Universidade Federal de Santa Catarina
Curso de Pós-Graduação em Matemática e Computação Científica

Algoritmos de Busca de Caminhos em Grafos aplicados aos problemas: Alinhamento de Proteínas e Quebra-cabeça de 15 peças

Diane Rizzotto Rossetto

Orientador: Prof. Dr. Clóvis Caesar Gonzaga

Florianópolis
2007

Universidade Federal de Santa Catarina
Curso de Pós-Graduação em Matemática e Computação Científica

Algoritmos de Busca de Caminhos em Grafos aplicados aos problemas: Alinhamento de Proteínas e Quebra-cabeça de 15 peças

Dissertação apresentada ao Curso de Pós Graduação em Matemática e Computação Científica, do Centro de Ciências Físicas e Matemáticas da Universidade Federal de Santa Catarina, para obtenção do grau de Mestre em Matemática, com Área de Concentração em Matemática Aplicada.

Diane Rizzotto Rossetto

Florianópolis
2007

Algoritmos de Busca de Caminhos em Grafos aplicados aos problemas: Alinhamento de Proteínas e Quebra-cabeça de 15 peças

por

Diane Rizzotto Rossetto

Esta Dissertação foi julgada adequada para obtenção do Título de "Mestre em Matemática", Área de Concentração em Matemática Aplicada, e aprovada em sua forma final pelo Curso de Pós-Graduação em Matemática e Computação Científica.

Prof. Dr. Clóvis Caesar Gonzaga
(Coordenador)

Comissão Examinadora

Prof. Dr. Clóvis Caesar Gonzaga
(MTM - UFSC, Orientador)

Prof. Dr. Paulo José da Silva e Silva
(IME - USP)

Prof. Dr. Jáuber Cavalcante de Oliveira
(MTM - UFSC)

Prof. Dr. Juliano de Bem Francisco
(MTM - UFSC)

Fevereiro de 2007

Aos meus pais, Caetano e Terezinha...
...“o amor não falha e a vida não falhará enquanto houver amor”.

Agradecimentos

Durante estes três anos em Florianópolis aprendi muitas coisas, tive muitas alegrias e também momentos tristes, mas nunca estava sozinha, sempre havia perto de mim alguém para me fazer sorrir. Por isso gostaria muito de agradecê-los...

Aos meus pais Caetano e Terezinha por serem "o porto seguro" em minha vida, sempre me apoiando, me guiando, me acolhendo e principalmente acreditando em mim. A vocês papai e mamãe queridos todo o meu amor.

Ao meu irmão Diego com quem compartilhei os momentos mais incríveis da minha infância. A minha irmã Daísa, o melhor presente que a vida me deu, amorosa e meiga, sempre lembrando-me que alguém no mundo precisava muito de mim. A vocês irmãos queridos peço a Deus que sempre os mantenham perto de mim.

Ao meu noivo Patrick, a pessoa que esteve presente em todos os momentos desta jornada, entendendo as horas ausentes, me incentivando e com muito amor e carinho sempre me deixando feliz.

Aos meus avôs Tranquilo e José, que já partiram, mas deixaram exemplos dos valores que eu devia seguir. As minhas avós Ancila e Tecla, que ao longo de seus noventa anos ainda são os maiores exemplos de força e garra.

Gostaria de agradecer aos demais familiares aos quais se uniram a mim por um sentimento muito mais nobre que é a amizade.

Aos amigos que deixei em Nonoai, aos que fiz durante a faculdade em Chapecó e aos que aqui conquistei, obrigada pelo ombro amigo e por compartilharem comigo meus instantes.

Aos colegas da pós-graduação, à vocês que sabem tanto quanto eu como não é fácil chegar até aqui, obrigada por também me ajudarem a crescer nas nossas longas discussões sobre listas e teoremas, por não me deixar sozinha nos momentos de insegurança e acima de tudo pelas tantas vezes que nossos bate-papos foram essenciais

para prosseguir. A vocês amigos parabéns por estarmos todos aqui.

Um agradecimento especial ao colega otimizador Rafael Casali que me ensinou muito sobre Matlab, Tex e Linux, e pelas discussões bem humoradas sobre grafos, joguinhos e proteínas.

Aos professores da UNOCHAPECÓ que sempre me incentivaram a continuar os estudos. Aos professores da UFSC que conheci durante o nivelamento e o mestrado minha gratidão por tudo aquilo que possibilitaram que eu aprendece, agradecimento especial àqueles que além disto se fizeram amigos compartilhando suas experiências.

A CAPES (Centro de Apoio a Pesquisa) pelo apoio financeiro.

A secretária da pós-graduação, Elisa Amaral pela dedicação com que nos serve.

Aos professores que participaram da banca agradeço por aceitarem corrigir este trabalho.

Finalmente ao professor mais apaixonante que eu já encontrei, Clóvis Caesar Gonzaga, pela maneira envolvente que conduz seus alunos e suas aulas, minha admiração pela maneira responsável que conduz sua vida de cientista, também pelo amigo e conselheiro que torna-se ao nos orientar.

Sumário

Lista de Figuras

Resumo

Neste trabalho foram estudados alguns conceitos fundamentais da Teoria de Grafos para o entendimento dos algoritmos de Busca de Caminhos em Grafos. Estudamos os algoritmos de Busca Horizontal, Busca em Profundidade, Dijkstra e A^*. Na aplicação do problema do alinhamento de proteínas usamos o algoritmo A^* para construir um emparelhamento entre os átomos de carbono de duas proteínas. Nosso objetivo era tornar este algoritmo mais rápido que o algoritmo de Programação Dinâmica, que é na literatura o algoritmo padrão para obter tal emparelhamento. Os algoritmos de Dijkstra e A^* foram aplicados ao problema do quebra-cabeça de 15 peças. Notamos que devido ao espaço de busca ser muito grande, o algoritmo A^* teve desempenho muito melhor que o algoritmo de Dijkstra, isto porque A^* faz uso de uma informação heurística. Mesmo assim a estimativa conhecida não é suficiente para resolver este problema. Apresentamos uma nova estimativa com que obtivemos resultados muito melhores mas que ainda não faz com que o algoritmo A^* resolva todas as instâncias do problema usando nossos recursos computacionais.

Abstract

In this work we study some fundamental concepts in Graph Theory needed to understand graph search algorithms. We studied breadth-first, depth-first, Dijkstra and A* algorithms. In the application to protein alignment we used the A* algorithm to construct a matching between the carbon atoms of two given proteins. Our objective was to obtain an algorithm faster than Dynamic Programming, which appears in the literature as the standard method for this matching. Dijkstra's algorithm and A* were applied to solving the fifteen puzzle. We noted that A* had a much better performance than Dijkstra, due to the use of the heuristic information. Even using this information, A* was not capable of solving the problem, and then we developed a process for obtaining better estimated costs. These estimates greatly improved the results, but were still unable to solve all instances of the problem with our computational resources.

Introdução

Esta dissertação iniciou a partir do interesse despertado pela teoria de otimização em grafos durante o curso de Pesquisa Operacional ministrado pelo professor Clóvis Gonzaga durante o segundo semestre de 2005.

Durante este curso tive meu primeiro contato com a teoria necessária para o estudo dos algoritmos de busca de caminhos em grafos, na oportunidade estudei os algoritmos de Busca Horizontal, Busca em Profundidade e Dijkstra. Neste curso também tive a oportunidade de ver a correspondência entre o algoritmo de Busca Horizontal e o algoritmo de Programação Dinâmica Progressiva para problemas discretos, na ocasião o exemplo tratava-se da decisão da quantidade de água a ser utilizada na geração de energia em uma usina hidrelétrica.

Assim como no exemplo acima a modelagem de grafos a problemas reais é muito interessante, permitindo adaptar problemas complexos a representações simples e gerais.

Entretanto em muitos problemas o grafo que o modela pode tornar-se muito grande e consequentemente o espaço de busca é muito vasto. Desta maneira, os algoritmos citados acima podem não ser suficientes para resolver determinados problemas. Um exemplo é o problema do quebra-cabeça de 15 peças que foi abordado por nós no quarto capítulo.

Com a informação que o algoritmo A^* com heurísticas convenientes encontra um caminho ótimo com um número menor de expansões que qualquer outro algoritmo de rotulação (Busca Horizontal, Busca em Profundidade e Dijkstra) iniciamos em 2006 o estudo deste algoritmo.

O desenvolvimento deste algoritmo, originado no contexto de Inteligência Artificial, começou com o seguinte problema: em uma sala encontra-se um robô, deseja-se que ele atravesse a sala percorrendo o menor caminho possível sem bater nos móveis. Assim, os cantos dos móveis bem como a entrada e a saída da sala são interpretados

como os nós do grafo, as retas sobre as quais o robô pode se movimentar entre os nós são os ramos do grafo. Para encontrar o menor caminho que ele deve percorrer para chegar até a saída da sala, em cada nó estima-se quanto ele ainda precisa movimentar-se para chegar ao destino se prosseguir num caminho através deste nó, obviamente o caminho com menor estimativa é o escolhido para ele continuar. Neste problema a estimativa usada foi a distância em linha reta entre o nó terminal do caminho e o nó que representa a saída da sala.

Neste trabalho aplicamos as técnicas de busca de caminhos de custos ótimos em um grafo a dois problemas bem distintos.

O primeiro, na área de Biologia Estrutural, é um sub-problema do problema de alinhamento de proteínas. Estudam-se métodos de alinhamento para medir o grau de semelhança entre duas proteínas, através de uma função de pontuação. Em nosso estudo buscamos o emparelhamento entre duas proteínas que obtem a maior pontuação possível, sendo assim buscamos o maior caminho em um grafo que represente o problema. Em geral este emparelhamento é feito utilizando Programação Dinâmica, desta forma a modelagem feita por nós é nova.

O segundo, de Inteligência Artificial, busca determinar o número mínimo de movimentos necessários para resolver o quebra-cabeça de 15 peças a partir de uma configuração dada arbitrária. A estimativa conhecida, baseada na distância de Manhattan, para este problema não é suficiente para resolvê-lo, utilizando um algoritmo bem geral, pois em muitos casos o erro em relação ao custo ótimo é grande. Desta forma no capítulo quatro propomos uma nova estimativa em que os resultados obtidos foram melhores mas ainda não suficientes para resolver este problema usando o algoritmo A^*.

Porém existem trabalhos que apresentam soluções para este problema usando outras técnicas. Em [8] Korf e Felner descrevem como encontraram soluções ótimas para o quebra-cabeça usando o algoritmo IDA^*, que é uma variação do algoritmo A^*. Em [5] Culberson e Schaeffer resolver este problema com algoritmos feitos explorando propriedades específicas do quebra-cabeça de 15 peças.

Capítulo 1

Introdução a Teoria de Grafos

Iniciamos com alguns conceitos e resultados que entendemos ser suficientes para o estudo de otimização em grafos, em especial ao problema de busca de caminhos em grafos que é o nosso objeto de estudo. A nomenclatura e as notações que padronizaremos agora, assim com as demonstrações dos teoremas seguem de [12] e [13].

A teoria de grafos é uma das ferramentas matemáticas mais simples e poderosas usadas para construção de modelos e resolução de problemas, tais como: fluxos em redes, escolha de uma rota ótima, tática e logística.

A mais antiga citação sobre a teoria ocorreu no ano de 1736, através do matemático suiço Leonhard Euler em seu artigo "The Seven Bridges of Königsberg".

Em Könisberg, Alemanha, havia uma ilha, um rio que passava pela cidade e que logo após passar por esta linha se bifurcava. Sobre este rio havia sete pontes, a pergunta era: é possível passar pelas sete pontes numa caminhada contínua sem passar duas vezes por qualquer uma das pontes?

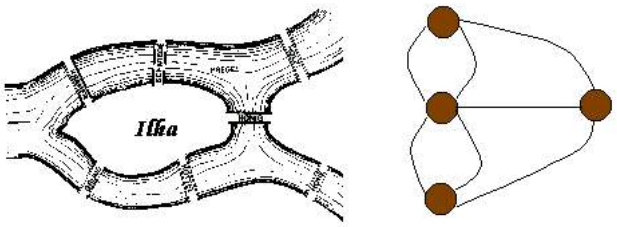


Figura 1.1: Pontes de Konisberg

Euler analisou o problema trocando as áreas de terra por pontos (nós) e as pontes por arcos (ramos), e modelando desta forma provou que o problema não tinha

solução, uma análise de como Euler chegou a esta conclusão pode ser vista em [15]. Teve início assim o que hoje conhecemos como *Teoria de Grafos*.

Porém o desenvolvimento desta teoria veio se dar somente na segunda metade do século XX, sob o impulso das aplicações de otimização organizacional. Tal desenvolvimento se deu com a invenção do computador, sem o qual a maioria das aplicações em grafos seria impossível.

1.1 Principais conceitos

Definição 1.1. *Um grafo é caracterizado por um par* $G = (\mathcal{N}, \mathcal{M})$ *em que* $\mathcal{N}$ *é um conjunto enumerável e* $\mathcal{M}$ *é um conjunto de pares ordenados de elementos de* $\mathcal{N}$*,* $\mathcal{M} \subset \mathcal{N} \times \mathcal{N}$*,*

$\mathcal{N}$ *é um conjunto* $(n_1, n_2, n_3, ...)$ *não vazio de nós.*

$\mathcal{M}$ *é o conjunto dos ramos* $r = (n_i, n_j)$ *em que* n_i *é a extremidade inicial e* n_j *a extremidade final, com* $r \in \mathcal{N} \times \mathcal{N}$*.*

G é um grafo *finito* se $\mathcal{N}$ e $\mathcal{M}$ forem finitos.

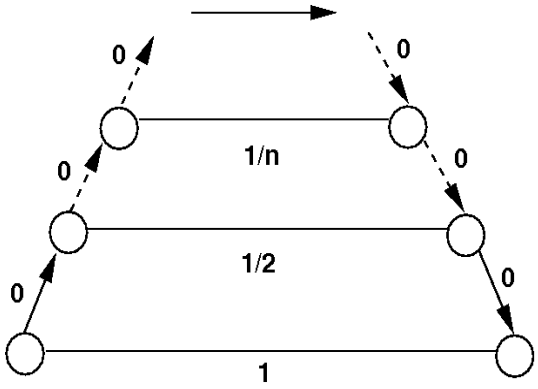


Figura 1.2: Exemplo de grafo infinito

G é um grafo *simples* se não existirem ramos diferentes associados aos mesmos nós, assim dizemos que ele não possui ramos múltiplos. Neste trabalho só consideramos grafos simples pois em problemas de busca de caminhos, ramos múltiplos podem sempre ser eliminados.

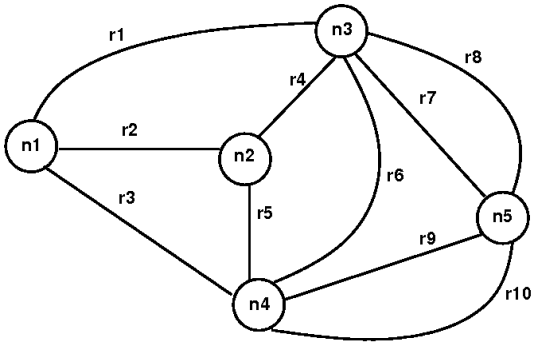


Figura 1.3: Grafo com ramos múltiplos

Definição 1.2. *O nó n é adjacente ao ramo r se n é uma extremidade de r.*

Observação 1.3. *Dizemos que um nó n_i é a cauda do ramo r_i se ele é o nó inicial do ramo. Se n_j é a extremidade final do ramo r_i dizemos que ele é a cabeça do ramo.*

Definição 1.4. *$r \in \mathcal{M}$ é incidente a $n \in \mathcal{N}$ se e somente se n é cabeça ou cauda de r.*

Definição 1.5. *Dados $n, \bar{n} \in \mathcal{N}$, dizemos que $\bar{n}$ é sucessor de n se e somente se $(n, \bar{n}) \in \mathcal{M}$.*

Dizemos que $\mathcal{P}$ é o conjunto das partes dos elementos de $\mathcal{N}$.

Definição 1.6. *Operador Sucessor é uma aplicação $\Gamma : \mathcal{N} \to \mathcal{P}(\mathcal{N})$ em que $\bar{n} \in \Gamma(n)$ se e somente se $\bar{n}$ é sucessor de n.*

Γ é uma regra para a construção dos sucessores. Um grafo simples pode ser caracterizado por seu operador sucessor, assim uma nova designação do grafo G é dada por $G = (\mathcal{N}, \Gamma)$, desprezando a descrição dos ramos.

Definição 1.7. *$G' = (\mathcal{N}, \mathcal{M}')$ é um grafo parcial de $G(\mathcal{N}, \mathcal{M})$ se e somente se $G' = (\mathcal{N}, \mathcal{M}')$ com $\mathcal{M}' \subset \mathcal{M}$.*

Definição 1.8. *Um grafo $G' = (\mathcal{N}', \mathcal{M}')$ é subgrafo de $G = (\mathcal{N}, \mathcal{M})$ se e somente se $\mathcal{N}' \subset \mathcal{N}$ e $\mathcal{M}' = \{(n_i, n_j)/n_i, n_j \in \mathcal{N}' \text{ e } (n_i, n_j) \in \mathcal{M}\}$.*

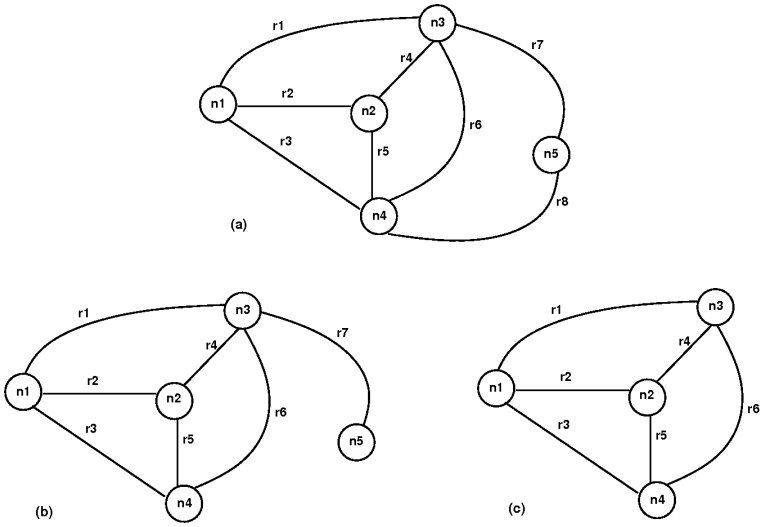


Figura 1.4: (a) Grafo, (b) Grafo parcial, (c) Subgrafo

Por nossa definição os grafos são sempre orientados, já que os ramos são pares ordenados. Para definir grafo não orientado, basta não utilizar a orientação, ou definir os nós como pares não orientados $\{n_i, n_j\}$, ou ainda exigir que sempre estejam presentes os ramos (n_i, n_j) e (n_j, n_i).

1.2 Conceitos em grafos não orientados

Definição 1.9. *Uma sequência $C = (n_1, r_1, n_2, r_2, ..., n_p, r_p, n_{p+1})$ é uma cadeia (ligando n_1 a n_{p+1}) se existir uma sequência de nós $(n = n_1, n_2, ..., n_{p+1})$, tais que r_i é adjacente a n_i e n_{i+1}, $i = 1, ..., p$.*

Dizemos que os nós n e $\bar{n}$ estão ligados se existe uma cadeia entre eles.

Definição 1.10. *Um grafo $G = (\mathcal{N}, \mathcal{M})$ é conexo se quaisquer dois nós são ligados por alguma cadeia, caso contrário o grafo é dito desconexo.*

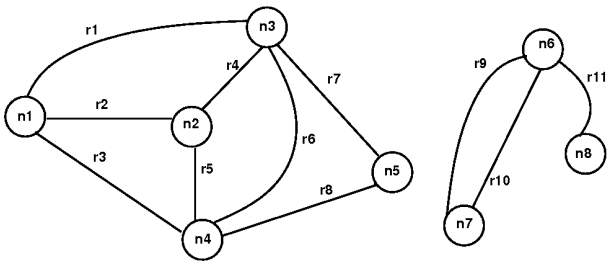


Figura 1.5: Grafo desconexo

Definição 1.11. *$G' = (\mathcal{N}', \mathcal{M}')$ é uma componente conexa de $G = (\mathcal{N}, \mathcal{M})$ se e somente se G' é um subgrafo conexo de G e nenhum nó de G' é ligado a nenhum nó em $\mathcal{N} - \mathcal{N}'$.*

Um *subgrafo conexo maximal* de G é uma componente conexa de G.

Definição 1.12. *Um ciclo é uma cadeia de um nó n a ele mesmo.*

Definição 1.13. *Uma árvore é um grafo conexo sem ciclos.*

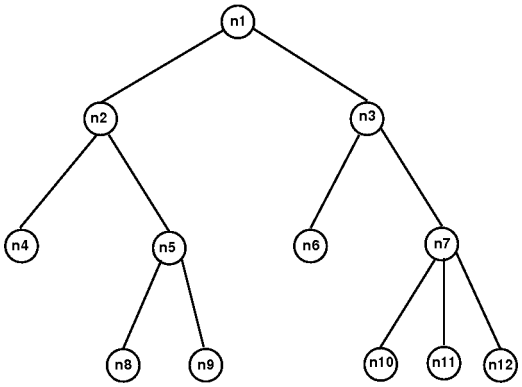


Figura 1.6: Árvore

Definição 1.14. *Uma folha é um nó com no máximo um ramo incidente.*

Na figura 1.6 os nós $n_4, n_6, n_8, n_9, n_{10}, n_{11}$ e n_{12} são folhas.

Lema 1.15. *Toda árvore tem folhas.*

Teorema 1.16. *Uma árvore com n nós tem exatamente $n - 1$ ramos.*

Demonstração:
Seja $G = (\mathcal{N}, \mathcal{M})$ uma árvore com n nós e m ramos. Como G tem uma folha $\bar{n}$, $G' = (\mathcal{N} - \{\bar{n}\}, \mathcal{M} - \{\bar{r}\})$, com $\bar{r}$ ramo ligado a $\bar{n}$, é uma árvore com $n-1$ nós e $m-1$ ramos.

Por recorrência, chega-se a um grafo com apenas um nó e nenhum ramo após $n - 1$ passos.

□

Teorema 1.17. *Seja $G = (\mathcal{N}, \mathcal{M})$ uma árvore. Então qualquer par de nós $n, \bar{n}$ é ligado por uma única cadeia.*

Demonstração:
Suponha que existe um par de nós $n, \bar{n}$ que está ligado por duas cadeias distintas.

$$n = n_1, r_1, n_2, r_2, ..., n_p = \bar{n},\ n = n_1^{'}, r_1^{'}, n_2^{'}, r_2^{'}, ..., n_p^{'} = \bar{n}$$

Se $r_1 = r_1^{'}$, façamos $n = n_2$ e reindexamos as cadeias. Repetindo o processo chegamos a situação em que $r_1 \neq r_1^{'}$.
Se $n_2 = n_2^{'}$ chegamos a um ciclo $n, r_1, n_2, n_2^{'}, r_1^{'}, n$.
Senão buscamos o primeiro nó comum as cadeias, sabemos que existe pois $n_p = n_p^{'}$, e assim chegamos a um ciclo.

Mas isto contradiz a definição de árvore.

□

1.3 Conceitos em grafos orientados

Definição 1.18. *Um caminho de $n \in \mathcal{N}$ a $\bar{n} \in \mathcal{N}$ é uma sequência $(n = n_1, n_2, ..., n_p = \bar{n})$ de nós tais que $(n_i, n_{i+1}) \in \mathcal{M}$, $i = 1, ..., p-1$.*

Definição 1.19. *$\bar{n} \in \mathcal{N}$ é acessível a partir de n se existir um caminho de n a $\bar{n}$.*

Definição 1.20. *Circuito é um caminho de um nó $n \in \mathcal{N}$ a ele mesmo.*

Definição 1.21. *Um nó $n \in \mathcal{N}$ é raiz do grafo se e somente se todos os nós são acessíveis a partir de n.*

Definição 1.22. *Uma arborescência é uma árvore com raiz.*

Uma arborescência possui somente uma raiz e cada nó é acessível a partir da raiz por um único caminho.

Dado um grafo $G = (\mathcal{N}, \mathcal{M})$, uma árvore (arborescência) maximal de G é um grafo parcial de G que é árvore (arborescência).

Dada uma aplicação $c : \mathcal{M} \to \mathbb{R}$, chamada de custo associado ao ramo, podemos definir o que é o custo de um caminho.

Definição 1.23. *Custo de um caminho $P = (n_1, n_2, ..., n_p)$ é dado por*

$$C(P) = \sum_{i=1}^{p-1} c(n_i, n_j),$$

isto é, é a soma dos custos de todos os ramos do caminho.

Observação 1.24. *Definimos caminho por uma sequência de nós $P = (n_1, n_2, ..., n_p)$. Poderíamos também caracterizar o caminho pela sequência de ramos $P = (r_1, r_2, ..., r_{p-1})$, com $r_i = (n_i, n_{i+1})$, $i = 1, 2, ..., p-1$. Utiliza-se esta caracterização quando for mais conveniente.*

Capítulo 2

Algoritmos de Busca de Caminhos

Após definir conceitos básicos da teoria de grafos, definiremos neste capítulo o nosso problema e em seguida os algoritmos de otimização em grafos que são usados para resolvê-lo.

Na teoria de grafos, o problema do *Caminho de Custo Mínimo* consiste na minimização do custo de travessia de um grafo entre dois nós.

Dizemos que um caminho P tem custo mínimo se $C(P) \leq C(P')$ para todo caminho P' que tenha a mesma origem e o mesmo destino que P.

Para definir nosso problema, através da caracterização do grafo pelo operador sucessor, considere:

- um grafo $G = (\mathcal{N}, \Gamma)$;
- um nó $s \in \mathcal{N}$, chamado *nó inicial ou raiz do grafo*;
- um subconjunto $T \subset \mathcal{N}$ chamado *alvo*.

Problema

Encontrar, se existir, um caminho de s a algum nó do alvo com custo mínimo entre todos os caminhos de s ao alvo.

Sempre que este caminho exista ele é dito ***caminho ótimo***.

Este problema sempre admite solução se o grafo for finito, sem circuitos de custo negativo, e se algum nó do alvo for acessível a partir de s. De agora em diante consideramos apenas grafos finitos.

2.1 Princípio de Otimalidade de Bellman para Grafos

Teorema 2.1. *Suponha que $s = n_1, n_2, ..., n_p = t$ é um caminho de custo mínimo de s ao alvo. Então $n_k, n_{k+1}, ..., n_{k+j}$, com $1 \leq k$, $k + j \leq p$ é um caminho de custo mínimo de n_k a n_{k+j}.*

Demonstração:
Suponha que $s = n_1, n_2, ..., n_p = t$ é um caminho de custo mínimo.
Por contradição, suponha que o caminho $n_k = n_1^{'}, n_2^{'}, ..., n_r^{'} = n_{k+j}$ satisfaz

$$C(n_1^{'}, n_2^{'}, ..., n_r^{'}) < C(n_k, n_{k+1}, ..., n_{k+j}),$$

ou seja, existe outro caminho de n_k a n_{k+j} com custo menor.
Temos que $n_1, n_2, ..., n_k = n_1^{'}, n_2^{'}, ..., n_r^{'} = n_{k+j}, ..., t$ é um caminho de s a t com custo

$$C(s, ..., t) - C(n_k, n_{k+1}, ..., n_{k+j}) + C(n_1^{'}, n_2^{'}, ..., n_r^{'}) < C(s, ..., t).$$

A desigualdade acima contradiz a hipótese que o caminho $s = n_1, n_2, ..., n_p = t$ tem custo mínimo.

□

2.2 Algoritmos de Rotulação

Definição 2.2. *Dados dois nós n, $\bar{n}$, define-se (usando a convenção $inf\ \emptyset = +\infty$):*

- $h(n, \bar{n}) = inf\{C(P)|\ P = (n = n_1, n_2, ..., n_p = \bar{n})\}$;
- $c^*(n) = h(s, n)$;
- $h(n) = inf\{h(n, t)|\ t\ \in\ T\}$.

Temos:

- $h(n, \bar{n})$ é o custo de um caminho de custo mínimo de n a $\bar{n}$, que chamaremos de "distância de n a $\bar{n}$";
- $c^*(n)$ é a distância do nó inicial a n;
- $h(n)$ é a distância de n ao alvo T;
- $h(s)$ é o custo de um caminho ótimo.

Utilizando a notação introduzida por Gonzaga em [17] representamos os caminhos por triplas ordenadas:

$$\eta = (n, c, p),$$

em que:

- η representa um caminho $(s = n_1, n_2, ..., n_q = n)$;
- n representa o nó terminal do caminho;
- c é o custo do caminho;
- p é um apontador para o caminho $(s = n_1, n_2, ..., n_{q-1})$.

Poderíamos guardar, ao invés do apontador, todo o caminho mas em grafos com muitos nós, isto geraria um gasto muito grande de memória. Assim, esta estrutura serve somente para memória de computador. Para construir um caminho completo, basta seguir os apontadores de trás para frente.

Exemplo:

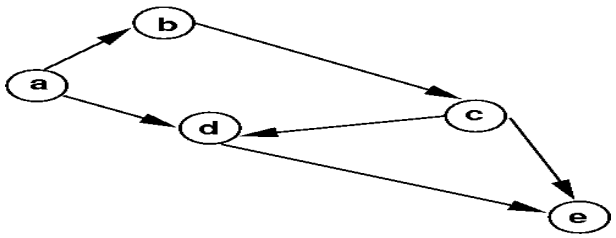


Figura 2.1: Caminhos guardados

a	-	$\eta_1 = (a, ., 0)$
ab	-	$\eta_2 = (b, ., 1)$
abc	-	$\eta_3 = (c, ., 2)$
ad	-	$\eta_4 = (d, ., 1)$
abcd	-	$\eta_5 = (d, ., 3)$
ade	-	$\eta_6 = (e, ., 4)$
abcde	-	$\eta_7 = (e, ., 5)$
abce	-	$\eta_8 = (e, ., 3)$

Definição 2.3. *Seja $G = (\mathcal{N}, \Gamma)$ um grafo finito em que o alvo é acessível a partir de s. Então um elemento $\eta = (n, c, p)$ é dito minimal se*

$$c = c^*(n) \; e \; c + h(n) = h(s),$$

ou seja, η coincide em parte com um caminho ótimo.

Como consequência imediata da Princípio de Otimalidade temos que se $\eta = (n, c, p)$ é minimal e aponta para $\tilde{\eta} = (\tilde{n}, \tilde{c}, \tilde{p})$, então $\tilde{\eta}$ também é minimal.

Definição 2.4. *Dados dois caminhos $\eta_i = (n_i, c_i, p_i)$, $\eta_j = (n_j, c_j, p_j)$, dizemos que η_i domina η_j se e somente se $n_i = n_j$ e $c_i \leq c_j$.*

Definição 2.5. *Dado um caminho $\eta = (n, c, p)$, os sucessores de η são os caminhos $\{\bar{\eta} = (\bar{n}, \bar{c}, \bar{p}) |\ \bar{n} \in \Gamma(n),\ \bar{p}$ apontador para $\eta\}$.*

Apresentaremos agora um algoritmo geral de rotulação. Esse algoritmo será estudado em detalhes adiante.

Dados $G = (\mathcal{N}, \Gamma)$, $s \in \mathcal{N}$, $T \subset \mathcal{N}$. Dizemos que um caminho já foi expandido por um algoritmo quando já foram gerados os seus sucessores.

Um algoritmo de rotulação gera iterativamente caminhos a partir de s, que são guardados em duas listas, chamadas ***Aberto*** e ***Fechado***.

A lista ***Aberto*** é a dos caminhos $\eta = (n, c, p)$ ainda não expandidos. Dizemos que um nó n está aberto quando existe um caminho aberto com nó terminal n.

A lista ***Fechado*** é a dos caminhos $\eta = (n, c, p)$ já expandidos. O nó terminal é dito nó fechado.

Desta forma cada caminho possui um rótulo, Aberto ou Fechado, aí está o motivo de também chamarmos o algoritmo de *Algoritmo de Rotulação.*

Nos Algoritmos de Rotulação os elementos[1] são gerados iterativamente. A cada iteração escolhe-se um elemento em Aberto para ser expandido, que passará a ter o rótulo Fechado. Os elementos sucessores gerados poderão ou não receber o rótulo Aberto, segundo um critério de eliminações.

Seja $G = (\mathcal{N}, \Gamma)$ um grafo finito, sem circuitos de custo negativo, cujo alvo é acessível a partir de s, então o algoritmo descrito abaixo resolve o problema de busca.

[1]Utilizaremos as palavras "elemento" e "caminho" de modo equivalente, lembrando que os elementos listados η representam caminhos no grafo.

Algoritmo 1 Algoritmo de Rotulação

1: Introduza em Aberto o elemento $\eta_1 = (s, 0, 0)$
2: **Enquanto** Aberto não está vazia
3: Escolha um elemento $\eta = (n, c, p)$ em Aberto para fechar.
4: Construa os elementos sucessores $\bar{\eta}_1, \bar{\eta}_2, ..., \bar{\eta}_q$, associados aos nós $\Gamma(n) = \{\bar{n}_1, \bar{n}_2, ..., \bar{n}_q\}$

$$\bar{\eta}_i = (\bar{n}_i, c + c(n, \bar{n}_i), \bar{p}_i),$$

$\bar{p}_i$ é o apontador para η
5: Elimine os elementos sucessores dominados por algum elemento listado (aberto ou fechado).
6: Elimine os elementos listados dominados por algum sucessor.
7: Introduza em Aberto os sucessores remanescentes
8: **Fim Enquanto**
9: Se há caminhos até o alvo em Fechado, encontrar um de custo mínimo e obter o caminho correspondente, seguindo os apontadores.

A regra de eliminação faz com que exista nas listas em cada instante no máximo um caminho listado até cada nó. Note que o fato de serem feitas eliminações não interfere no algoritmo, elas apenas são feitas com a finalidade de economizar memória de computador.

Este formato do algoritmo é muito geral. Adiante comentaremos o critério para a escolha do aberto a expandir em cada iteração e a regra de parada.

Proposição 2.6. *Um elemento listado minimal, $\eta_i = (n_i, c^*(n_i), .)$, nunca pode ser eliminado.*

Demonstração:
Para eliminar um elemento seria necessário um sucessor $\eta_j = (n_i, c_j, .)$ com

$$c_j < c^*(n_i).$$

Mas obrigatoriamente

$$c_j \geq c^*(n_i)$$

pois $c^*(n_i)$ é o custo de um caminho de custo mínimo de s até n_i.

□

Proposição 2.7. *O caminho correspondente a um elemento listado não tem circutos.*

A demonstração desta proposição pode ser vista em [12].

Teorema 2.8. *Seja G um grafo finito sem circuito de custo negativo. Então o algoritmo pára em um número finito de iterações.*

Demonstração:
Pela proposição 2.7, cada elemento listado corresponde um caminho sem circuitos. O mesmo caminho não pode ser listado duas vezes, pois o elemento correspondente a segunda obtenção do caminho seria eliminado.

O número de caminhos sem circuitos em um grafo finito é finito e portanto o número de iterações é finito, uma vez que a cada iteração fecha-se um aberto.

□

O lema a seguir é de grande valia. Ele afirma que enquanto não se encontrou um caminho de custo mínimo a um nó arbitrário, todos os caminhos de custo mínimo àquele nó estão sendo pesquisados pelo algoritmo.

Lema 2.9. *Em uma iteração qualquer, seja $\bar{n}$ um nó arbitrário acessível a partir de s. Seja $P = (s = n_1, n_2, ..., n_p = \bar{n})$ um caminho de custo mínimo de s a $\bar{n}$. Então:*
Se não existe um fechado com a forma $\bar{\eta} = (\bar{n}, c^*(\bar{n}), .)$ então existe um aberto com a forma $\eta_q = (n_q, c^*(n_q), .)$, $q = 1, 2, ..., p$.

Demonstração:
Seja $\bar{n}$ um nó arbitrário acessível a partir de s por um caminho de custo mínimo $(s = n_1, n_2, ..., n_p = \bar{n})$.

Na primeira iteração $(s, 0, 0)$ está aberto e nada há a provar.

Em outra iteração: suponha que não existe fechado $(\bar{n}, c^*(\bar{n}), .)$.

Seja $q-1$ o mais alto índice tal que algum $(n_{q-1}, c^*(n_{q-1}), .)$ está fechado, $1 \leq q-1 \leq p$

- $q-1$ existe pois $(s, 0, 0)$ está fechado
- $q-1 < p$ por hipótese

Como $(n_{q-1}, c^*(n_{q-1}), .)$ está fechado, foi expandido anteriormente, gerando o sucessor

$$(n_q, c^*(n_q), .) \text{ pois } c^*(n_q) = c^*(n_{q-1}) + c(n_{q-1}, n_q)$$

uma vez que P é um caminho de custo mínimo.
Se esse sucessor não foi eliminado, foi aberto.

Se foi eliminado, havia um aberto (n_q, c_q, p_q) que o eliminou, com $c_q \leq c^*(n_q)$. Obviamente $c_q = c^*(n_q)$.
Em qualquer caso, foi listado em aberto

$$(n_q, c^*(n_q), p_q).$$

Esse elemento não é eliminado e não é fechado por definição de q.
Portanto permanece aberto, completando a demonstração.

□

Corolário 2.10. *Quando o algoritmo termina, todos os elementos listados (fechados por construção) correspondem a caminhos de custo mínimo. Isto é, para todo* $\bar{\eta} = (\bar{n}, \bar{c}, \bar{p})$ *listado,*

$$\bar{c} = c^*(\bar{n}).$$

Além disso, todos os nós acessíveis a partir de s *estão fechados.*

Desta forma a lista Fechado descreve uma arborescência maximal mínima do subgrafo acessível por s.

Corolário 2.11. *Se* G *é grafo finito sem circuitos de custo negativo e existe nó do alvo acessível a partir de* s*, então o algoritmo termina com um caminho de custo mínimo entre* s *e o alvo, ou seja, um caminho ótimo.*

Em nenhum momento especificamos o procedimento de escolher um elemento para expandir em cada iteração. Esta escolha é um procedimento muito importante e pode ser feito de diferentes maneiras, é esta escolha que dá origem a diferentes algoritmos de rotulação.

A seguir apresentaremos os algoritmos de Busca Horizontal, Busca em Profundidade e Algoritmo de Dijkstra que não utilizam nenhuma informação quanto a proximidade do alvo em cada iteração. Em seguida apresentaremos o Algoritmo A^* que usa esta informação. O estudo dos três primeiros algoritmos foram iniciados por nós através de [13] e depois foi fortemente enriquecido pela abordagem feita em [12]. Em [12], Gonzaga faz um estudo bastante detalhado do algoritmo A^* ficando evidente a importância de pensar em caminhos os invés de nós.

2.3 Algoritmo de Busca Horizontal

Este algoritmo, também conhecido como método *Breadth-First* ou FIFO (*first in first out*), escolhe o elemento mais antigo da lista Aberto para expandir, ou seja, expande caminhos na ordem em que eles são gerados. Desta maneira o primeiro elemento a entrar na lista Aberto é também o primeiro a sair.

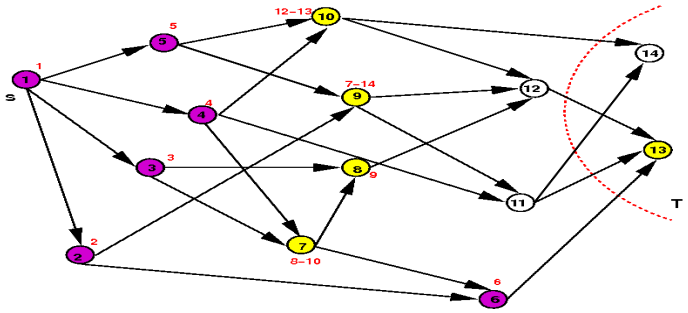


Figura 2.2: Busca Horizontal

No grafo da figura 2.2 mostramos seis iterações deste algoritmo. Os números em vermelho indicam a ordem em que eles foram abertos e conseqüentemente a ordem em que foram fechados. A cor amarela representa os nós abertos ao fim da sexta iteração e a magenta representa os nós fechados ao fim dessa iteração.

Note que este algoritmo desenvolve o grafo sistematicamente, evoluindo "de camada em camada". É conveniente se o alvo for pequeno, ou em grafos naturalmente estruturados em camadas, sem circuitos.

Neste caso, que ocorre por exemplo em problemas de decisões sequenciais, as listas podem ser manipuladas de maneira simplificada: em cada iteração só são processados nós de duas camadas. Os nós de uma camada somente são fechados quando todos os nós da camada anterior já estão fechados em caminhos de custo mínimo. O algoritmo então coincide com o método de programação dinâmica discreta progressiva.

Se prosseguirmos no grafo da figura 2.2 segundo o algoritmo 1 estaremos construindo uma arborescência maximal de custo mínimo para o subgrafo formado pelos nós acessíveis a partir de s descrita pela figura 2.3.

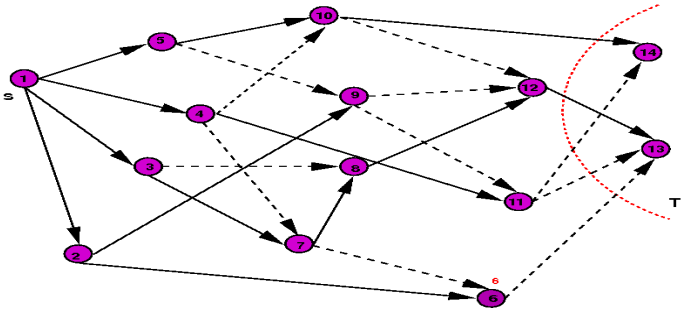


Figura 2.3: Arborescência maximal

2.4 Algoritmo de Busca em Profundidade

Este algoritmo, também conhecido como método *Depth-First* ou LIFO (*last in first out*), escolhe o elemento mais recente da lista Aberto para expandir. Assim o último elemento que entra em Aberto é o primeiro a sair.

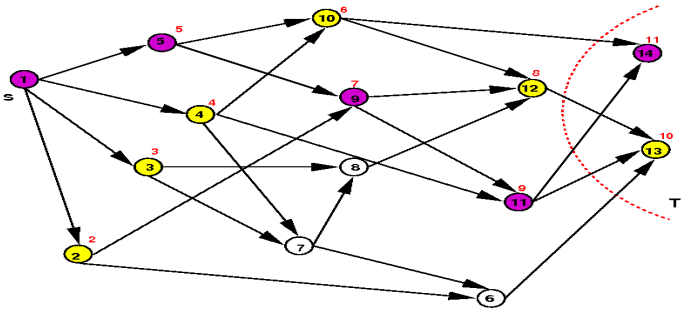


Figura 2.4: Busca em Profundidade

Na figura 2.4 representamos cinco iterações do algoritmo de Busca em profundidade. Os números em vermelho indicam a ordem em que eles foram abertos. A cor amarela representa os nós abertos e a magenta representa os nós fechados ao fim da quinta iteração.

Note que se algum sucessor do elemento expandido em uma iteração for listado, a próxima iteração vai fechar um desses sucessores, seguindo assim até atingir uma folha do grafo, ou até que todos os sucessores de um elemento sejam eliminados.

Esta é a busca mais "arrojada", procurando obter rapidamente soluções não ótimas. É conveniente se o alvo é grande e se o tempo de computação é crítico: pode-se parar o processamente antes de obter uma solução ótima.

Também neste método as listas podem ser manipuladas de maneira simplificada: em cada instante todos os nós abertos são sucessores de nós pertencentes a um caminho originado em s.

Se algum caminho, η de s a T é conhecido, seu custo pode ser usado como limitante na busca de novos caminhos. Desta forma elementos com custo, ou estimativa para o custo, maior que o custo de η podem ser eliminados, pois não nos levarão a caminhos ótimos até o alvo. Algoritmos que procedem desta forma são conhecidos como *Branch and Bound* e foram desenvolvidos para resolver problemas de Programação Linear Inteira.

2.5 Algoritmo de Dijkstra

O algoritmo de Dijkstra é o mais importante dos algoritmos para encontrar um caminho de custo mínimo entre nós de um grafo com custos não negativos e é o primeiro estudado por nós que leva em consideração nas expansões o custo.

Este algoritmo escolhe em cada iteração um elemento com custo mínimo entre os custos de todos os elementos da lista Aberto para expandir. Empates são resolvidos arbitrariamente, mas sempre dando preferência a caminhos cujos nós já estão no alvo.

Os próximos dois teoremas, extraídos de [13], nos transmitem as seguintes informações: para grafos com custos não negativos, todos os caminhos fechados são caminhos de custo mínimo. Ou seja, no momento em que o nó terminal de um caminho é expandido, já foi encontrado um caminho de custo mínimo até ele. Desta forma, no momento em que um nó do alvo for fechado, o problema de busca está resolvido.

Teorema 2.12. *Seja $G = (\mathcal{N}, \Gamma)$ um grafo finito tal que todos os ramos tem custos não negativos. Então o elemento fechado, $\eta = (n, c, p)$, em cada iteração tem custo mínimo, ou seja*

$$c = c^*(n).$$

Demonstração:

Suponha que em uma iteração é fechado um elemento

$$\eta = (n, c, p).$$

Por construção, todos os abertos η_i nessa iteração tem custos $c_i \geq c$.

O mesmo ocorrerá com todos os sucessores gerados no futuro, pois os custos são não negativos.

O elemento η nunca poderá ser eliminado.

Pelo corolário (2.10), η tem custo mínimo.

□

Teorema 2.13. *Seja $G = (\mathcal{N}, \Gamma)$ um grafo finito tal que todos os ramos tem custos não negativos e que o alvo é acessível a partir de s. Então o primeiro caminho cujo nó terminal n está no alvo, fechado pelo algoritmo de Dijkstra fornece uma solução ótima para o problema.*

Demonstração:
Pelo teorema (2.12), o elemento $\eta = (n, c, p)$, com $n \in T$, fechado tem custo mínimo. Assim qualquer caminho fechado posteriormente tem custo maior ou igual a c. Portanto, posteriormente nenhum caminho melhor poderá ser encontrado.

□

Com este teorema podemos estabelecer um novo critério de parada para o algoritmo de Dijkstra para grafos com custos não negativos: parar ao fechar um nó do alvo.

O algoritmo pode ser usado sobre grafos orientados ou não, e admite que todos os ramos possuem custos não negativos. Esta restrição é satisfeita por redes de transportes, onde os ramos representam normalmente distâncias ou tempos médios de percurso; poderão existir, no entanto, aplicações onde há ramos com custos negativos, nestes casos o algoritmo com este critério de parada não funcionará corretamente.

Ainda, em [12] Gonzaga conclui que o algoritmo de Dijkstra avança pelo grafo em "frentes de onda de custo constante" e observa que o algoritmo de Busca Horizontal corresponde à aplicação do algoritmo de Dijkstra ao problema de caminhos de *comprimento* mínimo.

Apresentamos abaixo uma ilustração do algoritmo de Dijkstra para o grafo dado pela figura 2.5.

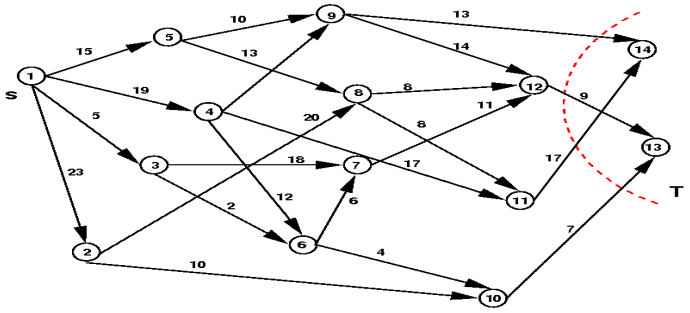


Figura 2.5: Grafo inicial

1. Primeira Iteração
   Fechou: $\eta_1 = (n_1, 0, 0)$, a raiz do grafo.
   Abriu:
   - $\eta_2 = (n_2, 23, 1)$
   - $\eta_3 = (n_3, 5, 1)$
   - $\eta_4 = (n_4, 19, 1)$
   - $\eta_5 = (n_5, 15, 1)$
2. Segunda Iteração
   Fechou: $\eta_3 = (n_3, 5, 1)$
   Abriu:
   - $\eta_6 = (n_6, 7, 3)$
   - $\eta_7 = (n_7, 23, 3)$
3. Terceira Iteração
   Fechou: $\eta_6 = (n_6, 7, 3)$
   Abriu:
   - $\eta_8 = (n_{10}, 11, 6)$
   - $\eta_9 = (n_7, 13, 6)$

   η_7 pode ser eliminado, pois $13 < 23$.
4. Quarta Iteração
   Fechou: $\eta_8 = (n_{10}, 11, 6)$
   Abriu:
   - $\eta_{10} = (n_{13}, 18, 8)$
5. Quinta Iteração
   Fechou: $\eta_5 = (n_5, 15, 1)$
   Abriu:

- $\eta_{11} = (n_9, 25, 5)$
- $\eta_{12} = (n_8, 28, 5)$

6. Sexta Iteração
   Fechou: $\eta_9 = (n_7, 13, 6)$
   Abriu:
   - $\eta_{13} = (n_{12}, 24, 9)$

7. Sexta Iteração
   Fechou: $\eta_{10} = (n_{13}, 18, 8)$

   Como n_{13} está no alvo, o algoritmo pára.

A figura 2.6 ilustra o grafo ao fim do algoritmo. Os nós em magenta representam os caminhos cujos nós terminais foram expandidos, os em amarelo representam os nós terminais dos caminhos que estão na lista Aberto e, o tracejado em vermelho representa os caminhos eliminados.

Observe que os nós em magenta com os ramos entre eles representam uma arborescência de custo mínimo. Porém esta arborescência não é maximal pois a lista Aberto não está vazia.

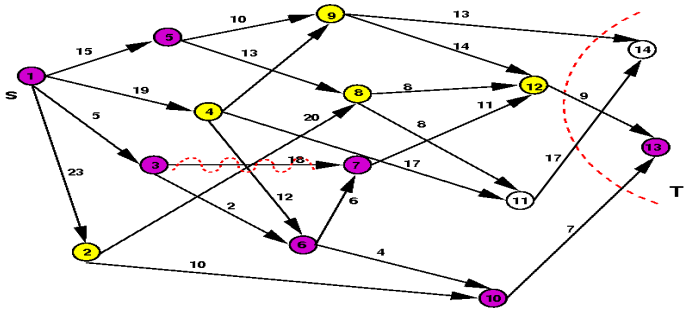


Figura 2.6: Grafo final

2.6 Algoritmo A*

O algoritmo A^* foi introduzido por Hart, Nilsson e Raphael em [9] e amplamente discutido por Nilsson em [14]. Este algoritmo é idêntico ao algoritmo de Dijkstra, exceto pelo uso de alguma informação heurística a respeito da estrutura do grafo, através de uma função "estimativa".

A abordagem adotada por nós, assim como definições e resultados que hora apresentamos, seguem as idéias de Gonzaga em [12] e [17].

Definição 2.14. *O custo de um caminho de custo mínimo de n ao alvo é definido pela aplicação:*

$$h : \mathcal{N} \longrightarrow \mathbb{R}$$

Em adição define-se para cada caminho $\eta = (n, c, p)$ a aplicação:

$$\eta \longmapsto f(\eta) = c^*(n) + h(n),$$

é o custo de um caminho de custo mínimo entre s e o alvo, que contém η como trecho inicial.

Em problemas em que é possível associar a cada nó n do grafo uma estimativa para o custo de um caminho ao alvo, o valor da estimativa de custo de um caminho entre s e o alvo, $\hat{f}$, será utilizado como critério de escolha do caminho a expandir em cada iteração. Esta estimativa é obtida em geral resolvendo um problema relaxado. Veremos exemplos de estimativas assim nos próximos capítulos.

Definição 2.15. *A estimativa de custo de um caminho de custo mínimo de n ao alvo é uma aplicação:*

$$\hat{h} : \mathcal{N} \longrightarrow \mathbb{R}$$

com $\hat{h}(t) = h(t) = 0$ para todo $t \in T$.

A estimativa de custo de um caminho de custo mínimo, entre s e o alvo que contém $\eta = (n, c, p)$ como trecho inicial, é definida por:

$$\eta \longmapsto \hat{f}(\eta) = c + \hat{h}(n).$$

Observação 2.16. *Note que:*

- *$\hat{h}(n)$ é associada ao nó n e deve estimar o valor de $h(n)$;*
- *$\hat{f}(\eta)$ é uma estimativa do custo do melhor caminho que se pode obter expandido o caminho η.*

2.6.1 Estimativa admissível

Definição 2.17. *Uma estimativa $\hat{h}(.)$ é dita admissível se para qualquer $n \in \mathcal{N}$:*

$$\hat{h}(n) \leq h(n).$$

Teorema 2.18. *Seja $G = (\mathcal{N}, \Gamma)$ um grafo finito sem circuitos de custo negativo. Se a estimativa é admissível e se o alvo é acessível a partir de s, então a qualquer iteração antes de A^* fechar o primeiro nó do alvo e para qualquer caminho ótimo P do nó s até um nó $t \in T$, existe um caminho aberto $\eta' = (n', c', p')$ com n' em P e*

$$\hat{f}(\eta') \leq h(s).$$

Demonstração:
Considere uma iteração $k \neq 1$ do algoritmo.
Seja $P = (s = n_1, n_2, ..., n_j = t)$ um caminho ótimo de s ao alvo.
Sabemos que s está fechado e que n_j não está fechado.

Pelo lema 2.9 existe um aberto com a forma $\eta_q = (n_q, c^*(n_q), p_q)$.

Ainda, por definição tem-se $\hat{f}(\eta_q) = c^*(n_q) + \hat{h}(n_q)$. Como a estimativa é admissível

$$\hat{f}(\eta_q) \leq c^*(n_q) + h(n_q) = h(s).$$

□

Em [14], Nilsson conclui que o teorema 2.18 garante que quanto maior a precisão da sub-estimativa menor será o número de elementos não minimais fechados.

Teorema 2.19. *Seja $G = (\mathcal{N}, \Gamma)$ um grafo finito sem circuitos de custos negativos. Se a estimativa é admissível e se o alvo é acessível a partir de s, então na primeira iteração em que o algoritmo A^* fecha um caminho até o alvo esse caminho é solução ótima do problema.*

Demonstração:
Pelo corolário 2.10 nós do alvo são fechados.
Seja $\bar{\eta} = (\bar{n}, \bar{c}, \bar{p})$ o primeiro caminho até o alvo a ser fechado, em alguma iteração $k > 1$.
Como $\bar{n} \in T$, $\hat{f}(\bar{\eta}) = \bar{c}$

Então pela regra de escolha

$$\bar{c} = \hat{f}(\bar{\eta}) \leq \hat{f}(\eta)$$

para todo η aberto nesta iteração.
Pelo teorema 2.18 existe um caminho $\tilde{\eta}$ aberto tal que $\hat{f}(\tilde{\eta}) \leq h(s)$.
Segue-se que $\bar{c} \leq h(s)$.
Como $\bar{c} \geq h(s)$ por definição de $h(s)$, só pode ocorrer $\bar{c} = h(s)$, como queríamos.

□

Desta forma o critério de parada para este algoritmo pode ser mudado para: até fechar um nó no alvo.

2.6.2 Estimativa consistente

A definição seguinte, extraída de [12], estabelece para o algoritmo A^* propriedades análogas àquelas do algoritmo de Dijkstra mostradas nos teoremas 2.12 e 2.13.

Definição 2.20. *Uma estimativa $\hat{h} : \mathcal{N} \longrightarrow \mathbb{R}$ é consistente se é admissível e se para todo $n_1, n_2 \in \mathcal{N}$,*

$$\hat{h}(n_1) \leq h(n_1, n_2) + \hat{h}(n_2).$$

Teorema 2.21. *Suponha que $\hat{h}$ é uma estimativa consistente e seja $\bar{\eta}$ o elemento escolhido em uma iteração do algoritmo A^*. Então nessa iteração $\hat{f}(\eta_i) \geq \hat{f}(\bar{\eta})$, $i = 1, 2, ..., q$, onde os η_i são os sucessores do elemento $\bar{\eta}$.*

Demonstração:
Seja $\eta_i = (n_i, c_i, p_i)$ um sucessor do elemento $\bar{\eta} = (\bar{n}, \bar{c}, \bar{p})$.

$$\begin{aligned}
\hat{f}(\eta_i) &= c_i + \hat{h}(n_i) \\
&= \bar{c} + c(\bar{n}, n_i) + \hat{h}(n_i) \\
&\geq \bar{c} + h(\bar{n}, n_i) + \hat{h}(n_i) \\
&\geq \bar{c} + \hat{h}(\bar{n}), \text{ pois a estimativa é consistente.} \\
&= \hat{f}(\bar{\eta})
\end{aligned}$$

logo $\hat{f}(\eta_i) \geq \hat{f}(\bar{\eta})$.

□

Corolário 2.22. *Se o algoritmo utiliza uma estimativa consistente, então $\hat{f}(\eta_j)$ é crescente com as iterações (onde η_j é o elemento escolhido em cada iteração).*

Teorema 2.23. *Se a estimativa é consistente, então todo elemento fechado $\bar{\eta} = (\bar{n}, \bar{c}, \bar{p})$ satisfaz $\bar{c} = c^*(\bar{n})$.*

Demonstração:
Suponha que $\bar{\eta} = (\bar{n}, \bar{c}, \bar{p})$ é fechado em alguma iteração, com $\bar{c} > c^*(\bar{n})$.
Pelo lema 2.9, existe um aberto $\eta = (n, c, p)$ tal que:

$$c = c^*(n),\ c + h(n, \bar{n}) = c^*(\bar{n}).$$

Portanto,

$$\begin{aligned}\hat{f}(\eta) &= c^*(n) + \hat{h}(n) \\ &\leq c^*(n) + h(n, \bar{n}) + \hat{h}(\bar{n}), \text{ pois a estimativa é consistente.} \\ &= c^*(\bar{n}) + \hat{h}(\bar{n}) \\ &< \hat{f}(\bar{\eta}) \text{ pois } \bar{c} > c^*(\bar{n}),\end{aligned}$$

o que contraria a escolha de $\bar{\eta}$ para ser expandido.

□

Do teorema 2.18 concluímos que:

- nenhum caminho η com $\hat{f}(\eta) > h(s)$ será expandido pelo algoritmo;
- se $\hat{h}(.) = h(.)$ (estimativa perfeita), então serão fechados somente caminhos η que são trechos iniciais de caminhos ótimos;
- se $\hat{h}(n) \leq h(n)$ (estimativa admissível), o número de elementos fechados depende do erro da estimativa: poderão ser fechados os elementos $\eta = (n, c, p)$ com $c + \hat{h}(n) \leq h(s)$.

A melhor situação ocorre no caso de estimativas consistentes. Nesse caso, todos os fechados são caminhos de custo mínimo (como ocorre no algoritmo de Dijkstra) e $\hat{f}(\eta_k)$ é crescente com as iterações k. O algoritmo explora o grafo em "frentes de onda" de $\hat{f}$ crescente.

Em [9] é demonstrado que nenhum algoritmo de rotulação pode encontrar um caminho ótimo ao alvo com menos expansões que A^*, se a única informação adicional disponível sobre o grafo é dada por $\hat{h}(.)$.

Capítulo 3

Alinhamento de Proteínas

Neste capítulo propomos uma abordagem matemática computacional, utilizando algoritmos de busca em grafos, para o problema de alinhamento de proteínas. A idéia para esta abordagem surgiu a partir de um seminário feito pelo Prof. José Mário Martinez em uma visita ao Dep. de Matemática da Universidade Federal de Santa Catarina. Em seu trabalho Martinez constrói o melhor emparelhamento entre átomos de duas proteínas utilizando programação dinâmica. Propomos uma nova modelagem para este problema via algoritmos de busca em grafos.

Proteínas são corpos rígidos definidos por sequências de aminoácidos. Em [20] encontramos a seguinte definição: um aminoácido é constituído por um átomo central de carbono ligado a quatro grupos: hidrogênio, grupo carboxílico $(COOH)$, grupo amínico (NH_2), comum a todos os aminoácidos, e um outro grupo R, ou cadeia lateral, que os distingue entre si.

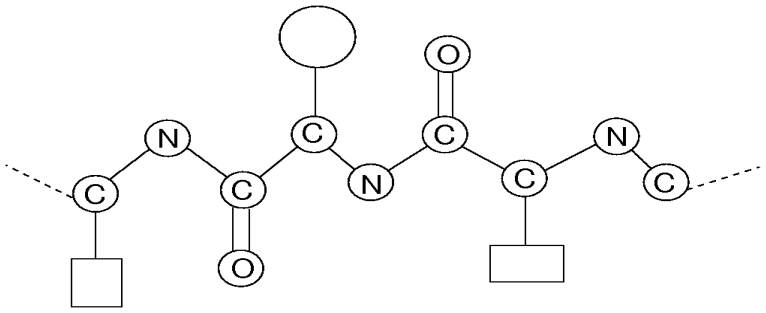


Figura 3.1: Proteína

Em sua estrutura primária cada proteína é representada por uma sequência de aminoácidos (uma sequência de letras). Em nosso estudo, como estamos interessados em alinhamento estrutural, uma proteína com N átomos de carbono é representada por N

pontos no espaço Euclidiano tri-dimensional, onde cada ponto refere-se às coordenadas espaciais de cada átomo de carbono. Desta forma uma proteína tem dimensão $N \times 3$.

O objetivo de alinhar proteínas [19] é estabelecer parâmetros de semelhança entre duas ou mais proteínas, respeitando critérios específicos. Então desenvolvem-se algoritmos para que sejam feitas pesquisas em bancos de dados e possam-se comparar proteínas. Comparação de proteínas é um importante problema em Biologia Estrutural [3] pois proteínas semelhantes ou com pedaços semelhantes podem ter a mesma função.

Tendo em vista que o número de proteínas obtidas experimentalmente está se tornando maior a cada ano [3], é preciso desenvolver algoritmos que sejam rápidos na pesquisa em bancos de dados e que tenham uma boa maneira de medir semelhanças entre proteínas.

Para alinhar proteínas, deve-se respeitar a seqüencialidade dos átomos. Também pode ocorrer que seja melhor não emparelhar algum átomo de uma das proteínas, neste caso dizemos que ocorreu um *gap*. Sempre que ocorrem gaps é preciso penalizá-los.

3.1 Definição do problema

Para definir o problema de alinhamento de proteínas considere:

- Uma proteína fixa espacialmente, caracterizada pelas coordenadas de seus átomos de carbono $y_1, y_2, ..., y_M \in \mathbb{R}^3$.
- Uma proteína caracterizada por pontos $x_1, x_2, ..., x_N \in \mathbb{R}^3$.
- Uma função de pontuação (score) que associa a um par de vetores $x, y \in \mathbb{R}^3$ o valor

$$f(x, y) = \frac{20}{1 + (\parallel x - y \parallel /2.24)^2} \tag{3.1}$$

  Esta função é usada na literatura em [3].

Vamos separar o problema em duas fases:

1. Emparelhamento: nesta fase consideram-se duas proteínas fixas no espaço e busca-se um emparelhamento entre pontos de duas proteínas

$$\{(x_{i_k}, y_{j_k}) | k = 1, 2, ..., p\} \text{ ou simplesmente } \{(i_k, j_k)\}_{k=1,2,...,p}$$

com $p \leq min\{M, N\}$ e i_k, j_k crescentes com k (respeita-se a sequencialidade).

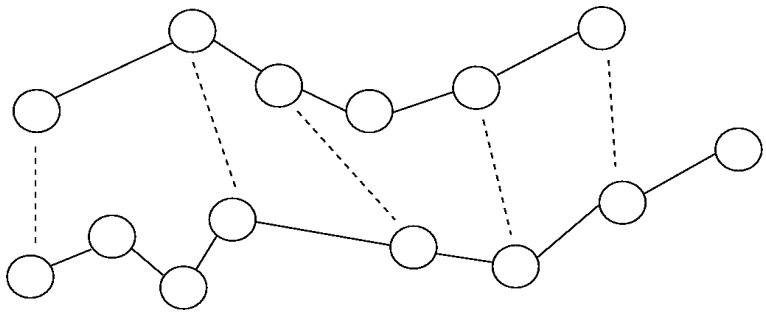


Figura 3.2: Emparelhamento

A figura 3.2 mostra um emparelhamento entre duas proteínas com a seguinte correspondência entre índices:

i_k	1	2	3	5	6
j_k	1	4	5	6	7

Neste emparelhamento vemos que x_4 não está emparelhado, o mesmo ocorrendo com y_2 e y_3. O emparelhamento apresenta um gap em cada proteína. Neste problema os gaps são penalizados com lucro igual a -10, que independe da extensão do gap.

A cada emparelhamento $\mathcal{E} = \{(i_k, j_k)\}_{k=1,2,...,p}$, associa-se uma pontuação

$$F(\mathcal{E}) = \sum_{k=1}^{p} f(x_{i_k}, y_{j_k}) - 10n_g, \tag{3.2}$$

em que n_g é o número total de gaps do emparelhamento.

O problema da primeira fase é: entre todos os emparelhamentos possíveis $\mathcal{E} = \{(i_k, j_k)\}_{k=1,2,...,p},\ p \leq min\{M, N\}$, encontrar um com o máximo valor de $F(\mathcal{E})$.

Este é o problema que vamos resolver nesta dissertação.

2. Deslocamento: uma vez estabelecido um emparelhamento dos átomos, desloca-se a proteína $(x_1, ..., x_N)$ por meio de uma translação e uma rotação (mantendo rígida a estrutura), de modo a otimizar um critério dado.

Existem duas abordagens na literatura:

- O problema de Procrustes[1]: é um problema de quadrados mínimos em que minimiza-se $\sum_{k=1}^{p} \parallel x_{i_k} - y_{j_k} \parallel^2$.

  A resolução deste problema é simples: faz-se uma translação de proteína para fazer coincidir os centros de massa e busca-se uma matriz de rotação da proteína em torno do centro de massa para minimizar o critério acima. Este problema é clássico (problema de Procrustes) e admite solução analítica (que faz parte do pacote Matlab).

- O critério de pontuação: busca-se um deslocamento que maximize o critério de pontuação.

O algoritmo completo consiste em ciclos *emparelhamento-deslocamento*. Dada uma posição da proteína, faz-se um emparelhamento seguido de um deslocamento. Após o deslocamento, o emparelhamento pode deixar de ser ótimo, e deve ser refeito. O processo é repetido até que se estabilize, ou entre em um ciclo.

O método Structal

O método Structal [10] é o mais utilizado atualmente, e usa o problema de Procrustes para o deslocamento. O método é descrito em Andreani, Martinez e Martinez [3]. Tem desvantagens óbvias: utiliza dois critérios distintos nas duas fases e não maximiza a pontuação [2]. Pode entrar em ciclo e isto de fato ocorre frequentemente[2]. Como vantagem, tem a simplicidade do problema de Procrustes.

Método LOVO para alinhamento de proteínas

Este método [3], [2] utiliza o mesmo critério em ambas as fases, produzindo resultados melhores. Mas o problema de deslocamento é muito mais difícil, devido à dificuldade do critério, que não é convexo.

Neste trabalho não mencionaremos mais o problema de deslocamento. Vamos descrever um novo método para o problema de emparelhamento utilizando a modelagem de grafos, que é comum a ambos métodos. A partir de agora, consideramos dadas duas proteínas fixas no espaço.

[1] Processo de obter a superposição de corpos rígidos que minimiza um critério entre correspondente átomos de carbono de duas proteínas [2].

[2] Afirmação de Andreani, Martinez e Martinez em [3].

Em ambas as abordagens citadas, o problema de emparelhamento é resolvido por programação dinâmica. Em um problema discreto, o algoritmo de programação dinâmica tem o desempenho de Busca Horizontal.

Como o problema é de maximização com custos positivos, o que equivale à minimização com custos negativos, não podemos utilizar o algoritmo de $Dijkstra$, mas mostraremos como usar o algoritmo A^*.

A matriz de pontuação

Para o problema de emparelhamento, construímos uma matriz de pontuação ou lucro (score) L, $(N \times M)$, com

$$L(i,j) = \frac{20}{1 + (d(i,j)/2,24)^2}, \;\; d(i,j) = \parallel x_i - y_j \parallel \tag{3.3}$$

A qualidade do alinhamento é medida através da equação (3.2), isto é pela soma dos pontos obtidos através deste emparelhamento menos as penalidades pela introdução de gaps.

3.2 Definindo o grafo

Estamos buscando um emparelhamento $\{(i_k, j_k)\}_{k=1,2,...,p}$ que maximize

$$\sum_{k=1}^{p} f(x_{i_k}, y_{j_k}) - 10n_g, \text{ com } p \leq min\{M, N\} \tag{3.4}$$

O lema a seguir, feito por nós, limita a distribuição de gaps em um emparelhamento ótimo.

Lema 3.1. *Suponha que $\{(i_k, j_k)\}_{k=1,2,...,p}$ é um emparelhamento ótimo. Então não podem ocorrer simultaneamente $i_{k+1} > i_k + 1$ e $j_{k+1} > j_k + 1$ para nenhum $k = 1, ..., p-1$.*

Demonstração:
Suponha que $\{(i_k, j_k)\}_{k=1,2,...,p}$ é ótimo e que para algum $q = 1, ..., p-1$, $i_{q+1} > i_q + 1$ (há um gap entre i_q e i_{q+1}) e $j_{q+1} > j_q + 1$ (há um gap entre j_q e j_{q+1}). Então podemos

construir um novo emparelhamento unindo os átomos $i_q + 1$ e $j_q + 1$:

$$\{(i_1, j_1), ..., (i_q, j_q), (i_q + 1, j_q + 1), (i_{q+1}, j_{q+1}), ..., (i_p, j_p)\}$$

Este novo emparelhamento tem um número de gaps menor ou igual ao do antigo, e pontuação excedente em $L(i_q + 1, j_q + 1)$, contradizendo a otimalidade do alinhamento.

□

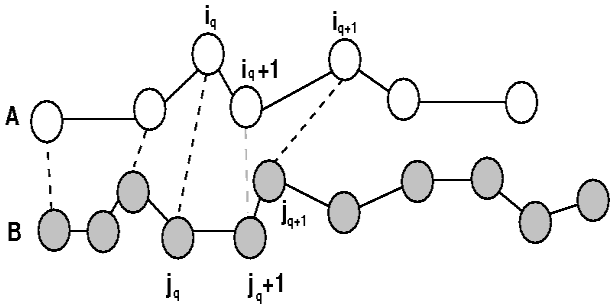


Figura 3.3: Ilustração do lema

Para construir um grafo que represente o problema, partimos do seguinte raciocínio: suponha que já decidimos como emparelhar os átomos $i = 1, 2, ...,$
$\alpha - 1$ e $j = 1, 2, ..., \beta - 1$, e queremos decidir se α será ligado a β ou não. Considerando o lema 3.1, existem três situações possíveis:

- $\alpha - 1$ e $\beta - 1$ estão emparelhados;
- $\alpha - 1$ não faz parte do emparelhamento (ocorreu gap à esquerda de α);
- $\beta - 1$ não faz parte do emparelhamento (ocorreu gap à esquerda de β).

3.2.1 Nó

Os nós $\mathcal{N}$ do grafo correspondem a todas as combinações de i, j e gaps compatíveis com as situações acima. Um nó n é uma trinca (α, β, gap) em que:

- α é um átomo da primeira proteína;
- β é um átomo da segunda proteína;
- gap indica se há ou não gaps à esquerda de α ou β:
  - gap$= 1$ se há gap à esquerda de α
  - gap$= 2$ se há gap à esquerda de β
  - gap$= 0$ se não há gaps à esquerda de α ou β.

Nó inicial: $s = (1, 1, 0)$.

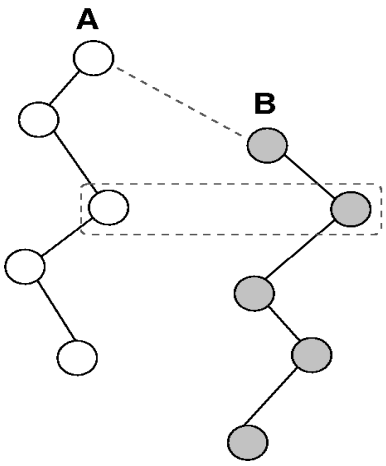


Figura 3.4: Definição do nó

O nó $n = (3; 2; 1)$ representa que no emparelhamento da figura 3.4, houve um gap a esquerda do terceiro átomo da proteína A.

3.2.2 Ramo

Os ramos do grafo correspondem às decisões que podemos tomar a partir de um nó (α, β, gap). As decisões possíveis são: ligar α e β ou rejeitar um dos átomos. O lucro da decisão (lucro do ramo) será $L(\alpha, \beta)$ se eles forem ligados, -10 se for criado um novo gap ou 0 se nenhuma dessas opções ocorrer.

3.2.3 Alvo

Sempre que emparelharmos o último átomo de uma das proteínas não há mais emparelhamentos a fazer pois devemos respeitar a sequencialidade dos átomos. Sendo assim qualquer nó que é cabeça de um ramo indicando que o último átomo de uma das proteína foi emparelhado, este nó é um nó do alvo.

Note que os nós do conjunto alvo correspondem as folhas do grafo.

Para simplificar a exposição, vamos definir os seguintes nós fictícios:

$$T = \{(N+1, j, gap), (i, M+1, gap) |\ i = 1, ..., N+1,\ j = 1, ..., M+1,\ gap = 0, 1, 2\}.$$

Considere duas proteínas tais que $N = 3$ e $M = 3$,

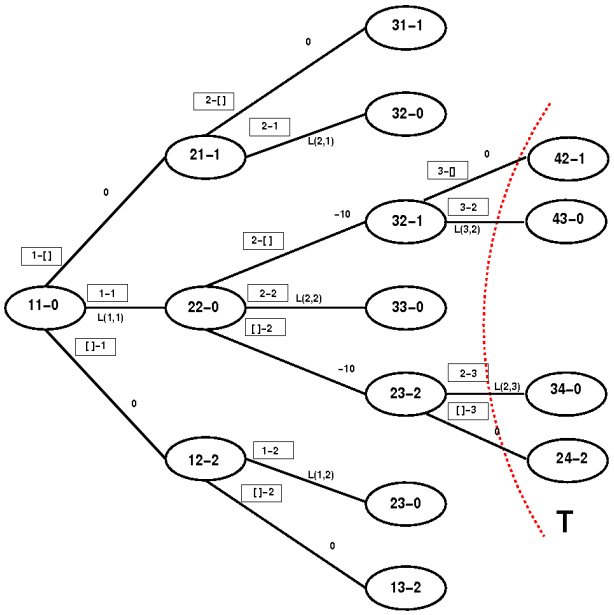


Figura 3.5: Representação do grafo

Desta forma, no grafo da figura 3.5:

- o ramo ligando o nó $(1, 2, 2)$ ao nó $(2, 3, 0)$ indica que emparelhamos o primeiro átomo da primeira proteína com o segundo átomo da segunda proteína;
- o ramo ligando o nó $(2, 2, 0)$ ao nó $(2, 3, 2)$ indica que houve um gap à esquerda do terceiro átomo da segunda proteína;
- o ramo ligando o nó $(2, 2, 0)$ ao nó $(3, 2, 1)$ indica que houve um gap à esquerda do segundo átomo da primeira proteína;
- os nós $(4, 2, 1)$, $(4, 3, 0)$, $(3, 4, 0)$ e $(2, 4, 2)$ são nós do alvo.

Observação 3.2. *Um nó (α, β, gap) não contém nenhuma informação sobre o emparelhamento dos átomos α e β. São os ramos que informam sobre a decisão que tomamos. Se examinarmos um emparelhamento das proteínas descrita por uma sequência de nós $(\alpha_i, \beta_i, gap_i)$, saberemos que α_{i-1} e β_{i-1} foram emparelhados se $gap_i = 0$.*

3.3 Operador sucessor

Cada nó tem no máximo três sucessores e nós fictícios não tem sucessores.

Definição 3.3. *Operador sucessor é uma aplicação* $\Gamma : n = (\alpha, \beta, gap) \in \mathcal{N} \longrightarrow \Gamma(n)$*, em que* $\Gamma(n)$ *tem elementos descritos a seguir com os custos dos ramos correspondentes:*

- *Ligação de* α *e* β*:*

$$u = (\alpha + 1, \beta + 1, 0) \textit{ com lucro } c(n, u) = L(\alpha, \beta).$$

- *Criação de um novo gap:*

  Se gap$= 0$

$$v = (\alpha + 1, \beta, 1),\ \textit{lucro} = \begin{cases} 0 \textit{ se } \alpha = N \textit{ ou } \alpha = 1 \\ -10 \textit{ no caso contrário} \end{cases}$$

$$w = (\alpha, \beta + 1, 2),\ \textit{lucro} = \begin{cases} 0 \textit{ se } \beta = M \textit{ ou } \beta = 1 \\ -10 \textit{ no caso contrário} \end{cases}$$

- *Aumento de um gap (respeitando o lema 3.1):*

  Se gap=1 e $\alpha < N$

$$v = (\alpha + 1, \beta, 1),\ \textit{lucro} = 0$$

  Se gap=2 e $\beta < M$

$$w = (\alpha, \beta + 1, 2),\ \textit{lucro} = 0$$

Observação 3.4. *No operador sucessor, não permitimos o aumento de gap na situações que contradizem o lema 3.1 e nem quando o átomo é o último da proteína. Poderíamos aceitar esses sucessores, mas eles nunca poderiam pertencer a um emparelhamento ótimo e aumentariam o tamanho do grafo.*

3.4 Caminho

Um caminho $(n_1, n_2, ..., n_p)$ no grafo corresponde a um emparelhamento. Sejam $n_r = (i_r, j_r, gap_r)$. Então para $r = 1, ..., p - 1$ (i_r, j_r) estão emparelhados se e somente se $gap_{r+1} = 0$. O lucro do emparelhamento é o lucro do caminho.

3.5 Super-estimativas

No problema de busca de caminhos de custo mínimo usando o algoritmo A^* procurávamos uma subestimativa que fosse a maior possível. Neste problema, em que

estamos interessados em maximizar o lucro, procuramos por uma super-estimativa que seja a menor possível. Para isto analisamos a matriz L.

Definição 3.5. *Considere um nó $n = (\alpha, \beta, gap)$. Queremos calcular uma super-estimativa $\hat{h}(\alpha, \beta, gap)$ para a lucro $h(\alpha, \beta, gap)$ de um caminho ótimo entre (α, β, gap) e o alvo, ou seja, de um emparelhamento ótimo entre os átomos $(\alpha, \alpha+1, ..., N)$ e os átomos $(\beta, \beta+1, ..., M)$.*

A estimativa é calculada relaxando a exigência de sequencialidade, e permitindo que um átomo seja ligado a vários outros. Temos duas estimativas:

- *Tomando como referência a primeira proteína e ligando cada um de seus átomos i ao átomo de máxima pontuação $L(i, j)$, obtemos*

$$h(\alpha, \beta, gap) \leq \sum_{i=\alpha}^{N} max\{L(i, j)|j = \beta, ..., M\}) \tag{3.5}$$

- *Tomando como referência a segunda proteína e agindo de modo similar, obtemos*

$$h(\alpha, \beta, gap) \leq \sum_{j=\beta}^{M} max\{L(i, j)|i = \alpha, ..., N\}) \tag{3.6}$$

A estimativa $\hat{h}(\alpha, \beta)$ será calculada pelo menor entre esses dois limitantes superiores. O valor de gap não é levado em conta no cálculo da estimativa.

O cálculo de $\hat{h}$ é feito facilmente a partir de L, gerando uma matriz *HCHAPEU* de mesma dimensão.

$$HCHAPEU(\alpha, \beta) = \hat{h}(n), \tag{3.7}$$

para todo $n = (\alpha, \beta, .)$, tal que n não está no alvo, pois
$\hat{h}(t) = 0$ para t no alvo.

Observação 3.6. *Obtemos esta matriz de uma maneira bastante prática. Primeiro somamos para todo par (α, β) o máximo por linhas da matriz $L(\alpha : N, \beta : M)$, obtendo uma matriz que denotamos Linha. Depois somamos o máximo por colunas obtendo uma matriz que denotamos Coluna. Temos HCHAPEU fazendo o mínimo entre as entradas correspondentes das duas matrizes.*

Vamos analisar o seguinte exemplo, onde a matriz representa que estamos tentando emparelhar uma proteínas com quatro átomos e outra com cinco:

$$L = \begin{pmatrix} 34 & 28 & 25 & 24 & 27 \\ 33 & 35 & 34 & 31 & 29 \\ 40 & 26 & 27 & 32 & 25 \\ 30 & 31 & 26 & 28 & 36 \end{pmatrix}$$

$$\mathit{Coluna} = \begin{pmatrix} 177 & 137 & 102 & 68 & 36 \\ 177 & 137 & 102 & 68 & 36 \\ 166 & 126 & 95 & 68 & 36 \\ 151 & 121 & 95 & 64 & 36 \end{pmatrix}$$

$$\mathit{Linha} = \begin{pmatrix} 145 & 131 & 129 & 126 & 117 \\ 111 & 103 & 102 & 99 & 90 \\ 76 & 68 & 68 & 68 & 61 \\ 36 & 36 & 36 & 36 & 36 \end{pmatrix}$$

$$\mathit{HCHAPEU} = \begin{pmatrix} 145 & 131 & 102 & 68 & 36 \\ 111 & 103 & 102 & 68 & 36 \\ 76 & 68 & 68 & 68 & 36 \\ 36 & 36 & 36 & 36 & 36 \end{pmatrix}$$

3.6 Testes numéricos

Neste problema não foi possível utilizar o algoritmo de Dijkstra puro pois como este grafo possui ramos com custos negativos, o critério de parada geralmente usado, *"até fechar nó do alvo"* não se aplica. Sendo assim, acreditando termos uma boa estimativa, testamos resolver este problema com o algoritmo A^*. Como estamos interessados em analisar o desempenho da modelagem feita com grafos em relação ao algoritmo de Programação Dinâmica, considerando o que foi exposto na seção 2.3, comparamos o algoritmo A^* com o algoritmo de Busca Horizontal.

Para podermos comparar o desempenho dos dois algoritmos guardamos o número de caminhos: abertos, fechados, reabertos, eliminados. Guardamos também o número de elementos sucessores guardados, o tempo gasto por cada algoritmo e, como neste problema o alvo é um conjunto, guardamos também o número de caminhos até o

alvo que foram abertos antes de se encontrar um caminho ótimo.

Estes testes foram feitos em um computador com um processador de 3Ghz e 512 MB de memória.

Exemplo 3.7. *Este exemplo se refere a uma proteína com 100 átomos de carbono e outra com 60 átomos de carbono que estão próximas em três trechos.*

Algoritmo	Abertos	Fechados	Reabertos	Alvos	Eliminados	Sucessores
Horizontal	51949	49499	34167	67	63073	115088
A^*	16013	15444	0	1	20024	36037

Algoritmo	Tempo total
Horizontal	48 segundos
A^*	14 segundos

Lucro ótimo = 1040

Exemplo 3.8. *Os próximos exemplos são idênticos ao do item anterior exceto pela introdução de uma perturbação aleatória.*

Algoritmo	Abertos	Fechados	Reabertos	Alvos	Eliminados	Sucessores
Horizontal	50949	46595	31222	66	57357	108371
A^*	16431	15444	0	1	19606	36037

Algoritmo	Tempo total
Horizontal	44 segundos
A^*	14 segundos

Lucro ótimo = 460,3477

Exemplo 3.9. *Os próximos exemplos são idênticos ao do item anterior exceto pela introdução de uma perturbação aleatória.*

Algoritmo	Abertos	Fechados	Reabertos	Alvos	Eliminados	Sucessores
Horizontal	43906	39011	23634	50	46702	90657
A^*	17212	15444	0	1	18824	36036

Algoritmo	Tempo total
Horizontal	31 segundos
A^*	12 segundos

Lucro ótimo = 263,4077

Exemplo 3.10. *Este exemplo se refere a uma proteína com 200 átomos de carbono e outra com 110 átomos de carbono que estão próximas em quatro trechos.*

Algoritmo	Abertos	Fechados	Reabertos	Alvos	Eliminados	Sucessores
Horizontal	342988	334448	273120	106	434469	777562
A^*	62514	61491	0	1	80966	143480

Algoritmo	Tempo total
Horizontal	6,7 minutos
A^*	47 segundos

Lucro ótimo = 2050

Exemplo 3.11. *O próximo exemplo é idêntico ao do item anterior exceto pela introdução de uma perturbação aleatória.*

Algoritmo	Abertos	Fechados	Reabertos	Alvos	Eliminados	Sucessores
Horizontal	340985	326147	264779	113	418717	759814
A^*	63384	61486	0	6	80077	143466

Algoritmo	Tempo total
Horizontal	6,3 minutos
A^*	42 segundos

Lucro ótimo = 857,5231

A seguir apresentamos o emparelhamentos obtido em cada um dos exemplos acima. Esta apresentação é importante para analisarmos o comportamento da estimativa.

Exemplo 3.7

Emparelhou	Lucro	Estimativa	$\hat{f}(\eta)$
90 53	1040	0	1040
89 52	1020	20	1040
88 51	1000	40	1040
87 50	980	60	1040
86 49	960	80	1040
85 48	940	100	1040
84 47	920	120	1040
83 46	900	140	1040
82 45	880	160	1040
81 44	860	180	1040
80 43	840	200	1040
79 42	820	220	1040
78 41	800	240	1040
77 40	780	260	1040
76 39	760	280	1040
75 38	740	300	1040
74 37	720	320	1040
73 36	700	340	1040
72 35	680	360	1040
71 34	660	380	1040
70 33	640	400	1040
50 32	630	420	1050
49 31	610	440	1050
48 30	590	460	1050
47 29	570	480	1050
46 28	550	500	1050
45 27	530	520	1050
44 26	510	540	1050
43 25	490	560	1050
42 24	470	580	1050
41 23	450	600	1050
40 22	430	620	1050
30 21	420	640	1060
29 20	400	660	1060
28 19	380	680	1060
27 18	360	700	1060
26 17	340	720	1060
25 16	320	740	1060
24 15	300	760	1060
23 14	280	780	1060
22 13	260	800	1060
21 12	240	820	1060
20 11	220	840	1060
19 10	200	860	1060
18 9	180	880	1060
17 8	160	900	1060
16 7	140	920	1060
15 6	120	940	1060
14 5	100	960	1060
13 4	80	980	1060
12 3	60	1000	1060
11 2	40	1020	1060
10 1	20	1040	1060

Exemplo 3.8

Emparelhou	Lucro	Estimativa	$\hat{f}(\eta)$
92 53	460	0	460
91 52	456	4	460
90 51	439	22	461
89 50	436	27	463
88 49	433	34	467
87 48	429	49	478
86 47	423	58	482
85 46	405	85	490
84 45	398	112	510
83 44	390	152	541
82 43	385	166	551
81 42	368	184	552
80 41	366	189	555
79 40	357	212	569
76 39	352	230	581
75 38	346	237	583
74 37	330	258	589
73 36	326	281	607
72 35	315	301	616
71 34	300	322	622
70 33	296	337	634
52 32	293	351	644
46 31	290	365	654
45 30	283	372	654
44 29	263	395	658
43 28	251	417	669
42 27	247	437	684
41 26	228	456	684
40 25	208	476	684
37 24	206	490	696
36 23	188	508	696
35 22	185	526	712
34 21	181	540	721
33 20	179	542	721
32 19	160	561	721
31 18	157	565	721
30 17	140	583	722
29 16	125	599	724
28 15	121	605	727
27 14	119	613	732
26 13	111	632	744
24 12	105	668	773
23 11	86	696	782
22 10	79	704	784
21 9	75	715	790
20 8	59	747	805
19 7	56	751	807
18 6	52	761	813
17 5	37	778	815
14 4	37	807	844
13 3	27	818	845
12 2	9	849	858
11 1	4	859	863

Exemplo 3.9

Emparelhou	Lucro	Estimativa	$\hat{f}(\eta)$
87 53	263	0	263
86 52	261	2	263
85 51	246	18	264
84 50	242	22	264
83 49	237	39	276
82 48	218	60	279
81 47	216	81	297
80 46	215	90	305
79 45	214	94	308
73 44	216	121	337
72 43	204	133	337
71 42	203	149	352
66 41	194	174	368
65 40	174	199	374
64 39	174	215	389
63 38	171	218	389
62 35	176	256	432
61 34	157	277	435
60 33	147	301	447
59 32	146	319	465
58 31	145	321	466
57 30	142	340	482
56 29	140	347	487
55 28	138	364	502
53 27	143	381	524
52 26	138	387	525
51 25	122	406	528
50 24	120	412	532
49 23	117	427	544
44 22	109	462	572
43 21	107	474	581
42 20	105	478	583
41 19	92	495	587
40 18	91	505	596
39 17	87	511	598
38 16	86	516	602
37 15	85	523	608
36 14	84	539	623
35 13	83	552	635
34 12	80	564	645
33 11	65	584	648
32 10	62	594	656
31 9	45	614	658
30 7	50	638	687
29 6	48	644	692
28 5	30	670	700
21 4	26	722	748
20 3	25	734	759
13 2	16	808	824
12 1	15	810	825

Exemplo 3.10

Emparelhou	Lucro	Estimativa	$\hat{f}(\eta)$
160 104	2050	0	2050
159 103	2030	20	2050
158 102	2010	40	2050
157 101	1990	60	2050
156 100	1970	80	2050
155 99	1950	100	2050
154 98	1930	120	2050
153 97	1910	140	2050
152 96	1890	160	2050
151 95	1870	180	2050
150 94	1850	200	2050
149 93	1830	220	2050
148 92	1810	240	2050
147 91	1790	260	2050
146 90	1770	280	2050
145 89	1750	300	2050
144 88	1730	320	2050
143 87	1710	340	2050
142 86	1690	360	2050
141 85	1670	380	2050
140 84	1650	400	2050
139 83	1630	420	2050
138 82	1610	440	2050
137 81	1590	460	2050
136 80	1570	480	2050
135 79	1550	500	2050
134 78	1530	520	2050
133 77	1510	540	2050
132 76	1490	560	2050
131 75	1470	580	2050
130 74	1450	600	2050
129 73	1430	620	2050
128 72	1410	640	2050
127 71	1390	660	2050
126 70	1370	680	2050
125 69	1350	700	2050
124 68	1330	720	2050
123 67	1310	740	2050
122 66	1290	760	2050
121 65	1270	780	2050
120 64	1250	800	2050
119 63	1230	820	2050
118 62	1210	840	2050
117 61	1190	860	2050
116 60	1170	880	2050
115 59	1150	900	2050
114 58	1130	920	2050
113 57	1110	940	2050
112 56	1090	960	2050
111 55	1070	980	2050
110 54	1050	1000	2050
90 53	1040	1020	2060
89 52	1020	1040	2060
88 51	1000	1060	2060

Continuação 3.10

Emparelhou	Lucro	Estimativa	$\hat{f}(\eta)$
87 50	980	1080	2060
86 49	960	1100	2060
85 48	940	1120	2060
84 47	920	1140	2060
83 46	900	1160	2060
82 45	880	1180	2060
81 44	860	1200	2060
80 43	840	1220	2060
79 42	820	1240	2060
78 41	800	1260	2060
77 40	780	1280	2060
76 39	760	1300	2060
75 38	740	1320	2060
74 37	720	1340	2060
73 36	700	1360	2060
72 35	680	1380	2060
71 34	660	1400	2060
70 33	640	1420	2060
50 32	630	1440	2070
49 31	610	1460	2070
48 30	590	1480	2070
47 29	570	1500	2070
46 28	550	1520	2070
45 27	530	1540	2070
44 26	510	1560	2070
43 25	490	1580	2070
42 24	470	1600	2070
41 23	450	1620	2070
40 22	430	1640	2070
30 21	420	1660	2080
29 20	400	1680	2080
28 19	380	1700	2080
27 18	360	1720	2080
26 17	340	1740	2080
25 16	320	1760	2080
24 15	300	1780	2080
23 14	280	1800	2080
22 13	260	1820	2080
21 12	240	1840	2080
20 11	220	1860	2080
19 10	200	1880	2080
18 9	180	1900	2080
17 8	160	1920	2080
16 7	140	1940	2080
15 6	120	1960	2080
14 5	100	1980	2080
13 4	80	2000	2080
12 3	60	2020	2080
11 2	40	2040	2080
10 1	20	2060	2080

3.6.1 Análise dos testes

Nestes testes o algoritmo de Busca Horizontal e o algoritmo A^* agiram dentro do esperado. Em todos os testes o algoritmo A^* foi superior pois o número de caminhos que ele precisa fechar até encontrar um caminho ótimo é menor que o algoritmo de Busca Horizontal, que precisa fechar todos os caminhos até o alvo.

Notamos também que o número de caminhos reabertos pelo algoritmo de Busca Horizontal é grande isto faz com que o tempo gasto por este algoritmo seja maior. Em relação ao tempo, observamos que a medida que aumentamos o tamanho do problema a diferença do tempo gasto pelo algoritmo de Busca Horizontal e pelo algoritmo A^* aumenta.

Em relação a estimativa, nos exemplos onde não inserimos uma perturbação aleatória ela foi excelente, nos outros exemplos ela começou alta em relação ao custo ótimo. Porém, mesmo assim não diminuiu a superioridade do algoritmo A^*.

Foram testados alguns exemplos fictícios, o que dá ao método alguma expectativa boa quanto ao seu comportamento. Para testar o método efetivamente seria necessário introduzi-lo nos algoritmos *Structal ou LOVO* e fazer testes práticos utilizando banco de dados de proteínas. Isso excede os objetivos deste trabalho.

A metodologia introduzida aqui pode também ser aplicada ao sequenciamento em genômica, em que há problemas semelhantes.

Capítulo 4

O quebra-cabeça de 15 peças

Na área de Inteligência Artificial, algoritmos de busca em espaço de estados e planejamento de trajetória e ações são usados para o desenvolvimento de jogos tais como: *8 puzzle, 15 puzzle, Resta 1, Xadrez*. Neste capítulo apresentamos uma aplicação do problema de busca de caminhos em grafos para o quebra-cabeça de 15 peças *(fifteen puzzle)*.

O quebra-cabeça de 15 peças consiste em 15 quadrados numerados de 1 a 15 mais um quadro branco que são colocados em uma caixa 4×4. Os quadrados numerados simbolizam as 15 peças do quebra-cabeça.

O jogo consiste no seguinte: dada uma configuração inicial arbitrária desejamos chegar à configuração original, figura 4.1, que chamaremos de configuração alvo. Em cada passo desliza-se uma peça do quebra-cabeça, fazendo-a ocupar o quadrado não numerado que de agora em diante chamaremos de posição vazia.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Figura 4.1: Quebra-cabeça de 15 peças

Em 1878 o americano Sam Loyd dizia estar *conduzindo o mundo à lou-*

cura, quando desafiava as pessoas a resolver o problema citado acima oferecendo recompensa de 1000 doláres a primeira solução correta. Porém a configuração que ele propunha ser resolvida era impossível, idêntica a da figura acima exceto pela troca da peça 14 com a peça 15. A loucura começou segundo [21] nos Estados Unidos em Janeiro de 1880 e na Europa em Abril do mesmo ano.

Sabe-se agora que se trocarmos a posição de duas peças, levantando-as da caixa, a configuração resultante não tem solução, pois para que exista solução o número de trocas deve ser par. Por esta razão ninguém conseguia resolver o jogo proposto por Sam Loyd.

Loyd foi realmente quem popularizou o quebra-cabeça mas segundo [25] ele foi inventado por Noyes Chapman, um agente de correio de Canastota no estado de Nova Iorque, em 1874. Em [21] temos a informação que Chapman obteve sua patente em Março de 1880.

Este problema chamou tanto a atenção do mundo como o *Cubo de Rubik ou Cubo mágico* cem anos mais tarde. O inventor (Erno Rubik) teve inspiração no quebra cabeça de 15 peças para desenvolver este jogo de raciocínio visto como uma versão tridimensional do quebra cabeça.

É evidente que uma configuração pode ser restaurada para a configuração alvo se ela foi obtida a partir da configuração alvo por meio de deslocamentos permitidos: para obter a solução é preciso apenas percorrer o caminho inverso.

Dada uma configuração inicial, existem regras simples (descritas em vários sites da internet) que levam a uma solução do quebra-cabeças. Nosso problema é encontrar uma solução com o menor número possível de movimentos. Neste trabalho tentaremos resolver este problema utilizando dois dos algoritmos estudados, algoritmo de Dijkstra e algoritmo A^*. Existem na literatura, [7] e [5], váriações do algoritmo A^* que resolvem este problema.

4.1 Definindo o grafo

O problema de encontrar uma sequência de operações tranformando uma configuração em outra é equivalente ao problema de encontrar um caminho em um grafo. Então tratamos este problema como o de busca de caminho de custo mínimo através de um grafo orientado. Esta modelagem para o quebra-cabeça de 15 peças é muito comum e pode ser vista por exemplo em [14], [23], [24].

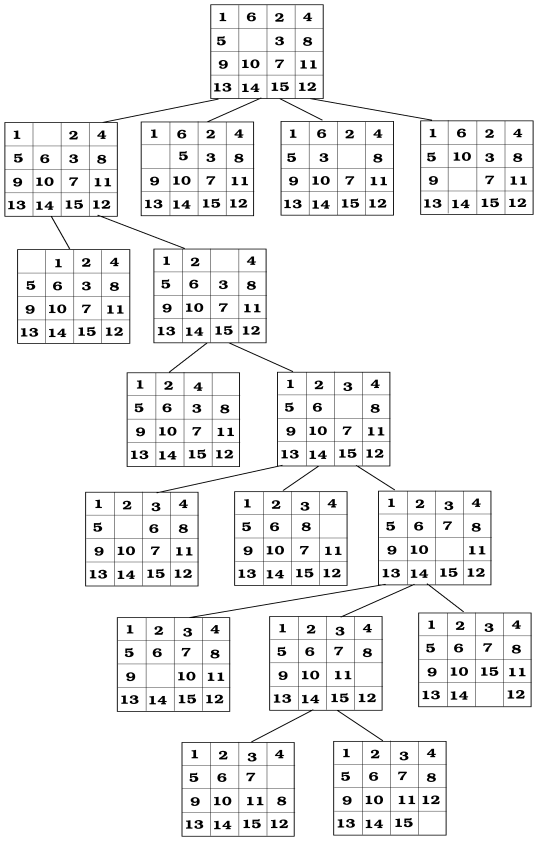


Figura 4.2: Grafo do quebra-cabeça

4.1.1 Nó e Ramo

Um nó do grafo representa uma configuração do quebra-cabeça e cada deslizamento de peça para uma determinada posição corresponde a um ramo.

Como é possível termos $16!$ configurações para o quebra-cabeça, este problema é representado por um grafo G de $16!$ nós. Este grafo é sempre bipartido em duas componentes conexas, uma consistindo das configurações que podem ser restauradas até a configuração alvo e outra das que não podem.

Representamos um nó por um vetor coluna $n \in \mathbb{R}^{16}$. Cada componente do vetor representa uma posição do quebra-cabeça e o valor de cada componente indica a peça que está ocupando tal posição. O número 16 indica qual posição está vazia.

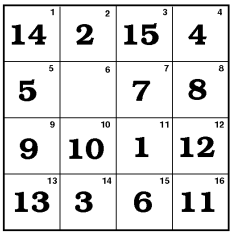


Figura 4.3: Representação do nó

Na figura 4.3 a peça 14 está ocupando a primeira posição do quebra-cabeça, assim o número 14 está na primeira componente do vetor n. Desta forma o nó correspondente a esta configuração é dado por:

$$n = (14, 2, 15, 4, 5, 16, 7, 8, 9, 10, 1, 12, 13, 3, 6, 11)^T.$$

Note que a posição 6 está vazia, logo o número 16 aparece na sexta componente do vetor.

O nó inicial é o nó que representa a configuração que queremos resolver.

4.2 Operador sucessor e custo

Observe que dada uma configuração arbitrária a partir dela com apenas um deslocamento é possível obter duas, três ou quatro configurações. Isto equivale a dizer que dado um nó n ele admite dois, três ou quatro sucessores. Para definir o operador sucessor usamos uma matriz dada, de dimensão 16×4, que chamamos de matriz A.

Na matriz A as 16 linhas indicam as 16 posições vazias possíveis para o quebra-cabeça e as 4 colunas indicam a peça de qual posição pode ser deslizada para a posição vazia. Cada componente $A(i, j),$ com $i = 1, 2, ..., 16$ e $j = 1, 2, 3, 4$, representa as possíveis mudanças para uma peça em função da posição que está vazia. Por exemplo:

- O elemento $A(1, 1)$ representa que a peça que ocupa a posição 2 pode ser deslizada até a posição 1;
- $A(1, 2)$ representa que a peça que ocupa a posição 5 pode ser deslizada até a posição 1;
- Os elementos $A(1, 3)$ e $A(1, 4)$ estão indicando que não é possível fazer mais deslizamentos se a posição que estiver vazia é a posição 1.

$$A = \begin{pmatrix} 2 & 5 & 0 & 0 \\ 1 & 3 & 6 & 0 \\ 2 & 4 & 7 & 0 \\ 3 & 8 & 0 & 0 \\ 1 & 6 & 9 & 0 \\ 2 & 5 & 7 & 10 \\ 3 & 6 & 8 & 11 \\ 4 & 7 & 12 & 0 \\ 5 & 10 & 13 & 0 \\ 6 & 9 & 11 & 14 \\ 7 & 10 & 12 & 15 \\ 8 & 11 & 16 & 0 \\ 9 & 14 & 0 & 0 \\ 10 & 13 & 15 & 0 \\ 11 & 14 & 16 & 0 \\ 12 & 15 & 0 & 0 \end{pmatrix}$$

Definição 4.1. *Definimos um nó $\bar{n}$ sucessor ao nó n pelo seguinte procedimento:*
Suponha que $n(i) = 16$ para um dado $i \in \{1, 2, ..., 16\}$.
Se $A(i, j) \neq 0$ para $j = 1, 2, 3, 4$

$$\bar{n}(i) = n(A(i, j)),$$

$$\bar{n}(A(i, j)) = 16$$

as demais componentes de n permanecem iguais as respectivas componentes de $\bar{n}$.

O operador sucessor $\Gamma(n)$ é definido pela aplicação do procedimento descrito acima a n.

Exemplo: $n = (6, 3, 14, 4, 10, 2, 16, 8, 9, 7, 13, 12, 11, 1, 15, 5)^T$

6	**3**	**14**	**4**
10	**2**		**8**
9	**7**	**13**	**12**
11	**1**	**15**	**5**

Nesta configuração $n(7) = 16$, ou seja, a sétima posição está vazia. Logo

- $A(7,1) = 3$, indica que a peça que ocupa a posição 3 pode ser deslizada até a posição 7, $\bar{n}_1(7) = n(3) = 14$ e $\bar{n}_1(3) = 16$, assim
  $\bar{n}_1 = (6,3,16,4,10,2,14,8,9,7,13,12,11,1,15,5)^T$.

- $A(7,2) = 6$, indica que a peça que ocupa a posição 6 pode ser deslizada até a posição 7, $\bar{n}_2(7) = n(6) = 2$ e $\bar{n}_2(6) = 16$, assim
  $\bar{n}_2 = (6,3,14,4,10,16,2,8,9,7,13,12,11,1,15,5)^T$.

- $A(7,3) = 8$, indica que a peça que ocupa a posição 8 pode ser deslizada até a posição 7, $\bar{n}_3(7) = n(8) = 8$ e $\bar{n}_3(8) = 16$, assim
  $\bar{n}_3 = (6,3,14,4,10,2,8,16,9,7,13,12,11,1,15,5)^T$.

- $A(7,4) = 11$, indica que a peça que ocupa a posição 11 pode ser deslizada até a posição 7, $\bar{n}_4(7) = n(11) = 13$ e $\bar{n}_4(11) = 16$, assim
  $\bar{n}_4 = (6,3,14,4,10,2,13,8,9,7,16,12,11,1,15,5)^T$.

Note que ao gerarmos os sucessores de qualquer um dos nós: $\bar{n}_1$, $\bar{n}_2$, $\bar{n}_3$, $\bar{n}_4$, um destes sucessores é o nó n. Desta forma ao expandirmos um caminho $\eta = (n, c, p)$, excluímos o caminho sucessor cujo nó terminal corresponde ao predecessor de n em η, este caminho sucessor seria sempre eliminado.

Observe também que ao gerarmos um sucessor $\bar{n}$ do nó n estamos apenas deslocando uma peça do quebra-cabeça entre duas configurações. Desta forma o custo associado ao ramo que representa este deslizamento é 1.

4.3 Caminho e custo de um caminho

Um caminho $P = (n_1, n_2, ..., n_p)$ no grafo corresponde a uma sequência de movimentos para a partir da configuração representada pelo nó n_1 se chegar na configuração representada pelo nó n_p. O custo do caminho P é $p - 1$, pois para se chegar em n_p a partir de n_1 foram precisos $p - 1$ deslocamentos.

4.4 Sub-estimativa

Procuramos uma sub-estimativa para o custo de um caminho até o alvo a partir de um determinado nó n que seja a maior possível.

Um sub-estimativa conhecida se baseia na *distância de Manhattan* também conhecida como *distância pombalina ou distância de táxi* e foi desenvolvida por Hermann Minkowski [26] no século 19. Ela recebeu esse nome pois define a menor distância possível que um carro é capaz de percorrer numa malha urbana reticulada ortogonal, tal como se encontram em zonas como Manhattan ou a Baixa Pombalina.

Para um sistema de coordenadas fixo, a distância de Manhattan entre dois pontos S e Q é simplesmente $\parallel S - Q \parallel_1$.

Pensando em uma configuração do quebra-cabeça, consiste em determinar quantos movimentos seriam necessários para mover cada peça da posição que ela ocupa até a posição em que desejamos que ela esteja, se fosse possível levantá-la com a mão e mover.

Definiremos agora a estimativa de custo de caminho entre um nó n e o alvo T, onde T não necessariamente é o nó que representa a configuraçao da figura 4.1, que dizemos ser o alvo padrão.

Para isso representamos as 16 posições do quebra-cabeça (as 16 componentes do vetor n) por 16 pontos em um sistema de coordenadas: a posição 1 corresponde ao ponto $S_1 = (1, 1)$, a posição 15 corresponde ao ponto $S_{15} = (4, 3)$.

1 (1,1)	2 (1,2)	3 (1,3)	4 (1,4)
5 (2,1)	6 (2,2)	7 (2,3)	8 (2,4)
9 (3,1)	10 (3,2)	11 (3,3)	12 (3,4)
13 (4,1)	14 (4,2)	15 (4,3)	16 (4,4)

Dados dois nós n e t, com n representado por 16 pontos S_i, com $i = 1, 2, ..., 16$, e $t \in T$ representado por 16 pontos Q_i, com $i = 1, 2, ..., 16$, cada peça $n(i)$ ocupa uma posição que corresponde a um ponto S_i em n, gostaríamos de deslocá-la até a posição que corresponde a um ponto Q_j tal que $t(j) = n(i)$, com $j = 1, 2, ..., 16$, em t. Logo,

Definição 4.2. *A estimativa de custo $\hat{h}$ de um caminho de custo mínimo entre n e um alvo t é:*

$$\hat{h}(n,t) = \sum_{\substack{i=1 \\ n(i)\neq 16 \\ t(j)=n(i)}}^{16} \parallel S_i - Q_j \parallel_1,\ j = 1, 2, ..., 16 \tag{4.1}$$

A estimativa de custo entre o nó n e o alvo T é dada por:

$$\hat{h}(n,T) = min_{t \in T}\{\hat{h}(n,t)\} \tag{4.2}$$

Exemplo:

9	3	1	7
6	5	2	4
13		12	8
14	11	10	15

(a) n

2	3	1	4
5	6	7	8
9	10	11	12
13	14	15	

(b) t

Figura 4.4: Estimativa entre o nó n e o nó t

Para n e t acima tem-se que:

- A peça $n(1) = 9$ está no ponto S_1 gostaríamos que estivesse em Q_9, $\| S_1 - Q_9 \|_1 = 2$;
- A peça $n(2) = 3$ está no ponto S_2 gostaríamos que estivesse em Q_2, $\| S_2 - Q_2 \|_1 = 0$;

Prosseguindo desta maneira chega-se a $\hat{h}(n,t) = 19$.

Se o alvo for dado pelo nó que representa a configuração da figura 4.1 utilizamos uma matriz D, de dimensão 16×16, para obter esta estimativa. Neste caso desejamos sempre que a peça i esteja na posição i, $i = 1, 2, ..., 15$. Os elementos $D(i,j)$ indicam a distância da peça j (que deveria estar na posição j) até a posição i.

$$D = \begin{pmatrix} 0 & 1 & 2 & 3 & 1 & 2 & 3 & 4 & 2 & 3 & 4 & 5 & 3 & 4 & 5 & 6 \\ 1 & 0 & 1 & 2 & 2 & 1 & 2 & 3 & 3 & 2 & 3 & 4 & 4 & 3 & 4 & 5 \\ 2 & 1 & 0 & 1 & 3 & 2 & 1 & 2 & 4 & 3 & 2 & 3 & 5 & 4 & 3 & 4 \\ 3 & 2 & 1 & 0 & 4 & 3 & 2 & 1 & 5 & 4 & 3 & 2 & 6 & 5 & 4 & 3 \\ 1 & 2 & 3 & 4 & 0 & 1 & 2 & 3 & 1 & 2 & 3 & 4 & 2 & 3 & 4 & 5 \\ 2 & 1 & 2 & 3 & 1 & 0 & 1 & 2 & 2 & 1 & 2 & 3 & 3 & 2 & 3 & 4 \\ 3 & 2 & 1 & 2 & 2 & 1 & 0 & 1 & 3 & 2 & 1 & 2 & 4 & 3 & 2 & 3 \\ 4 & 3 & 2 & 1 & 3 & 2 & 1 & 0 & 4 & 3 & 2 & 1 & 5 & 4 & 3 & 2 \\ 2 & 3 & 4 & 5 & 1 & 2 & 3 & 4 & 0 & 1 & 2 & 3 & 1 & 2 & 3 & 4 \\ 3 & 2 & 3 & 4 & 2 & 1 & 2 & 3 & 1 & 0 & 1 & 2 & 2 & 1 & 2 & 3 \\ 4 & 3 & 2 & 3 & 3 & 2 & 1 & 2 & 2 & 1 & 0 & 1 & 3 & 2 & 1 & 2 \\ 5 & 4 & 3 & 2 & 4 & 3 & 2 & 1 & 3 & 2 & 1 & 0 & 4 & 3 & 2 & 1 \\ 3 & 4 & 5 & 6 & 2 & 3 & 4 & 5 & 1 & 2 & 3 & 4 & 0 & 1 & 2 & 3 \\ 4 & 3 & 4 & 5 & 3 & 2 & 3 & 4 & 2 & 1 & 2 & 3 & 1 & 0 & 1 & 2 \\ 5 & 4 & 3 & 4 & 4 & 3 & 2 & 3 & 3 & 2 & 1 & 2 & 2 & 1 & 0 & 1 \\ 6 & 5 & 4 & 3 & 5 & 4 & 3 & 2 & 4 & 3 & 2 & 1 & 3 & 2 & 1 & 0 \end{pmatrix}$$

Lembrando a definição 4.2, onde padronizamos $h(n)$ sendo o custo de um caminho de custo mínimo entre n e o alvo padrão, define-se:

Definição 4.3. *A estimativa de custo de um caminho entre um nó n até o alvo é:*

$$\hat{h}(n) = \sum_{i=1, n(i)\neq 16}^{16} D(i, n(i)) \tag{4.3}$$

Exemplo:

2	**3**	**1**	**4**
5	**6**	**7**	**8**
9	**10**	**11**	**12**
13	**14**	**15**	

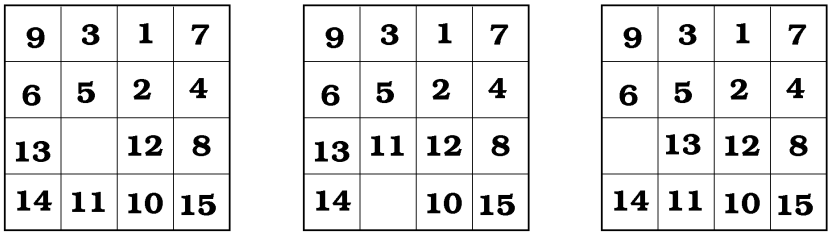
$$n = (2, 3, 1, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16)'$$

- Para $i = 1$, $n(1) = 2$ na posição 1 está a peça 2, temos que $D(1, 2) = 1$.

- Para $i = 2$, $n(2) = 3$ na posição 2 está a peça 3, temos que $D(2, 3) = 1$.
- Para $i = 3$, $n(3) = 1$ na posição 3 está a peça 1, temos que $D(3, 1) = 2$.
- Para i de 4 a 16 $n(i) = i$ e $D(i, n(i)) = 0$.

Portanto a estimativa de custo de um caminho a partir do nó $\bar{n}$ até o alvo é 4.

Observação 4.4. *É importante observar que a estimativa de um nó n e de seu nó sucessor $\bar{n}$ varia em 1 ou -1 para um alvo fixo.*



(a) $\hat{h}(n) = 21$ (b) $\hat{h}(\bar{n}) = 20$ (c) $\hat{h}(\bar{n}) = 22$

Figura 4.5: Variação da estimativa

Para um alvo T a variação é -1, 0 ou 1.

Na seção 2.6.2 definimos estimativa consistente e constatamos que com estimativas consistentes se estabelece propriedades análogas entre o algoritmo de Dijkstra e o algoritmo A^*.

Teorema 4.5. *A sub-estimativa dada pela definição 4.2 é consistente.*

Demonstração:
Seja $P = (n = n_0, n_1, ..., n_p = n^{'})$ um caminho de custo ótimo entre n e $n^{'}$. Pela observação acima temos que $\hat{h}(n_{i+1}, t) \geq \hat{h}(n_i, t) - 1$ para todo $t \in T$, e portanto

$\hat{h}(n_{i+1}, T) \geq \hat{h}(n_i, T) - 1$.

Desta forma $\hat{h}(n^{'}, T) \geq \hat{h}(n, T) - p$, mas $h(n, n^{'}) = p$ já que P é um caminho de custo mínimo.
Portanto $\hat{h}(n^{'}, T) + h(n, n^{'}) \geq \hat{h}(n, T)$, o que prova que a sub-estimativa é consistente.

□

Então, de acordo com o teorema 2.23, uma vez que um caminho é fechado já se encontrou um caminho de custo mínimo até seu nó terminal. Sendo assim, nenhum caminho pode ser reaberto pelo algoritmo A^*.

Tendo uma sub-estimativa consistente para o quebra-cabeça pode-se usar o algoritmo A^* para "tentar" resolver este problema.

4.5 Estrutura de dados

Neste problema a lista aberto cresce rapidamente pois o grafo é muito grande. Para se ter uma idéia: para armazenarmos todas as configuração do quebra-cabeça que estão na mesma componente conexa que a configuração alvo seriam necessários 150000 Gigabytes de memória de computador, ou, se pudessemos escrever cada configuração em 1 cm^2 de papel, seria necessário uma área maior que a área da Ilha de Santa Catarina para armazenarmos todas as configuração que podem ser resolvidas. Na implementação apenas os caminhos gerados pelo algoritmo são armazenados em uma estrutura $\{\eta_i\}_{i=1,\ldots,q_k}$, em que q_k é o número de elementos listados na iteração k.

Há duas operações de alto custo computacional em cada iteração, para as quais utilizamos estruturas de dados apropriadas:

1. busca de um aberto de custo (ou estimativa) mínimo, que utiliza a estrutura *heap*;
2. eliminação ou introdução de um sucessor na lista, que utiliza a estrutura *hashing*.

Os custos, ou as estimativas de custos, e um indicador para um caminho da lista Aberto são armazenados em um *Heap*. Os custos, ou as estimativas, representam as chaves do *Heap* sendo os itens os indicadores para os caminhos. Os itens são dispostos de maneira que um caminho com chave mínima esteja na raiz do heap.

Assim com apenas uma consulta ao *Heap* pode-se extrair um caminho com mínima chave, e é este caminho que escolhemos para expandir. Após extrair este elemento de um maneira muito rápida se reordena o *Heap*.

Uma breve explicação sobre *Heap* encontra-se no Apêndice A.

No processo de eliminações é preciso varrer listas para verificar se os caminhos já foram listados. Fazemos isto usando um vetor de *Hashing*.

A cada caminho $\eta_i = (n_i, c_i, p_i)$ uma função de espalhamento (*Hash*) associa um valor $H(n_i)$. O *Hashing* é um vetor inicialmente com componentes nulas. Na linha deste vetor correspondente ao valor $H(n_i)$ guarda-se o valor i. Assim sempre que um caminho $\bar{\eta} = (\bar{n}, \bar{c}, \bar{p})$ for gerado, a componente i do *Hashing* correspondente a $H(\bar{n})$ nos dá a seguinte informação: se a componente é nula, o nó não está listado; senão, está listado um nó n com $H(n) = H(\bar{n})$ é é necessário comparar n e $\bar{n}$ diretamente.

Uma explicação sobre *Hash* pode ser vista em [11].

4.6 Testes numéricos

Como dados relevantes a observar guardamos os seguintes números: caminhos abertos, caminhos fechados, caminhos eliminados, colisões feitas pela função de *hashing*. Guardamos também a quantidade de nós gerados pelo operador sucessor, a estimativa de custo de um caminho através do nó inicial e o alvo.

Os próximos dois exemplos obtivemos deslizando as peças do quebra-cabeça a partir da configuração alvo pois sabemos que eles têm solução pois estão na mesma componente conexa que a configuração alvo e, deslizando as peças temos uma super-estimativa para o custo de um caminho de custo ótimo. Para estes testes adotamos como 100000 o número máximo de iterações, a ausência do número de passos indica que o algoritmo termina sem encontrar uma solução.

Estes testes foram feitos em Matlab em um computador com um processador de 3Ghz e 512 MB de memória.

Exemplo 4.6.

9	3	1	7
6	5	2	4
13		12	8
14	11	10	15

Algoritmo	Abertos	Fechados	Eliminados	Sucessores	Passos	Colisões	$\hat{f}$
Dijkstra	197085	100001	13709	210793	-	2314	-
A^*	2108	1040	28	2141	27	1	21

Algoritmo	Tempo total
Dijkstra	68,3 minutos
A^*	2,2 segundos

Exemplo 4.7.

3	11	15	4
14	9	7	
2	6	5	8
1	13	10	12

Algoritmo	Abertos	Fechados	Eliminados	Sucessores	Passos	Colisões	$\hat{f}$
Dijkstra	198782	100001	12434	211215	-	2326	-
A^*	10016	4972	230	10267	36	5	28

Algoritmo	Tempo total
Dijkstra	68,8 minutos
A^*	17,2 segundos

Os próximos exemplos foram obtidos fazendo-se um número par de trocas entre as peças do quebra-cabeça pois desta forma as configurações apresentadas estão na mesma componente conexa que a configuração alvo e assim possuem solução.

Exemplo 4.8. *Duas trocas:*

2	3	1	4
5	6	7	8
9	10	11	12
13	14	15	

Algoritmo	Abertos	Fechados	Eliminados	Sucessores	Passos	Colisões	$\hat{f}$
Dijkstra	198075	100001	13469	211543	-	2380	-
A^*	3536	1755	122	3684	20	0	4

Algoritmo	Tempo total
Dijkstra	66,4 minutos
A^*	4 segundos

Nos próximos testes aumentamos o número máximo de iterações para 200000.

Exemplo 4.9. *Quatro trocas:*

2	**3**	**1**	**4**
5	**7**	**8**	**6**
9	**10**	**11**	**12**
13	**14**	**15**	

Algoritmo	Abertos	Fechados	Eliminados	Sucessores	Passos	Colisões	$\hat{f}$
Dijkstra	393417	200001	28939	422355	-	9184	-
A^*	36576	18774	2011	38872	28	103	8

Algoritmo	Tempo total
Dijkstra	4,4 horas
A^*	2,7 minutos

Exemplo 4.10. *Seis trocas:*

2	**3**	**1**	**4**
5	**7**	**8**	**6**
9	**11**	**12**	**10**
13	**14**	**15**	

Algoritmo	Abertos	Fechados	Eliminados	Sucessores	Passos	Colisões	$\hat{f}$
A^*	316278	164812	24722	341179	34	6024	12

Algoritmo	Tempo total
A^*	3,1 horas

No próximo exemplo usamos 550000 para o limite de iterações.

Exemplo 4.11. *Oito trocas:*

2	**3**	**1**	**4**
5	**7**	**8**	**6**
9	**11**	**12**	**10**
15	**13**	**14**	

Algoritmo	Abertos	Fechados	Eliminados	Sucessores	Passos	Colisões	$\hat{f}$
A^*	1036695	550001	93101	1129795	-	63827	16

Algoritmo	Tempo total
A^*	54,6 horas

O algoritmo pára com 550000 iterações sem resolver o problema.

4.6.1 Análise dos testes

Ao fim destes testes confirmamos a superioridade do algoritmo A^* em relação ao algoritmo de Dijkstra na tentativa de resolver este problema. É importante salientar que devido ao espaço de busca ser muito grande é indispensável o uso desta informação quanto a proximidade do alvo para se chegar nele com maior rapidez e consequentemente diminuir o esforço computacional.

Embora, em algumas configurações, o algoritmo A^* tem sido muito eficiente, notamos que a medida que aumentamos o número de trocas o número de iterações e o tempo computacional aumentaram significativamente. Além disso encontramos configurações que o algoritmo A^* não conseguiu resolver, exemplo 4.11 onde o algoritmo atingiu 550000 iterações em mais de 54 horas.

Analisando o comportamento da estimativa dos primeiros caminhos fechados observamos que o erro em relação ao custo ótimo é grande, em especial a estimativa do caminho a partir da raiz. Por exemplo, no exemplo 4.8 a estimativa do caminho através de seu nó inicial é 4, enquanto que o custo ótimo é 20, no exemplo 4.9 a estimativa é 8 e o custo ótimo é 28. Sendo assim propomos uma nova estimativa que continue sendo admissível mas que diminua o erro em relação ao custo ótimo.

4.7 Uma nova sub-estimativa

Sugerimos aumentar o alvo, que originalmente é $T = \{t_0\}$, t_0 a configuração padrão, construindo um novo alvo definido por:

$$\tilde{T} = \{n \in \mathcal{N} | h(n, t_0) = k\},$$

com $k \in \mathbb{N}$. Utilizaremos valores de k entre 8 e 16.

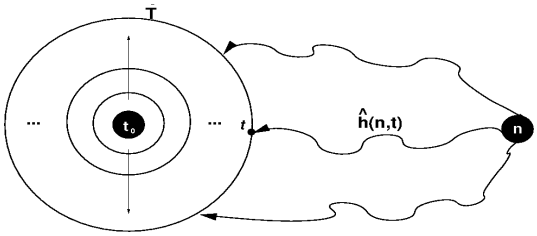


Figura 4.6: Representação da nova estimativa

Para um dado k, esse conjunto pode ser construído pelo algoritmo de Dijkstra, que neste caso equivale a Busca Horizontal. Utilizamos o algoritmo até fechar o último elemento de custo $k-1$. Neste momento os nós abertos são a frente de onda de custo k, que coincide com $\tilde{T}$.

Seja $\tilde{\mathcal{N}} = \{n \in \mathcal{N} | h(n,t_0) < k\}$. Se $s \in \tilde{\mathcal{N}}$ ou $s \in \tilde{T}$, então o problema de otimização é simples: basta procurar s entre os nós listados pelo algoritmo de Dijkstra e lá se encontrará um caminho ótimo de t a s (que invertido é um caminho ótimo de s a t).

Vamos supor que o nó inicial s é tal que $s \in \mathcal{N} - \tilde{\mathcal{N}}$. Então $h(s) \geq k$.

Definição 4.12. *A estimativa de custo $\tilde{h}$ associada a cada nó $n \in \mathcal{N} - \tilde{\mathcal{N}}$ é:*

$$\tilde{h}(n) = min_{t \in \tilde{T}}\{\hat{h}(n,t)\} \tag{4.4}$$

Os seguintes fatos são imediatos:

- $\tilde{h}(.)$ é admissível, pois para qualquer $t \in \tilde{T}$, $\tilde{h}(n) \leq \hat{h}(n,t) \leq h(n,t)$.
- $\tilde{h}(.)$ é consistente. Isso se demonstra exatamente como no teorema 4.5.
- $h(n,t_0) = k + h(n,\tilde{T})$ para todo $n \in \mathcal{N} - \tilde{\mathcal{N}}$. Todo caminho ótimo de s a t tem seus k últimos nós na lista gerada pelo algoritmo de Dijkstra.
- $k + \tilde{h}(n)$ é sub-estimativa para $h(n,t_0)$.

O algoritmo

Roda-se o algoritmo A^* com alvo $\tilde{T}$ e estimativa $\tilde{h}(.)$. Ao se fechar um caminho $\bar{\eta} = (\bar{n}, \bar{c}, \bar{p})$ com $\bar{n} \in \tilde{T}$, obtém-se um caminho ótimo de custo $h(s) = k + \bar{c}$, recuperando o caminho de s a $\bar{n}$ associado a $\bar{\eta}$ e o caminho de $\bar{n} \in \tilde{T}$ a t_0 com k nós a partir da lista construída por Dijkstra.

4.7.1 Novos testes

Iniciamos guardando alguns conjuntos $\tilde{T}$. Analisamos o tempo e o número de iterações gastos para gerar este conjunto:

k	Alvos	Fechados	Tempo
8	446	414	1 seg
9	946	860	1,9 seg
10	1948	1806	3,8 seg
11	3938	3754	9,8 seg
12	7808	7692	29,8 seg
13	15544	15500	2,1 min
14	30821	31044	7,1 min
15	60842	61865	27 min
16	119000	122707	1,7 horas

Observação 4.13. *É importante ressaltar que estes conjuntos obtidos acima não são gerados toda vez que vamos resolver um problema, eles são gerados uma única vez e em seguida guardados na memória do computador. Sendo assim para analisarmos o tempo para resolver o problema é necessário apenas olhar para o tempo gasto pelo algoritmo* A^*.

Para comparar com as dados obtidos na seção 4.6 guardamos o número de caminhos fechados, o tempo total gasto pelo algoritmo (A^* e suas funções: listagem de caminho, heap, hashing,...), o valor da estimativa de custo de cada caminho a partir do nó inicial e, considerando que o cálculo da estimativa é muito mais complexo, o tempo gasto para a estimativa.

Para cada exemplo analisamos diferentes conjuntos de alvos. A primeira linha da tabela repete os resultados obtidos pelo algoritmo A^* na seção 4.6.

Exemplo 4.6, custo ótimo 27.

k	Fechados	$\tilde{f}$	Estimativa	Total
1	1040	21	-	2,2 seg
9	208	23	0,2 seg	2 seg
10	173	21	0,2 seg	2 seg
11	132	23	0,2 seg	2 seg
12	133	23	0,3 seg	3 seg

Exemplo 4.7 custo ótimo 36.

k	Fechados	$\tilde{f}$	Estimativa	Total
1	4972	28	-	17,2 seg
10	1985	30	1,6 seg	8 seg
11	975	30	1,1 seg	5 seg
12	898	30	1,4 seg	6 seg
13	772	32	1,7 seg	6 seg

Exemplo 4.8 custo ótimo 20.

k	Fechados	$\tilde{f}$	Estimativa	Total
1	1755	4	-	4 seg
8	507	16	0,4 seg	3 seg
9	122	16	0,2 seg	1 seg
10	83	16	0,2 seg	1 seg
11	59	16	0,2 seg	2 seg
12	39	16	0,1 seg	3 seg
15	12	20	0,2 seg	7 seg

Exemplo 4.9 custo ótimo 28.

k	Fechados	$\tilde{f}$	Estimativa	Total
1	18774	8	-	2,7 min
10	2500	18	2,1 seg	11 seg
11	2207	20	2,6 seg	10 seg
12	1647	20	2,4 seg	8 seg
13	644	20	1,6 seg	6 seg
14	552	20	2,3 seg	8 seg

Exemplo 4.10 custo ótimo 34.

k	Fechados	$\tilde{f}$	Estimativa	Total
1	164812	12	-	3,1 h
10	17832	20	13 seg	2,9 min
11	20028	22	19 seg	3,8 min
12	16694	22	21,4 seg	2,9 min
13	11154	22	22,4 seg	1,6 min
14	4522	22	16,5 seg	36 seg
15	4056	24	31 seg	52 seg

Exemplo 4.11

k	Fechados	$\tilde{f}$	Estimativa	Total
13	39315	26	1,3 min	13,8 min
14	44932	26	2,5 min	18,5 min
15	36863	28	4,4 min	15,4 min
16	27442	28	8,8 min	15,3 min

Este problema é resolvido em 40 passos.

Exemplo 4.14. *oito trocas*

	15	**14**	**13**
12	**11**	**10**	**9**
8	**7**	**6**	**5**
4	**3**	**2**	**1**

k	Fechados	$\tilde{f}$	Estimativa	Total
13	200001	64	5.6 min	3,4 horas
14	200001	64	10.6 min	4,01 horas
15	200001	64	21.9 min	4,2 horas
16	200001	64	57.7 min	4,9 horas

Com 200000 iterações este problema não foi resolvido.

4.7.2 Análise dos testes

Comparando com as testes feitos na seção 4.6 vemos que com a nova estimativa, o desempenho do algoritmo A^* teve uma melhora bem considerável, principalmente nos exemplos obtidos fazendo-se trocas. Isto se deve ao fato que o erro entre o custo ótimo e a estimativa de custo do caminho entre o nó inicial e o alvo ter diminuído, já que para estas configurações iniciais $\hat{f}(\eta)$ é bem menor que o custo ótimo do caminho representado pelo caminho inicial η.

Outro dado importante a comentar é o número de caminhos Fechados (equivalente ao número de iterações). Como a estimativa está mais próxima do custo ótimo o algoritmo tende a fechar um número menor de caminhos que não fazem parte de caminhos ótimos, esta é uma explicação para o número de caminhos fechados ter diminuído em relação ao testes da seção 4.6, contribuindo assim para o melhora do tempo computacional.

Note também que a medida que aumentamos o custo de $\tilde{T}$ o tempo gasto para calcular $\tilde{h}(.)$ aumenta, isto porque o número de nós em $\tilde{T}$ cresce bastante ao aumentar o custo de $k-1$ para k.

Entretanto a melhora não foi tão boa como esperávamos, pois ainda encontramos configurações que não conseguimos obter a solução. No exemplo 4.14 com oito trocas, testamos diferentes custos para o alvo com 200000 iterações e para nenhum deles conseguimos obter solução.

Conclusão

Este trabalho propiciou-me uma boa oportunidade para entender como são feitas as modelagens em problemas de grafos. Percebi que o estudo do modelo deve ser feito com muito cuidado pois a resolução do problema depende da qualidade do modelo.

Em relação aos algoritmos estudados e implementados, os resultados obtidos com os problemas propostos vieram confirmar o que havíamos visto na teoria. Confirmamos que se é possível obter uma estimativa de custo para um problema, nenhum algoritmo de rotulação supera o algoritmo A^*, pois ele realiza menos expansões.

Concluo também que devido ao fato de que o tamanho das listas pode ficar muito grande é necessário sabermos administrá-las usando ferramentas essenciais, por isso o uso do *Heap* e do *Hashing* foi fundamental. Desta forma creio ser indispensável o conhecimento de estruturas de dados que facilitem o trabalho diminuindo o esforço computacional.

As aplicações que foram feitas possibilitaram-me entender como é construído um modelo e quão rica a Matemática pode ser ao descrevê-los. No problema de alinhamento de proteínas não obtive grandes surpresas quanto ao desempenho dos algoritmos, pois esperava pela superioridade do algoritmo A^*. No problema do quebra-cabeça percebi como um problema pode ficar grande e entendi porque o algoritmo A^* não é suficiente para resolver este problema.

Apêndice A

Heap

Um *heap* é uma arborescência com as seguintes propriedades:

A cada nó x associam-se:

- Um item i (elemento de algum conjunto pré-definido como vetores ou estruturas).
- Uma chave $c(i) \in \mathbb{R}$ (custos).
- Se um nó y contendo o item j é sucessor de x então $c(j) \geq c(i)$.

Da definição acima conclui-se que a chave da raiz do heap é mínima. Mais ainda, a raiz de qualquer sub-arborescência do heap tem chave mínima nesta sub-arborescência.

Heaps são usados para ordenar listas, e são muito eficientes quando se necessita manter atualizada a informação sobre itens de custo mínimo em uma lista, como ocorre nesta dissertação: a cada iteração dos algoritmos escolhe-se um elemento aberto de mínimo custo ou estimativa.

Como de costume vamos associar a um nó do grafo à tripla $\eta = (n, c, p)$ em que p é um apontador para o antecessor do nó na arborescência. [1]

d-Heap

Um d-heap é um heap em que cada nó possui no máximo d sucessores, $d \in \mathbb{N}$. O caso mais comum, que utilizaremos neste trabalho, e d=2 (árvore binária).

[1] No texto η representa um caminho da raiz ao nó. Aqui, como o grafo é uma arborescência, caminhos e nós são bi-univocamente associados.

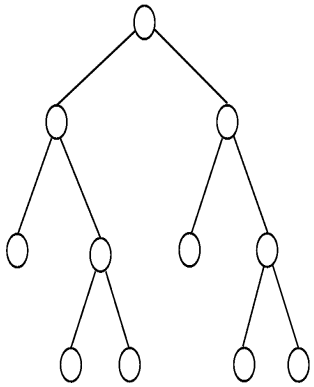


Figura A.1: 2-Heap

Um 2-heap é armazenado em um vetor *(array)* com componentes $h(j)$, $j = 1, 2, ..., 2^p$: os sucessores do nó $h(j)$ são $h(2j)$ e $h(2j + 1)$.

Procedimentos básicos de reodenação: siftup e siftdown

Dado um heap, suponha que $\eta = (i, c(i), p)$ é um nó. Suponha que por alguma razão mudamos o valor de $c(i)$ de modo a romper a ordenação exigida para o heap. Podem ocorrer dois casos que necessitam de reordenação:

1. Seja $\bar{\eta} = (j, c(j), \bar{p})$ o antecessor de η (apontado por η). Se $c(j) \geq c(i)$, deve-se trocar os itens nos dois nós, empurrando para cima o item i que está barato. Agora $\eta = (j, c(j), p)$ não viola a regra de ordenação, mas $\bar{\eta} = (i, c(i), \bar{p})$ pode violar e o procedimento deve ser executado novamente.

   Constrói-se o seguinte procedimento recursivo para $\eta = (i, c(i), p)$
   $siftup(\eta)$:
   se $p = 0$, pare (o nó é raiz).
   recupere o nó $\bar{\eta} = (j, c(j), \bar{p})$ antecessor de η.
   se $c(j) > c(i)$

   $\bar{\eta} = (i, c(i), \bar{p})$

   $\eta = (j, c(j), p)$

   $siftup(\bar{\eta})$

2. Seja $\bar{\eta} = (j, c(j), \bar{p})$ o sucessor de menor custo de η. Se $c(i) > c(j)$, devem-se trocar os ítens dos dois nós, empurrando para baixo o item i que está caro. Procedimento:

$siftdown(\eta)$:
se η é folha do grafo, pare.
recupere o nó $\bar{\eta} = (j, c(j), \bar{p})$ de menor custo entre os sucessores de η.
se $c(j) < c(i)$

$\bar{\eta} = (i, c(i), \bar{p})$

$\eta = (j, c(j), p)$

$siftdown(\bar{\eta})$

Com esses procedimentos pode-se operar no heap:

- Inserir um novo item i: precisamos criar um novo nó $\bar{\eta} = (i, c(i), \bar{p})$:

  Basta escolher arbitrariamente um nó da arborescência que não tem ainda o número máximo de sucessores (em um 2-heap, adiciona-se uma componente nova no final do *array*). Seja $\bar{p}$ um apontador para esse nó.

  adiciona-se o nó $\bar{\eta} = (i, c(i), \bar{p})$ ao grafo

  $siftup(\bar{\eta})$

- Descartar (deletar) um item i: seja $\eta = (i, c(i), p)$ o nó que contém o item i.
  Se $\bar{\eta}$ é uma folha, basta eliminá-lo da árvore.
  Senão, escolhe-se uma folha qualquer $\bar{\eta} = (j, c(j), \bar{p})$ e faz-se:

  eliminar $\bar{\eta}$ da árvore

  $\eta = (j, c(j), p)$

  $siftdown(\eta)$

Complexidade: em um 2-heap com profundidade p guardam-se $2^p - 1$ itens. Cada inserção faz um máximo de p passos ***(siftup)*** consistindo de comparação e troca de elementos.

Em termos do número de itens, cada inserção tem complexidade $O(log_2 p)$. Para ordenar uma lista de p itens fazendo p inserções a partir de um heap vazio, fazem-se $O(p.log_2 p)$ operações.

Para descartar itens, a complexidade é semelhante.

Este estudo foi baseado em [16].

Referências Bibliográficas

[1] ANDREANI, R., MARTÍNEZ, J. M., MARTÍNEZ, L., YANO F., *Continuous Optimization Methods for Structural Alignments,* To appear im Mathematical Programming, 2006. Currently available at http://www.ime.unicamp.br/ martinez/

[2] ANDREANI, R., MARTÍNEZ, J. M., MARTÍNEZ, L., *Low Order Value Optimization Methods for Protein Alignment.* To be published, 2006. Currently available at http://www.ime.unicamp.br/ martinez/

[3] ANDREANI, R., MARTÍNEZ, J. M., MARTÍNEZ, L., *Protein structal alignment: Interpretation as a Low Order Value Optimization problem ans resulting practial algorithms.* Currently available at http://www.ime.unicamp.br/ martinez/

[4] ARCHER, A. F., *A Modern Treatment of the 15 puzzle*, The American Mathematical Montly, november 1999.

[5] CULBERSON, J., C., SCHAEFFER, J., *Efficiently Searching the 15-Puzzle,* Departament of Computing Science, University of Alberta, 1994.

[6] KARLEMO, F.,R.,W., OSTERGARD, P., R., J., *On Sliding block puzzles,* Journal of Combinatorial Mathematics and Combinatorial Computing, vol 34, 97-107, 2000.

[7] KORF, R., *Depth-First Iterative-Deepening an Optimal Admissible Treesearch,* Artificial Inteligence, vol. 27, n^o 1: 97-109, 1985.

[8] KORF, R., FELNER, A., *Disjoint Pattern Database Heuristics,* Artificial Inteligence, vol. 134, n^o 1-2: 9-22, 2002.

[9] HART, P., NILSSON, N., RAPHAEL, B., *A formal basis for the Heuristic Determination of Minimun Cost Paths,* IEEE Trans. Syst. Cybernetics 4, N^o 2: 100-107, julho, 1968.

[10] SUBBIAH, S., LAURENTS, D. V., LEVITT, M., *Structural similarity of DNA-binding domains of bacteriophage repressors and globin core* Curr. Biol. 1993;3:141-148.

[11] AHO, A., V., H0PCROFT, J., E., ULLMANN, J., D., *Data Structures ans Algorithms*, Addison-Wesley, Boston, 1983.

[12] GONZAGA, C., *Busca de Caminhos em Grafos e Aplicações*, I Reunião de Matemática Aplicada, IBM, Rio de Janeiro, 1978.

[13] GONZAGA, C., *Notas de aula de Pesquisa Operacional*, Universidade Federal de Santa Catarina, Florianópolis, 2005.

[14] NILSSON, N., *Problem-Solving Methods in Artificial Intelligence*, McGRAW-HILL, 1971.

[15] RABUSKE, M. A., *Introdução à teoria de grafos*, Editora da UFSC, 1992.

[16] TARJAN, R., *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, 1983.

[17] GONZAGA, C., *Estudo de Algoritmos de Busca em Grafos e sua Aplicação a Problemas de Planejamento*, Tese de Doutorado, Dpto. de Engenharia de Sistemas e Computação, COPPE, Universidade Federal do Rio de Janeiro, 1973.

[18] KAGOIKI, K. C., *Estudo e implementação de algoritmos de busca em grafos*, Monografia de Graduação, Dpto. de Matemática, Universidade Federal de Santa Catarina, 2005.

[19] OLIVEIRA, D. C., *Alinhamento de Sequências*, Monografia de Graduação, Dpto. de Ciência da Computação, Universidade Federal de Lavras, 2002.

[20] SILVA, S. G. O., *Previsão da Estrutura Secundária de Proteínas utilizando Redes Neurais*, Dissertação de Mestrado, Dpto. de Informática, Universidade de Lisboa, 1999.

[21] http://bd.thrijswijk.nl/15puzzle/15puzzen.htm, acessado em 17/12/2006.

[22] http://www.cut-the-knot.org/pythagoras/history15.shtml, acessado em 20/06/2006.

[23] www.eternallyconfuzzled.com/brain.html, acessado em 20/06/2006.

[24] http://mathworld.wolfram.com/15Puzzle.html, acessado em 20/06/2006.

[25] http://en.wikipedia.org/wiki/N-puzzle, acessado em 20/06/2006.

[26] http://pt.wikipedia.org/wiki/Geometria_pombalina, acessado em 28/12/2006.