mais Promissor

1 para j := 0 até d faça

   1.1 envia nó n_i pelo canal j e recebe nó n_r pelo canal j;

   1.2 se n_r é mais promissor que n_i então $n_i := n_r$;

2 $N := n_i$.

N = nó mais promissor da lista $OPEN$ global,

i = número do processo P_i , que varia de 0 até $2^d - 1$,

canal j = canal de comunicação entre o processo i e o processo m, tal que as representações binárias de i e m diferem apenas no bit j.

O algoritmo proposto acima é totalmente adequado à uma arquitetura do tipo hipercubo, mas pode ser utilizado em outros tipos, se for possível estabelecer as rotas de comunicação necessárias.

Como os conjuntos Ψ_i são disjuntos, para qualquer nó n :

- se n está na lista $OPEN$ de um processo i, o mesmo nó n não se encontrará em nenhuma outra lista $OPEN$ de outro processo $j, j \neq i$.
- um nó n nunca estará em ambas as listas $OPEN$ e $CLOSED$ de um mesmo processo simultaneamente.

A partir das afirmações acima podemos assegurar que o nó selecionado pelo algoritmo acima é o melhor nó global.

O fim da lista $OPEN$ global é detectado quando o nó selecionado pelo algoritmo de seleção do nó mais promissor for um nó vazio.

Informados sobre qual o nó n selecionado, cada processo i coloca n em sua lista $CLOSED_i$ e inicia a geração dos sucessores de n que pertencem ao seu conjunto Ψ_i, inserindo-os na sua lista $OPEN_i$.

O algoritmo termina quando o nó selecionado pelo algoritmo for um nó objetivo ou um nó vazio.

Ao término do algoritmo, todos os processos possuem a resposta, já que todos os nós do caminho solução estão na lista $CLOSED$ e todos os processos possuem uma cópia idêntica desta lista.

Como a ordem de escolha dos nós a expandir e o cálculo das funções de avaliação permanecem iguais ao da versão seqüencial, esta versão paralela do algoritmo A^* mantém as mesmas propriedades do método seqüencial.

Processo do Algoritmo A^* de Lista Distribuída

1 insira nó raiz na lista $OPEN_i$;

2 selecione o nó n_i mais promissor da lista $OPEN_i$;

3 execute o algoritmo de seleção do nó mais promissor;

4 se o nó mais promissor global N é um nó vazio ou N é um nó objetivo, então pule para o passo 10..

5 se N estiver na lista $OPEN_i$, então retira N da lista $OPEN_i$;

6 insira N na lista $CLOSED_i$;

7 gere todos os sucessores de N que pertençam a Ψ_i;

8 para cada nó sucessor m gerado que não está em $CLOSED_i$ faça

8.1 calcule $h(m)$;

8.2 $a := g(N) + c(N, m)$;

8.3 $b := a + h(m)$;

8.4 se (m está em $OPEN_i$ e $g(m) \leq a$), então pule para o passo 8.

8.5 $g(m) := a$;

8.6 $f(m) := b$;

8.7 $p(m) := N$;

8.8 se (m não está em $OPEN_i$), então insira m em $OPEN_i$;

9 pule para o passo 2.

10 se N é um nó vazio, então não há solução, senão a solução é o caminho de N até s através dos ponteiros p;

III.2 Versões Paralelas do Algoritmo BS^*

A idéia básica é que existam dois conjuntos de processos, cada um expandindo a árvore de busca em uma direção, que se comuniquem periodicamente para atualizar a informação de melhor caminho completo encontrado. Utilizando um princípio simples, essa comunicação pode ocorrer sempre que ambos os conjuntos acabarem de expandir um nó de sua árvore.

Outras mensagens deverão ser trocadas entre os processos para a implementação das operações de *PRUNING*, *NIPPING*, *TRIMMING* e *SCREENING*.

III.2.1 Lista Centralizada

Como o algoritmo BS^* pode ser encarado como duas árvores sendo expandidas pelo método A^*, uma das alternativas de paralelização é que cada árvore execute as expansões em paralelo como descrito na versão paralela do A^* com lista centralizada.

Existirão dois processos mestres (M_d e $M_{d'}$), um para cada árvore, que gerenciarão as listas $OPEN$. Cada processo M_d mestre terá um conjunto de k processos escravos $E_{d,i}$, i de 1 até k, que realizarão as expansões de nó.

Cada processo escravo $E_{d,i}$ da árvore d gera os nós sucessores que pertencem ao subconjunto Ψ_i do conjunto de nós que formam o espaço do problema.

O esquema dos processos é apresentado na figura III.3.

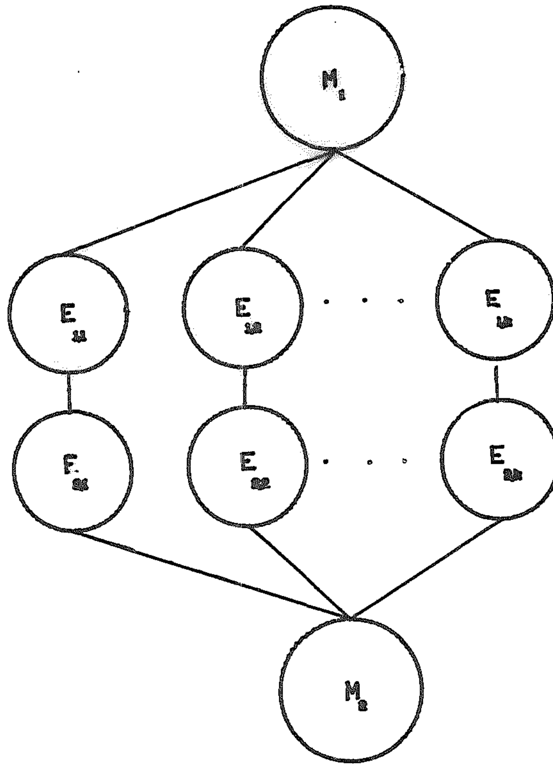


Figura III.3: Processos do Algoritmo BS^* Paralelo de Lista Centralizada

Operações de *NIPPING* e *PRUNING*

Cada processo mestre M_d enviará o nó N_d mais promissor de sua lista $OPEN$ para os processos escravos $E_{d,i}$ realizarem sua expansão. Porém, se o nó N_d estiver fechado na árvore oposta (d'), sua expansão pode ser cancelada. Para verificar se N_d sofrerá *NIPPING*, cada processo $E_{d,i}$ envia o nó N_d recebido para o processo $E_{d',i}$, que enviará uma mensagem informando se o nó está na sua lista $CLOSED$. Como já foi visto na versão do algortmo A^*, todos os processos escravos possuem uma cópia completa da lista $CLOSED$ global da árvore que estão expandindo. Logo todos os processos $E_{d,i}$ receberão uma resposta igual e correta. Caso a resposta do processo $E_{d',i}$ seja afirmativa, o processo $E_{d,i}$ não irá expandir o nó N_d e o processo $E_{d',i}$ irá informar o processo $M_{d'}$, que por sua vez irá retirar todos os descendentes de N_d de sua lista $OPEN$. O processo $E_{d',i}$ também retira todos os descendentes de N_d que estão na sua lista $OPEN$.

Operações de *TRIMMING* e *SCREENING*

Se N_d não sofrer *NIPPING*, então cada processo $E_{d,i}$ irá gerar o conjunto $\Omega_{d,i}$ dos nós sucessores de N_d que :

- pertencem ao subconjunto Ψi,
- não pertencem à lista $CLOSED_{d,i}$ e
- cujo valor da função f é menor ao seu $LMIN_{d,i}$.

Cada nó do conjunto $\Omega_{d,i}$ é inserido na lista *OPEN* do processo $E_{d,i}$. Se N_d sofreu *NIPPING*, o conjunto $\Omega_{d,i}$ será vazio. Após a expansão, cada processo $E_{d,i}$ envia o conjunto $\Omega_{d,i}$ para o processo escravo da árvore oposta $E_{d',i}$ e para o processo mestre M_d.

Um processo escravo, ao receber o conjunto dos nós gerados pelo processo escravo da árvore oposta, verifica se acha algum novo caminho completo, isto é, se algum dos nós gerados está na sua lista $OPEN_{d,i}$ ou $CLOSED_{d,i}$. Se o nó gerado estiver na lista $CLOSED_d$ global, ele estará na lista $CLOSED_{d,i}$ do processo, pois são idênticas. Se o nó gerado estiver na lista $OPEN_d$ global, ele estará na lista $OPEN_{d,i}$ do processo, pois ele pertence ao subconjunto Ψ_i e todos os nós desse conjunto são gerados pelo processo $E_{d',i}$, que armazena todos os nós gerados na sua lista $OPEN_{d,i}$. A cada novo caminho completo identificado, se o seu custo for menor do que o valor de $LMIN_{d,i}$, o processo atualiza o seu valor e guarda o nó comum na variável $MEET_{d,i}$. Após esse passo, cada processo escravo pode ter um valor de $LMIN_{d,i}$ diferente. O ideal é que todos os processos utilizem o menor valor de $LMIN$ encontrado, para acelerar o algoritmo, pois quanto menor o valor de $LMIN$, mais nós poderão ser descartados pelas operações de *TRIMMING* e *SCREENING*, diminuindo a lista *OPEN* global.

Realizamos esta unificação do valor de $LMIN$ em duas fases. Na primeira fase, determinamos o melhor valor $LMIN_{d,i}$ de cada árvore. Cada processo escravo envia o seu valor de $LMIN_{d,i}$ e o seu $MEET_{d,i}$ para o processo mestre, que estabelece o melhor valor $LMIN_d$ e o divulga para todos os processos escravos. Na segunda fase, cada par de processos $E_{d,i}, E_{d',i}$ trocam os seus valores de $LMIN_d$, que agora são o melhor de cada árvore, entre si e registram o melhor valor. Desta maneira, todos os processos terão registrado o melhor valor global de $LMIN$.

Após determinar o melhor valor de $LMIN$, um dos processos escravos de cada árvore informa ao seu processo mestre qual o valor de $LMIN$, que o utilizará para realizar a operação de $TRIMMING$ na sua lista $OPEN$.

O algoritmo do processo mestre M_d obedece os seguintes passos :

Processo Mestre do Algoritmo BS^* Paralelo de Lista Centralizada

1 insira o nó inicial s na sua lista $OPEN$.

2 $LMIN := \infty$.

3 selecione o nó N_d mais promissor de sua lista $OPEN_d$.

4 envie o nó N_d para todos os seus k processos escravos $E_{d,i}$.

5 se N_d for um nó vazio então pule para o passo 16.

6 $recebidos := 0$.

7 se $recebidos = k$, então pule para o passo 12 .

8 recebe conjunto $\Omega_{d,i}$ do processo $E_{d,i}$ $\forall i, 1 \leq i \leq k \rightarrow$

8.1 para cada nó m do conjunto $\Omega_{d,i}$ faça

8.1.1 $f_d(m) := f_{d,i}(m)$.

8.1.2 $g_d(m) := g_{d,i}(m)$.

8.1.3 $p_d(m) := N_d$.

8.2 volte para o passo 7.

9 recebe valor de $LMIN_{d,i}$ e $MEET_{d,i}$ do processo $E_{d,i}$ $\forall i, 1 \leq i \leq k \rightarrow$

9.1 se $LMIN_{d,i} < LMIN$, então $LMIN := LMIN_{d,i}$ e $MEET := MEET_{d,i}$.

9.2 $recebidos := recebidos + 1$.

9.3 volte para o passo 7.

10 recebe nó $N_{d'}$ que sofreu $NIPPING$ do processo $E_{d,1} \rightarrow$

10.1 retira todos os descendentes de $N_{d'}$ de sua lista $OPEN$.

10.2 volte para o passo 7.

11 recebe mensagem de fim da fila oposta do processo $E_{d,1} \rightarrow$

11.1 pule para o passo 16.

12 envie valor de $LMIN$ e $MEET$ para todos os seus processos escravos $E_{d,i}$.

13 recebe valor de $LMIN$ e de $MEET$ do processo $E_{d,1}$.

14 retire de sua lista $OPEN$ todos os nós cujo valor da função f é maior ou igual a $LMIN$.

15 volte para o passo 3.

16 se $LMIN \neq \infty$, então parte da resposta é o caminho do nó $MEET$ até o nó inicial s pelos ponteiros p_d.

O algoritmo de um processo escravo $E_{d,i}$ obedece os seguintes passos :

Processo Escravo i do Algoritmo BS^* Paralelo de Lista Centralizada

1 $LMIN_{d,i} := \infty$.

2 receba nó N_d do processo mestre M_d.

3 envie o nó N_d para o processo $E_{d',i}$ e receba o nó $N_{d'}$ do processo $E_{d',i}$.

4 se N_d ou $N_{d'}$ for um nó vazio então pule para o passo 20.

5 se o nó $N_{d'}$ não está na sua lista $CLOSED$, então $F_p := TRUE$ senão $F_p := FALSE$.

6 envie F_p para o processo $E_{d',i}$ e receba $F_{p'}$ do processo $E_{d',i}$.

7 se $F_p = TRUE$ e $i = 1$, então envia nó $N_{d'}$, que sofrerá $NIPPING$, para o processo mestre M_d.

8 se $F_{p'} = TRUE$, então pule para o passo 11.

9 gere o conjunto $\Omega_{d,i}$ de todos os nós sucessores do nó N_d que pertençam ao subconjunto Ψ_i e nao pertençam à lista $CLOSED_{d,i}$.

10 para todos o nó $m \in \Omega_{d,i}$ faça :

10.1 $g := g_d(N_d) + c_d(N_d, m)$.

10.2 $f := g + h_d(m)$.

10.3 se $[(m \in OPEN)$ e $(g \geq g_d(m))]$ ou $(f > LMIN_i)$, então retire m de $\Omega_{d,i}$ e pule para passo 10.

10.4 $g_{d,i}(m) := g.f_{d,i}(m) := f$.

10.5 se $m \notin OPEN$ então insira m em $OPEN$.

11 envie $\Omega_{d,i}$ para o processo $E_{d',i}$ e recebe $\Omega_{d',i}$ do processo $E_{d',i}$.

12 envie $\Omega_{d,i}$ para o processo mestre M_d.

13 para todo o nó $m \in \Omega_{d',i}$ faça :

13.1 se $m \in CLOSED$ ou $m \in OPEN$ e $g_d(m) + g_{d'}(m) < LMIN_{d,i}$, então $LMIN_{d,i} := g_d(m) + g_{d'}(m)$ e $MEET_{d,i} := m$.

14 envie $LMIN_{d,i}$ e $MEET_{d,i}$ para o processo mestre M_d.

15 receba $LMIN_d$ e $MEET_d$ do processo mestre M_d.

16 envie $LMIN_d$ e $MEET_d$ para o processo $E_{d',i}$ e receba $LMIN_{d'}$ e $MEET_{d'}$ do processo $E_{d',i}$.

17 se $LMIN_{d'} < LMIN_d$, então $LMIN_{d,i} := LMIN_{d'}$ e $MEET_{d,i} := MEET_{d',i}$.

18 se i = 1, então envie $LMIN_{d,i}$ e $MEET_{d,i}$ para o processo mestre M_d.

19 volte para passo 2.

20 se $i = 1$ então envie mensagem de fim da fila oposta para o processo mestre M_d.

21 termine o processo.

III.2.2 Lista Distribuída

Podemos dividir a tarefa de expandir a árvore de uma direção por mais de um processo, utilizando a idéia de lista distribuída do descrita na versão paralela do algoritmo A^*.

A configuração básica é que cada processo $E_{d,i}$ de uma árvore expandindo na direção d tenha um processo equivalente $E_{d',i}$ na árvore de direção oposta d', sendo que :

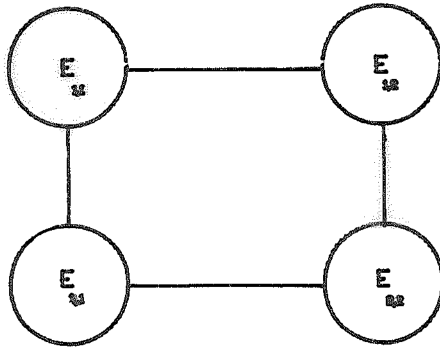


Figura III.4: Processos do Algoritmo BS^* Paralelo de Lista Distribuída

- existe um canal de comunicação entre os todos os pares de processos $E_{d,i}, E_{d',i}$ e
- os processos $E_{d,i}$ e $E_{d',i}$ sómente geram sucessores que pertençam ao subconjunto Ψ_i dos nós que formam o espaço do problema.

O esquema dos processos é apresentado na figura III.4.

Dessa forma, cada processo de uma árvore terá um processo respectivo na árvore oposta. Note-se que esta configuração é muito adequada para configurações hipercúbicas.

Seja um hipercubo de dimensão dim. Podemos dividir os processadores do hipercubo em dois conjuntos de 2^{dim-1} processadores, de tal maneira que o bit mais significativo do identificador de todos os processadores de um conjunto seja igual. Notamos que os elementos de um conjunto assim montado, formam um hipercubo de dimensão $(dim - 1)$, e que cada processadores de um conjunto possuem um canal de comunicação com um processador distinto do conjunto oposto, e a representação binária dos identificadores destes processos diferem apenas no bit mais significativo.

Por exemplo, considere um hipercubo de dimensão 3 :

representação binária dos identificadores dos processadores : 000, 001, 010, 011, 100, 101, 110, 111,

teremos os seguintes grupos :	grupo 0	grupo 1
	000	100
	001	101
	010	110
	011	111

Utilizando um grupo de processadores para cada árvore, podemos aplicar o algoritmo A^* paralelo de lista distribuída para expandir cada árvore , a partir dos quais, com algumas modificações, implementa-se uma busca bidirecional.

Operações de *NIPPING* e *PRUNING*

Sabemos que no início de uma iteração de expansão de uma árvore, todos os processos terão uma cópia do nó n selecionado para expansão. Para implementarmos as operações de *NIPPING* e *PRUNING*, antes de expandir o nó, cada par de processos $E_{d,i}$,$E_{d',i}$ devem trocar a informação de qual nó vai expandir. O processo $E_{d,i}$ ao receber o nó $n_{d'}$, a ser expandido pelo processo $E_{d',i}$, consulta se $n_{d'}$ está na sua lista *CLOSED* e informa ao processo $E_{d',i}$. Se o processo $E_{d',i}$ recebe uma mensagem de $E_{d,i}$, informando que $n_{d'}$ está fechado na árvore oposta, ele não expande o nó, colocando-o diretamente na fila *CLOSED*, e realiza a operação *PRUNING* na sua lista *OPEN*. Como todos os processos de uma mesma árvore possuem listas *CLOSED* idênticas, todos os processos de uma mesma árvore recebem a mesma resposta dos processos da árvore oposta. Logo podemos afirmar que a operação de *PRUNING* é realizada na lista *OPEN* global de uma árvore.

Operações de *TRIMMING* e *SCREENING*

Se o nó selecionado para expansão numa árvore não sofreu *NIPPING*, todos os processos da árvore realizam a sua partição da expansão do nó. Durante a expansão, cada processo pode realizar a operação de *SCREENING* nos nós gerados cujo valor da função f for maior ou igual ao valor de *LMIN*. Ao acabar a expansão, todo o processo $E_{d,i}$ envia os sucessores gerados para o processo $E_{d',i}$, a fim de verificar se existem novos caminhos completos entre o nó inicial e o nó objetivo.

Se um sucessor gerado pelo processo $E_{d,i}$ está na árvore oposta, ele pertence a uma das listas do processo $E_{d',i}$, pois esses dois processos geram os nós que pertencem ao

mesmo subespaço Ψ_i do espaço do problema.

O processo $E_{d',i}$, ao receber os nós gerados pelo processo $E_{d,i}$, verifica se estes pertencem a alguma de suas listas ($OPEN$ ou $CLOSED$). Caso pertençam, verifica se os caminhos que passam por estes nós são melhores que o melhor caminho já registrado na sua variável $LMIN$. Se encontrar um caminho melhor, registra o custo do melhor caminho em $LMIN$ e o nó comum na variável $MEET$.

Nota-se que ao final da verificação dos nós gerados pela árvore oposta em busca de novos caminhos completos, cada processo poderá ter um valor de $LMIN$ e $MEET$ diferente. O ideal é que todos os processos utilizem o menor valor de $LMIN$ encontrado, para acelerar o algoritmo, pois quanto menor o valor de $LMIN$, mais nós poderão ser descartados pelas operações de $TRIMMING$ e $SCREENING$, diminuindo a lista $OPEN$. Para que todos os processos utilizem o menor valor de $LMIN$ encontrado, utilizaremos o mesmo algoritmo de seleção do nó mais promissor para expansão, englobando todos os processos.

Após todos os processos registrarem o menor valor de $LMIN$ encontrado, cada um pode realizar a operação de $TRIMMING$ em sua lista $OPEN$ local. Isto corresponderá a realizar a operação sobre ambas as listas $OPEN$ globais.

Notamos que a única diferença em relação ao algoritmo seqüencial está na operação de $SCREENING$, pois o valor de $LMIN$ só é atualizado ao final da expansão de um nó. Mas isso não afeta o estado das listas no início da próxima iteração de expansão de nó, pois os nós que não sofreram $SCREENING$ na versão paralela e sofreriam na versão seqüencial, sofrerão $TRIMMING$ na versão paralela. O algoritmo de um processo $E_{d,i}$ segue os seguintes passos :

dicionário de dados

d :	direção de busca ;
d' :	direção oposta à atual direção de busca $= 3 - d$;
$c(m,n)$:	custo do nó m até o seu nó sucessor n ;
$g_i^*(n)$:	custo ótimo do nó s até o nó n se $i = 1$, ou do nó n até o nó β se $i = 2$;
$h_i^*(n)$:	custo ótimo do nó s até o nó n se $i = 2$, ou do nó n até o nó β se $i = 1$;
$g_i(n)$ e $h_i(n)$:	estimativas de $g_i^*(n)$ e $h_i^*(n)$, respectivamente;
$p_i(m)$:	pai do nó m na árvore i ;
$f_i(n)$:	$g_i(n) + h_i(n)$;

Processo do Algoritmo BS^* Paralelo de Lista Distribuída

1 insira nó inicial s (ou t) na sua lista $OPEN$.

2 $LMIN := \infty$.

3 selecione o nó $n_{d,i}$ mais promissor de sua lista $OPEN$.

4 participe do algoritmo para selecionar o nó N_d mais promissor da árvore d.

5 se $N_d = n_{d,i}$, então retire $n_{d,i}$ da sua lista $OPEN$.

6 insira N_d na sua lista $CLOSED$.

7 envie nó N_d para o processo $E_{d',i}$ e recebe nó $N_{d'}$ do processo $E_{d',i}$.

8 se N_d ou $N_{d'}$ for um nó vazio, então pule para passo 20.

9 se $N_{d'}$ estiver na sua lista $CLOSED$, então $F_p := TRUE$, senão $F_p := FALSE$.

10 envie F_p para processo $E_{d',i}$ e receba $F_{p',i}$ do processo $E_{d',i}$.

11 se $F_p = TRUE$, então retire da sua lista $OPEN$ todos os descendentes de $N_{d',i}$.

12 se $F_{p'} = TRUE$, então $\Omega_i =$ e pule para passo 15.

13 gere o conjunto $\Omega_{d,i}$ de todos os sucessores do nó N_d que pertençam ao subconjunto Ψ_i e nao pertençam à lista $CLOSED$.

14 para todos o nó $m \in \Omega_{d,i}$ faça :

14.1 $g := g_d(N_d) + c_d(N_d, m)$.

14.2 $f := g + h_d(m)$.

14.3 se $m \in OPEN$ e $g \geq g_d(m)$ ou $f > LMIN$, então retire m de $\Omega_{d,i}$ e pule para passo 14.

14.4 $g_d(m) := g.f_d(m) := f.p_d(m) := N_d$.

14.5 se $m \notin OPEN$, então insira m em $OPEN$.

15 envie $\Omega_{d,i}$ para o processo $E_{d',i}$ e receba $\Omega_{d',i}$ do processo $E_{d',i}$.

16 para todo o nó $m \in \Omega_{d',i}$ faça :

16.1 se $m \in CLOSED$ ou $m \in OPEN$ e $g_d(m)+g_{d'}(m) < LMIN$, então $LMIN := g_d(m) + g_{d'}(m)$ e $MEET := m$.

17 participe do algoritmo de seleção do menor $LMIN$ e atualize o valor de $MEET$.

18 retire de sua lista $OPEN$ todos os nós cujo valor da função f for maior ou igual a $LMIN$.

19 volta para o passo 3.

20 envie para o processo $E_{d',i}$ o caminho P_d entre $MEET$ e s apontado pelos ponteiros p_d e receba do processo $E_{d',i}$ o caminho $P_{d'}$.

21 se $LMIN \neq \infty$, então a solução é dada pela concatenação dos caminhos P_d e $P_{d'}$.

Capítulo IV

Implementação e Resultados

IV.1 OCCAM

OCCAM é uma linguagem de programação baseada na notação de CSP (Hoare) [8] e foi utilizada para escrever os algoritmos paralelos apresentados neste trabalho. É uma ferramenta projetada para descrever sistemas e algoritmos concorrentes, oferecendo a confiabilidade de uma linguagem de alto nível.

A unidade básica da linguagem OCCAM é o processo, que realiza uma série de ações e termina. Os processos primitivos são a atribuição e os comandos de entrada e saída.

Um processo pode ser constituído de vários processos mais simples, através de uma das seguintes construções :

a) seqüencial : os processos são executados um após o outro, de forma seqüencial;

b) condicional : os processos são guardados por expressões booleanas, que são avaliadas seqüencialmente, sendo executado apenas o processo cuja guarda for satisfeita primeiro;

c) seletiva : os processos são associados a um conjunto de valores e uma expressão é usada como seletor, o processo cujo um dos valores associados é igual ao valor da expressão seletora, é executado;

d) iterativa : essa construção executa repetidamente um processo, enquanto a expressão booleana associada a ele for avaliada como verdadeira;

e) paralela : os processos são executados de forma concorrente e o pro-

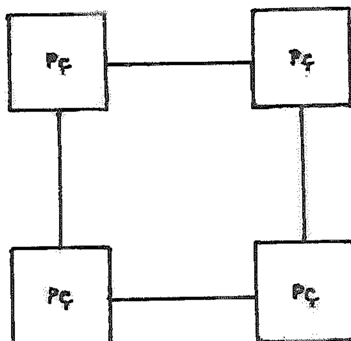


Figura IV.1: Configuração da rede de processadores do tipo Transputer

cesso formado por esta construção só termina quando todos os processo que o compõem terminam;

f) alternada : cada processo é guardado por um comando de entrada, que pode ou não estar combinado a uma expressão booleana, sendo executado aquele cuja guarda estiver pronta primeiro,; o processo só termina quando uma das guardas estiver pronta.

A construção paralela introduz a necessidade de fornecer um meio de comunicação entre os processos. Essa comunicação é feita através de canais de comunicação ponto a ponto, unidirecionais e de capacidade zero entre dois processos concorrentes.

IV.2 O Transputer

Os algoritmos foram executados em uma rede de processadores do tipo Transputer. O transputer é um componente VLSI programável, com canais de comunicação para conexão ponto a ponto com outros transputers. A sua arquitetura é totalmente definida utilizando como referência a linguagem OCCAM.

Cada transputer possue um processador, memória interna ao circuito integrado e um conjunto de canais padrão. Um transputer é capaz de implementar um processo OCCAM, inclusive concorrência interna através de 'time-sharing'.

IV.3 Aplicação Implementada

Foi escolhida uma aplicação, o planejador de rotas, que tivesse imediata identificação com o mundo real e fosse adequadamente tratada pelos algoritmos A^* e BS^*. O planejador de rotas serve como modelo imediato para problemas reais, como a movimentação robótica, e existe uma função heurística admissível que satisfaz à condição de monotonicidade.

A função da aplicação é encontrar o melhor caminho entre dois nós da rede. O custo associado a um arco entre dois nós do grafo é igual à distância em linha reta entre as coordenadas geográficas dos dos nós. A função heurística utilizada foi a distância em linha reta entre as coordenadas geográficas do nó e do nó objetivo.

Um grafo, representando uma rede de nós geograficamente dispersos, foi gerado da seguinte forma :

- foram geradas aleatoriamente as coordenadas geográficas (x_i, y_i) dos N nós g_i da rede;
- os nós da rede forma conectados em forma de árvore, sendo cada nó g_i conectado a um nó g_j, sendo $2 \leq i \leq N$ e j um número entre 1 e $(i-1)$, gerado aleatoriamente;
- em cima desta árvore, foram aleatoriamente adicionadas conexões, de tal maneira que todo nó g_i tem uma probabilidade P_c de estar conectado a um nó g_j, $j > i$.

A probabilidade P_c é aqui chamada de probabilidade de conexão.

Os algoritmos foram aplicados na solução do problema de roteamento sobre cinqüenta e quatro diferentes grafos de rede, os quais foram gerados através da variação da probabilidade de conexão P_c, da semente do gerador de números aleatórios e do número de nós da rede. Dessa maneira foi possível observar o comportamento dos algoritmos em grafos de busca com características, como o tamanho do caminho ótimo, diferentes. Sobre cada um do grafos de rede foram encontrados os caminhos ótimos entre todos os pares ordenados de nós distintos da rede.

Para cada tipo de algoritmo foi medido o tempo de execução de cada grafo.

IV.4 Distribuição de Processos

O algoritmo A^* de lista centralizada foi executado utilizando-se dois e quatro processadores, com as seguintes configurações :

- *dois processadores :* foram utilizados um processo mestre e dois processos escravos assim distribuídos :

  no processador T_0 foram executados o processo mestre e um processo escravo,

  o processador T_1 executou um processo escravo.

- *quatro processadores :* foram utilizados um processo mestre e quatro processos escravos assim distribuídos :

  no processador T_0 foram executados o processo mestre, um processo escravo e um processo roteador de mensagens,

  o processador T_1 executou um processo escravo e um processo de roteamento de mensagens,

  os processadores T_2 e T_3 executaram um processo escravo cada um.

O processo roteador de mensagens é necessário, pois na configuração da rede, apresentada na figura IV.1, não há um processador que se comunique diretamente com todos os outros.

Para algoritmo BS^{*} paralelo de lista centralizada, utilizamos as configurações a seguir :

- *dois processadores :* foram utilizados dois processos, um para cada direção :

  no processador T_0 foi executado o processo da árvore de direção d e

  o processador T_1 executou o processo da árvore de direção d'.

- *quatro processadores :* foram utilizados um processo mestre e dois processos escravos para cada direção, assim distribuídos :

no processador T_0 foram executados o processo mestre M_d e o processo escravo $E_{d,1}$,

o processador T_1 executou o processo mestre $M_{d'}$ e o processo escravo $E_{d',1}$,

o processador T_2 executou o processo escravo $E_{d,2}$ e

o processador T_3 executou o processo escravo $E_{d',2}$.

A versão paralela dos algoritmos A^* e BS^* de lista distribuída também foi implementada em dois e quatro processadores. Nesses casos, cada processador executou apenas um processo, pois nesta versão são todos iguais.

IV.5 Resultados

O parâmetro utilizado para a avaliação dos algoritmos, aqui denominado de aceleração, consiste na razão entre o tempo de processamento do algoritmo A^* seqüencial e o tempo de processamento do algoritmo sendo avaliado.

Para cada algoritmo foram agrupados os tempos de execução obtidos na solução de grafos de rede com o mesmo número de nós e probabilidade de conexão. Foram calculados os tempos de execução médios de cada grupamento, os quais foram utilizados no cálculo dos valores de aceleração. Para identificar cada um dos algoritmos paralelos desenvolvidos neste trabalho, foram associados os seguintes mnemônicos :

APLC	algoritmo A^* paralelo de lista centralizada
APLD	algoritmo A^* paralelo de lista distribuída
BSPLC	algoritmo BS^* paralelo de lista centralizada
BSPLD	algoritmo BS^* paralelo de lista distribuída

A figura IV.2 apresenta a aceleração do algoritmo BS^* seqüencial para cada probabilidade de conexão P_c. Observamos que o seu desempenho é um pouco inferior ao do A^* e que a aceleração decresce com o aumento da probabilidade de conexão. Isto deve-se ao fato de que , quanto maior o valor de P_c, maior o número de caminhos que devem ter tamanhos comparáveis.

Os gráficos de aceleração em função do número de nós do grafo, quando utilizados dois processadores, são apresentados nas figuras IV.3, IV.4 e IV.5. Nestes gráficos não é apresentada a curva do algoritmo BS^* paralelo de lista centralizada, pois não há sentido

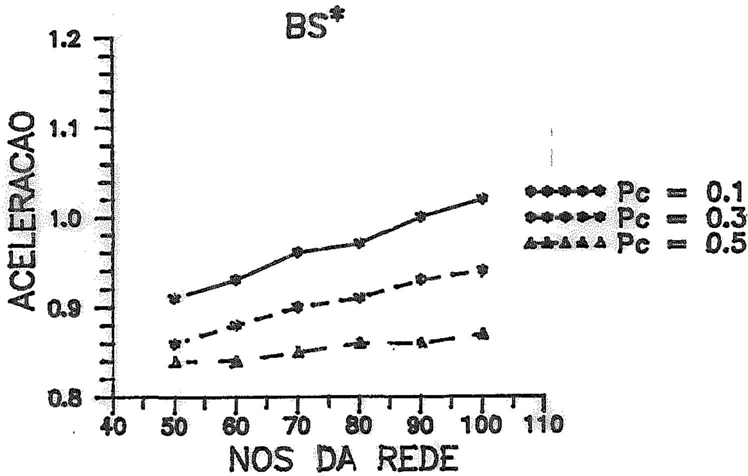


Figura IV.2: Acelerações do Algoritmo BS^* seqüencial

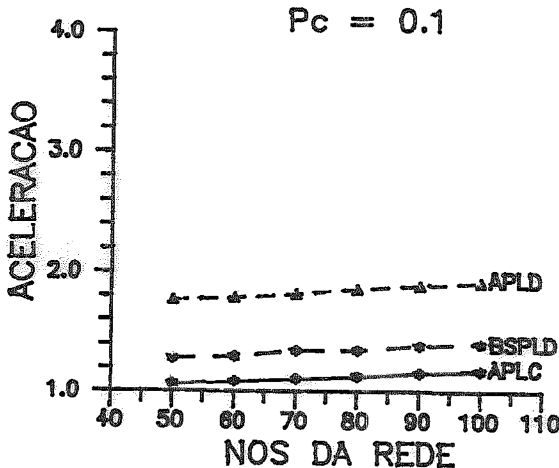


Figura IV.3: Acelerações dos Algoritmos Paralelos, utilizando 2 processadores, para grafos com probabilidade de conexão igual a 0.1

em dividir a tarefa de expansão de nós por mais de um processo em um único processador.

Os gráficos das figuras IV.6, IV.7 e IV.8 correspondem ao resultados obtidos quando utilizados quatro processadores. Cada um dos gráficos corresponde a uma probabilidade de conexão entre os nós do grafo.

Todos os resultados nos mostram que os algoritmos da classe de lista distribuída apresentam um melhor desempenho do que os da classe de lista centralizada. Este resultado era esperado, já que a lista distribuída evita o gargalo de comunicação ("bottleneck") com o processo mestre. Notamos também que a relação entre os algoritmos unidirecionais, baseados no A^*, e os bidirecionais, baseados no BS^*, se invertem para cada classe. En-

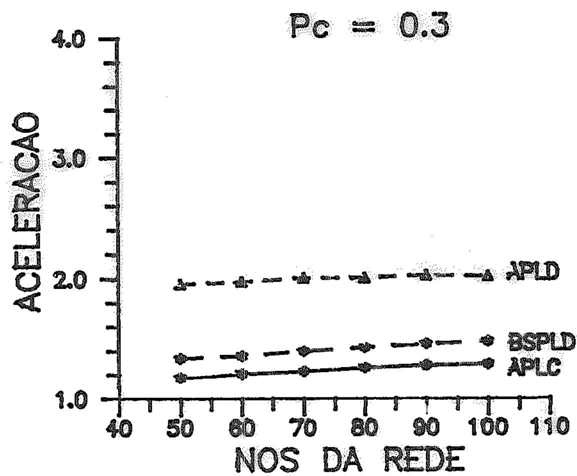


Figura IV.4: Acelerações dos Algoritmos Paralelos, utilizando 2 processadores, para grafos com probabilidade de conexão igual a 0.3

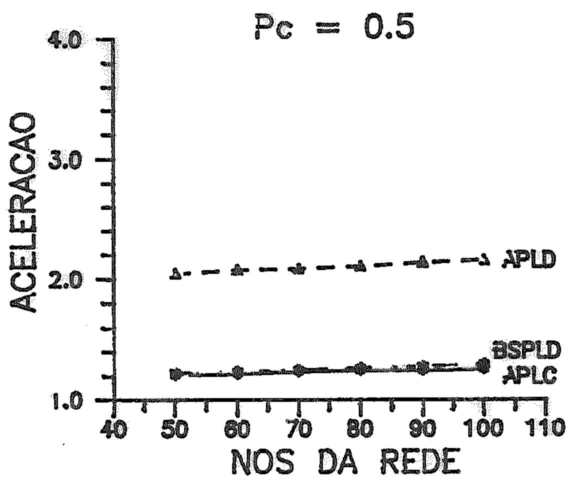


Figura IV.5: Acelerações dos Algoritmos Paralelos, utilizando 2 processadores, para grafos com probabilidade de conexão igual a 0.5

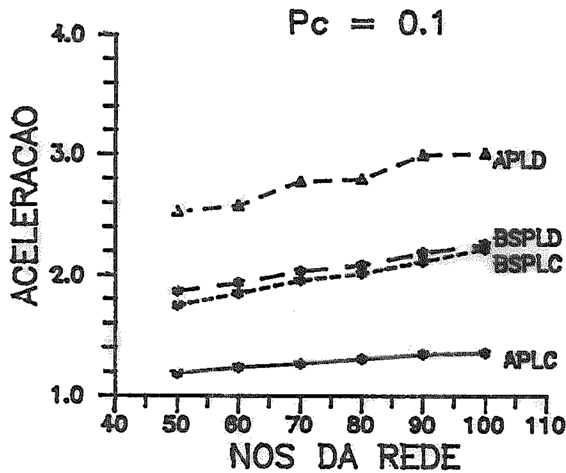


Figura IV.6: Acelerações dos Algoritmos Paralelos, utilizando 4 processadores, para grafos com probabilidade de conexão igual a 0.1

quanto na lista centralizada os algoritmos bidirecionais apresentam melhores resultados, na classe de lista distribuída os unidirecionais mostram-se superiores.

Observamos que, no intervalo pesquisado, os algoritmos possuem uma tendência de pequena variação linear da aceleração, em relação ao número de nós do grafo. Sendo que, em alguns casos, o algoritmo unidirecional de lista distribuída ultrapassa o valor máximo esperado, que é igual ao número de processadores utilizados. Isto se deve ao fato das listas manipuladas por cada processador serem menores, diminuindo os tempos das operações de pesquisa do melhor nó de uma lista.

Os valores obtidos para os algoritmos unidirecionais também apresentam uma variação positiva em função da probabilidade de conexão, sendo que a taxa de variação decresce com a elevação do valor da probabilidade. Tal comportamento não se verifica nos algoritmos bidirecionais. Estes últimos apresentam uma variação positiva com o aumento da probabilidade de 0.1 para 0.3, que se torna negativa ao passarmos de 0.3 para 0.5.

Notamos também que a diferença entre os algoritmos unidirecionais e bidirecionais centralizados tende a aumentar e os da classe de lista distribuída a diminuir, com o aumento da probabilidade de conexão. Isto deve-se ao fato de que o número de caminhos de tamanho comparável aumenta, diminuindo o desempenho dos algoritmos baseados no BS^*.

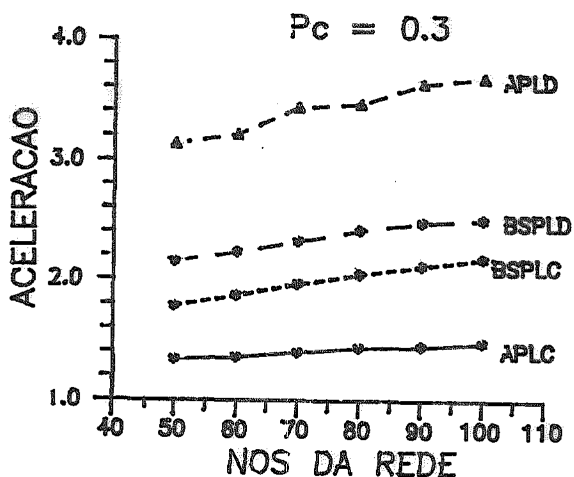


Figura IV.7: Acelerações dos Algoritmos Paralelos, utilizando 4 processadores, para grafos com probabilidade de conexão igual a 0.3

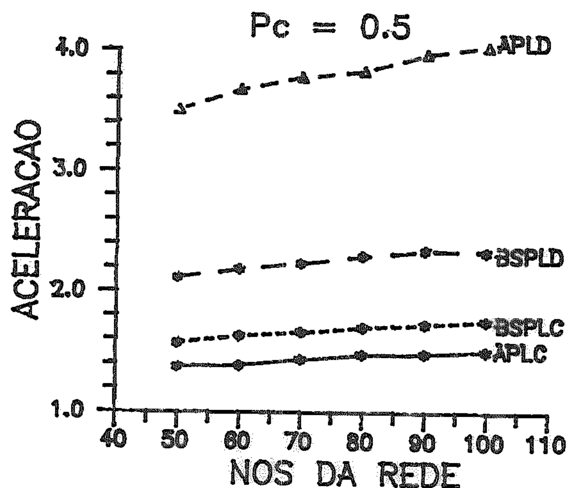


Figura IV.8: Acelerações dos Algoritmos Paralelos, utilizando 4 processadores, para grafos com probabilidade de conexão igual a 0.5

IV.6 Conclusões

A aplicação escolhida, mesmo sendo um problema válido, apresenta algumas características que certamente influem nos resultados obtidos. Em primeiro lugar, a probabilidade de conexão entre nós utilizada foi constante para todos os nós, o que pode não corresponder aos casos reais. Outro fator, decorrente do primeiro, é que o espaço do problema foi dividido em conjuntos Ψ_i, de maneira que todos os nós possuem praticamente o mesmo número de sucessores em cada um desses conjuntos. Essa situação pode não ser facilmente obtida em problemas reais.

Os algoritmos da classe de lista distrbuída apresentaram os melhores resultados, de onde concluímos que o problema de gargalos de comunicação realmente restringe o desempenho e que foi contornado.

O problema de convergência das frentes de busca, para os algoritmos bidirecionais foi satisfatoriamente resolvido pelo método utilizado de sincronização ao final da expansão de um nó. Isso foi comprovado pelos fatos de que para todas as corridas realizadas, os número de nós expandidos por cada árvore ser praticamente idêntico e do tamanho da parte do caminho solução encontrado por cada árvore ser também muito próximo.

Os métodos de busca utilizados mostraram-se muito adequados para uma adaptação para arquiteturas paralelas, apresentando um considerável aumento no desempenho. Isto torna-se mais evidente com o aumento do tamanho do problema.

O algoritmo A^* mostrou-se ainda o melhor dos métodos de busca admissíveis, tanto na versão seqüencial quanto na versão paralela, desde o momento que conseguimos um meio de eliminar os gargalos de comunicação.

Referências Bibliográficas

[1] KWA, J.B.H, 'BS*: An Admissible Bidirectional Staged Heuristic Search Algorithm', *Artificial Intelligence*,no. 38,pp 95-109, 1989.

[2] ABDELRAHAM, T.S, e MUDGE, T.N., 'Parallel Branch and Bound Algorithms on Hypercube Multiprocessors', *Proc. of the Third Conference On Hypercube Concurrent Computers and Applications*,vol. 3,pp 1492-1499, California, USA, 1988.

[3] FELTEN, E.W., 'Best-First Branch and Bound on a Hypercube', *Proc. of the Third Conference On Hypercube Concurrent Computers and Applications*,vol. 3,pp 1500-1504, California, USA, 1988.

[4] MA, R.P., TSUNG, F-S. e MA, M-H., 'A Dynamic Load Balancer For A Parallel Branch and Bound Algorithm', *Proc. of the Third Conference On Hypercube Concurrent Computers and Applications*,vol. 3,pp 1505-1513, California, USA, 1988.

[5] PARGAS, R.P. e WOOSTER, D.E., 'Branch and Bound Algorithms on a Hypercube', *Proc. of the Third Conference On Hypercube Concurrent Computers and Applications*,vol. 3,pp 1514-1519, California, USA, 1988.

[6] PEARL, J., *HEURISTICS: Intelligent Seach Strategies for Computer Problem Solving*, Addison-Wesley, MA, USA, 1984.

[7] NILSSON, N.J., *Principles of Artificial Intelligence*, Tioga, CA, USA, 1980.

[8] HOARE, C.A.R., 'Communicating Sequential Processes', *Comm. ACM*,vol 21,no. 8,pp 666-677, 1978.