Pedro

Estudo sobre Algoritmos de Menor Caminho em Grafos

Vitória, ES

2017

Pedro

Estudo sobre Algoritmos de Menor Caminho em Grafos

Monografia apresentada ao Curso de Ciência da Computação do Departamento de Informática da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do Grau de Bacharel em Ciência da Computação.

Universidade Federal do Espírito Santo – UFES

Centro Tecnológico

Departamento de Informática

Orientador: Prof. Dra. Maria Cláudia Silva Boeres

Vitória, ES

2017

Pedro
Estudo sobre Algoritmos de Menor Caminho em Grafos/ Pedro. – Vitória, ES, 2017-
39 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dra. Maria Cláudia Silva Boeres

Monografia (PG) – Universidade Federal do Espírito Santo – UFES
Centro Tecnológico
Departamento de Informática, 2017.

1. Palavra-chave1. 2. Palavra-chave2. I. Souza, Vítor Estêvão Silva. II. Universidade Federal do Espírito Santo. IV. Estudo sobre Algoritmos de Menor Caminho em Grafos

CDU 02:141:005.7

Pedro

Estudo sobre Algoritmos de Menor Caminho em Grafos

Monografia apresentada ao Curso de Ciência da Computação do Departamento de Informática da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do Grau de Bacharel em Ciência da Computação.

Trabalho aprovado. Vitória, ES, 25 de setembro de 2017:

Prof. Dra. Maria Cláudia Silva Boeres
Orientador

Professor
Convidado 1

Professor
Convidado 2

Vitória, ES
2017

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis malesuada laoreet leo at interdum. Nullam neque eros, dignissim sed ipsum sed, sagittis laoreet nisi.

Agradecimentos

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis malesuada laoreet leo at interdum. Nullam neque eros, dignissim sed ipsum sed, sagittis laoreet nisi. Duis a pulvinar nisl. Aenean varius nisl eu magna facilisis porttitor. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Ut mattis tortor nisi, facilisis molestie arcu hendrerit sed. Donec placerat velit at odio dignissim luctus. Suspendisse potenti. Integer tristique mattis arcu, ut venenatis nulla tempor non. Donec at tincidunt nulla. Cras ac dignissim neque. Morbi in odio nulla. Donec posuere sem finibus, auctor nisl eu, posuere nisl. Duis sit amet neque id massa vehicula commodo dapibus eu elit. Sed nec leo eu sem viverra aliquet. Nam at nunc nec massa rutrum aliquam sed ac ante.

Vivamus nec quam iaculis, tempus ipsum eu, cursus ante. Phasellus cursus euismod auctor. Fusce luctus mauris id tortor cursus, volutpat cursus lacus ornare. Proin tristique metus sed est semper, id finibus neque efficitur. Cras venenatis augue ac venenatis mollis. Maecenas nec tellus quis libero consequat suscipit. Aliquam enim leo, pretium non elementum sit amet, vestibulum ut diam. Maecenas vitae diam ligula.

Fusce ac pretium leo, in convallis augue. Mauris pulvinar elit rhoncus velit auctor finibus. Praesent et commodo est, eu luctus arcu. Vivamus ut porta tortor, eget facilisis ex. Nunc aliquet tristique mauris id sollicitudin. Donec quis commodo metus, sit amet accumsan nibh. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

“Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Duis malesuada laoreet leo at interdum. Nullam neque eros, dignissim
sed ipsum sed, sagittis laoreet nisi.
(Lipsum generator)

Resumo

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis malesuada laoreet leo at interdum. Nullam neque eros, dignissim sed ipsum sed, sagittis laoreet nisi. Duis a pulvinar nisl. Aenean varius nisl eu magna facilisis porttitor. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Ut mattis tortor nisi, facilisis molestie arcu hendrerit sed. Donec placerat velit at odio dignissim luctus. Suspendisse potenti. Integer tristique mattis arcu, ut venenatis nulla tempor non. Donec at tincidunt nulla. Cras ac dignissim neque. Morbi in odio nulla. Donec posuere sem finibus, auctor nisl eu, posuere nisl. Duis sit amet neque id massa vehicula commodo dapibus eu elit. Sed nec leo eu sem viverra aliquet. Nam at nunc nec massa rutrum aliquam sed ac ante.

Vivamus nec quam iaculis, tempus ipsum eu, cursus ante. Phasellus cursus euismod auctor. Fusce luctus mauris id tortor cursus, volutpat cursus lacus ornare. Proin tristique metus sed est semper, id finibus neque efficitur. Cras venenatis augue ac venenatis mollis. Maecenas nec tellus quis libero consequat suscipit. Aliquam enim leo, pretium non elementum sit amet, vestibulum ut diam. Maecenas vitae diam ligula.

Fusce ac pretium leo, in convallis augue. Mauris pulvinar elit rhoncus velit auctor finibus. Praesent et commodo est, eu luctus arcu. Vivamus ut porta tortor, eget facilisis ex. Nunc aliquet tristique mauris id sollicitudin. Donec quis commodo metus, sit amet accumsan nibh. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

Duis elementum dictum tristique. Integer mattis libero sit amet pretium euismod. Curabitur auctor eu augue ut ornare. Integer bibendum eros ullamcorper rhoncus convallis. Pellentesque non pretium ligula, sit amet bibendum eros. Nam venenatis ex felis, quis blandit nunc auctor sit amet. Maecenas ut eros pharetra, lobortis neque id, fermentum arcu. Cras neque dui, rhoncus feugiat leo id, semper facilisis lorem. Fusce non ex turpis. Nullam venenatis sed ligula ac lacinia.

Palavras-chaves: lorem. ipsum. dolor. sit. amet.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

UML	Unified Modeling Language

Sumário

APÊNDICES

1 Introdução

Na computação muitas aplicações necessitam considerar um conjunto um conjunto de conexões entre dados e uso de algoritmos nesses dados para poder responder perguntad como, se existe um caminho entre um objeto a outro seguindo por essas conexões, qual a menor dinstância entre eles ou ainda quantos objetos podemos alcançar a partir de um determinado ponto. Para modelar tais situações, utilizamos um tipo abstrato chamado grafo (ZIVIANI et al., 2004).

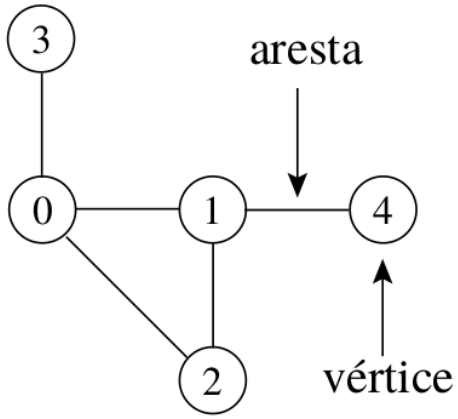

Figura 1 – Exemplo de um grafo.

Um problema clássico da literatura envolvendo grafos é o cálculo de caminho mínimo (MOURA; RITT; BURIOL, 2010). Nele desejamos obter o menor caminho entre dois pontos específicos do grafo representado. A sua modelagem é feita representando as arestas com determinados pesos que podem significar o tempo decorrente entre executar tarefas, o custo de transmitir informações entre locais, quantidades específicas a serem transportadas entre um local e outro e etc (DROZDEK, 2012).

Dentre as modelagens realizadas para resolver o problema do caminho mínimo, uma que é abordada é a definição do menor caminho entre dois pontos geográficos tendo como aplicação prática o uso por softwares do tipo GPS. Nela representamos as interseções das ruas, caminhos ou rodovias como o vértices do grafo e as distâncias entre essas interseções (as próprias ruas, caminhos ou rodovias) como as arestas e tendo seus respectivos pesos como sendo a distâncias entre essas interseções.

A figura X mostra o exemplo desta modelagem abordada.

A motivação deste trabalho é o estudo dos algoritmos de menor caminho em grafos, desde os clássicos como o algoritmo de Dijkstra (DIJKSTRA, 1959) até algoritmos mais

recentes propostos como o Anytime Dynamic A* (AD*) (LIKHACHEV et al., 2008). Tem por objetivo verificar o desempenho e a eficácia destes algoritmos, averiguar o impacto que as estruturas de dados utilizadas para resolver o problema causam e analisar quais situações os algoritmos estudados melhor se aplicam.

1.1 Metodologia de pesquisa

Para a realização do estudo a que este trabalho propõe, será implementado os algoritmos a serem estudados na linguagem Java, versão 1.8.0_131. Os testes a serem realizados e as instâncias utilizadas estão descritos no capítulo 5. A configuração do computador em que será submetido os testes está descrito na tabela 1.

Tabela 1 – Configuração da máquina onde será realizado os testes computacionais.

Sistema Operacional	Linux Mint 18.2 Sonya 64-bit
Processador	Intel Core i5-2430M @ 2.40 GHz. Cache 6MB
Memória	2.0 GB
Compilador	Java 1.8.0_131

1.2 Organização dos capítulos

Este trabalho está estruturado da seguinte forma:

Capítulo 2: Apresenta a descrição do algoritmo clássico de Dijkstra, as versões implementadas baseadas em estrutura de dados diferentes e suas respectivas descrições;

Capítulo 3: Apresenta a descrição do algoritmo busca A*, sua diferença em relação ao Dijkstra e uma descrição do uso da estratégia de heurísticas e como elas são classificadas;

Capítulo 4: Apresenta a descrição dos algoritmos ARA* e AD*, a estratégia de heurística inflada e uma breve descrição de grafos dinâmicos, nos quais o algoritmo AD* se aplica;

Capítulo 5: Apresenta a descrição dos testes realizados, seus respectivos resultados e uma análise desses para cada algoritmo abordado;

Capítulo 6: Traz a conclusão geral deste trabalho e possíveis trabalhos futuros.

2 Algoritmo de Dijkstra

2.1 O Algoritmo

O algoritmo de Dijkstra foi proposto por Edgar W. Dijkstra em 1959 (DIJKSTRA, 1959). Ele tem por objetivo definir o menor caminho partindo do vértice origem v_s e chegando a todos os demais vértices v_i do grafo $G = (V, E)$. Para garantir a viabilidade do algoritmo, assume-se que todos os pesos $w(u, v)$ sejam maiores ou iguais a zero para toda aresta E do grafo G (CORMEN, 2009).

A seguir é apresentado o pseudocódigo do algoritmo conforme descrito em Drozdek (2012).

Listagem 2.1 – Algoritmo de Dijkstra.

```
1  DijkstraAlgorithm(weighted simple digraph, vertex first)
2    for all vertices v
3      currDist(v) = ∞;
4    currDist(first) = 0;
5    toBeChecked = all vertices;
6    while toBeChecked is not empty
7      v = a vertex in toBeChecked with minimal currDist(v);
8      remove v from toBeChecked;
9      for all vertices u adjacent to v and in toBeChecked
10       if currDist( u ) > currDist( v ) + weight( edge(vu) )
11         currDist( u ) = currDist( v ) + weight( edge(vu) );
12         predecessor( u ) = v;
```

O algoritmo inicia atribuindo o valor inicial de cada distância de cada vértice do grafo igual a ∞, com exceção do vértice inicial v_s que será iniciado por 0. Em seguida todos os vértices são adicionados ao conjunto dos "toBeChecked"("aSeremChecados"). Feito isso, inicia-se o processo iterativo: seleciona-se o vértice v de menor custo que esteja dentro do conjunto "toBeChecked", retira-se ele do conjunto e a partir dele, para cada vértice adjacente u de v, verifica-se se a distância atual calculada de u é maior do que a distância calculada de v mais o valor referente ao peso da aresta de v e u (origem em v). Caso seja verdade, a distância atual de u é substituída pela soma da distância atual de v mais o peso da aresta de v e u (este valor corresponde a distância do vértice de origem v_s até u), além de definir o antecessor u como sendo v. Repete-se o passo iterativo até que o conjunto "toBeChecked"esteja vazio[1].

Ao final do algoritmo, teremos o conjunto de predecessores de cada vértice do grafo, e a partir deste, poderemos definir a rota para qualquer vértice do grafo partindo de v_s. A figura 2 mostra um exemplo de aplicação do algoritmo a um grafo.

[1] Em linguagens de programação, é costume substituir o valor ∞ pelo maior número representativo do tipo

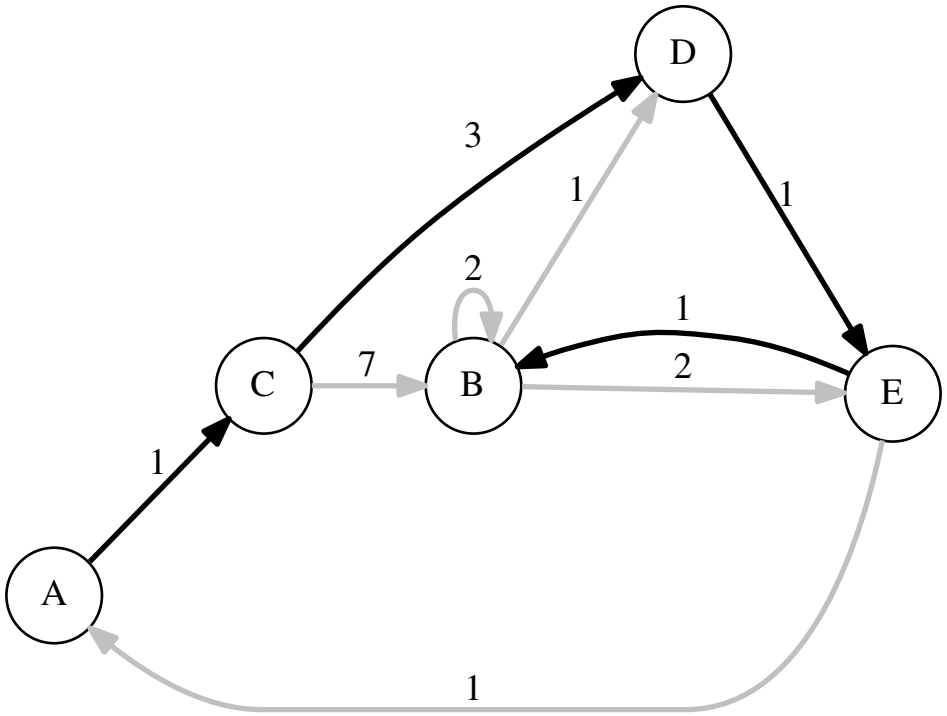

Figura 2 – Aplicação do algoritmo de Dijkstra tendo o vértice "A"como origem. As arestas pintadas de preto correspondem a rota calculada a todos os demais vértices.

2.1.1 Garantia do algoritmo retorna o menor valor

[Professora, a prova de que algoritmo de Dijkstra retorna o menor caminho entre todos os pontos eu penso colocar nesta subseção.]

2.2 Versões do Algoritmo implementadas e suas Estrutura de Dados

Para este projeto de graduação será implementada três versões do algoritmo de Dijkstra baseados em estruturas de dados diversas que implicam em tempos computacionais diferentes (CORMEN, 2009).

da variável selecionado para representar a distância. Por exemplo na linguagem C, caso se utilize o valor int (inteiro) para representar a distância, a atribuição inicial será dado pela constante INT_MAX definida pela biblioteca "limits.h", que representa o maior valor numérico representado por esse tipo de variável.

As versões implementadas são o Dijkstra Canônico (descrito a seguir), Dijkstra Heap Binário (subseção 2.2.1) e Dijkstra Heap de Fibonacci (subseção 2.2.2), todas baseadas em Cormen (2009), Drozdek (2012).

Para a versão Dijkstra Canônico o algoritmo utiliza de um vetor para armazenar as distâncias calculadas pelo algoritmo (o indíce dos vértices correspondem ao indíce do vetor em que são armazenados), e a cada passo iterativo (conforme demonstrado pelo algoritmo na seção 2.1), uma busca linear é realizada para determinar o vértice (fora do conjunto "toBeChecked") cuja a distância é menor dentre todas as outras. O tempo computacional para esse caso é $O(|V^2|)$ (DROZDEK, 2012).

2.2.1 Dijkstra Heap Binário

Para esta implementação, será utilizada a estrutura de dados heap binária mínima como fila de prioridade. Heaps binária podem ser descritas como árvores binárias que possuem as seguintes propriedades (DROZDEK, 2012):

1. O valor de cada nodo não é maior do que os valores guardados em cada um de seus filhos.
2. A árvore é perfeitamente balanceada, e as folhas no último nível estão todas mais a esquerda.

Um exemplo de estrutura Heap Binário representada tanto como árvore como vetor pode ser visualizado nas figuras 3 e 4 respectivamente.

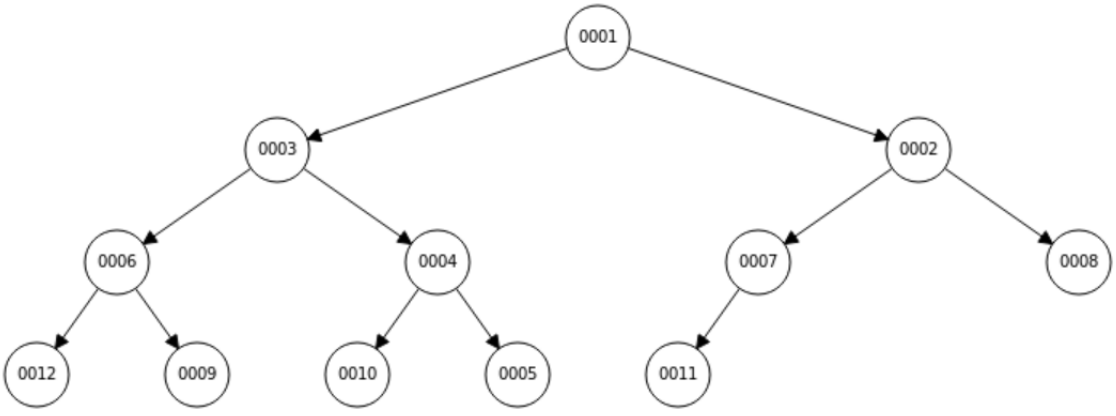

Figura 3 – Exemplo de Heap Binário representado como árvore.

0001	0003	0002	0006	0004	0007	0008	0012	0009	0010	0005	0011
1	2	3	4	5	6	7	8	9	10	11	12

Figura 4 – Representação do Heap Binário da figura 3 como vetor.

A disposição dos elementos da árvore no vetor segue as seguintes relações entre nós pai, filho-direita e filho-esquerda:

Pai(i): $\lfloor i/2 \rfloor$

Filho-esquerda(i): $2 * i$

Filho-direita(i): $2 * i + 1$

Onde $i \in \mathbb{N}$ e $i \subset [1, n]$, sendo que i representa o índice do elemento no vetor e n o número de elementos da árvore.

Para efeito de exemplo (observe as figuras 3 e 4 para constatação), o nó que está contido na posição 4 do vetor possui como pai o nó de posição 2 ($\lfloor 4/2 \rfloor = 2$), tem como filho da esquerda o nó de posição 8 ($2 * 4 = 8$) e filho da direita o nó de posição 9 ($2 * 4 + 1 = 9$).

A vantagem de se usar essa estrutura de dados reside no fato de suas operações de inserção, extração de mínimo e reconstrução da heap possuirem tempo computacional de $O(\lg n)$. Por consequência, o tempo computacional para este caso é de $O(|E| \lg |V|)$ (CORMEN, 2009).

2.2.2 Dijkstra Heap de Fibonacci

A Heap de Fibonacci consiste de uma coleção de árvores que seguem a regra de árvore heap mínima, ou seja, os nós pais são maiores ou iguais aos nós filhos. Os nós raízes de cada árvore são interligados por uma lista circular duplamente encadeada. Um ponteiro chamado "raiz mínima"aponta para o nó de menor valor.

Sua característica é que operações de adição são executadas de uma maneira "preguiçosa", não procurando criar uma forma para as árvores (como por exemplo, deixa-la balanceada), apenas as adicionados-as a lista principal de raízes. Por consequência, operações de inserção possuem tempo computacional $O(1)$ (CORMEN, 2009).

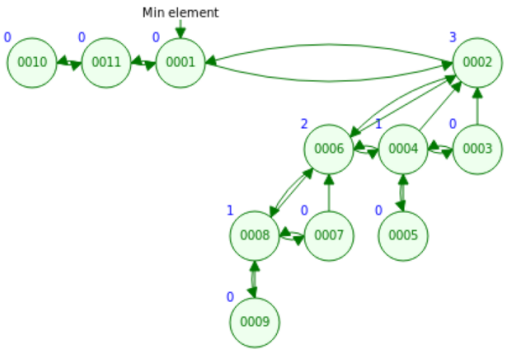

Figura 5 – Exemplo de Heap de Fibonacci (os números no canto superior esquerdo de cada nodo correspondem ao grau de cada um, ou seja, o número de filhos).

Para operações de extração de mínimo o tempo computacional é mais custoso. Isso é devido ao fato de que quando o mínimo é retirado, a heap precisa ser reorganizada de forma que sua propriedade principal não seja violada e o um novo mínimo seja determinado. Para isso a operação de extração de mínimo se dá em três etapas. Primeiro é retirado o mínimo da heap (se caso o mínimo possua nós filhos, eles são colocados na lista principal de raízes) e o seu vizinho é assimilado como o novo mínimo provisório. Agora precisamos definir quem é o novo mínimo e para isso teremos que verificar todos os demais nós raízes. Com o intuito de diminuir o número de nós raízes é que o segundo passo é aplicado. Ele consiste em lincar raízes com o mesmo número de grau (grau corresponde ao número de filhos que cada nodo possui) e para cada par de nodos licandos, verifica-se qual dos dois é menor. O que for o menor será o nodo pai e outro por consequência será o nodo filho. Após a lincagem de todos os nodos com mesmo número de grau, uma busca linear é realizada para se determinar o menor elemento entre os nodos raízes restantes[2]. O tempo computacional para a extração de mínimo é $O(\lg n)$ (CORMEN, 2009).

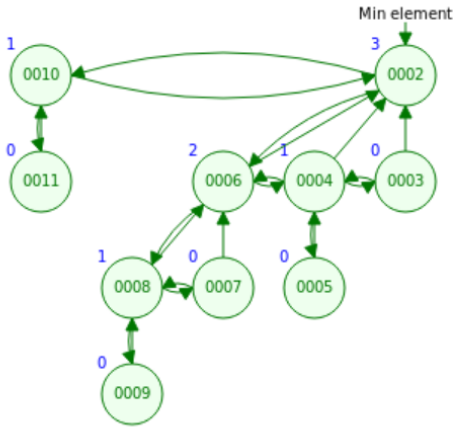

Figura 6 – Heap de Fibonacci da figura 5 após a operação de extração de mínimo.

Finalmente para a operação de mudança de chave de um determinado nodo, e após a mudança realizada em tempo constante ($O(1)$), é verificado se a propriedade heap foi violada. Se caso sim, esse nodo é cortado de seu nodo pai e colocado junto a lista principal. Se o pai não pertencer a lista de nodos raízes, ele é "marcado"("pintado") e caso já estivesse "marcado"ele também é cortado e seu pai é "marcado". Esse processo continua subindo até encontrarmos um nodo pai "não marcado"ou um nodo raiz. Após esse processo recursivo ter terminado, verifica-se se nodo modificado inicialmente é menor do que o nodo mínimo

[2] Para otimizar a busca de nodos com o mesmo número de grau é utilizado um vetor auxiliar de tamanho mínimo ao maior grau de um nodo da estrutura. Esse vetor contém ponteiros para os nodos e a posição desse nodo no ponteiro corresponde ao grau do nodo. Por exemplo, se um nodo possui grau 3, ele ocupará a posição de número 3 no vetor. Quando um nodo da lista é referenciado na posição que já está ocupado, o processo de lincagem é feito conforme descrito.

atual. Se caso sim, o novo nodo mínimo é o modificado. O tempo computacional é $O(1)$ (CORMEN, 2009).

Por consequência, o tempo computacional aplicado para o algoritmo de Dijkstra é de $O(|V| \lg |V| + |E|)$ (CORMEN, 2009).

3 Algoritmo A*

3.1 O Algoritmo

O algoritmo A* (lê-se "A estrela"), também conhecido como busca A*, é um algoritmo de busca informatizada em grafos. Foi proposta originalmente em Hart, Nilsson e Raphael (1968) e pode ser visto como uma adaptação do algoritmo de Dijkstra (apresentado no capítulo 2) em que, ao invés de se calcular a melhor rota de um ponto de partida para todos os demais vértices do grafo, se estabelece uma boa rota (ou mesmo a rota ótima[1]) partindo do vértice origem a um vértice destino. Isso é feito realizando "podas"do caminho de forma que não seja necessário visitar todos os vértices, apenas os mais promissores.

A seguir é apresentado o algoritmo A* adaptado de Likhachev et al. (2008) sobre o algoritmo de Dijkstra apresentado na seção 2.1.

Listagem 3.1 – Algoritmo A*

```
1  A*Algorithm(weighted simple digraph, vertex first, vertex goal)
2    for all vertices v
3      g(v) = ∞;
4    g(first) = 0;
5    toBeChecked = all vertices;
6    while goal is in toBeChecked
7      v = a vertex in toBeChecked with minimal f(v);
8      remove v from toBeChecked;
9      for all vertices u adjacent to v and in toBeChecked
10       if g( u ) > g( v ) + weight( edge(vu) )
11         g( u ) = g( v ) + weight( edge(vu) );
12         predecessor( u ) = v;
13         update u in toBeChecked with f(u) = g(u) + h(u);
```

O algoritmo segue em sua essência como um Dijkstra adaptado. Iniciamos a distância de todos os vértices g(v) (valor que corresponde ao valor da distância calculada do vértice origem "first"até o vértice "v") como sendo ∞[2], com exceção do vértice origem, cujo valor atribuído é zero. Adicionamos todos os vértices ao grupo dos "toBeChecked"[3]. Feito isso inicia-se o processo iterativo: enquanto o vértice "goal"estiver dentro do conjunto do "toBeChecked"(ou seja, o vértice "goal"não foi alcançado ainda pelo algoritmo), o vértice com menor valor f(v) é retirado do conjunto "toBeChecked"e para cada vértice adjacente u de v, verifica-se se o valor de g(u) atual é maior que g(v) mais o peso da aresta entre v e u (edge(vu)). Se caso for verdade, o valor de g(u) é atualizado para g(v) mais o peso da aresta entre v e u, e v é marcado como o predecessor de u. O valor do peso do vértice u

[1] A garantia do valor ótimo do algoritmo depende de fatores que serão discutidos na subseção 3.1.1.
[2] Vide nota de rodapé da seção 2.1.
[3] Algumas literaturas designam esse conjunto como OPEN.

é atualizado na fila de prioridades utilizada (como a heap binária, descrita na subseção 2.2.1) com o valor f(u) = g(u) + h(u).

Observe que para o algoritmo A*, diferente do que ocorre em Dijkstra, não se utiliza o valor de g(u) (valor da distância calculada do vértice origem "first"até o vértice "u") como chave de ordenamento da fila de prioridade, mas sim esse valor acrescido de h(u). Observe também que o algoritmo termina ao ser removido o vértice destino ("goal") da lista do "toBeChecked"em contrapartida ao Dijkstra que calcula para todos os vértices do grafo.

O termo h(u) significa o valor heurístico que corresponde a uma estimativa da distância de u ao vértice destino "goal". É devido a esse valor que o algoritmo A* realiza "podas"no número de vértices a serem checados, buscando os mais promissores, já que esse valor faz com que os vértices cujas estimativas sejam mais próximas do vértice destino ("goal") sejam colocados mais a frente na fila de prioridades e por consequência, sejam calculadas primeiros. E assim é mais provável que o vértice destino seja alcançado antes e tenha sua rota calculada, terminando o algoritmo. O valor h(u) é classificado como admissível e não-admissível cujo significado será dado na subseção 3.1.1.

A figura 7 contida em Russell e Norvig (1995) mostra um exemplo de aplicação do algoritmo.

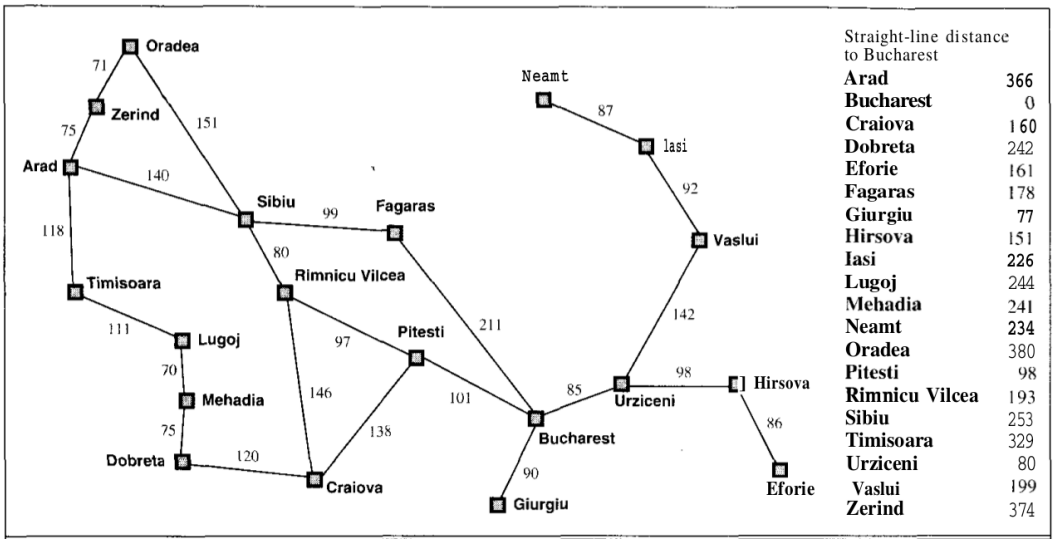

Figura 7 – Mapa da Romênia com os valores das distâncias entre as cidades e a distância euclidiana de todas elas até Bucareste.

Um viajante deseja partir da cidade de Arad com destino a Bucareste buscando percorrer o menor caminho entre essas duas cidades. Para isso é utilizado o algoritmo A* que explora o grafos conforme descrito na figura 8[4], tendo como heurística utilizada, a

[4] A figura 8 foi obtida do vídeo "Algoritmo A*"contido no sítio eletrônico <https://www.youtube.com/watch?v=6CqZ5AaTfhQ>, acesso em 25 de abril de 2017.

distância euclidiana entre todas as cidades e Bucareste (distâncias também contida na figura 7).

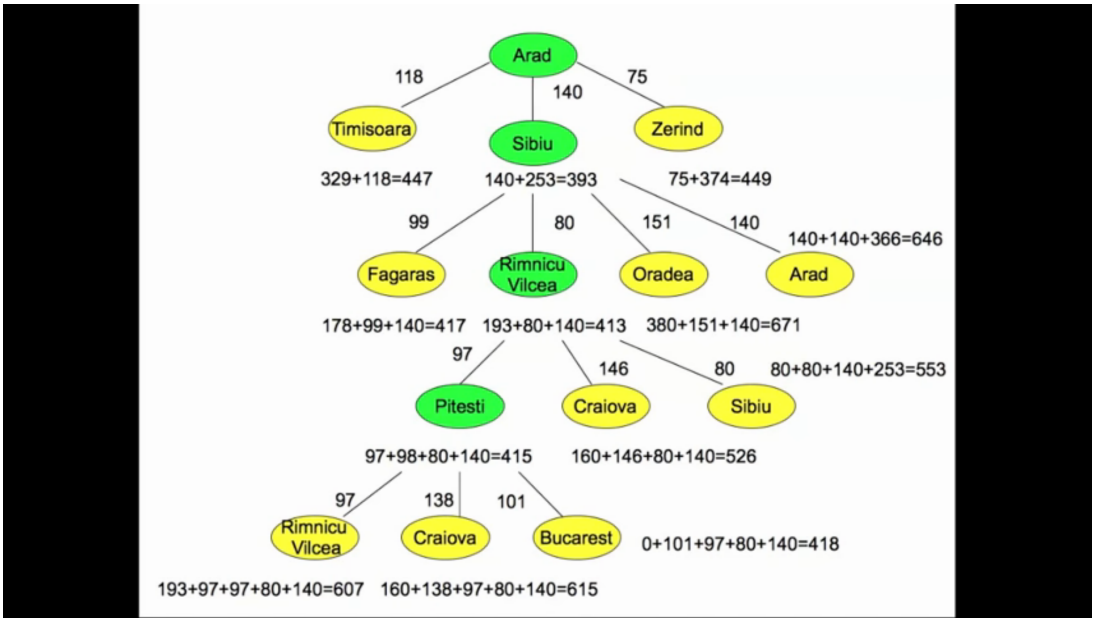

Figura 8 – Desenvolvimento do algoritmo A*.

3.1.1 Heurísticas admissíveis e não-admissíveis

O fator heurístico h(u) é uma estimativa da distância entre o vértice "u"e o vértice "goal". Ela é chamada de **admissível** quando o valor da estimativa garantidamente não superestima o valor da distância real entre "u"e "goal"(RUSSELL; NORVIG, 1995). Um exemplo clássico usado de heurística admissível é a distância euclidiana, já que a menor distância entre dois pontos é uma reta (RUSSELL; NORVIG, 1995).

O cálculo da distância euclidiana porém, nem sempre é a forma mais rápida em termos computacionais, já que geralmente ela é calculada em termos dos pontos geográficos do vértice e esse cálculo envolve exponenciação e radiação. É por isso que existe o uso de heurísticas **não-admissíveis** que são estimativas que visam a usar cálculos mais simples e que, porém, não há a garantia que essa distância superestime a distância real entre "u"e "goal". Por consequência não há garantia que o caminho ótimo seja encontrado.

Exemplos de heurísticas não-admissíveis:

- Distância Manhattan:
  - h(u) $= |x_u - x_{goal}| + |y_u - y_{goal}|$;
- Atalho Diagonal:
  - h(u) $= \sqrt{2} * |y_u - y_{goal}| + (|x_u - x_{goal}| - |y_u - y_{goal}|)$ [Se a distância $|x_u - x_{goal}| > |y_u - y_{goal}|$];

- h(u) = $\sqrt{2} * |x_u - x_{goal}| + (|y_u - y_{goal}| - |x_u - x_{goal}|)$ [Se a distância $|x_u - x_{goal}| < |y_u - y_{goal}|$];

4 Algoritmos Dinâmicos

4.1 Grafos Dinâmicos

Os algoritmos apresentados até agora trataram apenas com grafos estáticos, ou seja, grafos em que o peso de suas arestas não mudam. Mas em situações reais podemos ter casos em que a modelagem feita por grafos requerem que o peso de suas arestas variem com tempo. Para esses casos, temos grafos ditos **dinâmicos**. Exemplos de modelagem por grafos dinâmicos são o sistema de tempo real de trânsito em que os pesos das arestas correspondem ao tempo médio para percorrer um determinado trecho e esse tempo está diretamente ligado ao trânsito local em uma determinada hora; e o fluxo de em uma rede interna onde as arestas representam o caminho entre roteadores e os seus pesos correspondem ao uso desta linha, ou seja, o quão congestionada está.

Para calcularmos o menor caminho entre dois vértices em grafos dinâmicos, poderíamos utilizar os algoritmos de Dijsktra e A* (discutidos nos capítulos 2 e 3) para acharmos a rota, e quando houver uma detecção de mudança do peso de arestas, recalcularíamos novamente o trajeto reutilizando esses algoritmos. Porém, ás vezes desejamos (ou necessitamos) que esse recálculo seja mais rápido. Para esse fim existem os algoritmos dinâmicos que calculam uma solução rápida porém não garantidamente ótima (elas são ditas sub-ótimas).

Exemplos de algoritmos são o D* (STENTZ, 1994), D* Lite (KOENIG; LIKHACHEV, 2002) e o AD* (LIKHACHEV et al., 2008). Esse último apresentado a seguir.

4.2 Algoritmo ARA*

O algoritmo ARA* proposto em Likhachev et al. (2008), está descrito nas listagens 4.1, 4.2 e 4.3.

Começando pela função principal (listagem 4.3), temos as variáveis g(s) e v(s) que correspondem ao valor da distância real calculada da origem s_{start} até o vértice s (o valor de v(s), apesar de estar descrito no algoritmo conforme a literatura de origem, não é utilizado pelo algoritmo e por isso será ignorado na explicação que se segue). O valor de g(s_{goal}) (distância real calculada da origem ao vértice destino) é atribuído como ∞ (infinito)[1], enquanto que o valor de g(s_{start}) é atribuído com o valor 0 (já que o valor da distancia real calculada se baseia a partir da origem). Em seguida são criados o conjunto "OPEN", "CLOSED"e "INCONS", correspondendo respectivamente ao conjunto dos "ABERTOS",

[1] Vide nota de rodapé da seção 2.1.

Listagem 4.1 – Algoritmo ARA* - função de cálculo de caminho

```
1  procedure ComputePath()
2    while(key(s_goal) > min_{s∈OPEN(key(s))})
3      remove s with smallest key(s) from OPEN;
4      v(s) = g(s); CLOSED = CLOSED ∪ {s};
5      for each successor s' of s
6        if s' was never visited by ARA* before then
7          v(s') = g(s') = ∞;
8        if g(s') > g(s) + c(s, s')
9          g(s') = g(s) + c(s, s');
10         if s' ∉ CLOSED
11           insert/update s' in OPEN with key(s');
12         else
13           insert s' into INCONS;
```

Listagem 4.2 – Algoritmo ARA* - função da chave ordenadora da fila de prioridades

```
1  procedure key(s)
2    return g(s) + ϵ * h(s);
```

Listagem 4.3 – Algoritmo ARA* - função principal

```
1  procedure Main()
2    g(s_goal) = v(s_goal) = ∞; v(s_start) = ∞;
3    g(s_start) = 0; OPEN = CLOSED = INCONS = ∅;
4    insert s_start into OPEN with key(s_start);
5    ComputePath();
6    publish current ϵ-suboptimal solution;
7    while ϵ > 1
8      decrease ϵ;
9      Move states from INCONS into OPEN
10     Update the priorities for all s ∈ OPEN according to key(s);
11     CLOSED = ∅;
12     ComputePath();
13     publish current ϵ-suboptimal solution;
```

"FECHADOS"e "INCONSISTENTES". O vértice s_{start} é inserido na fila de prioridades do conjunto OPEN utilizado a função de chave ordenadora descrito na listagem 4.2, a qual utiliza a estratégia da heurística inflada (descrito a seguir, na subseção 4.2.1).

A função de cálculo de caminho é então invocada pela função principal (listagem 4.1). Nesta função, o processo iterativo se faz enquanto o valor da chave de s_{goal} for maior do que o valor da chave do termo mínimo da fila de prioridades. Enquanto isso for verdade, o vértice s, que corresponde ao valor do vértice com menor chave de OPEN, é removido deste e adicionado ao conjunto CLOSED. Para cada s' (que representa um vértice do conjunto de vértices adjacentes de s) é verificado se o mesmo já foi visitado pelo algoritmo e em caso negativo, o seu valor de g(s') é atribuído como ∞ (infinito). Em seguida, é verificado se a distância calculada até o momento para s', g(s'), é maior do que a soma de g(s) mais o peso da aresta entre s e s'. Se caso positivo, g(s') é atribuído com esse novo valor. A seguir é verificado se o vértice s' não pertence ao conjunto dos fechados e em caso positivo, ele é adicionado/atualizado no conjunto OPEN baseando-se na função de chave

ordenadora. Em caso contrário, ele é adicionado ao conjunto dos INCONS.

Após o cálculo do caminho e continuando a função principal (listagem 4.3), a solução ϵ sub-ótima é apresentada pelo algoritmo. A partir daí começa o processo iterativo em que o valor de ϵ é verificado se é maior do que 1. Em caso positivo, o valor de ϵ é decrescido por um fator de corte estipulado pelo programador. Os vértices pertencentes a INCONS são movidos para OPEN e todas as chaves da fila de prioridades são atualizados considerando o novo valor de ϵ. O conjunto CLOSED é limpo e a função de cálculo de caminho é invocada novamente e seu respectivo resultado é apresentado.

4.2.1 Heurística inflada e considerações sobre o algoritmo

A principal diferença entre o algoritmo ARA* e o A* está na utilização da estratégia da heurística inflada. Ela consiste em utilizar a mesma função de ordenação de chaves da fila de prioridades do algoritmo A*, com a exceção de que o valor heurístico é multiplicado por um fator ϵ. Isso faz com que menos vértices sejam visitados, já que os menos "promissores"serão jogados mais para o final da fila de prioridades enquanto que os mais "promissores"serão jogados para frente, e com isso é mais provável que um caminho até o vértice destino, s_{goal}, seja encontrado mais rápido do que pelo algoritmo A* (já que menos vértices deverão ser visitados). Entretanto, ao se utilizar esta técnica perdemos a garantia do resultado ótimo do algoritmo (algo semelhante ao que ocorre quando não utilizamos heurísticas não-admissíveis. Vide subseção 3.1.1).

Porém, "Uma grande vantagem de se utilizar a estratégia apresentada é que temos um limite superior para a solução encontrada. Suponha que a melhor solução tenha custo C*. Se utilizarmos uma busca com o A* com um valor real $\epsilon \geq 1$ multiplicando o valor heurístico na função de avaliação, então há a garantia que para a nova solução C, C* $\leq$ C $\leq \epsilon$ x C*. Portanto, se $\epsilon = 1$, a solução encontrada é ótima."(MOURA; RITT; BURIOL, 2010).

Portanto a ideia principal é achar uma solução rápida, porém não-garantidamente ótima, atribuindo um valor ϵ maior do que 1, e a mediada que mais tempo é nos dado, diminuímos o valor de ϵ até termos $\epsilon = 1$, em que a solução será garantidamente ótima.

Outra estratégia utilizada é que para ganhar tempo, a cada novo valor de ϵ em que a função para cálculo de caminho é invocada (listagem 4.1), não é necessário recalcular todos os vértices visitados anteriormente, já que os mesmos permanecem na lista de abertos (tendo apenas suas chaves atualizadas com o novo valor de ϵ e consequentemente a ordem dos vértices também é reconfigurada de acordo com esses novos valores). Além disso, para caso haja um melhor caminho encontrado para um vértice que pertence aos fechados, este são enviados para os INCONS dos quais serão enviados para OPEN para serem

recalculados na próxima busca[2].

A figura 9, contida em Likhachev et al. (2008), mostra a aplicação da estratégia dita anteriormente.

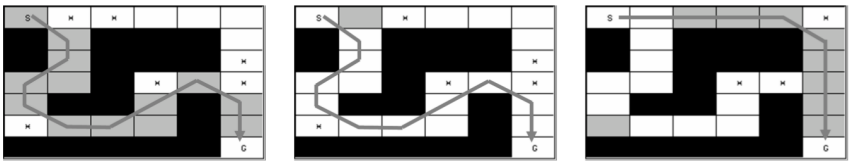

Figura 9 – Exemplo de aplicação da estratégia de diminuição do valor de ϵ.

4.3 Algoritmo AD*

O algoritmo AD* também proposto em Likhachev et al. (2008), está descrito nas listagens 4.4, 4.5, 4.6 e 4.7.

Listagem 4.4 – Algoritmo AD* - função para determinar o conjunto ao qual vértice pertencerá

```
1 procedure UpdateSetMembership(s)
2   if (v(s) ≠ g(s))
3     if (s ∉ CLOSED) insert/update s in OPEN with key(s);
4     else if (s ∉ INCONS) insert s in INCONS;
5   else
6     if (s ∈ OPEN) remove s from OPEN;
7     else if (s ∈ INCONS) remove s from INCONS;
```

O algoritmo é muito semelhante em sua execução ao ARA*, já que o AD* é uma adaptação do mesmo (MOURA; RITT; BURIOL, 2010). A função principal (listagem 4.7) começa iniciando os valores de g(s_{goal}), v(s_{goal}) e v(s_{start}) como ∞, bp(s_{start}) como nulo e g(s_{start}) como 0. Em seguida o vértice s_{start} é inserido na fila de prioridades de acordo com a função de chave ordenadora descrita na listagem 4.6. Disso, é invocado a função de cálculo de caminho (listagem 4.5).

Essa função trabalha da seguinte forma: inicialmente se verifica se o valor da chave de s_{goal} é maior do que a menor chave da fila de prioridades (condição semelhante ao do algoritmo ARA*, vide listagem 4.1) ou se o valor de v(s_{goal}) é menor do que o valor de g(s_{goal}). Se caso for verdade, o processo iterativo é inciado. O vértice s, que corresponde ao vértice com menor chave na fila de prioridades é removido do conjunto OPEN, e é verificado se o valor de v(s) é maior do que o g(s) (ou seja, se o valor da distância calculada

[2] Busca aqui, se refere ao número da iteração da invocação da função de cálculo de caminho (listagem 4.1), e consequentemente ao valor de ϵ a ele atribuído.

Listagem 4.5 – Algoritmo AD* - função de cálculo de caminho

```
1  procedure ComputePath()
2    while(key(s_goal) > min_{s∈OPEN(key(s))} OR v(s_goal) < g(s_goal))
3      remove s with smallest key(s) from OPEN;
4      if v(s) > g(s)
5        v(s) = g(s); CLOSED = CLOSED ∪ {s};
6        for each successor s' of s
7          if s' was never visited by AD* before then
8            v(s') = g(s') = ∞;bp(s') = null;
9          if g(s') > g(s) + c(s, s')
10           g(s') = g(s) + c(s, s');
11           bp(s') = s;
12           g(s') = g(bp(s')) + c(bp(s'),s'); UpdateSetMembership(s');
13     else
14       v(s) = ∞; UpdateSetMembership(s);
15       for each successor s' of s
16       if s' was never visited by AD* before then
17         v(s') = g(s') = ∞;bp(s') = null;
18       if bp(s') = s
19         bp(s') = argmin_{s''∈pred(s')} v(s'') + c(s'',s');
20         g(s') = v(bp(s')) + c(bp(s'),s'); UpdateSetMembership(s');
```

Listagem 4.6 – Algoritmo AD* - função da chave ordenadora da fila de prioridades

```
1  procedure key(s)
2    if (v(s) ≥ g(s))
3      return [g(s) + ϵ * h(s); g(s)];
4    else
5      return [v(s) + h(s); v(s)];
```

Listagem 4.7 – Algoritmo AD* - função principal

```
1  procedure Main()
2    g(s_goal) = v(s_goal) = ∞; v(s_start) = ∞;bp(s_goal) = bp(s_start) = null;
3    g(s_start) = 0; OPEN = CLOSED = INCONS = ∅; ϵ = ϵ0;
4    insert s_start into OPEN with key(s_start);
5    forever
6      ComputePath();
7      publish current ϵ−suboptimal solution;
8      if ϵ = 1
9        wait changes in edge costs;
10     for all directed edges (u,v) with changed edge costs
11       update the edge cost c(u,v);
12       if ( v ≠ s_start AND v was visited by AD* before)
13         bp(v) = argmin_{s''∈pred(v)} v(s'') + c(s'',v);
14         g(v) = v(bp(v)) + c(bp(v),v); UpdateSetMembership(v);
15     if significant edge cost changes were observed
16       increase ϵ or re−plan from scratch (i.e., re−excute Main function);
17     else if ϵ > 1
18       decrease ϵ;
19     Move states from INCONS into OPEN
20     Update the priorities for all s ∈ OPEN according to key(s);
21     CLOSED = ∅;
```

da busca anterior é maior do que o valor da busca atual, esse é o caso normal do algoritmo). Sendo verdade, o algoritmo segue a mesma forma de execução do ARA*, conforme descrito na seção 4.2. A diferença de tratamento ocorre quando a condição inicial não ocorre, e neste caso, o valor de v(s) é atribuído como ∞, e para cada vértice s' adjacente de s é verificado se este já havia sido visitado pelo algoritmo. Se caso não havia sido visitado, os valores de v(s'), g(s') são inciados como ∞ e bp(s') é inciado como nulo. Seguindo, é verificado se o bp(s') é igual o vértice s. Se caso sim, o bp(s') é atualizado com o vértice predecessor de s', s", cuja a função v(s") + c(s", s') é a menor dentre todos os vértices predecessores de s'. Feito isso, o valor de g(s') é atualizado com esse novo valor. Em seguida a função de determinação para qual conjunto pertencerá s' é chamado (função "UpdateSetMembership").

A função de determinação ao qual conjunto pertencerá funciona de uma maneira muito simples. Caso os valores de v(s) e g(s) sejam diferentes, a função é igual ao que ocorre no processo de atribuição de conjunto do ARA* (vide listagem 4.1), a diferença ocorre quando esses valores são iguais (ou seja, não houve mudança de valores entre a busca atual e a anterior), o vértice é removido de OPEN caso esteja nele ou removido de INCONS caso pertença a este. Isso feito para que este vértice não precise ser calculada nesta busca ou na próxima (caso esteja em INCOS).

Voltando ao método principal (listagem 4.7), a função segue o mesmo padrão da ARA*, a única exceção se dá quando mudanças no grafo são detectadas (mudança dos pesos das arestas). Neste caso, para cada aresta (u,v) mudada, é atribuído ao valor bp(v), o vértice s", predecessor de v, cuja a função v(s") + c(s", v) seja mínima. Consequentemente, o valor de g(v) é atualizado conforme esse valor e a função de atribuição de conjunto é invocada.

5 Testes Computacionais

5.1 Algoritmo de Dijkstra

Para a realização dos experimentos computacionais será rodado instâncias de grafos que representam malhas rodoviárias reais. Todas elas descritas nas tabelas 2 e 3, e disponíveis no sítio eletrônico <http://www.dis.uniroma1.it/challenge9/download.shtml> (acesso em 28 de janeiro de 2017).

Tabela 2 – Instâncias a serem rodadas pelo algoritmo de Dijkstra em suas três versões.

Nome Instância	**Descrição**
USA-road-d.NY.gr	Representa a malha viária do estado de Nova Iorque, Estados Unidos
USA-road-d.BAY.gr	Representa a malha viária da bahia de São Francisco, Califórnia, Estados Unidos
USA-road-d.COL.gr	Representa a malha viária do estado do Colorado, Estados Unidos
USA-road-d.FLA.gr	Representa a malha viária do estado da Flórida, Estados Unidos

Tabela 3 – Configuração dos grafos correspondestes as malhas viárias descritas na tabela 2.

Nome Instância	**Número de Vértices |V|**	**Número de Arestas |E|**
USA-road-d.NY.gr	264.346	733.846
USA-road-d.BAY.gr	321.270	800.172
USA-road-d.COL.gr	435.666	1.057.066
USA-road-d.FLA.gr	1.070.376	2.712.798

5.1.1 Resultados obtidos

Os resultados dos testes obtidos estão descritos a seguir.

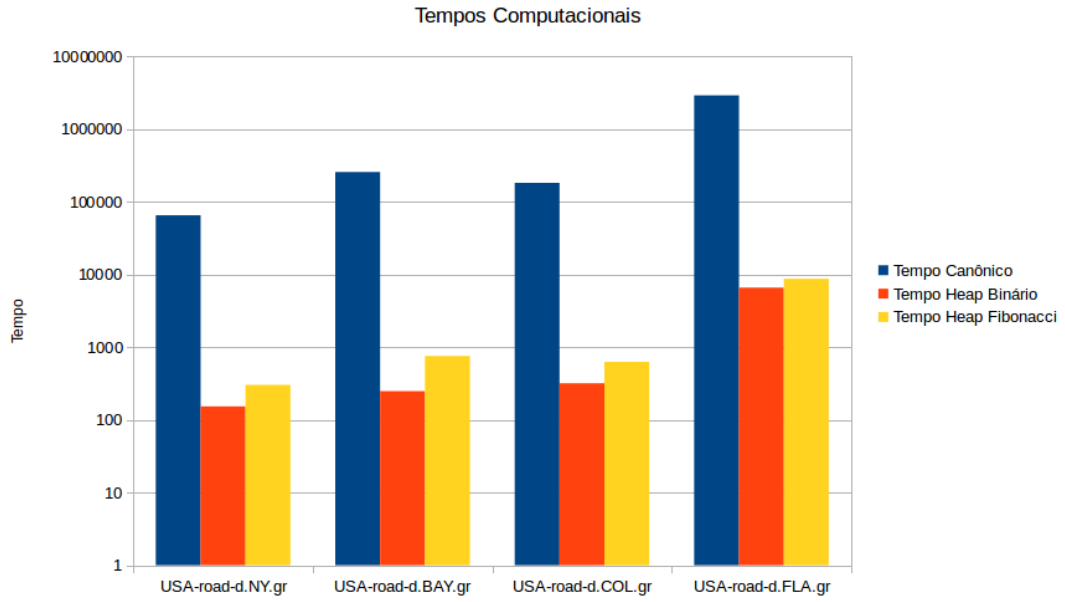

Figura 10 – Tempos computacionais obtidos pelos Métodos de Dijkstra empregados (tempo em escala logarítmica).

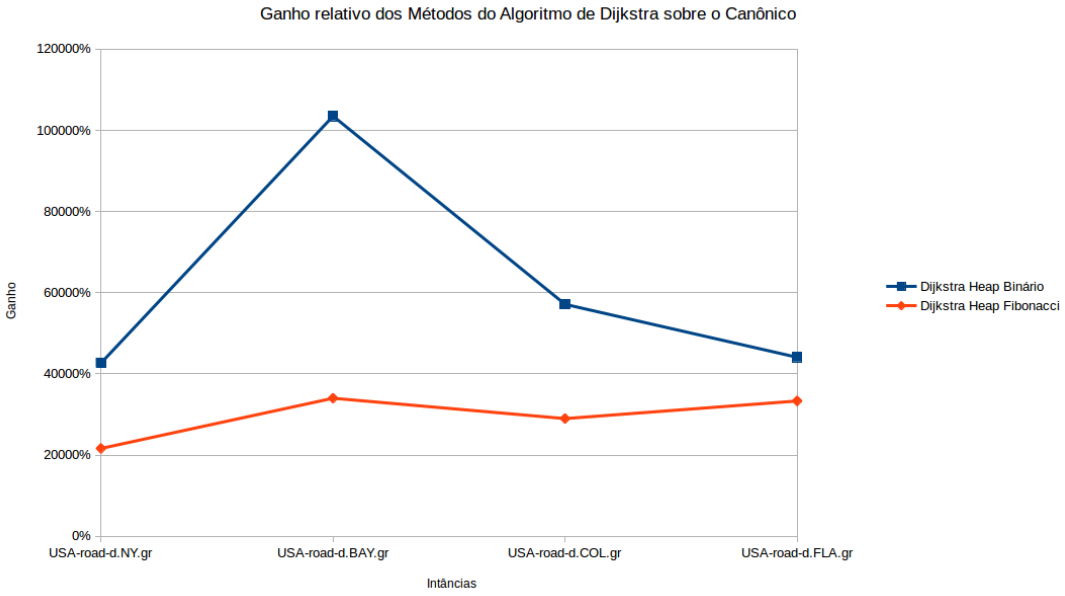

Figura 11 – Ganho relativo dos Métodos do Algoritmo de Dijkstra sobre o Canônico.

Tabela 4 – Tempos Computacionais obtidos pelo algoritmo de Dijkstra em suas diferentes versões (tempo em milissegundos).

Nome Instância	**Tempo Canônico**	**Tempo Heap Binário**	**Tempo Heap Fibonacci**
USA-road-d.NY.gr	65287	153	302
USA-road-d.BAY.gr	256690	248	755
USA-road-d.COL.gr	181687	318	627
USA-road-d.FLA.gr	2909655	6601	8732

Tabela 5 – Ganho relativo sobre o Dijkstra Canônico.

Nome Instância	**Ganho Heap Binário**	**Ganho Heap Fibonacci**
USA-road-d.NY.gr	42.671%	21.618%
USA-road-d.BAY.gr	103.504%	33.999%
USA-road-d.COL.gr	57.134%	28.977%
USA-road-d.FLA.gr	44.079%	33.322%

5.1.2 Análise dos resultados

Podemos observar que o Dijkstra Canônico elevou um tempo consideravelmente grande (a instância USA-road-d.FLA.gr por exemplo levou aproximadamente 48 minutos). Já o uso de estrutura de dados impactou consideravelmente no ganho do tempo sendo que o Heap Binário teve um ganho médio de 61.847% com relação ao Dijkstra Canônico enquanto Dijkstra Heap de Fibonacci teve um ganho médio de 29.479% (vide tabela 5).

Com relação ao resultado obtido pelos métodos do Heap Binário e Heap de Fibonacci, ele é de certo modo inesperado, já que conforme demonstrados nas subseções 2.2.1 e 2.2.2, o heap de Fibonacci possui tempo computacional, aplicado ao algoritmo de Dijkstra, de $O(|V| \lg |V| + |E|)$ enquanto o heap binário possui $O(|E| \lg |V|)$. Como para todas as instâncias rodadas $|E| > |V|$ (vide tabela 3), era de se esperar do ponto de vista teórico que a heap de Fibonacci apresentasse tempos mais rápidos do que o Heap Binário.

Porém, conforme também constatado por Larkin, Sen e Tarjan (2014), a aplicação prática das estruturas de dados nem sempre corresponde a esperada descrita na teoria. Larkin, Sen e Tarjan (2014) mostram que estrutura de dados heaps baseadas em vetor são, na prática, mais eficientes do que a Heap de Fibonacci (vide referência para mais detalhes). É o que os testes realizados por este trabalho também constatam.

5.1.3 Conclusões

Conforme demonstramos pelos experimentos descritos nesta seção, o algoritmo de Dijkstra que obteve o melhor resultado foi o Heap Binário, sendo mais rápido do que o próprio Heap de Fibonacci que teoricamente deveria ser mais rápido. O Dijkstra canônico obteve tempos que para aplicações como sistema de ponto global (em inglês, GPS) é indesejado, sendo sua implementação interessante apenas para fins de aprendizado e entendimento do algoritmo.

Em termos de implementação, sem dúvida o mais complicado para se implementar foi o Heap de Fibonacci devido a sua própria estrutura que contém uma lista circular duplamente encadeada (a lista de raízes), e pelas funções de restruturação da estrutura que possui muitas movimentações de nodos e tratamento de casos de desvio de condição.

Com isso, colocando em termos práticos, das formas de implementação apresentadas

e testadas, a que melhor se sobressai é o Heap Binário que não só foi melhor no tempo dentre outros, como sua implementação é simples.

5.2 Algoritmo A*

Para os experimentos computacionais serão utilizados as mesmas instâncias descritas em subseção 5.1. Serão comparados quatro versões de algoritmos: o algoritmo de Dijkstra descrito no capítulo 2, o algoritmo de Dijkstra adaptado onde o algoritmo é parado assim que se é explorado o vértice objetivo, o algoritmo A* onde se utiliza a heurística admissível distância euclidiana e o algoritmo A* onde se utiliza a heurística não-admissível distância Manhattan, todas sumarizadas na tabela 6.

Tabela 6 – Descrição das versões a serem testadas neste capítulo.

Nome Instância	**Descrição**
Dijkstra	Versão de Dijsktra conforme descrito no capítulo 2
Dijkstra Adaptado	Versão de Dijkstra adaptado para parar quando o vértice destino é encontrado
Algoritmo A*	Algoritmo A* utilizando a distância euclidiana
Algoritmo A* Manhattan	Algoritmo A* utilizando a distância Manhattan

Será rodado dez vezes cada algoritmo para cada instância da subseção 5.1 em que para cada rodada, será afixado o vértice origem como o sendo de valor de identificação "0"e terá como vértice destino um vértice escolhido aleatoriamente, sendo que não haverá repetição de vértices[1] (ou seja, supomos que o vértice de valor de identificação "180"tenha sido escolhido na primeira rodada. Esse vértice não será escolhido como destino nas demais 9 rodadas. Caso esse vértice seja "sorteado"na próxima iteração, um novo vértice será escolhido aleatoriamente). Nessas dez rodadas será verificado o tempo médio de execução, o número médio de vértices abertos por cada versão e para o algoritmo A* com a heurística Manhattan, será verificado a qualidade da solução.

Para todas as versões será utilizada a estrutura de dados Heap Binário (descrito na subseção 2.2.1) pois conforme mostrado no capítulo 2, foi a estrutura que melhor se sobressaiu entre as outras em termos de tempo computacional.

5.2.1 Resultados obtidos

Os resultados dos testes computacionais descritos anteriormente podem ser vistos nas tabelas 7, 8 e 9.

[1] Para o caso do algoritmo de Dijkstra especificado na primeira linha da tabela 6, não será estabelecido um vértice destino já que o algoritmo calcula a melhor rota para todos os vértices do grafo.

Tabela 7 – Tempo médio obtido pelos métodos descritos na tabela 6 (tempo em milissegundos).

Nome Instância	Dijkstra	Dijkstra Adaptado	Algoritmo A*	Algoritmo A* Manhattan
USA-road-d.NY.gr	236	109	29	14
USA-road-d.BAY.gr	321	187	54	23
USA-road-d.COL.gr	494	229	117	27
USA-road-d.FLA.gr	2858	1026	886	101

Tabela 8 – Diferença média de solução obtida pelo A* Manhattan com relação a solução ótima.

Nome Instância	Qualidade Solução
USA-road-d.NY.gr	5%
USA-road-d.BAY.gr	4%
USA-road-d.COL.gr	3%
USA-road-d.FLA.gr	3%

Tabela 9 – Número de Vértices Abertos (NVA) médio por cada método.

Nome Instância	NVA Dijkstra Adptado	NVA A*	NVA A* Manhattan
USA-road-d.NY.gr	140501	31995	9079
USA-road-d.BAY.gr	225754	53206	17044
USA-road-d.COL.gr	242674	108343	21577
USA-road-d.FLA.gr	580010	349201	79077

5.2.2 Análise dos resultados

Como apontado pelos testes realizados, o algoritmo A* teve um desempenho computacional melhor do que o algoritmo Dijkstra, inclusive sobre o Dijkstra Adaptado (descrito anteriormente). Podemos ver que esse resultado está diretamente ligado ao número de vértices abertos por cada algoritmo (tabela 9). Aqueles que abriram mais vértices, obtiveram um tempo computacional maior. Isso já esperado, já que o algoritmo teve que processar mais etapas até que sua condição de parada fosse encontrada.

Dos quatro métodos testados, o que obteve menor tempo computacional foi o algoritmo A* aplicando a heurística não-admissível Distância Manhattan. Isso se deve ao fato de o cálculo da distância (que é realizado em tempo de execução) ser mais simples do que o empregado pela distância euclidiana (que envolve radiação e exponenciação). Porém esse resultado possui um "preço a ser pago"que é, conforme descrito na subseção 3.1.1, a não garantia do menor caminho entre os vértices pesquisados. Mas, conforme demonstra a tabela 8, a diferença média entre as soluções encontradas e suas respectivas soluções ótimas giram em torno de 4%, o que pode ser considerado como um bom resultado.

5.2.3 Conclusões

O algoritmo A* demonstra ser um ótimo algoritmo para o cálculo de menor caminho entre dois vértices (origem e destino), superando em tempo computacional o algoritmo de Dijsktra (mas que retorna a menor rota para todos os demais vértices), sendo mais indicado para esse tipo de cálculo. Sua implementação é simples e é praticamente uma adaptação do algoritmo de Dijsktra.

Com relação as heurísticas, para a garantia do melhor caminho como o algoritmo de Dijkstra o faz, é obrigatório o uso de uma heurística admissível, mesmo que isso tenha um impacto negativo no tempo com relação ao uso de outras heurísticas (as não-admissíveis). Porém, conforme mostrado na subseção refsec-aestrela-instancias-resultados, a diferença entre as soluções ótimas e obtidos pelo método Manhattan giraram em torno de 4%, o que é um bom resultado para quem deseja menor tempo computacional e não possui a obrigatoriedade do menor caminho.

5.3 Algoritmos Dinâmicos

A ser realizado.

6 Considerações Finais

Referências

CORMEN, T. H. *Introduction to algorithms*. [S.l.]: MIT press, 2009. Citado 6 vezes nas páginas 15, 16, 17, 18, 19 e 20.

DIJKSTRA, E. W. A note on two problems in connexion with graphs. *Numerische mathematik*, Springer, v. 1, n. 1, p. 269–271, 1959. Citado 2 vezes nas páginas 13 e 15.

DROZDEK, A. *Data Structures and algorithms in C++*. [S.l.]: Cengage Learning, 2012. Citado 3 vezes nas páginas 13, 15 e 17.

HART, P. E.; NILSSON, N. J.; RAPHAEL, B. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, IEEE, v. 4, n. 2, p. 100–107, 1968. Citado na página 21.

KOENIG, S.; LIKHACHEV, M. D* lite. *AAAI/IAAI*, v. 15, 2002. Citado na página 25.

LARKIN, D. H.; SEN, S.; TARJAN, R. E. A back-to-basics empirical study of priority queues. In: SIAM. *2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. [S.l.], 2014. p. 61–72. Citado na página 33.

LIKHACHEV, M. et al. Anytime search in dynamic graphs. *Artificial Intelligence*, Elsevier, v. 172, n. 14, p. 1613–1643, 2008. Citado 4 vezes nas páginas 14, 21, 25 e 28.

MOURA, L.; RITT, M.; BURIOL, L. S. Estudo experimental de algoritmos em tempo real de caminho mınimo ponto a ponto em grafos dinâmicos. *Anais do XLII Simpósio Brasileiro de Pesquisa Operacional, ser. SBPO*, 2010. Citado 3 vezes nas páginas 13, 27 e 28.

RUSSELL, S. J.; NORVIG, P. *Artificial Intelligence: A Modern Approach*. [S.l.]: Prentice-Hall, 1995. Citado 2 vezes nas páginas 22 e 23.

STENTZ, A. Optimal and efficient path planning for partially-known environments. In: IEEE. *Robotics and Automation, 1994. Proceedings., 1994 IEEE International Conference on*. [S.l.], 1994. p. 3310–3317. Citado na página 25.

ZIVIANI, N. et al. *Projeto de algoritmos: com implementações em Pascal e C*. [S.l.]: Thomson, 2004. v. 2. Citado na página 13.

Apêndices