

Pedro Henrique Brunoro Hoppe

Estudo sobre Algoritmos de Menor Caminho em Grafos

Vitória, ES

2017

Pedro Henrique Brunoro Hoppe

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. **a.** Maria Claudia Silva Boeres

Vitória, ES

2017

Pedro Henrique Brunoro Hoppe

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**, 03 de agosto de 2017:

Prof. Dr^a. Maria Claudia Silva Boeres
Orientador

Prof. Dr^a. Maria Cristina Rangel
Convidado 1

Prof. Me. Edmar Hell Kampke
Convidado 2

Vitória, ES
2017

Resumo

Este trabalho propõe um estudo de algoritmos de menor caminho em grafos, tanto algoritmos de grafos estáticos quanto dinâmicos, descrevendo sua forma, estudando seu funcionamento, testando seu desempenho computacional e analisando em quais situações melhor se aplicam. Inicialmente é descrito o algoritmo clássico de Dijkstra, no qual é apresentado seu funcionamento, seguido de suas versões de implementação baseados em estrutura de dados que representam a fila de prioridade de ordenamento dos vértices. Em seguida é apresentado o algoritmo busca A^* , que pode ser visto como uma adaptação do algoritmo de Dijkstra. É apresentada a estratégia do uso de heurística, tanto a admissível quanto não-admissível, a qual o algoritmo busca A^* utiliza, e o impacto causado por ela. Em seguida, é apresentado o conceito de grafo dinâmico e os algoritmos ARA^* e AD^* , sendo que o AD^* é uma adaptação do ARA^* . São descritos seus funcionamentos e a utilização da estratégia da heurística inflada. Finalmente então, é descrito os testes computacionais realizados para cada algoritmo, a descrição das instâncias escolhidas, análise dos resultados obtidos e as conclusões sobre em quais situações o uso desses algoritmos melhor se aplicam.

Palavras-chaves: grafos estáticos, grafos dinâmicos, algoritmos de menor caminho, desempenho, busca A^* , Dijkstra, ARA^* , AD^* .

Lista de ilustrações

| | |
|---|----|
| Figura 1 – Exemplo de um grafo. | 8 |
| 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. | 12 |
| Figura 3 – Exemplo de Heap Binário representado como árvore. | 14 |
| Figura 4 – Representação do Heap Binário da figura 3 como vetor. | 14 |
| 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). | 15 |
| Figura 6 – Heap de Fibonacci da figura 5 após a operação de extração de mínimo. | 16 |
| Figura 7 – Mapa da Romênia com os valores das distâncias entre as cidades e a distância euclidiana de todas elas até Bucareste. | 18 |
| Figura 8 – Desenvolvimento do algoritmo A*. | 19 |
| Figura 9 – Exemplo de aplicação da estratégia de diminuição do valor de ϵ | 24 |
| Figura 10 – Tempos computacionais obtidos pelos Métodos de Dijkstra empregados (tempo em escala logarítmica). | 28 |
| Figura 11 – Ganho relativo dos Métodos do Algoritmo de Dijkstra sobre o Canônico. | 28 |

Lista de tabelas

| | |
|--|----|
| Tabela 1 – Instâncias a serem rodadas pelo algoritmo de Dijkstra em suas três versões. | 27 |
| Tabela 2 – Configuração dos grafos correspondentes as malhas viárias descritas na tabela 1. | 27 |
| Tabela 3 – Tempos Computacionais obtidos pelo algoritmo de Dijkstra em suas diferentes versões (tempo em milissegundos). | 28 |
| Tabela 4 – Ganho relativo sobre o Dijkstra Canônico. | 29 |
| Tabela 5 – Descrição das versões a serem testadas neste capítulo. | 30 |
| Tabela 6 – Tempo médio obtido pelos métodos descritos na tabela 5 (tempo em milissegundos). | 31 |
| Tabela 7 – Diferença média de solução obtida pelo A* Manhattan com relação a solução ótima. | 31 |
| Tabela 8 – Número de Vértices Abertos (NVA) médio por cada método. | 31 |
| Tabela 9 – Resultado testes ARA*. | 33 |
| Tabela 10 – Resultados obtidos para AD*. | 34 |

Sumário



| | | |
|----------|--|-----------|
| 1 | INTRODUÇÃO | 8 |
| 1.1 | Revisão bibliográfica e trabalhos correlatos | 9 |
| 1.2 | Metodologia de pesquisa | 9 |
| 1.3 | Organização dos capítulos | 10 |
| 2 | ALGORITMO DE DIJKSTRA | 11 |
| 2.1 | O Algoritmo | 11 |
| 2.2 | Versões do Algoritmo implementadas e suas Estrutura de Dados | 13 |
| 2.2.1 | Dijkstra Heap Binário | 13 |
| 2.2.2 | Dijkstra Heap de Fibonacci | 14 |
| 3 | ALGORITMO A* | 17 |
| 3.1 | O Algoritmo | 17 |
| 3.1.1 | Heurísticas admissíveis e não-admissíveis | 20 |
| 4 | ALGORITMOS DINÂMICOS | 21 |
| 4.1 | Grafos Dinâmicos | 21 |
| 4.2 | Algoritmo ARA* | 21 |
| 4.2.1 | Heurística inflada e considerações sobre o algoritmo | 23 |
| 4.3 | Algoritmo AD* | 24 |
| 5 | TESTES COMPUTACIONAIS | 27 |
| 5.1 | Algoritmo de Dijkstra | 27 |
| 5.1.1 | Resultados obtidos | 27 |
| 5.1.2 | Análise dos resultados | 29 |
| 5.1.3 | Conclusões | 29 |
| 5.2 | Algoritmo A* | 30 |
| 5.2.1 | Resultados obtidos | 31 |
| 5.2.2 | Análise dos resultados | 31 |
| 5.2.3 | Conclusões | 32 |
| 5.3 | Algoritmos Dinâmicos | 32 |
| 5.3.1 | ARA* | 32 |
| 5.3.1.1 | Resultados obtidos | 33 |
| 5.3.1.2 | Análise dos resultados | 33 |
| 5.3.2 | AD* | 33 |
| 5.3.2.1 | Resultados obtidos | 34 |



| | | |
|----------|---|-----------|
| 5.3.2.2 | Análise dos resultados | 34 |
| 5.3.3 | Conclusões | 35 |
| 6 | CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS | 36 |
| | REFERÊNCIAS | 38 |

1 Introdução



Na computação muitas aplicações necessitam considerar um conjunto de conexões entre dados e uso de algoritmos **nesses** dados para poder responder perguntas **como**, se existe um caminho entre **um objeto a outro** seguindo por essas conexões, qual a menor distância 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), que é representado em termos de seus vértices e arestas.

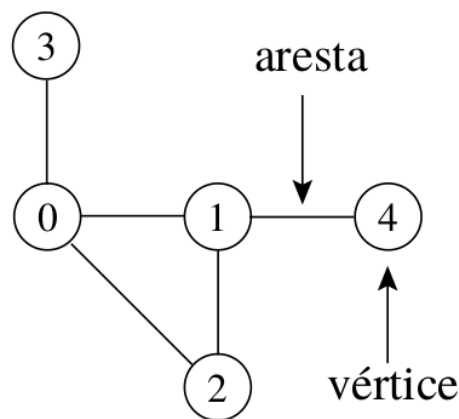


Figura 1 – Exemplo de um grafo.

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 muito utilizada 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 os 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 as distâncias entre essas interseções.

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 Revisão bibliográfica e trabalhos correlatos

O algoritmo de Dijkstra é amplamente difundido na literatura, sendo que é utilizado como base para o estudo desse algoritmo os livros de [Cormen \(2009\)](#) e [Drozdek \(2012\)](#), em que abordam a descrição do algoritmo, estruturas de dados e a análise computacional das mesmas.

O algoritmo A* é bastante difundido na literatura. Foi proposto por [Hart, Nilsson e Raphael \(1968\)](#). Ele é descrito no livro de [Russell e Norvig \(1995\)](#) e no artigo de [Likhachev et al. \(2008\)](#). O pseudo-código descrito neste livro e artigo são usados como base para a implementação desse algoritmo neste projeto.



O estudo dos algoritmos ARA* e AD* é baseado no artigo no qual esses algoritmos foram propostos ([LIKHACHEV et al., 2008](#)), além do artigo em que investiga esses algoritmos ([MOURA; RITT; BURIOL, 2010](#)).

O trabalho de [Larkin, Sen e Tarjan \(2014\)](#) analisa o uso de diversas estruturas de dados como fila de prioridades e seus respectivos impactos, ressaltando diferenças entre a proposta teórica dessas estruturas e a aplicação prática.

1.2 Metodologia de pesquisa

Todos os algoritmos estudados foram implementados na linguagem Java, a versão do compilador utilizado é Java 1.8.0_131. Os testes a serem realizados e as instâncias utilizadas estão descritos no capítulo 5. A configuração do computador onde os testes foram realizados **é o seguinte: o** sistema operacional **é o** Linux Mint 18.2 Sonya 64-bit. **O** computador possui processador Intel Core i5-2430M @ 2.40 GHz, cache 6MB, memória de 4.0 GB.

1.3 Organização dos capítulos

Este trabalho está estruturado da seguinte forma:

~~Capítulo 1:~~ Esta introdução.

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 \(2012\)](#).

Algoritmo 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 à distância do vértice de origem v_s até u), além de definir o antecessor **u** como v . Repete-se o passo iterativo até que o conjunto “toBeChecked” esteja vazio¹.

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

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 . O caminho retornado é garantidamente ótimo como mostra [Cormen \(2009\)](#).

A figura 2 mostra um exemplo de aplicação do algoritmo a um grafo.

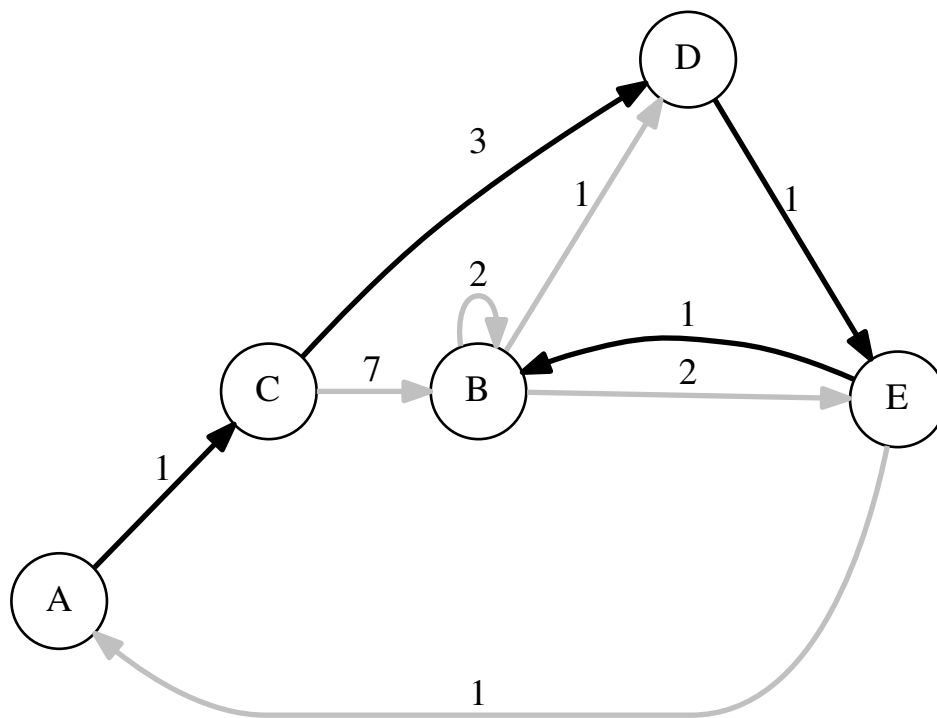


Figura 2 – Aplicação do algoritmo de Dijkstra tendo o vértice “A” como origem. As arestas pintadas de preto correspondem a rota calculada a todos os demais vértices.

Inicialmente a distância do vértice “A” é atribuído como zero enquanto de todos os demais é atribuído como ~~infinito~~ ∞ . Inicia-se o processo iterativo a partir de “A” que explora os seus vértices adjacentes, que neste caso só tem um que é o vértice “C”. A distância de “C” é calculada como o valor da distância de “A” mais o peso da aresta “AC” (que é igual a 1), e o vértice que é atribuído como antecessor de “C” é “A”. Em seguida é escolhido o vértice cuja a distância seja a menor dentro do conjunto “toBeChecked” que neste caso é o próprio “C”. Do vértice “C” explora-se os vértices adjacentes **dele** “D” e “B”. Suas respectivas distâncias são atribuídas como a distância de “C” mais o peso de suas respectivas arestas ($\text{currDist}(B) = 7 + 1$; $\text{currDist}(D) = 3 + 1$), além de atribuir como “C” os seus respectivos vértices antecessores. **Disso novamente**, escolhe-se o vértice com

menor distância em “toBeChecked” que neste caso será o D ($\text{currDist}(D) = 4 < \text{currDist}(C) = 8$). De “D” é explorado seu vértice adjacente “E” que é atribuído seu valor $\text{currDist}(E)$ como sendo 5 ($3 + 1 + 1$) e seu vértice antecessor como “D”. Busca-se novamente o menor valor dos vértices em “toBeChecked” que neste caso será o “E” ($\text{currDist}(E) = 5 < \text{currDist}(B) = 8$). Dele explora-se os seus vértices adjacentes “B” e “A”. O único valor da distância que é alterado é o de “B” pois o caminho vindo por “E” ($A \rightarrow C \rightarrow D \rightarrow E = 1 + 3 + 1 + 1$) é menor do que o vindo por “C” ($A \rightarrow C \rightarrow B = 1 + 7$). Finalmente, o vértice “B” é explorado, mas nenhum de seus vértices adjacentes tem o valor de sua distância alterado pois o menor caminho para eles já foi encontrado.

2.2 Versões do Algoritmo implementadas e suas Estrutura de Dados

Neste projeto de graduação foram implementadas três versões do algoritmo de Dijkstra usando estruturas de dados diversas.

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 um vetor para armazenar as distâncias calculadas pelo algoritmo (o índice dos vértices correspondem ao índice 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 distância é menor dentre todas as outras. A complexidade 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 binárias 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 posicionadas mais a esquerda.

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

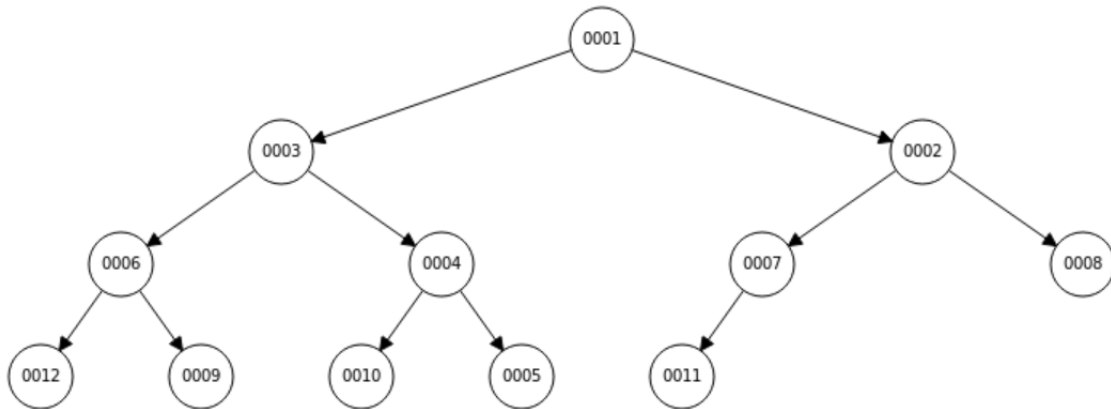


Figura 3 – Exemplo de Heap Binário representado como árvore.

| | | | | | | | | | | | |
|------|------|------|------|------|------|------|------|------|------|------|------|
| 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 $i \in [1, n]$, 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 possuírem complexidade 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, deixá-la balanceada), apenas as adicionando à lista principal de raízes. Por consequência, operações de inserção possuem tempo computacional $O(1)$ (CORMEN, 2009).

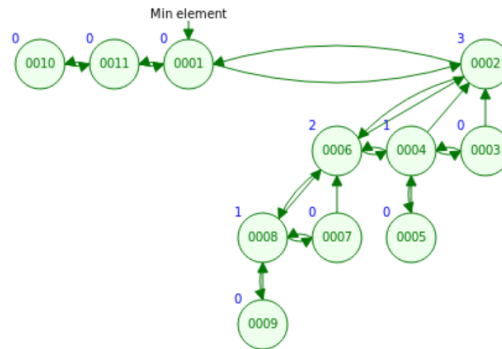


Figura 5 – Exemplo de Heap de Fibonacci (os números no canto superior esquerdo de cada nodo correspondem ao grau de cada um, ou seja, o número de filhos).

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 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. Em seguida, é preciso 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 agrupar raízes com o mesmo número de grau (grau corresponde ao número de filhos que cada nodo possui) e para cada par de nodos agrupados, 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 o agrupamento de todos os nodos com mesmo número de grau, uma busca linear é realizada para se determinar o menor elemento entre os nodos raízes restantes². O tempo computacional para a extração de mínimo é $O(\lg n)$ (CORMEN, 2009).

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 linkagem é feito conforme descrito.

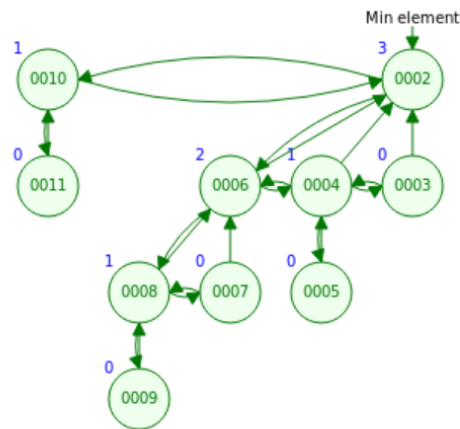


Figura 6 – Heap de Fibonacci da figura 5 após a operação de extração de mínimo.

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 da heap foi violada. Em caso afirmativo, esse nodo é cortado de seu nodo pai e colocado junto à lista principal. Se o pai não pertencer a lista de nodos raízes, ele é “marcado”, e caso já estivesse “marcado” ele também é cortado e seu pai é “marcado”. Esse processo continua até encontrarmos um nodo pai “não marcado” ou um nodo raiz. No final do processo, verifica-se se o nodo modificado inicialmente é menor do que o nodo mínimo atual. Neste caso, o novo nodo mínimo é o modificado. O tempo computacional é $O(1)$ (CORMEN, 2009).

Por consequência, a complexidade aplicado, neste caso, 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 informada 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 ótima¹) 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 do grafo.

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

Algoritmo 3.1 – Algoritmo A*

```

1 A*Algorithm(weighted simple digraph, vertex first, vertex goal)
2   for all vertices v
3     g(v) =  $\infty$ ;
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
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 “toBeChecked”³. Feito isso inicia-se o processo iterativo: enquanto o vértice “goal” estiver dentro do conjunto “toBeChecked” (ou seja, o vértice “goal” não foi alcançado ainda), 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 ($\text{edge}(vu)$). Em caso afirmativo, o valor de $g(u)$ é atualizado para $g(v)$ mais o peso da

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.

aresta entre v e u , e v é marcado como o predecessor de u . O valor do peso do vértice u 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 as distâncias para todos os vértices do grafo. O termo $h(u)$ consiste no 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 calculados primeiro. 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 discutido na subseção 3.1.1.

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

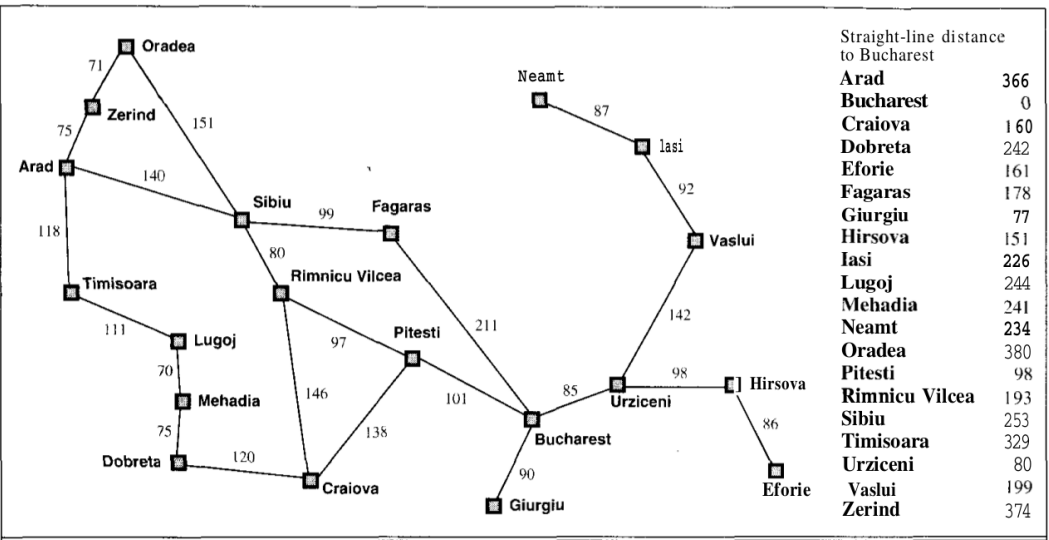


Figura 7 – Mapa da Romênia com os valores das distâncias entre as cidades e a distância euclidiana de todas elas até Bucareste.

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 grafo conforme descrito na figura 8⁴, tendo como heurística utilizada, a distância euclidiana entre todas as cidades e Bucareste (distâncias também representadas na figura 7).

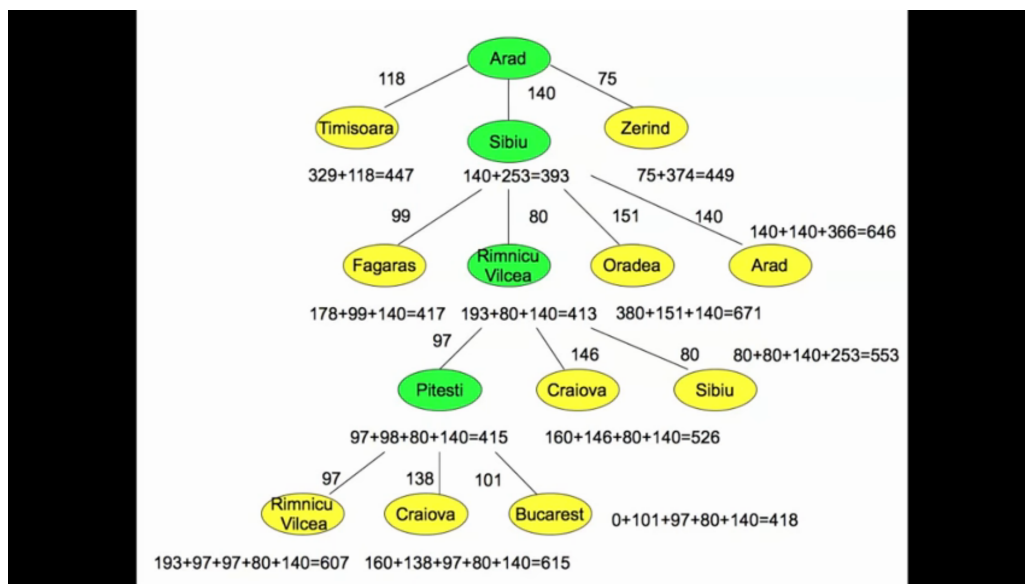




Figura 8 – Desenvolvimento do algoritmo A*.

O procedimento é bem semelhante ao de Dijkstra (descrito no capítulo 2). Inicialmente é atribuído o valor de distância de todos os vértices $g(v)$ como ∞ , com exceção do vértice de origem que é atribuído como zero. Neste exemplo da figura 8, o vértice de origem é a cidade de “Arad”. Dela se expande para seus vértices vizinhos, que no caso são as cidades de “Timisoara”, “Sibiu” e “Zerind”. A fila de prioridades é atualizado conforme a função $f(u) = g(u) + h(u)$. Sendo assim o próximo vértice a ser explorado é a cidade de “Sibiu”. Vemos o fator heurístico “pesando” na escolha do próximo vértice a ser escolhido, pois por Dijkstra, o próximo vértice a ser escolhido seria “Zerind”, ao invés de “Sibiu”, pois aquela possui aresta com peso menor do que esta. Dela se expande para as cidades vizinhas de “Fagaras”, “Rimnicu Vilcea”, “Oradea” e “Arad”. “Rimnicu Vilcea” é tida a cidade mais promissora e dela continua a expansão para as suas cidades vizinhas de “Pitesti”, “Craiova” e “Sibiu”. “Pitesti” é escolhida. E dela se expande para as suas cidades vizinhas de “Rimnicu Vilcea”, “Craiova” e finalmente “Bucarest” que é a cidade destino e que tem fator heurístico igual a zero, sendo portanto mais provável a sua escolha na próxima iteração, terminando assim o algoritmo.

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.

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  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:

$$\text{■ } h(u) = |x_u - x_{goal}| + |y_u - y_{goal}|;$$



- Atalho Diagonal:



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

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

4 Algoritmos Dinâmicos

4.1 Grafos Dinâmicos

Os algoritmos apresentados até agora tratam apenas de 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 requer que os pesos de suas arestas variem com o 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 Dijkstra e A* (discutidos nos capítulos 2 e 3) para acharmos a rota, e quando houver uma detecção de mudança do peso de arestas, recalculá-los novamente o trajeto reutilizando esses algoritmos. Em geral 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).

Será discutidos os algoritmos ARA* e AD* (LIKHACHEV et al., 2008), ~~que foram escolhidos para serem estudados por este trabalho.~~

4.2 Algoritmo ARA*

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



A função principal é descrita no **Algoritmo 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. Ela é utilizada, na verdade, no próximo algoritmo, AD*, que está descrito no mesmo artigo do ARA* e foi do critério de seus autores originais colocar essa variável em sua descrição. Portanto será ignorado na explicação que se segue).** O valor de $g(s_{goal})$ (distância real calculada da origem ao vértice destino) é atribuído como

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

```

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

```

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

```

1 procedure key( $s$ )
2   return  $g(s) + \epsilon * h(s)$ ;

```

Algoritmo 4.3 – Algoritmo ARA* - função principal

```

1 procedure Main()
2    $g(s_{goal}) = v(s_{goal}) = \infty$ ;  $v(s_{start}) = \infty$ ;
3    $g(s_{start}) = 0$ ; OPEN = CLOSED = INCONS =  $\emptyset$ ;
4   insert  $s_{start}$  into OPEN with key( $s_{start}$ );
5   ComputePath();
6   publish current  $\epsilon$ -suboptimal solution;
7   while  $\epsilon > 1$ 
8     decrease  $\epsilon$ ;
9     Move states from INCONS into OPEN
10    Update the priorities for all  $s \in \text{OPEN}$  according to key( $s$ );
11    CLOSED =  $\emptyset$ ;
12    ComputePath();
13    publish current  $\epsilon$ -suboptimal solution;

```

∞ (infinito)¹, enquanto que o valor de $g(s_{start})$ é atribuído com o valor 0 (já que o valor da distancia real é calculada a partir da origem). Em seguida são criados os conjuntos “OPEN”, “CLOSED” e “INCONS”, correspondendo respectivamente ao conjunto dos “ABERTOS”, “FECHADOS” e “INCONSISTENTES”. O vértice s_{start} é inserido na fila de prioridades do conjunto OPEN utilizando a função de chave ordenadora descrito no [algoritmo 4.2](#). Essa função utiliza a estratégia da heurística inflada (descrito a seguir, na subseção 4.2.1).

A função de cálculo de caminho é então acionada pela função principal (algoritmo 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

¹ Vide nota de rodapé da seção 2.1.

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 $g(s)$ com o peso da aresta entre s e s' . Se caso positivo, $g(s')$ é atualizado com esse novo valor. Em seguida é verificado se o vértice s' não pertence ao conjunto dos fechados e em caso positivo, ele é adicionado/atualizado no conjunto OPEN de acordo com a função de chave ordenadora. Caso contrário, ele é adicionado ao conjunto dos INCONS.

Após o cálculo do caminho e de acordo com função principal (algoritmo 4.3), a solução ϵ sub-ótima (a explicação do motivo de a solução ser sub-ótima será dada na subseção 4.2.1) é apresentada pelo algoritmo. A partir daí começa o processo iterativo em que se verifica se o valor de ϵ é maior que 1. Em caso positivo, o valor de ϵ é decrescido por um fator de corte estipulado como parâmetro de entrada. Os vértices pertencentes a INCONS são movidos para OPEN e todas as chaves da fila de prioridades são atualizadas considerando o novo valor de ϵ . O conjunto CLOSED é esvaziado e a função de cálculo de caminho é acionada 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 posicionados mais para o final da fila de prioridades enquanto que os mais “promissores” serão posicionados mais 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 essa estratégia é que teremos um limite superior para a solução encontrada. Considere que o custo ótimo de um determinado caminho seja C^* . Se o utilizarmos a função $g(s) + \epsilon * h(s)$, com o valor de $\epsilon > 1$, então há a garantia de que a nova solução C , $C^* \leq C \leq \epsilon \times C^*$. Com isso, se $\epsilon = 1$, a solução encontrada será garantidamente 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 caso a aplicação permita o uso de um tempo maior para aprimorarmos o resultado², diminuimos o valor de ϵ . E se o tempo ainda permitir, diminuimos até termos $\epsilon = 1$, em que a solução será garantidamente ótima.

² Por exemplo, uma determinada aplicação necessita que se ache um caminho em no máximo 2 segundos. Se o algoritmo achar uma solução sub-ótima em 3 ms, dispomos de mais 1997 ms para podermos aprimorar ela.

Outra estratégia utilizada para ganhar tempo, a cada novo valor de ϵ em que a função para cálculo de caminho é acionada (algoritmo 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, caso haja um melhor caminho encontrado para um vértice que pertence aos fechados, este são colocados em INCONS dos quais serão enviados para OPEN para serem recalculados na próxima busca (a busca se refere ao número da iteração da invocação da função de cálculo de caminho, contido no algoritmo 4.1, e consequentemente ao valor de ϵ a ele atribuído).

A figura 9, contida em Likhachev et al. (2008), mostra a aplicação da estratégia da heurística inflada com seus respectivos resultados encontrados pelo algoritmo.

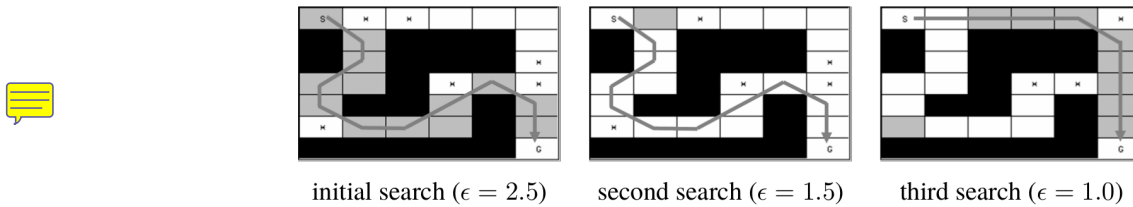


Figura 9 – Exemplo de aplicação da estratégia de diminuição do valor de ϵ .

4.3 Algoritmo AD*

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

Algoritmo 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 (algoritmo 4.7) começa iniciando os valores de $g(s_{goal})$, $v(s_{goal})$ e $v(s_{start})$ como ∞ , $bp(s_{start})$ como ponteiro 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 no algoritmo 4.6. Disso, é acionado a função de cálculo de caminho (algoritmo 4.5).

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

```

1 procedure ComputePath()
2   while (key( $s_{goal}$ ) >  $\min_{s \in OPEN} (key(s))$  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  $\cup \{s\}$ ;
6       for each successor  $s'$  of  $s$ 
7         if  $s'$  was never visited by AD* before then
8            $v(s') = g(s') = \infty$ ;  $bp(s') = \text{null}$ ;
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          $v(s) = \infty$ ; UpdateSetMembership( $s$ );
15         for each successor  $s'$  of  $s$ 
16           if  $s'$  was never visited by AD* before then
17              $v(s') = g(s') = \infty$ ;  $bp(s') = \text{null}$ ;
18             if  $bp(s') = s$ 
19                $bp(s') = \operatorname{argmin}_{s'' \in pred(s')} v(s'') + c(s'', s')$ ;
20                $g(s') = v(bp(s')) + c(bp(s'), s')$ ; UpdateSetMembership( $s'$ );

```

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

```

1 procedure key( $s$ )
2   if ( $v(s) \geq g(s)$ )
3     return [ $g(s) + \epsilon * h(s)$ ;  $g(s)$ ];
4   else
5     return [ $v(s) + h(s)$ ;  $v(s)$ ];

```

Algoritmo 4.7 – Algoritmo AD* - função principal

```

1 procedure Main()
2    $g(s_{goal}) = v(s_{goal}) = \infty$ ;  $v(s_{start}) = \infty$ ;  $bp(s_{goal}) = bp(s_{start}) = \text{null}$ ;
3    $g(s_{start}) = 0$ ; OPEN = CLOSED = INCONS =  $\emptyset$ ;  $\epsilon = \epsilon_0$ ;
4   insert  $s_{start}$  into OPEN with key( $s_{start}$ );
5   forever
6     ComputePath();
7     publish current  $\epsilon$ -suboptimal solution;
8     if  $\epsilon = 1$ 
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 ( $v \neq s_{start}$  AND  $v$  was visited by AD* before)
13         $bp(v) = \operatorname{argmin}_{s'' \in pred(v)} v(s'') + c(s'', v)$ ;
14         $g(v) = v(bp(v)) + c(bp(v), v)$ ; UpdateSetMembership( $v$ );
15    if significant edge cost changes were observed
16      increase  $\epsilon$  or re-plan from scratch (i.e., re-execute Main function);
17    else if  $\epsilon > 1$ 
18      decrease  $\epsilon$ ;
19    Move states from INCONS into OPEN
20    Update the priorities for all  $s \in OPEN$  according to key( $s$ );
21    CLOSED =  $\emptyset$ ;

```

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 algoritmo 4.1) ou se o valor de $v(s_{goal})$ é menor do que o valor de $g(s_{goal})$. Caso seja verdade, o processo iterativo é iniciado. 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 da busca anterior é maior do que o valor da busca atual (esse é o caso padrão 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 atualizado como ∞ , e para cada vértice s' adjacente de s é verificado se este já havia sido visitado pelo algoritmo. Caso contrário, os valores de $v(s')$, $g(s')$ são iniciados como ∞ e $bp(s')$ é iniciado como ponteiro nulo. Em seguida, é verificado se $bp(s')$ é igual ao vértice s . Em caso positivo, o $bp(s')$ é atualizado com o vértice predecessor de s' , s'' , cujo valor da 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. Finalmente a função de determinação para qual conjunto pertencerá s' é chamada (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 tem comportamento similar ao processo de atribuição de conjunto do ARA* (vide algoritmo 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 pertença a ele ou removido de INCONS caso pertença a este. Isso feito para que este vértice não precise ser explorado nesta busca ou na próxima (caso esteja em INCOS).

Voltando ao método principal (algoritmo 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 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 acionada.



5 Testes Computacionais

5.1 Algoritmo de Dijkstra

Para a realização dos experimentos computacionais são utilizadas instâncias de grafos que representam malhas rodoviárias reais. Todas elas descritas nas tabelas 1 e 2, e disponíveis no sítio eletrônico <<http://www.dis.uniroma1.it/challenge9/download.shtml>> (acesso em 28 de janeiro de 2017).

A tabela 1 mostra o nome dos arquivos das instâncias que foram executadas para os testes e suas respectivas descrições ~~dos grafos que representam malhas viárias reais~~. A tabela 2 mostra a configuração dos respectivos grafos apresentados na tabela 1, com seus respectivos número de vértices $|V|$ e número de arestas $|E|$.

Tabela 1 – 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 2 – Configuração dos grafos correspondentes as malhas viárias descritas na tabela 1.

| 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.

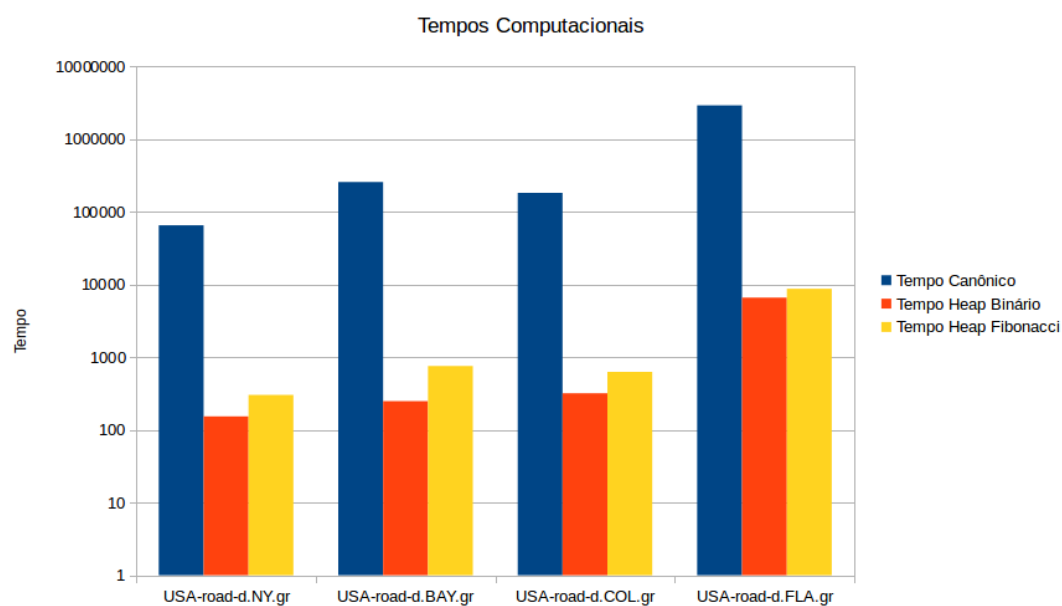


Figura 10 – Tempos computacionais obtidos pelos Métodos de Dijkstra empregados (tempo em escala logarítmica).

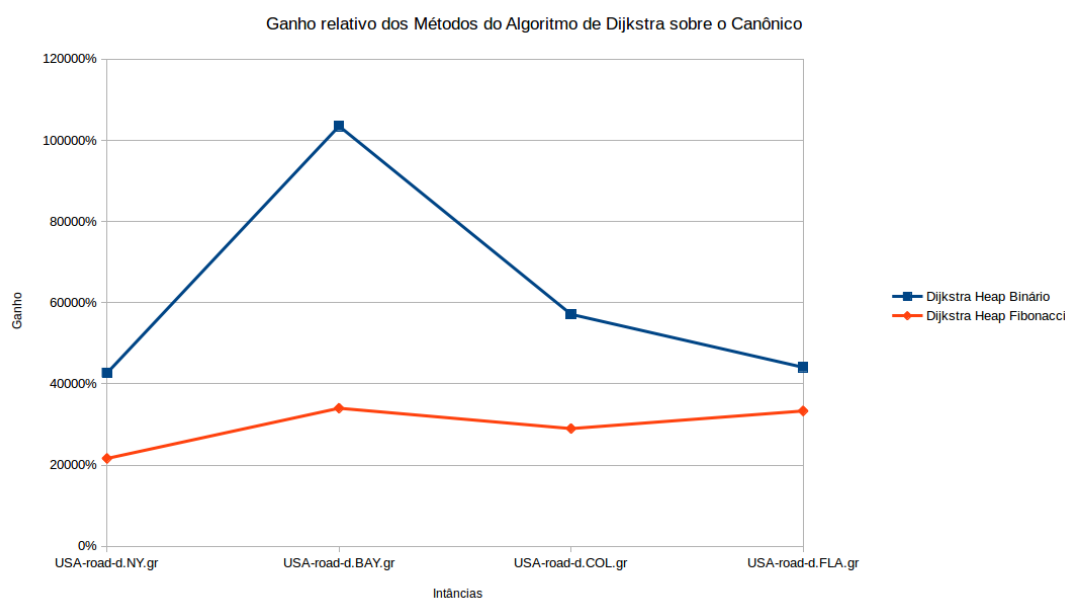


Figura 11 – Ganho relativo dos Métodos do Algoritmo de Dijkstra sobre o Canônico.

Tabela 3 – 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 4 – 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% |

A figura 10 mostra o comparativo entre os tempos computacionais obtidos por cada método em escala logarítmica. A figura 11 mostra o ganho relativo em tempo entre os métodos Dijkstra Heap Binário e Dijkstra Heap de Fibonacci sobre o Dijkstra Canônico. As tabelas 3 e 4 mostram os resultados, em números, dos tempos computacionais obtidos e os ganhos relativos sobre o Dijkstra Canônico, respectivamente.

5.1.2 Análise dos resultados

Podemos observar que o Dijkstra Canônico foi executado em um tempo elevado (a instância USA-road-d.FLA.gr por exemplo foi executado em 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 4).

Com relação ao resultado obtido pelos métodos do Heap Binário e Heap de Fibonacci, ele é de certo modo inesperado, já que conforme mostrado 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 executadas $|E| > |V|$ (vide tabela 2), 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 constata.

5.1.3 Conclusões

Conforme mostramos pelos experimentos descritos nesta seção, o algoritmo de Dijkstra que obteve o melhor resultado foi aquele com 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 reestruturação 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 5.

Tabela 5 – Descrição das versões a serem testadas neste capítulo.

| Nome Instância | Descrição |
|------------------------|--|
| Dijkstra | Versão de Dijkstra 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 |

Cuidado com o tempo verbal

Será executado dez vezes cada algoritmo para cada instância da subseção 5.1 em que para cada execução, será afixado o vértice origem como  sendo de valor de identificação “0” e terá como vértice destino um vértice escolhido aleatoriamente, sendo que não haverá repetição de vértices¹ (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.

¹ Para o caso do algoritmo de Dijkstra especificado na primeira linha da tabela 5, não será estabelecido um vértice destino já que o algoritmo calcula a melhor rota para todos os vértices do grafo.

5.2.1 Resultados obtidos

Os resultados dos testes computacionais descritos anteriormente podem ser vistos nas tabelas 6, 7 e 8.

Tabela 6 – Tempo médio obtido pelos métodos descritos na tabela 5 (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 7 – 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 8 – 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 |

A tabela 6 mostra os tempos, em milissegundos, médios obtidos por cada método testado. A tabela 7 mostra a porcentagem da diferença média entre a solução obtida pelo algoritmo A* aplicado com a heurística não-admissível e a solução ótima. Já a tabela 8 mostra o número médio de vértices abertos por cada método.

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 8). 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 mostra a tabela 7, 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* mostra ser um ótimo algoritmo para o cálculo de menor caminho entre dois vértices (origem e destino), superando em tempo computacional o algoritmo de Dijkstra (~~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 Dijkstra.

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

Para o capítulo de algoritmo dinâmicos serão [rodados](#) dois tipos testes, um para cada algoritmo. Serão utilizadas as mesmas instâncias descritas na seção 5.1.


5.3.1 ARA*

Para os testes com o algoritmo ARA*, será comparado o desempenho da utilização da heurística inflada com relação ao desempenho do algoritmo ~~de~~ A*. Será ~~afixado~~ um conjunto arbitrário de ϵ e será medido quanto tempo o algoritmo ARA* acha uma solução (mesmo não sendo a ótima) para aquele determinado ϵ e comparar com o tempo que o algoritmo A* acha uma solução ótima, além do número médio de vértices abertos por cada algoritmo. Será utilizada a mesma forma de [bateria de](#) testes descritas na seção 5.2 em que [serão](#) escolhidos 10 vértices diferentes [s](#) aleatórios como vértice destino [e](#) 0 como vértice origem e depois disso, [tirado a](#) média dos tempo total gasto para achar [a rota](#).

5.3.1.1 Resultados obtidos

Os resultados dos experimentos estão descritos a seguir na tabela 9. O tempo é descrito em nanosegundos (ns). Observe também que o valor do tempo médio do algoritmo A* não varia dentro da mesma instância. Isso é devido a ser utilizado o resultado único para cada instância, já que o valor de ϵ não é utilizado pelo A*.

Tabela 9 – Resultado testes ARA*.



| Nome Instância | Epsilon | NVA A* | Tempo médio A* | NVA ARA* | Tempo médio ARA* | Ganho de tempo com relação ao A* |
|-------------------|---------|--------|----------------|----------|------------------|----------------------------------|
| USA-road-d.NY.gr | 4.0 | 29394 | 26900000 | 252 | 980000 | 2.700% |
| | 3.5 | 29394 | 26900000 | 257 | 320000 | 8.400% |
| | 3.0 | 29394 | 26900000 | 265 | 820000 | 3.200% |
| | 2.5 | 29394 | 26900000 | 281 | 840000 | 3.200% |
| | 2.0 | 29394 | 26900000 | 296 | 560000 | 4.800% |
| | 1.5 | 29394 | 26900000 | 360 | 940000 | 2.800% |
| USA-road-d.BAY.gr | 1.0 | 29394 | 26900000 | 4825 | 3140000 | 800% |
| | 4.0 | 43896 | 45980000 | 239 | 1180000 | 3.800% |
| | 3.5 | 43896 | 45980000 | 236 | 800000 | 5.700% |
| | 3.0 | 43896 | 45980000 | 224 | 800000 | 5.700% |
| | 2.5 | 43896 | 45980000 | 233 | 620000 | 7.400% |
| | 2.0 | 43896 | 45980000 | 252 | 1040000 | 4.400% |
| USA-road-d.COL.gr | 1.5 | 43896 | 45980000 | 299 | 740000 | 6.200% |
| | 1.0 | 43896 | 45980000 | 10867 | 7920000 | 500% |
| | 4.0 | 69991 | 68140000 | 99 | 580000 | 11.700% |
| | 3.5 | 69991 | 68140000 | 93 | 860000 | 7.900% |
| | 3.0 | 69991 | 68140000 | 92 | 780000 | 8.700% |
| | 2.5 | 69991 | 68140000 | 92 | 720000 | 9.400% |
| USA-road-d.FLA.gr | 2.0 | 69991 | 68140000 | 90 | 500000 | 13.600% |
| | 1.5 | 69991 | 68140000 | 172 | 580000 | 11.700% |
| | 1.0 | 69991 | 68140000 | 2080 | 1720000 | 3.900% |
| | 4.0 | 255312 | 711120000 | 808 | 3500000 | 20.300% |
| | 3.5 | 255312 | 711120000 | 840 | 2820000 | 25.200% |
| | 3.0 | 255312 | 711120000 | 913 | 2460000 | 28.900% |
| | 2.5 | 255312 | 711120000 | 596 | 2220000 | 32.000% |
| | 2.0 | 255312 | 711120000 | 524 | 1940000 | 36.600% |
| | 1.5 | 255312 | 711120000 | 687 | 2060000 | 34.500% |
| | 1.0 | 255312 | 711120000 | 18909 | 14360000 | 4.900% |

5.3.1.2 Análise dos resultados

Podemos observar na tabela 9 que o algoritmo ARA* possui um ganho no tempo consideravelmente alto. Ele conseguiu achar uma solução (não garantidamente ótima) em um tempo 36.600% mais rápido do que o A*, reduzindo o tempo a ordem de nanosegundos. É notável também a quantidade reduzida de vértices abertos pelo ARA*, mostrando que de fato o uso da heurística inflada “poda” mais ainda os vértices a serem visitados com relação ao A*.

5.3.2 AD*

Para o algoritmo AD*, será comparado o desempenho computacional deste algoritmo com A*. Serão escolhidos também 10 vértices aleatórios seguindo os mesmos critérios descritos na seção 5.2. Porém para este caso, em cada vértice escolhido aleatoriamente, será rodada 15 vezes cada algoritmo, e nestas 15 vezes a cada iteração, o grafo sofrerá alteração dinâmica dos pesos de suas arestas de acordo com uma porcentagem dos vértices do grafo. Para o algoritmo A*, a cada iteração será rodado o algoritmo novamente para recálculo da rota enquanto que para o AD*, o próprio algoritmo, que estará em regime

de funcionamento, dará o devido tratamento para essa mudança conforme foi descrito na seção 4.2.

5.3.2.1 Resultados obtidos

Os resultados obtidos estão descritos na tabela 10. **Modo compreende** o tipo de teste rodado quanto a alteração do grafo. “Normal” significa que o grafo não sofre alteração, “Diminui” significa que as mudanças de pesos nas arestas compreende uma diminuição do valor do peso enquanto que “Aumenta” significa aumento delas. A porcentagem define a quantidade percentual de vértices do grafo que sofrerão alteração a cada nova iteração do teste (conforme descrito no início desta subseção). O tempo é dado em nanosegundos (ns).

Tabela 10 – Resultados obtidos para AD*.

| Nome Instância | Modo | Porcentagem de mudança de vértices | Tempo A* | NVA A* | Tempo AD* | NVA AD* | Ganho em relação ao A* |
|-------------------|---------|------------------------------------|------------|---------|------------|---------|------------------------|
| USA-road-d.NY.gr | Normal | 0% | 1359866662 | 2087605 | 157800000 | 139458 | 800% |
| | Diminui | 2% | 1385299996 | 2114495 | 198100000 | 181197 | 600% |
| | Diminui | 20% | 1350633328 | 1993419 | 859800000 | 781839 | 100% |
| | Diminui | 50% | 183033328 | 151199 | 9500000 | 4845 | 1.900% |
| | Diminui | 70% | 105533329 | 27898 | 1600000 | 5096 | 6.500% |
| | Aumenta | 2% | 139399995 | 79134 | 59300000 | 58691 | 200% |
| | Aumenta | 20% | 1252799994 | 1832418 | 625600000 | 559774 | 200% |
| | Aumenta | 50% | 1386799995 | 2112880 | 168400000 | 153857 | 800% |
| USA-road-d.BAY.gr | Aumenta | 70% | 1378066662 | 2095219 | 273000000 | 143918 | 500% |
| | Normal | 0% | 1238633329 | 1591980 | 148300000 | 106581 | 800% |
| | Diminui | 2% | 1248599995 | 1573329 | 199800000 | 118749 | 600% |
| | Diminui | 20% | 1078899995 | 1350961 | 275900000 | 257416 | 300% |
| | Diminui | 50% | 223599995 | 177938 | 10800000 | 4887 | 2.000% |
| | Diminui | 70% | 140199995 | 44969 | 4500000 | 5670 | 3.100% |
| | Aumenta | 2% | 172633328 | 112561 | 13900000 | 15388 | 1.200% |
| | Aumenta | 20% | 1035999995 | 1329812 | 213700000 | 169195 | 400% |
| USA-road-d.COL.gr | Aumenta | 50% | 1364833329 | 1742031 | 165300000 | 124424 | 800% |
| | Aumenta | 70% | 1358966661 | 1756298 | 227600000 | 122366 | 500% |
| | Normal | 0% | 2393133328 | 2595777 | 362000000 | 177983 | 600% |
| | Diminui | 2% | 2408266662 | 2580674 | 279300000 | 207273 | 800% |
| | Diminui | 20% | 2108899994 | 2323321 | 962400000 | 752105 | 200% |
| | Diminui | 50% | 413566660 | 349176 | 3100000 | 2858 | 13.300% |
| | Diminui | 70% | 188266660 | 20794 | 9500000 | 3682 | 1.900% |
| | Aumenta | 2% | 309799993 | 230532 | 625600000 | 512411 | 0% |
| USA-road-d.FLA.gr | Aumenta | 20% | 2194799995 | 2500123 | 520900000 | 379449 | 400% |
| | Aumenta | 50% | 2506366661 | 2748988 | 251000000 | 189233 | 900% |
| | Aumenta | 70% | 2528133328 | 2756270 | 329300000 | 183643 | 700% |
| | Normal | 0% | 5970399996 | 5999935 | 1130900000 | 407767 | 500% |
| | Diminui | 2% | 5885366661 | 5919297 | 898500000 | 453850 | 600% |
| | Diminui | 20% | 5396366662 | 5532391 | 4405600000 | 2598656 | 100% |
| | Diminui | 50% | 1202133329 | 766740 | 25000000 | 18739 | 4.800% |
| | Diminui | 70% | 552966662 | 99583 | 14000000 | 16163 | 3.900% |
| USA-road-d.FLA.gr | Aumenta | 2% | 661233329 | 327988 | 51500000 | 38827 | 1.200% |
| | Aumenta | 20% | 4870366662 | 5181622 | 3001600000 | 1574949 | 100% |
| | Aumenta | 50% | 5791899995 | 6000935 | 1641100000 | 420677 | 300% |
| | Aumenta | 70% | 5881499994 | 6118269 | 1451000000 | 413480 | 400% |

5.3.2.2 Análise dos resultados

Podemos observar que o algoritmo AD* possui um tempo computacional em geral melhor do que A* quando usado para recalculer a rota em grafos dinâmicos. No geral tivemos que essa diferença foi maior para quando o grafo tem o peso de suas arestas diminuídos do que quando o peso é aumentado. Isso é justificável, já que a rotina de tratamento para arestas com o peso aumentado é mais complexa do que para o peso diminuído (MOURA; RITT; BURIOL, 2010).

Observamos também que existe um ganho computacional mesmo quando o grafo não sofre alteração de seus pesos (caso “normal” descrito na tabela). Isso ocorre por causa

da estratégia da heurística inflada que acha uma solução sub-ótima rápida e a medida que o valor de ϵ é diminuído, a solução é melhorada, porém, ao contrário do ~~que executa o A*~~, o AD* não recalcula **todo mundo** novamente, mas usa vértices já abertos anteriormente, **tornado** mais rápido esse novo cálculo. Observamos ainda que o número de vértices abertos pelo AD*, em geral, é menor do que o A*.

Os resultados mostram também, que o ganho foi maior quando houve muitas mudanças dos vértices (50%-70%), mostrando que esse algoritmo se **ad**equa bem a grandes mudanças no grafo mantendo um bom desempenho.

5.3.3 Conclusões

O algoritmo ARA* se mostra um excelente algoritmo para acharmos resultados rapidamente, tendo um desempenho de nanosegundos. Seu uso se torna ideal para aplicações que necessitam de respostas rápidas e não tem a obrigatoriedade de se ter o resultado ótimo. Mais interessante ainda é que, conforme descrito no capítulo 4, o algoritmo vai melhorando o resultado a medida que mais tempo de cálculo é permitido pela aplicação, podendo assim achar um resultado rápido e este pode acabar **se tornando** ótimo (quando $\epsilon = 1$).

O algoritmo AD*, que na verdade é uma adaptação do ARA*, mostrou-se um excelente algoritmo para grafos dinâmicos conseguindo superar em tempo computacional o algoritmo A* (que na verdade é um algoritmo estático, o uso dele em grafos dinâmicos é uma adaptação, conforme descrito no capítulo 4) **que consegue com o uso da estratégia da heurística inflada, mais o não recálculo de todos vértices visitados anteriormente pelo algoritmo.**

Mas o uso do AD* só deve mesmo ser usado para grafos dinâmicos já que sua implementação é complexa e suscetível a erros de programação. Sendo assim, para grafos estáticos, é mais recomendado o uso dos algoritmos Dijkstra e o A*.



6 Considerações Finais e Trabalhos Futuros

Neste trabalho foram realizados estudos de algoritmos de caminho mínimo para grafos estáticos e dinâmicos, abordando seus funcionamentos, suas estratégias e o impacto que o uso de determinadas estruturas de dados ocasionam. Por meio dos testes computacionais realizados foi possível constatar a eficácia desses algoritmos e analisar em quais situações melhor se aplicam.

Foram estudados os algoritmos de Dijkstra, busca A^* , *Anytime Repairing* A^* (ARA*) e o *Anytime Dynamic* A^* (AD*) sobre instâncias que representam malhas rodoviárias reais, sendo o número de vértices e arestas desses grafos ~~nos quais foram adaptados essas malhas viárias~~, da ordem de grandeza de 100.000 a 1.000.000, possibilitando assim verificar o desempenho desses algoritmos em uma situação real e verificando também o desempenho para diferentes tamanhos de grafos.

Este trabalho constatou que a escolha de uma determinada estrutura de dados impacta fortemente no desempenho do algoritmo e que certas estrutura de dados propostas, como é o caso da heap de Fibonacci, apesar de teoricamente possuírem análise computacional melhor do que outras estruturas de dados como a heap binária, na prática, nem sempre é essa situação que ocorre, tendo fatores práticos computacionais **pesando no** desempenho. **Constatou** também que o algoritmo A^* , em geral, tem um desempenho melhor do que o algoritmo de Dijkstra devido ao uso da estratégia de heurística e de ser objetivo ao se determinar um vértice destino a ser buscado um caminho (em detrimento do Dijkstra que busca a solução calculando a melhor rota para todos os caminhos, sendo seu uso mais recomendado quando a aplicação exige a melhor rota para mais de um vértice). Foi verificado também que o uso de heurísticas não-admissíveis, apesar de se perder a garantia do resultado ótimo, obtêm bons ganhos de tempos computacionais a um preço baixo de deterioramento de qualidade de solução que gira em torno de 4%. Para os algoritmos dinâmicos, constatou-se que o algoritmo AD* se **ade**qua muito bem a esse tipo de grafo, tendo que ao se detectar mudança no peso de suas arestas, o algoritmo consegue calcular uma nova rota sem ter que recalcular todos os vértices que antes seriam necessários para achar uma nova rota, já que se baseia no uso de vértices já calculados anteriormente. Já o algoritmo ARA* se mostra muito eficiente para o cálculo rápido de soluções, que não são garantidamente ótimas, mas que a medida que o tempo for passando, essa solução tem a possibilidade de ser melhorada a um custo menor do que se fosse recalculada utilizando todos os vértices necessários.

Para trabalhos futuros sugerimos realizar testes para comparar o desempenho dos algoritmos AD* e ARA* com o D* e o D* Lite, que são algoritmos dinâmicos propostos

em [Stentz \(1994\)](#) e [Koenig e Likhachev \(2002\)](#) respectivamente, verificando também em quais situações cada um melhor se aplica. Além disso, vale também um estudo do uso de outras heurísticas além das que foram aplicadas, tanto para o algoritmo A^* quanto para o ARA^* e o AD^* , incluindo também, nestes dois últimos casos, o uso de heurísticas não-admissíveis. Para o algoritmo de Dijkstra **serão** realizados testes com outras estruturas de dados propostas que não foram testadas por este trabalho.

Referências

- CORMEN, T. H. *Introduction to algorithms*. [S.l.]: MIT press, 2009. Citado 7 vezes nas páginas 9, 11, 12, 13, 14, 15 e 16.
- 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 8 e 11.
- DROZDEK, A. *Data Structures and algorithms in C++*. [S.l.]: Cengage Learning, 2012. Citado 4 vezes nas páginas 8, 9, 11 e 13.
- 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 2 vezes nas páginas 9 e 17.
- KOENIG, S.; LIKHACHEV, M. D* lite. *AAAI/IAAI*, v. 15, 2002. Citado 2 vezes nas páginas 21 e 37.
- 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 2 vezes nas páginas 9 e 29.
- LIKHACHEV, M. et al. Anytime search in dynamic graphs. *Artificial Intelligence*, Elsevier, v. 172, n. 14, p. 1613–1643, 2008. Citado 5 vezes nas páginas 8, 9, 17, 21 e 24.
- MOURA, L.; RITT, M.; BURIOL, L. S. Estudo experimental de algoritmos em tempo real de caminho mínimo ponto a ponto em grafos dinâmicos. *Anais do XLII Simpósio Brasileiro de Pesquisa Operacional, ser. SBPO*, 2010. Citado 5 vezes nas páginas 8, 9, 23, 24 e 34.
- RUSSELL, S. J.; NORVIG, P. *Artificial Intelligence: A Modern Approach*. [S.l.]: Prentice-Hall, 1995. Citado 3 vezes nas páginas 9, 18 e 20.
- 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 2 vezes nas páginas 21 e 37.
- ZIVIANI, N. et al. *Projeto de algoritmos: com implementações em Pascal e C*. [S.l.]: Thomson, 2004. v. 2. Citado na página 8.