Karla Christina da Costa Kagoiki

Estudo e implementação de algoritmos de busca em grafos

Universidade Federal de Santa Catarina

Orientador: Prof. Dr. Clóvis Caesar Gonzaga

Florianópolis

Novembro de 2005

Karla Christina da Costa Kagoiki

Estudo e implementação de algoritmos de busca em grafos

Trabalho de Conclusão de Curso
apresentado ao Curso de Matemática
Habilitação Licenciatura
Departamento de Matemática
Centro de Ciências Físicas e Matemáticas

Universidade Federal de Santa Catarina
Florianópolis
Novembro de 2005

Ao meu esposo, Franco Yukio.

Ao meu pai, Luiz.

À minha mãe, Vânia.

Agradecimentos

Aos amigos Grasielli, João, Mael e Rodrigo, que me deram "suporte técnico"e todo o apoio, acreditando na minha capacidade. Que estejamos sempre juntos.

Às minhas irmãs, Kelly e Sabrina, que sempre me dão o carinho que preciso para continuar a caminhar.

Ao meu professor-tutor-coordenador-amigo Pinho, que me deu oportunidade de aprender no PET e no Projeto das Olimpíadas de Matemática, por suas palavras de incentivo e por nunca falhar, mesmo nas missões mais difíceis.

Às minhas madrinhas, Iara e Sílvia D'Ávila, pela amizade, confiança, conversas e esclarecimentos durante todos esses anos.

Ao professor do Departamento de Matemática, Eliezer Batista, que despertou boa parte do meu conhecimento geométrico e por sua valiosa amizade.

Aos meus amigos do PET Matemática, especialmente Felipe e Louise, por todo o companheirismo e momentos de alegria.

Aos meus sogros, Noriaki e Miyochi, que me acolheram como a uma filha e sempre me animaram para seguir em frente.

Ao meu orientador, Clóvis Gonzaga, que com muita paciência me orientou e mostrou um novo horizonte de conhecimento.

Sumário

Introdução

Este trabalho consiste no estudo de algoritmos de busca de caminhos de custo mínimo em grafos dirigidos e sua implementação em Matlab.

Estudaremos grafos com grande número de nós, especificados por meio da descrição do conjunto de nós e do operador sucessor, que associa a cada nó os nós atingíveis a partir dele atravessando um ramo. Essa especificação é conveniente em problemas com grande número de nós e aplica-se muito bem a problemas de decisões seqüênciais. Os grafos têm custos associados aos ramos e supõe-se que não há circuitos de custo negativo (pois nesse caso o problema é NP-completo). Deve-se encontrar um caminho de custo mínimo entre um nó *inicial* dado e um conjunto de nós *alvo*.

O algoritmo básico para esses problemas é o método de Dijkstra, que visita o menor número possível de nós até atingir uma solução ótima. Vamos apresentá-lo, implementá-lo e testá-lo com o problema de quebra-cabeças de 9 peças.

Vamos apresentar situações em que além dos custos associados aos ramos dispõe-se de sub-estimativas para o custo de um caminho ótimo entre cada nó do grafo e o alvo. Essas sub-estimativas (também chamadas de distâncias heurísticas) são comuns em problemas práticos, sendo obtidas através da resolução de problemas relaxados. As sub-estimativas são usadas para guiar a busca, evitando trabalhar sobre nós muito afastados do alvo. O algoritmo

mais importante de programação heurística é conhecido por Algoritmo A^*.

Embora essas estimativas sejam heurísticas, demonstra-se que se forem sub-estimativas, então os algoritmos encontram sempre uma solução ótima, se existir uma.

Os algoritmos foram implementados para o exemplo do quebra-cabeças de 9 peças.

Capítulo 1

Definições de grafos

Neste capítulo definiremos grafo e apresentaremos algumas definições e propriedades associadas a um grafo.

1.1 Um pouco sobre Leonhard Euler

Leonhard Euler foi provavelmente o maior matemático de todos os tempos. Suíço, nasceu em 15 de abril de 1707, na cidade de Basiléia. Seu pai, Paul Euler, um pastor calvinista, queria que seu filho estudasse teologia com o objetivo de seguir a carreira eclesiástica, apesar de Euler demonstrar um grande talento para a matemática. Euler foi matriculado na Universidade de Basel a fim de estudar teologia e hebraico. Nesse período, foi aluno de Jean Bernoulli. Os filhos de Jean Bernoulli, percebendo que um brilhante matemático estava se transformando no mais medíocre dos teólogos, apelaram ao pai de Euler para que permitisse que seu filho abandonasse o clero. Como no passado Paul Euler havia sido discípulo de Jakob Bernoulli, o patriarca dos Bernoulli, e seu respeito pela família era grande, decidiu que seu filho poderia seguir a carreira dos números.

Euler escreveu mais de quinhentos livros e artigos sobre os mais diversos assuntos da Matemática. Redigia com tanta facilidade que costumava dizer ironicamente que seu lápis era muito mais inteligente que ele. Além da teoria de números, Euler ocupou-se de quase todos os ramos da Matemática Pura e Aplicada, sendo o maior responsável pela linguagem e notações que usamos hoje. Foi o primeiro a empregar a letra e como base do sistema de logaritmos naturais, a letra π para razão entre comprimento e diâmetro da circunferência e o símbolo i para $\sqrt{-1}$. Deve-se a ele também, o uso de letras minúsculas designando lados do triângulo e maiúsculas para seus ângulos opostos. Simbolizou logaritmo de x por $ln\ x$, usou Σ para indicar adição e $f(x)$ para função de x, além de outras notações em Geometria, Álgebra, Trigonometria e Análise, sendo um dos pioneiros também da topologia algébrica e da teoria de grafos.

1.2 O problema das Pontes de Königsberg

Consta que em Konigsberg, Alemanha, um rio que passava pela cidade, tinha uma ilha e logo depois de passar por essa ilha se bifurcava como mostra a figura abaixo:

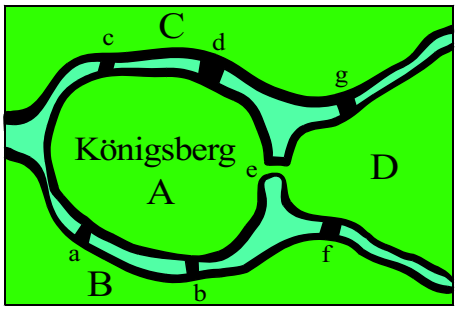


Figura 1.1: Pontes de Konigsberg

Por muito tempo, os habitantes daquela cidade perguntavam-se se era possível cruzar as sete pontes numa caminhada contínua sem passar duas vezes por qualquer uma delas.

Euler percebeu que seria mais simples analisar o problema trocando as áreas de terra por pontos (nós) e as pontes por arcos (ramos).

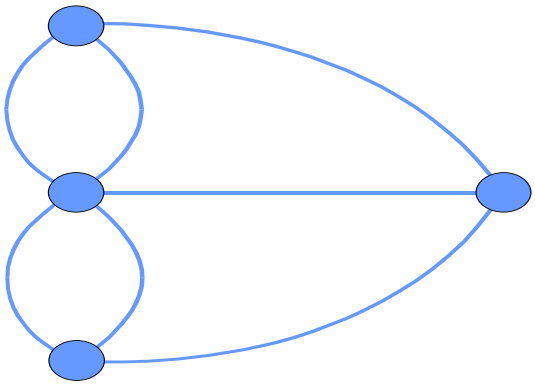


Figura 1.2: Esquema das pontes

O problema agora consiste em percorrer todos os ramos, passando por cada um deles uma única vez, continuamente. Cada nó tem um número ímpar de ramos ligados a ele. Considere um nó com três ramos. A primeira vez que chegarmos a esse nó por um desses ramos, poderemos sair por um dos dois ramos restantes. Mas quando chegarmos a esse mesmo nó pelo ramo restante, não teremos como sair sem reutilizarmos um dos ramos.

Existem duas possibilidades para analisarmos:

- Se esse nó com três ramos for o último, alcançado pela primeira vez ao final da caminhada. Assim, os dois ramos restantes serviriam para atravessar uma ponte e voltar pela outra.

- Se começarmos por esse nó e depois, em um momento posterior, chegarmos novamente nele, sair e não voltar mais.

Portanto, um nó com um número ímpar de ramos deve ser o primeiro ou o último da trajetória. Sendo assim, pode haver, no máximo, dois nós com um número ímpar de ramos ligados a eles.

No caso das pontes de Königsberg, existem quatro nós com um número ímpar de ramos. Logo, não tem solução.

Na teoria dos grafos, um caminho completo, com as propriedades descritas acima, é chamado *Trajetória de Euler*.

1.3 Algumas definições sobre Teoria de Grafos

Grafos são uma forma de representar elementos de um conjunto e relações entre esses elementos de forma sucinta. Formalmente, temos a seguinte definição.

Definição 1 *Um grafo é uma estrutura $G = (N, M)$ em que:*

- *N é o conjunto $(n_1, n_2, n_3, ...)$ não-vazio de nós.*
- *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 $r \in N \times N$.*

A partir da definição de M, podem aparecer ramos múltiplos, ou seja, ramos diferentes associados aos mesmos nós. G é um grafo simples se não houver ramos múltiplos. Serão usados neste trabalho somente grafos simples.

Definição 2 *O nó n é adjacente ao ramo r se n é extremidade de r.*

Definição 3 *Os ramos r_1 e r_2 são adjacentes se tiverem ao menos uma extremidade em comum.*

Definição 4 *Os nós n_1 e n_2 são adjacentes de forem extremidades de um ramo.*

Definição 5 *Um subgrafo de um grafo* (N, M) *é qualquer grafo* (N', M') *tal que* $N' \subseteq N$ *e* $M' \subseteq M$.

Definição 6 *Uma seqüência de ramos* $C = (r_1, ..., r_p)$ *é uma cadeia se existir uma seqüência de nós* $(n_1, ..., n_{p+1})$, *tais que* r_i *é adjacente a* n_i *e* n_{i+1}, $i = 1, ..., p$.

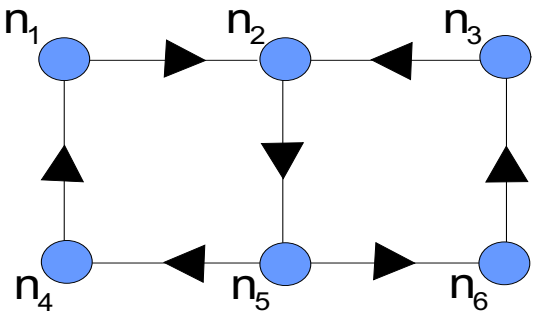


Figura 1.3: Grafo

A seqüência de nós (n_6, n_5, n_4, n_1) é um exemplo de cadeia na figura acima.

Definição 7 *Um grafo é conexo se, para qualquer par* (n_i, n_j) *de seus nós, existe uma cadeia com extremos* n_i *e* n_j.

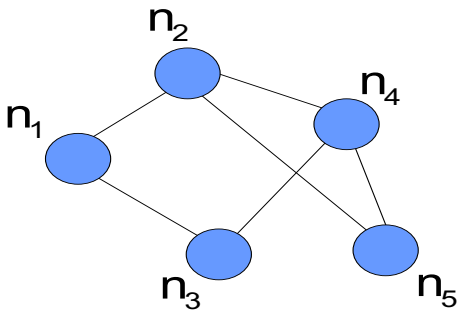


Figura 1.4: Grafo conexo

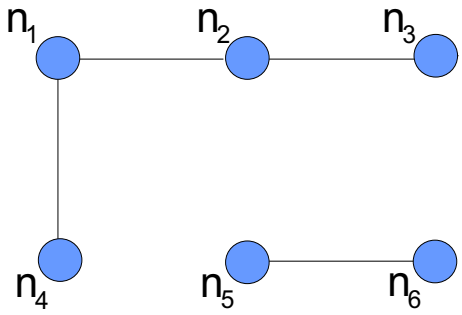


Figura 1.5: Grafo desconexo

Definição 8 *Um ciclo é uma cadeia em que* $n_1 = n_{p+1}$. *Na figura 1.3, a seqüência de nós* $(n_1, n_2, n_3, n_6, n_5, n_4, n_1)$ *é um exemplo de ciclo.*

Definição 9 *A seqüência* P *é um caminho (de* n_1 *a* n_{p+1}*) se* $r_i = (n_i, n_{i+1})$, $i = 1, ..., p$. *A seqüência de nós* $(n_1, n_2, n_5, n_6, n_3)$ *na figura 1.3 indica um caminho.*

Definição 10 *Uma árvore é um grafo conexo sem ciclos. Em uma árvore com* n *nós, o número de ramos é* $n - 1$.

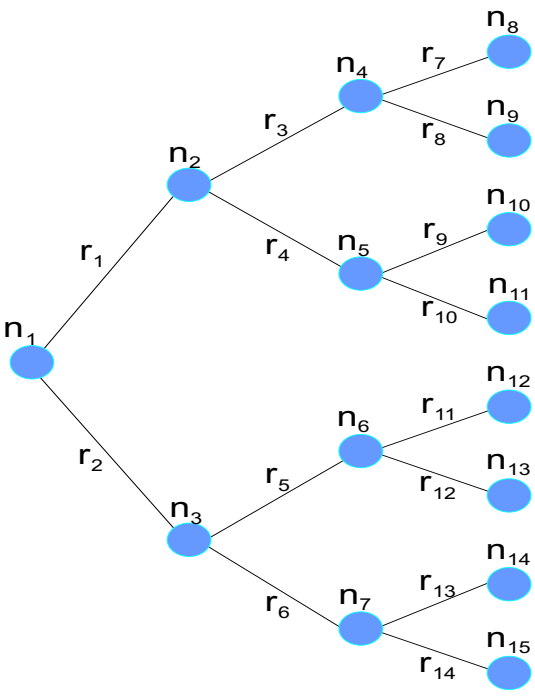


Figura 1.6: Árvore

Definição 11 *Um subgrafo que é uma árvore e que alcança todos os nós do grafo é chamado árvore maximal.*

Definição 12 *Um nó* $n \in N$ *é acessível a partir de um nó* $s \in N$ *se e só se existe um caminho de* s *a* n.

Definição 13 *Um nó é uma raiz de um grafo se qualquer nó for acessível a partir dele.*

Definição 14 *Uma arborescência é uma árvore com uma raiz.*

Definição 15 *O operador sucessor* Γ *de um grafo* (N, M) *associa a cada nó* n *um conjunto de nós* $\Gamma(n)$ *tais que* $n_j \in \Gamma(n)$ *se e só se* $(n, n_j) \in M$.

Propriedade 1 *Um grafo* $G = (N, M)$ *é perfeitamente caracterizado por* N *e* Γ. *Podemos escrever* $G = (N, \Gamma)$.

Propriedade 2 *Se um grafo* (N, Γ) *possui uma raiz* n_0, *então é perfeitamente caracterizado por* (n_0, Γ).

Neste caso, o conjunto N pode ser construído por aplicações sucessivas do operador Γ.

Capítulo 2

Caminhos de custo mínimo

Veremos neste capítulo o problema de busca de caminhos de custo mínimo e o problema do quebra-cabeças de 9 peças.

Suponha que temos uma função custo associada aos ramos do grafo G:

$$(n_i, n_j) \in M \mapsto c(n_i, n_j) \in \mathbb{R}.$$

Dado um caminho $P = (r_1, r_2, ..., r_p)$ com $r_i \in M$, $i = 1, ..., p$, o custo do caminho é definido por

$$c(P) = \sum_{i=1}^{p} c(r_i).$$

Um caminho P tem custo mínimo (o caminho P é ótimo) se $c(P) \leq c(P')$ para todo caminho P' que tenha a mesma origem e o mesmo destino que P.

Neste trabalho estudaremos o problema de busca de caminho de custo mínimo de um grafo.

Problema: Dado um nó inicial $s \in N$ e um conjunto alvo $T \subset N$, encontrar, se existir, um caminho de custo mínimo entre todos os caminhos de s a algum nó de T.

Para que sempre exista solução (desde que T seja acessível a partir de s),

o grafo não deve ter circuitos de custo negativo.

Dado um nó n, definimos:

- $c^*(n)$ = custo de um caminho ótimo de s até n.
- $h^*(n)$ = custo de um caminho ótimo de n até T.

Os caminhos encontrados serão listados. Guardaremos os caminhos em triplas ordenadas

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

em que:

- η_i representa um caminho;
- n é o nó terminal do caminho;
- c é o custo do caminho;
- p é um apontador para outro elemento listado (n', c', p') tal que (n', n) é o ramo terminal do caminho correspondente a η_i.

Dados dois caminhos $\eta_1 = (n_1, c_1, p_1)$ e $\eta_2 = (n_2, c_2, p_2)$, dizemos que η_1 domina η_2 se, e somente se, $n_1 = n_2$ e $c_1 \leq c_2$. Isto é, quando os dois caminhos levam ao mesmo nó e o custo do primeiro não é maior que o do segundo.

Arborescência maximal ótima

O problema de busca citado anteriormente está associado ao problema de encontrar uma arborescência maximal de G em que o caminho associado a cada nó $n \in N$ tenha custo $c^*(n)$.

2.1 Problema do quebra-cabeças

O problema do quebra-cabeças enunciado a seguir será tratado como um problema de busca de caminho de custo mínimo através de um grafo orientado.

Dado o quebra-cabeças de 9 peças com uma posição inicial n_0, colocá-lo na seguinte ordem:

1	2	3
8		4
7	6	5

As casas do quebra-cabeças são numeradas de 1 a 9 de acordo com essa configuração, com a casa 9 no centro. Uma configuração qualquer é representada por um vetor que associa a cada posição a peça que se encontra nela.

Ao vetor que representa uma configuração, adicionamos uma décima componente que indica a posição da casa vazia.

Por exemplo, dada a posição inicial, representada pelo vetor n_0, em que os números de 1 a 8 indicam as peças do quebra-cabeças, o número 9 indica a casa vazia e a última entrada do vetor indica a posição que está vazia,

1	3	4
8	6	2
7	5	

$$n_0 = \begin{pmatrix} 1 \\ 3 \\ 4 \\ 2 \\ 9 \\ 5 \\ 7 \\ 8 \\ 6 \\ 5 \end{pmatrix}$$

chegar a posição final indicada no início do problema.

Em cada passo da resolução do quebra-cabeças move-se uma peça adjacente à posição vazia para esta posição. Equivalentemente, pode-se pensar que "a posição vazia é movida para uma casa adjacente".

A matriz A a seguir indica os movimentos possíveis para a casa vazia. A linha A_i lista as posições para onde a casa vazia pode mover-se a partir da casa i. Os elementos nulos de A devem ser ignorados.

Um movimento a partir de n_0 corresponde à troca das peças que estavam nas posições $b = n_0(10)$ e em cada posição adjacente a b, dadas por A_b.

$$A = \begin{pmatrix} 2 & 8 & 0 & 0 \\ 1 & 3 & 9 & 0 \\ 2 & 4 & 0 & 0 \\ 3 & 5 & 9 & 0 \\ 4 & 6 & 0 & 0 \\ 5 & 7 & 9 & 0 \\ 6 & 8 & 0 & 0 \\ 1 & 7 & 9 & 0 \\ 2 & 4 & 6 & 8 \end{pmatrix}$$

O grafo: o problema é representado por um grafo (N, Γ), em que N é o conjunto de todas as configurações possíveis para o quebra-cabeças, e Γ associa a cada configuração os sucessores indicados pela matriz A, como descrevemos acima.

Neste exemplo, os sucessores de n_0 são os seguintes:

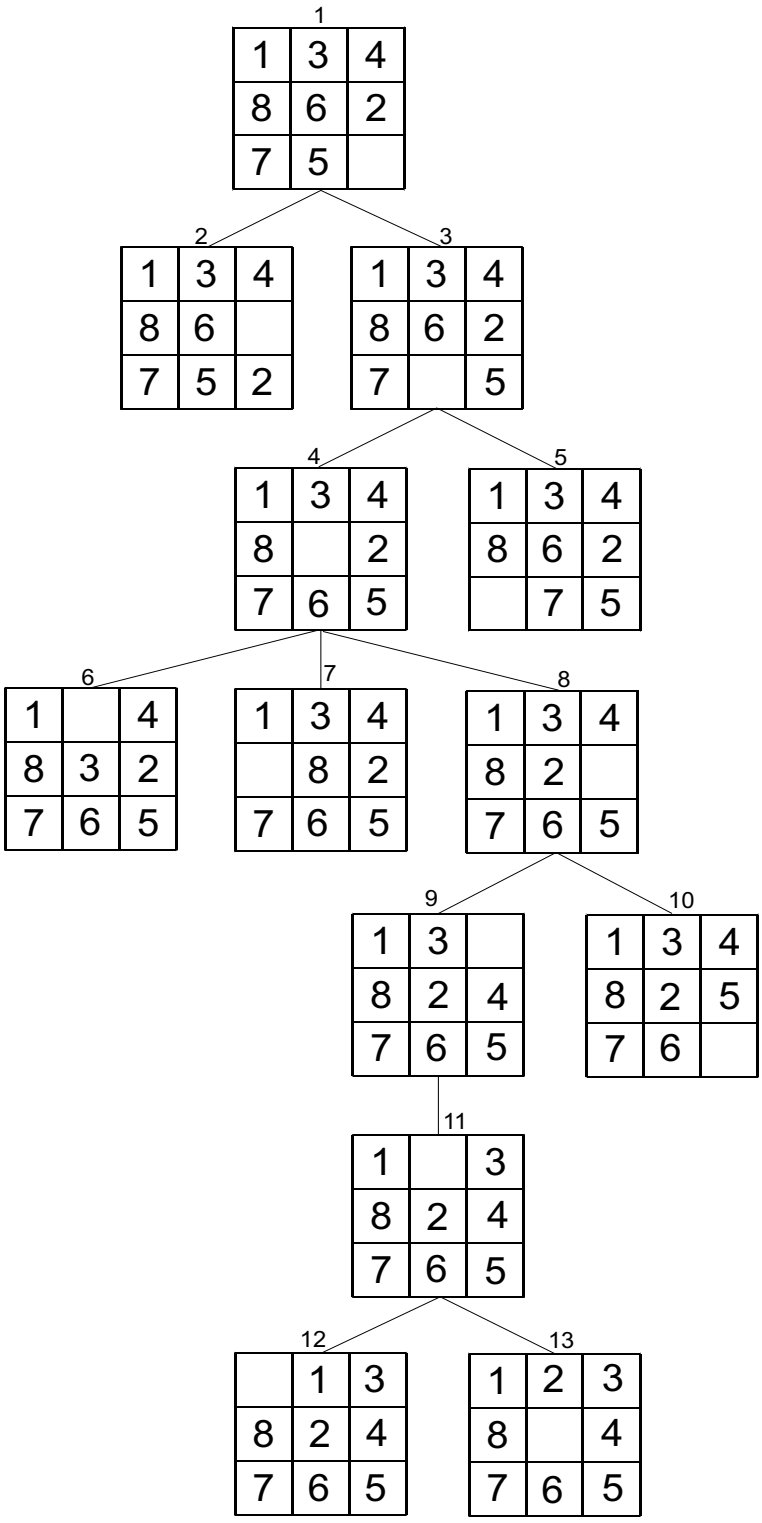
1	3	4
8	6	
7	5	2

$$n_1 = \begin{pmatrix} 1 \\ 3 \\ 4 \\ 9 \\ 2 \\ 5 \\ 7 \\ 8 \\ 6 \\ 4 \end{pmatrix}$$

1	3	4
8	6	2
7		5

$$n_2 = \begin{pmatrix} 1 \\ 3 \\ 4 \\ 2 \\ 5 \\ 9 \\ 7 \\ 8 \\ 6 \\ 6 \end{pmatrix}$$

O diagrama a seguir, mostra uma parte do grafo gerado a partir de n_0 por operações sucessivas de Γ.


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

$$
\begin{aligned}
\eta_1 &= ([1\ 3\ 4\ 2\ 9\ 5\ 7\ 8\ 6],\ 0,\ 0) \\
\eta_2 &= ([1\ 3\ 4\ 9\ 2\ 5\ 7\ 8\ 6],\ 1,\ 1) \\
\eta_3 &= ([1\ 3\ 4\ 2\ 5\ 9\ 7\ 8\ 6],\ 1,\ 1) \\
\eta_4 &= ([1\ 3\ 4\ 2\ 5\ 6\ 7\ 8\ 9],\ 2,\ 3) \\
\eta_5 &= ([1\ 3\ 4\ 2\ 5\ 7\ 9\ 8\ 6],\ 2,\ 3) \\
\eta_6 &= ([1\ 9\ 4\ 2\ 5\ 6\ 7\ 8\ 3],\ 3,\ 4) \\
\eta_7 &= ([1\ 3\ 4\ 2\ 5\ 6\ 7\ 9\ 8],\ 3,\ 4) \\
\eta_8 &= ([1\ 3\ 4\ 9\ 5\ 6\ 7\ 8\ 2],\ 3,\ 4) \\
\eta_9 &= ([1\ 3\ 9\ 4\ 5\ 6\ 7\ 8\ 2],\ 4,\ 8) \\
\eta_{10} &= ([1\ 3\ 4\ 5\ 9\ 6\ 7\ 8\ 2],\ 4,\ 8) \\
\eta_{11} &= ([1\ 9\ 3\ 4\ 5\ 6\ 7\ 8\ 2],\ 5,\ 9) \\
\eta_{12} &= ([9\ 1\ 3\ 4\ 5\ 6\ 7\ 8\ 2],\ 6,\ 11) \\
\eta_{13} &= ([1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9],\ 6,\ 11)
\end{aligned}
$$

O caminho que une n_0 ao nó alvo é obtido através dos apontadores de trás para diante, partindo do elemento listado η_{13}, e corresponde às configurações em $\eta_1,\ \eta_3,\ \eta_4,\ \eta_8,\ \eta_9,\ \eta_{11},\ \eta_{13}$.

Capítulo 3

Algoritmos de busca em grafos

Vamos descrever os algoritmos utilizado para busca do caminho de custo mínimo.

Algoritmo de rotulação: Dados $G = (N, \Gamma)$, $s \in N$, $T \subset N$.

Um algoritmo de rotulação parte do caminho construído somente pelo nó s, e constrói iterativamente uma lista de caminhos. Em cada iteração escolhe-se um caminho listado $\overline{\eta}$, que é expandido, obtendo-se um conjunto de caminhos sucessores de $\overline{\eta}$. Os sucessores de $\overline{\eta}$ são os caminhos obtidos adicionando um ramo a $\overline{\eta}$, usando o operador sucessor.

Os caminhos são listados como descrevemos acima, e utilizaremos as expressões "caminho listado"ou "elemento listado"como equivalentes.

Diremos que "um nó n está listado"se estiver listado algum caminho de s a n.

O algoritmo utiliza duas listas, uma chamada aberto e outra chamada fechado.

- Lista aberto: contêm os caminhos que ainda não geraram os sucessores (caminhos abertos).

- Lista fechado: contêm os caminhos que já foram expandidos (caminhos fechados).

Vamos descrever um algoritmo típico:

Inicialmente, fechado está vazia e aberto contém somente $\eta_1 = (s, 0, 0)$.

ENQUANTO aberto não está vazia,

Escolha um elemento $\overline{\eta} = (\overline{n}, \overline{c}, \overline{p})$ em aberto, transfira $\overline{\eta}$ para fechado.

Construa os elementos sucessores $\eta^1, \eta^2, ..., \eta^q$ associados aos nós $\Gamma(\overline{n}) = n^1, n^2, ..., n^q$:

$\eta^i = (n^i, \overline{c} + c(\overline{n}, n^i), p^i)$

$p^i =$ apontador para $\overline{\eta}$

Eliminações: elimine os elementos sucessores dominados por algum elemento listado (aberto ou fechado). Elimine os elementos listados dominados por algum sucessor.

Introduza em aberto os sucessores remanescentes.

FIM

Se há caminhos até nós do alvo em fechado, encontrar um de custo mínimo entre eles e obter o caminho correspondente seguindo os apontadores.

Observações:

- Não especificamos o critério de escolha do elemento a expandir em cada iteração. Isto é muito importante e será visto posteriormente.
- Um caminho só é eliminado de aberto ou fechado se for encontrado um novo caminho com custo estritamente menor.
- Um caminho ótimo nunca é eliminado das listas.

Teorema 1 *Quando o algoritmo termina, todos os elementos listados (fechados por construção) correpondem a caminhos ótimos. Isto é, para todo* $\eta_i = (n_i, c_i, p_i)$ *listado,*

$$c_i = c^*(n_i).$$

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

Demonstração: Suponha por contradição que um nó $\overline{n}$ acessível com custo $\overline{c}$ não está listado com custo $c^*(\overline{n})$ (ou porque não está listado ou está listado com custo $\overline{c} > c^*(\overline{n})$).

Seja $(s = n_1, n_2, ..., n_q = \overline{n})$ um caminho ótimo de s a $\overline{n}$.

Seja n_j, $j > 1$, o primeiro nó deste caminho que não está listado com custo ótimo. Sabemos que $j > 1$ porque s está listado.

Temos: o nó n_{j-1} está listado em um elemento

$$\widetilde{\eta} = (n_{j-1}, c^*(n_{j-1}), \widetilde{p}),$$

fechado.

Em alguma iteração, $\widetilde{\eta}$ foi expandido, pois está fechado. Nessa iteração, gerou-se o sucessor ótimo

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

com $c = c(n_{j-1} + c(n_{j-1}, n_j) = c^*(n_j)$.

Esse elemento não foi eliminado, pois por hipótese n_j não está listado com custo ótimo. Portanto foi listado, com custo ótimo e nunca poderia ser eliminado.

Isso contradiz a hipótese sobre n_j, completando a demonstração.

■

Como o algoritmo gera um caminho ótimo até cada nó acessível a partir de s, a lista fechado representa uma arborescência maximal ótima.

3.1 Critérios de escolha do elemento a expandir

Há quatro critérios para a escolha do elemento a expandir:

- Busca horizontal: À medida que os caminhos são expandidos, os sucessores vão para o final da lista aberto, expandindo sempre o primeiro caminho dessa lista (o elemento mais antigo).

- Busca em profundidade: Nessa busca, ao contrário da horizontal, expande-se sempre o último aberto da lista (o elemento mais recente).

- Algoritmo de Dijkstra: neste algoritmo escolhe-se em cada iteração um elemento de custo mínimo em aberto. Empates são decididos arbitrariamente.

- Algoritmo A^*: Em certos problemas, existem procedimentos que associam a cada nó $n \in N$ uma estimativa $\hat{h}(n)$ para o custo de um caminho ótimo entre n e o alvo T. Tais estimativas são obtidas pela resolução de um problema mais simples, obtido relaxando retiradas algumas restrições. O algoritmo A^* escolhe um elemento $\overline{\eta}$ com mínimo valor de $c_j + \hat{h}(n_j)$ entre todos os abertos η_j . Assim, a comparação de custos é feita sobre uma estimativa do custo total de um caminho que inclui o nó n_j.

3.1.1 Algoritmo de Dijkstra

Vamos analisar o algoritmo de Dijkstra.

ESCOLHA: escolher um elemento $\overline{\eta} = (\overline{n}, \overline{c}, \overline{p})$ com $\overline{c}$ mínimo entre os custos de todos os elementos abertos.

Este algoritmo é especialmente interessante quando todos os custos são não negativos.

Teorema 2 *Suponha que todos os ramos tem custos não negativos. Então o elemento $\overline{n}$ fechado em cada iteração tem custo $\overline{c} = c^*(\overline{n})$.*

Demonstração: Suponha que em uma iteração é fechado um elemento $\overline{\eta} = (\overline{n}, \overline{c}, \overline{p})$. Por construção, todos os abertos η_i nessa iteração tem custos $c_i \geq \overline{c}$. O mesmo ocorrerá com todos os sucessores gerados no futuro, pois os custos são não negativos. O elemento $\overline{\eta}$ nunca poderá ser eliminado. Pelo teorema anterior, $\overline{\eta}$ é ótimo.

■

No momento em que um nó do alvo for fechado, o problema de busca está resolvido. Podemos então, parar o algoritmo e recuperar o caminho ótimo seguindo os apontadores.

Este teorema também é válido para o algoritmo A^*, que veremos a seguir.

O algoritmo de Dijkstra é o melhor possível no seguinte sentido: cada iteração obtem um caminho ótimo até um nó, com custo menor ou igual ao custo ótimo $h(s)$. O número de iterações é portanto menor ou igual ao número de nós acessíveis a partir de s com custo menor ou igual a $h(s)$. Mostra-se que é impossível que um algoritmo expanda menos nós que este. O algoritmo é, portanto, ótimo entre todos os algoritmos que usam a mesma informação.

3.2 O algoritmo A^*

O algoritmo A^* foi desenvolvido por Hart, Nilsson e Raphael [3] quando estudavam algoritmos para um robô que devia atravessar uma sala com

obstáculos. O problema é representado por um grafo cujos nós são as quinas dos obstáculos (móveis, por exemplo), e o alvo é um ponto determinado. Dado um ponto qualquer, a distância euclidiana entre o ponto e o alvo fornece uma sub-estimativa do custo de um caminho viável.

A estimativa para o problema do quebra-cabeças estudado neste trabalho é dada pela soma dos custos das peças da posição em que estão até a sua posição final. A matriz H descreve as estimativas das peças do quebra-cabeças.

$$H = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 3 & 2 & 1 & 2 \\ 1 & 0 & 1 & 2 & 3 & 2 & 3 & 2 & 1 \\ 2 & 1 & 0 & 1 & 2 & 3 & 4 & 3 & 2 \\ 3 & 2 & 1 & 0 & 1 & 2 & 3 & 2 & 1 \\ 4 & 3 & 2 & 1 & 0 & 1 & 2 & 3 & 2 \\ 3 & 2 & 3 & 2 & 1 & 0 & 1 & 2 & 1 \\ 2 & 3 & 4 & 3 & 2 & 1 & 0 & 1 & 2 \\ 1 & 2 & 3 & 2 & 3 & 2 & 1 & 0 & 1 \\ 2 & 1 & 2 & 1 & 2 & 1 & 2 & 1 & 0 \end{pmatrix}$$

A posição h_{ij} da matriz H é a estimativa do custo, em que i é a posição na qual a peça está no quebra-cabeças e j é a posição na qual a peça deve estar na configuração final do quebra-cabeças.

Estimativas do custo de um caminho ótimo de cada nó de um grafo até o alvo são associadas a muitos problemas práticos. Para descrever seu uso, vamos fixar a notação.

Considere um problema de busca em um grafo (N, Γ) com um nó inicial $s \in N$ e alvo $T \subset N$. Sejam n_i, n_j dois nós arbitrários. Definimos:

- $g(n_i) = c^*(n_i)$: custo de um caminho ótimo de s a n_i.
- $h(n_i)$: custo de um caminho ótimo de n_i a T.
- $k(n_i, n_j)$: custo de um caminho ótimo de n_i a n_j.

Em todos os casos, os custos são infinitos se não existirem caminhos entre os nós considerados.

Dado um nó $n \in N$, definimos também

$$f(n) = g(n) + h(n)$$

o custo de um caminho de custo mínimo entre os caminhos de s a T que passam por n.

Observamos que se $(s, n_1, n_2, ..., n_q)$ é um caminho ótimo de s a T, então

$$f(s) = f(1) = ... = f(n_q).$$

O problema de busca pode ser enunciado como "buscar um caminho P entre s e T com $c(P) = h(s)$".

A estimativa: suporemos que para o grafo (N, Γ) conhecemos uma função

$$n \in N \mapsto \hat{h}(n) \in \mathbb{R},$$

que associa a cada nó n uma estimativa de $h(n)$.

Dado um caminho $\eta = (n, c, p)$, associamos a η a estimativa

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

O algoritmo A^* escolhe em cada iteração um elemento com o mínimo de valor de $\hat{f}(.)$ entre os caminhos abertos.

Para esse algoritmo temos o seguinte resultado:

Teorema 3 *Se $\hat{h}(n) \leq h(n)$ para todo $n \in N$, então em qualquer iteração em que nenhum nó de T está em fechado e para qualquer caminho ótimo P do nó s até o alvo, existe um nó aberto n' em P com $\hat{f}(n') \leq f(s)$.*

Demonstração: Suponha um caminho ótimo P representado por uma sequência ordenada $(s = n_0,\ n_1,\ n_2,\ ...,\ n_k)$, onde n_k é um nó alvo. Seja n' o primeiro nó aberto na seqüência. (Deve haver pelo menos um, já que n_k não está em fechado.) Seja $\eta' = (n', c', p')$ o caminho correspondente. Pela definição de $\hat{f}$, temos

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

Sabemos que A^* achou um caminho ótimo até n' já que n' está em P e todos os seus antecessores em P estão fechados. Portanto, $c' = g(n')$ e

$$\hat{f}(n') = g(n') + \hat{h}(n').$$

Como supusemos $\hat{h}(n') \leq h(n')$, podemos escrever

$$\hat{f}(\eta') \leq g(n') + h(n') = f(n').$$

Mas o valor f de qualquer nó em um caminho ótimo é igual a $f(s)$, o custo mínimo, e portanto temos $\hat{f}(n') \leq f(s)$, como alegamos no teorema.

■

Deste teorema concluímos o seguinte, se $\hat{h}$ subestima h:

- Quando um nó $\overline{n} \in T$ é fechado pelo algoritmo, o caminho $\overline{\eta} = (\overline{n}, \overline{c}, \overline{p})$ é ótimo. De fato, se $\hat{f}(\overline{\eta} = \overline{c} > h(s)$, então haverá um aberto η' com $\hat{f}(\eta') < \hat{f}(\overline{\eta})$.

- Se terminarmos o algoritmo quando um nó do alvo é fechado, somente estarão fechados nós com $g(n) + \hat{h}(n) \leq g(s)$. Se a estimativa é boa, essa região pode ser pequena.

Capítulo 4

Programas e Implementações

Veremos nesse capítulo, os programas desenvolvidos no Matlab para aplicações no problema do quebra-cabeças de 9 peças e suas implementações.

4.1 Programas

- gera_no.m: É uma função que gera aleatoriamente um nó inicial para os programas. Esse programa utiliza a matriz A descrita no capítulo 2.

```
function s=gera_no(p);
% funcao para gerar um nó inicial com p passos aleatórios

s= [1;2;3;4;5;6;7;8;9;9;0;0];
A=[2 8 0 0;1 3 9 0;2 4 0 0;3 5 9 0;4 6 0 0;5 7 9 0;6 8 0 0;1 7 9 0;2 4 6 8];

for k=1:p
V=sucessor(s,A,0);
nsuc=size(V,2);
%gerar um inteiro aleatório entre 1 e nsuc
i=1+floor(rand*nsuc-eps);
s=V(:,i);
s(end-1)=0;
disp([s' i])
end;
```

- pertence.m: Função utilizada para verificar se um nó já está listado na lista de aberto ou fechado.

```
function y=pertence(x,P)

y = 0;
[m,n] = size(P);

no=x(1:m-3);
for i=1:n
  if P(1:m-3,i) == no,
       y = i ;
    end
end
```

- sucessor.m: É a função que calcula os sucessores de um nó qualquer.

```
function suc=sucessor(x,A,pai)

n=length(x);
u=x(n-2);
custo=x(n-1);

for i=1:4
    if A(u,i)~=0
        y=x;
        p=y(u);
        y(u)=y(A(u,i));
        y(A(u,i))=p;
        y(n-2)=A(u,i);
        suc(:,i)=y;
        suc(n-1,i)=custo+1;
        suc(n,i)=pai;
    end
end
```

- dijkstra.m: Busca um caminho de custo mínimo. Para esse programa, os elementos que representam as posições do quebra-cabeças são dados por um vetor da seguinte forma: as entradas de 1 a 9 são as posições das peças no quebra-cabeças; a entrada 10 indica a posição que está vazia; a entrada 11 é o custo e a 12 é o apontador.

```
go;
kmax=2000;
Ab=x;  %coloca o vetor x na matriz de Abertos.
F = [];  %a matriz de Fechados esta vazia.
alvo=[1;2;3;4;5;6;7;8;9;9];  %alvo.
nfechados=0;
while ~isempty(Ab) % enquanto Ab nao estiver vazia
    % Escolher um aberto de custo minimo
    [cmin,i]=min(Ab(11,:));
    etabarra=Ab(:,i);
```

```
    Ab(:,i)=[];
    nfechados=nfechados+1;
    if mod(nfechados,100)==0, disp(nfechados); end;
    if nfechados>kmax, disp('Atingiu o numero maximo de iteracoes');break;end;
    F(:,nfechados)=etabarra;
    % Parar se etabarra chegou ao alvo
    if etabarra(1:10,1)== alvo, break; end;
    %coloca os sucessores de etabarra na matriz suc.
    suc = sucessor(etabarra,A,nfechados);
    [n,q] = size(suc);  %indica a dimensao da matriz suc.

    for i=1:q
       eliminado=false;
       y=pertence(suc(:,i),Ab);
       if y>0
         if suc(11,i)>=Ab(11,y), eliminado=true;
         else Ab(:,y)=[];
         end;
       else
         y=pertence(suc(:,i),F);
         if y>0
            if suc(11,i)>=F(11,y), eliminado=true; end;
         end;
       end;
       if ~eliminado
          Ab(:,end+1) = suc(:,i);  %coloca o sucessor na matriz de Abertos.
       end
     end
end
% Obter o caminho
caminho=etabarra;
ultimo=caminho(:,end);
apontador=ultimo(end);
while apontador>0
  ultimo=F(:,apontador);
  caminho=[caminho ultimo];
  apontador=ultimo(end);
end;
caminho=caminho(:,end:-1:1);
disp(caminho)
```

- hchapeu.m: Calcula as estimativas do custo para o algoritmo A^*. Utiliza a matriz H descrita no Capítulo 3.

```
function h=hchapeu(no)
dist=[...
0 1 2 3 4 3 2 1 2
1 0 1 2 3 2 3 2 1
2 1 0 1 2 3 4 3 2
3 2 1 0 1 2 3 2 1
4 3 2 1 0 1 2 3 2
3 2 3 2 1 0 1 2 1
2 3 4 3 2 1 0 1 2
1 2 3 2 3 2 1 0 1
2 1 2 1 2 1 2 1 0];
h=0;
for i=1:9
```

```
14      if no(i)<9
15         h=h+dist(i,no(i));
16      end;
17   end;
```

- aestrela.m: Este algoritmo também busca um caminho de custo mínimo. O vetor que indica a posição do quebra-cabeças tem a mesma estrutura descrita no dijkstra.m.

```
1   %go;
2   if ~exist('x'),
3   clear all
4   A=[2 8 0 0;1 3 9 0;2 4 0 0;3 5 9 0;4 6 0 0;5 7 9 0;6 8 0 0;1 7 9 0;2 4 6 8];
5   x=gera_no(20);
6   end;
7   % exemplo interessante:
8   % x=[2;3;1;4;5;6;7;8;9;9;0;0];
9   kmax=2000;
10  Ab=x;  %coloca o vetor x na matriz de Abertos.
11  fchapeu=hchapeu(x);
12  F = [];  %a matriz de Fechados esta vazia.
13  alvo=[1;2;3;4;5;6;7;8;9;9];  %alvo.
14  nfechados=0;
15  while ~isempty(Ab) % enquanto Ab nao estiver vazia
16      % Escolher um aberto de fchapeu minimo
17      [cmin,i]=min(fchapeu);
18      etabarra=Ab(:,i);
19      Ab(:,i)=[];
20      fchapeu(i)=[];
21      nfechados=nfechados+1;
22      if mod(nfechados,100)==0, disp(nfechados); end;
23      if nfechados>kmax, disp('Atingiu o numero maximo de iteracoes');break;end;
24      F(:,nfechados)=etabarra;
25      % Parar se etabarra chegou ao alvo
26      if etabarra(1:10,1)== alvo, break; end;
27      %coloca os sucessores de etabarra na matriz suc.
28      suc = sucessor(etabarra,A,nfechados);
29      [n,q] = size(suc);  %indica a dimensao da matriz suc.
30
31      for i=1:q
32         eliminado=false;
33         y=pertence(suc(:,i),Ab);
34         if y>0
35           if suc(11,i)>=Ab(11,y), eliminado=true;
36           else
37              Ab(:,y)=[];
38              fchapeu(y)=[];
39           end;
40         else
41           y=pertence(suc(:,i),F);
42           if y>0
43              if suc(11,i)>=F(11,y), eliminado=true; end;
44           end;
45         end;
46         if ~eliminado
47            Ab(:,end+1) = suc(:,i);  %coloca o sucessor na matriz de Abertos.
```

```
48            fchapeu(end+1)=suc(11,i)+hchapeu(suc(:,i));
49          end
50        end
51    end
52    % Obter o caminho
53    caminho=etabarra;
54    ultimo=caminho(:,end);
55    apontador=ultimo(end);
56    while apontador>0
57      ultimo=F(:,apontador);
58      caminho=[caminho ultimo];
59      apontador=ultimo(end);
60    end;
61    caminho=caminho(:,end:-1:1);
62    comprimento=size(caminho,2);
63    estimativas=zeros(1,comprimento);
64    for i=1:comprimento
65      estimativas(i)=caminho(11,i)+hchapeu(caminho(:,i));
66    end;
67
68    disp(caminho)
69    disp('estimativas')
70    disp(estimativas);
```

4.2 Implementações

Veremos aqui a implemetação dos algoritmos de Dijkstra e A^* e seus resultados.

Abaixo está a seqüência de nós dada pelo programa gera_no. A última linha da matriz abaixo foi tomado como nó inicial para os este exemplo.

$$
\text{gera\_no} = \begin{pmatrix}
1 & 2 & 3 & 9 & 5 & 6 & 7 & 8 & 4 & 4 & 0 & 0 & 2 \\
1 & 2 & 3 & 5 & 9 & 6 & 7 & 8 & 4 & 5 & 0 & 0 & 2 \\
1 & 2 & 3 & 9 & 5 & 6 & 7 & 8 & 4 & 4 & 0 & 0 & 1 \\
1 & 2 & 3 & 5 & 9 & 6 & 7 & 8 & 4 & 5 & 0 & 0 & 2 \\
1 & 2 & 3 & 9 & 5 & 6 & 7 & 8 & 4 & 4 & 0 & 0 & 1 \\
1 & 2 & 3 & 5 & 9 & 6 & 7 & 8 & 4 & 5 & 0 & 0 & 2 \\
1 & 2 & 3 & 9 & 5 & 6 & 7 & 8 & 4 & 4 & 0 & 0 & 1 \\
1 & 2 & 9 & 3 & 5 & 6 & 7 & 8 & 4 & 3 & 0 & 0 & 1 \\
1 & 9 & 2 & 3 & 5 & 6 & 7 & 8 & 4 & 2 & 0 & 0 & 1 \\
1 & 4 & 2 & 3 & 5 & 6 & 7 & 8 & 9 & 9 & 0 & 0 & 3 \\
1 & 4 & 2 & 3 & 5 & 6 & 7 & 9 & 8 & 8 & 0 & 0 & 4 \\
9 & 4 & 2 & 3 & 5 & 6 & 7 & 1 & 8 & 1 & 0 & 0 & 1 \\
4 & 9 & 2 & 3 & 5 & 6 & 7 & 1 & 8 & 2 & 0 & 0 & 1 \\
4 & 8 & 2 & 3 & 5 & 6 & 7 & 1 & 9 & 9 & 0 & 0 & 3 \\
4 & 9 & 2 & 3 & 5 & 6 & 7 & 1 & 8 & 2 & 0 & 0 & 1 \\
4 & 2 & 9 & 3 & 5 & 6 & 7 & 1 & 8 & 3 & 0 & 0 & 2 \\
4 & 2 & 3 & 9 & 5 & 6 & 7 & 1 & 8 & 4 & 0 & 0 & 2 \\
4 & 2 & 3 & 5 & 9 & 6 & 7 & 1 & 8 & 5 & 0 & 0 & 2 \\
4 & 2 & 3 & 5 & 6 & 9 & 7 & 1 & 8 & 6 & 0 & 0 & 2 \\
4 & 2 & 3 & 5 & 9 & 6 & 7 & 1 & 8 & 5 & 0 & 0 & 1
\end{pmatrix}
$$

A matriz *caminho* descreve o caminho de custo mínimo obtido através do algoritmo de Dijkstra. Cada coluna representa uma posição do quebra-cabeças.

$$\text{caminho} = \begin{pmatrix} 4 & 4 & 4 & 4 & 9 & 1 & 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 9 & 4 & 4 & 4 & 9 & 2 & 2 & 2 \\ 3 & 3 & 9 & 2 & 2 & 2 & 2 & 2 & 9 & 3 & 3 \\ 5 & 9 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 9 & 4 \\ 9 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 \\ 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 \\ 7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 \\ 1 & 1 & 1 & 1 & 1 & 9 & 8 & 8 & 8 & 8 & 8 \\ 8 & 8 & 8 & 8 & 8 & 8 & 9 & 4 & 4 & 4 & 9 \\ 5 & 4 & 3 & 2 & 1 & 8 & 9 & 2 & 3 & 4 & 9 \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 0 & 1 & 2 & 4 & 8 & 16 & 32 & 53 & 92 & 156 & 274 \end{pmatrix}$$

O caminho de custo mínimo obtido através do algoritmo A^* está descrito na matriz *caminho**.

$$\text{caminho}^* = \begin{pmatrix} 4 & 4 & 4 & 4 & 9 & 1 & 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 9 & 4 & 4 & 4 & 9 & 2 & 2 & 2 \\ 3 & 3 & 9 & 2 & 2 & 2 & 2 & 2 & 9 & 3 & 3 \\ 5 & 9 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 9 & 4 \\ 9 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 \\ 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 \\ 7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 \\ 1 & 1 & 1 & 1 & 1 & 9 & 8 & 8 & 8 & 8 & 8 \\ 8 & 8 & 8 & 8 & 8 & 8 & 9 & 4 & 4 & 4 & 9 \\ 5 & 4 & 3 & 2 & 1 & 8 & 9 & 2 & 3 & 4 & 9 \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 0 & 1 & 2 & 4 & 8 & 12 & 15 & 17 & 18 & 19 & 20 \end{pmatrix}$$

$$\text{estimativas} = \begin{pmatrix} 6 & 6 & 8 & 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \end{pmatrix}$$

Observe que neste último, a estimativa vai se aproximando do custo do caminho de custo mínimo.

Um exemplo interessante é obtido quando mudamos duas peças de lugar duas vezes, levantando cada uma delas. Por essas mudanças, obtemos um grafo com duas componentes conexas.

$$x = \begin{pmatrix} 2 & 3 & 1 & 4 & 5 & 6 & 7 & 8 & 9 & 9 & 0 & 0 \end{pmatrix}$$

O algoritmo de Dijkstra não consegue resolver esse exemplo. Já o algoritmo A^* continua eficiente na resolução, como podemos ver na matriz a seguir:

$$\begin{pmatrix}
2 & 2 & 2 & 2 & 9 & 8 & 8 & 8 & 8 & 8 & 8 & 8 & 9 & 1 & 1 & 1 & 1 \\
3 & 3 & 3 & 9 & 2 & 2 & 2 & 2 & 2 & 9 & 1 & 1 & 1 & 9 & 2 & 2 & 2 \\
1 & 1 & 9 & 3 & 3 & 3 & 3 & 3 & 9 & 2 & 2 & 2 & 2 & 2 & 9 & 3 & 3 \\
4 & 9 & 1 & 1 & 1 & 1 & 1 & 9 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 9 & 4 \\
5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 \\
6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 \\
7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 \\
8 & 8 & 8 & 8 & 8 & 9 & 4 & 4 & 4 & 4 & 4 & 9 & 8 & 8 & 8 & 8 & 8 \\
9 & 4 & 4 & 4 & 4 & 4 & 9 & 1 & 1 & 1 & 9 & 4 & 4 & 4 & 4 & 4 & 9 \\
9 & 4 & 3 & 2 & 1 & 8 & 9 & 4 & 3 & 2 & 9 & 8 & 1 & 2 & 3 & 4 & 9 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\
0 & 1 & 3 & 8 & 16 & 18 & 31 & 57 & 65 & 113 & 199 & 245 & 266 & 276 & 283 & 287 & 290
\end{pmatrix}$$

Com a seguinte estimativa:

$$\begin{pmatrix} 4 & 6 & 8 & 8 & 8 & 10 & 12 & 12 & 14 & 16 & 16 & 16 & 16 & 16 & 16 & 16 & 16 \end{pmatrix}$$

Conclusão

Trabalhar com grafos foi um grande desafio, pois o assunto não é abordado durante a graduação e meu primeiro contato foi quando procurei o Prof. Clóvis para orientar-me neste trabalho.

O conteúdo é muito interessante e pode ser aplicado em várias situações. Uma delas é o quebra-cabeças de 9 peças, um jogo que conheci na infância e jamais poderia imaginar toda a matemática existente por trás de sua resolução.

Os algoritmos trabalhados trouxeram um aprendizado totalmente novo, principalmente na parte de programação em Matlab, pois o único contato com o software foi na disciplina de Métodos Numéricos em Cálculo e na disciplina optativa de Programação Linear.

Verificamos através dos algoritmos programados a eficiência do algoritmo A^* na resolução de grafos, especialmente aqueles com duas componentes conexas. Já o algoritmo de Dijkstra é um ótimo algoritmo de busca de caminhos de custo mínimo, mas não tão eficiente em algumas configurações do quebra-cabeças.

Referências Bibliográficas

[1] Clóvis Gonzaga, *Busca de Caminhos em Grafos e Aplicações,* I Reunião de Matemática Aplicada, IBM, Rio de Janeiro, 1978

[2] Nils Nilsson, *Problem-solving methods in artificial intelligence,* McGraw-Hill, 1971

[3] P. Hart; N. Nilsson e B. Raphael, *A Formal Basis for the Heuristic Determination of Minimun Cost Paths,* IEEE Trans., 1968

[4] Clóvis Gonzaga, *Métodos de busca em grafos e sua aplicação a problemas de planejamento,* Tese de Doutorado, Depto. de Engenharia de Sistemas e Computação, COPPE, Universidade Federal do Rio de Janeiro, 1973

[5] Paulo Oswaldo Boaventura Netto, *Grafos - Teoria, modelos, algoritmos,* Edgard Blücher Ltda., 1996