Estudo Experimental de Algoritmos em Tempo Real de Caminho Mínimo Ponto a Ponto em Grafos Dinâmicos

Leonardo Moura, Marcus Ritt, Luciana S. Buriol

Instituto de Informática - Universidade Federal do Rio Grande do Sul - UFRGS

lfsmoura@inf.ufrgs.br, mrpritt@inf.ufrgs.br, buriol@inf.ufrgs.br

Resumo

Calcular caminhos mínimos em grafos é um problema amplamente estudado na literatura. No entanto, aplicações específicas ainda requerem novas variações desses problemas. Recentemente, um algoritmo foi proposto para o cálculo de caminhos mínimos ponto-a-ponto em tempo real em grafos dinâmicos originados de aplicações de robótica. Este trabalho apresenta um estudo experimental deste algoritmo em dados rodoviários, comprovando sua eficácia também para dados desta origem.

1 Introdução

Um dos problemas mais comuns em grafos é calcular o caminho mínimo entre um par de vértices. Diversos problemas práticos são modelados via grafos e requerem o cálculo de caminhos mínimos para os mais diversos fins. Por exemplo, ambientes de trânsito podem ser modelados através de grafos valorados em que as arestas representam ruas e os vértices cruzamentos entre essas ruas. Nesse caso o peso associado a cada aresta corresponde ao comprimento ou tempo de travessia do trecho de rua correspondente.

Sistemas de tempo real, como agentes robóticos em um terreno ou motoristas no trânsito, precisam de soluções rápidas para os problemas de caminhos mínimos. Entretanto, nem sempre é possível encontrar a solução ótima com um tempo limitado de planejamento em grafos de grande dimensão. Uma possível solução para esse problema é encontrar uma solução subótima, limitando o número de vértices verificados no grafo. Tendo uma solução subótima já calculada, se mais tempo for disponibilizado, o algoritmo pode melhorar esta solução.

Ainda, para modelar ambientes que mudam com o tempo, como é o caso dos de trânsito, são utilizados grafos dinâmicos — grafos que podem ter os pesos das arestas modificados. Porém, recalcular a solução a cada mudança pode ser custoso. Como existe uma limitação de tempo, é possível reutilizar soluções anteriores modificando apenas as partes do grafo afetadas.

1.1 Literatura relacionada

Existem diversos problemas de caminhos mínimos, e cada um deles requerem algoritmos específicos de solução. As variações mais comuns são o caminho mínimo ponto-a-ponto (entre um par de pontos), caminhos mínimos de fonte única (entre um nó e todos os demais do grafo) e caminhos mínimos entre todos os pares de nós de um grafo. Todos requerem um grafo $G = (V, A)$ valorado. Entretanto, dependendo da estrutura topológica do grafo ou dos pesos dos arcos, alguns algoritmos são mais indicados que outros. De forma geral, os algoritmos mais eficientes são o `A*`, `Dijkstra` e `Floyd-Warshal`, respectivamente para os três problemas de caminhos mínimos mencionados.

Em situações específicas, variações destes algoritmos, ou mesmo outros algoritmos podem ser mais indicados. Neste caso, não só características do grafo podem influenciar, mas características da aplicação prática que originou o problema. Alguns exemplos são apresentados a seguir.

Suponha a situação no qual um grafo estático é fornecido e um número grande de requisições de caminhos mínimos são efetuadas. Neste caso, pode-se efetuar um pré-processamento do grafo, armazenando informações intermediárias que auxiliem o cálculo de caminhos mínimos, de forma que cada requisição de caminho mínimo pode ser calculada mais rapidamente. Este é o caso dos algoritmos de caminhos mínimo com pré-processamento [Köhler et al., Lauther, Goldberg et al.]. Uma busca de rota (caminho mínimo ponto a ponto) em mapas disponíveis online são exemplos de aplicação destes algoritmos.

Outra situação acontece quando o(s) caminho(s) mínimo(s) já foram calculados, e um ou mais arcos do grafo têm seus pesos alterados. Neste caso, é necessário atualizar o grafo de caminhos mínimos, ao invés de recalculá-lo. Os algoritmos de caminhos mínimos dinâmicos [Ramalingam et al., Bauer e Wagner] tem este objetivo. Por exemplo, numa rede de tráfego, quando o tráfego em um ou mais trechos de rua aumenta ou diminui, o caminho mínimo deve ser atualizado considerando estas modificações, em vez de recalculado.

Já caminhos mínimos em tempo real calculam um caminho subótimo rapidamente, e melhoram o caminho encontrado a medida que mais tempo é disponibilizado. [Likhachev et al.] apresenta o algoritmo AD*, o primeiro a ser em tempo real e dinâmico, i.e. utiliza uma solução incremental para o problema de caminhos mínimos em grafos dinâmicos e oferece um possível controle no tempo total de busca, enquanto ainda possui uma garantia da solução encontrada. O artigo também apresenta um teste experimental do algoritmo utilizando-o para planejar a navegação de um agente robótico.

1.2 Contribuições deste trabalho

Neste artigo é descrito um estudo experimental e comparativo, em redes rodoviárias, entre algoritmos em tempo real de caminho mínimo ponto a ponto. Dois algoritmos são investigados: *Anytime Repairing A*(ARA*)*, que dá suporte a limitações de tempo mas é estático, e *Anytime Dynamic A*(AD*)*, que é uma variação incremental do ARA*. Ambos os algoritmos foram originalmente propostos em [Likhachev et al.]. Entretanto, no artigo original, apenas grafos originados de

aplicações de robótica foram testados. Além disso, será avaliado experimentalmente o ganho de desempenho, ao se calcular um caminho mínimo após diferentes quantidades de mudanças no grafo de entrada, do AD* em relação a um algoritmo tradicional.

O artigo está organizado da seguinte forma: a Seção 2 contém a descrição dos algoritmos ARA* e AD*. Os resultados computacionais são apresentados na Seção 3, juntamente com suas análises. Finalmente na Seção 4 será apresentada as principais conclusões deste trabalho, bem como a descrição de trabalhos futuros.

2 Algoritmos de Caminho Mínimo Ponto a Ponto em Tempo Real

Um algoritmo é dito **anytime** se calcula uma solução rapidamente (não necessariamente a ótima) e melhora essa solução a medida que mais tempo é disponibilizado. Uma maneira de desenvolver um algoritmo com essa propriedade é forçar a busca a analisar menos estados. Isso pode ser obtido com uma modificação do algoritmo A*.

A* é um algoritmo de busca em grafos proposto em [Hart et al.], que se utiliza de uma função heurística para determinar a ordem de visita dos vértices, verificando assim menos vértices: $f(x) = g(x) + h(x)$, onde g(x) representa a distância do vértice x até a origem e h(x) estima a distância até o vértice final.

2.1 Anytime Repairing A*

Trocando a função de avaliação do A* por $f(x) = g(x) + \varepsilon \times h(x)$, sendo ε um valor real maior que um, menos vértices serão verificados, reduzindo o tempo de execução. Perde-se, porém, a garantia de otimalidade oferecida. Quanto maior o valor de ε, menos vértices serão verificados. Isso ocorre porque um peso maior é dado ao valor de distância. Dessa forma, os vértices verificados primeiramente serão os que terão menor custo, ignorando vértices com custo semelhante que poderiam fazer parte de um caminho de menor custo. Essa técnica é chamada de **heurística inflada**.

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 $\varepsilon \geq 1$ multiplicando o valor heurístico na função de avaliação, então há a garantia que para a nova solução C, C*$\leq$C$\leq \varepsilon \times$C*. Portanto, se $\varepsilon = 1$, a solução encontrada é ótima.

Uma solução pode ser encontrada rapidamente usando um valor de ε alto; e melhorada conforme o tempo restante, reduzindo-se esse valor e recalculando-se uma nova solução. Utilizando esse procedimento repetidas vezes, a solução encontrada será garantidamente ótima para $\varepsilon = 1$. No entanto, muitos vértices que já foram expandidos uma vez não precisam ser expandidos novamente. Portanto, ao se utilizar essa abordagem, muitos cálculos são feitos mais de uma vez desnecessariamente. O algoritmo Anytime Repairing A*(ARA*), proposto em [Likhachev et al.] resolve esse problema, reutilizando as soluções anteriores a cada iteração.

Assim como no A*, ARA* utiliza uma fila de prioridade ordenada pela função de avaliação $f(s) = g(s) + \varepsilon \times h(s)$ para definir a ordem de visita dos vértices do grafo. Porém, essa função tem a heurística inflada. Primeiramente a fila de prioridade só contém o vértice inicial. O valor de multiplicação da heurística é inicialmente definido com um valor ε_0. A cada iteração do algoritmo é obtida uma solução e o valor ε é diminuído por um pequeno valor δ. Esse procedimento é executado até que ε se torne 1 e se obtenha, então, a solução ótima.

A função `ComputePath` (Figura 1) faz a busca para cada um dos valores de ε. OPEN é uma fila de prioridade ordenada pela função de avaliação. A cada iteração de `ComputePath`, o vértice s com o menor valor da função de avaliação em OPEN é expandido: ao serem expandidos, todos os vértices vizinhos que tiverem sua distância diminuída com esse novo vértice são atualizados e adicionados no conjunto OPEN. Esse procedimento repete-se até que o vértice final seja encontrado.

```
1  Procedimento ComputePath()
2  enquanto  g(goal) > min_{s∈OPEN}(g(s) + ε × h(s)) faça
3  |  remova s com menor g(s) + ε × h(s) em OPEN ;
4  |  v(s) = g(s); CLOSED = CLOSED ∪ {s} ;
5  |  para cada sucessor s' de s faça
6  |  |  se  s' não foi visitado então
7  |  |  |  v(s') = g(s') = ∞ ;
8  |  |  se  g(s') > g(s) + c(s, s') então
9  |  |  |  g(s') = g(s) + c(s, s') ;
10 |  |  fim
11 |  |  se  s' ∈ CLOSED então
12 |  |  |  insira/atualize s' em OPEN com g(s') + ε × h(s') ;
13 |  |  senão
14 |  |  |  INCONS = INCONS ∪ {s'} ;
15 |  |  fim
16 |  fim
17 fim
```

Figura 1: Anytime Repairing A*(ARA*)[Likhachev et al.]

Quando a função `ComputePath` é executada com valores de ε maiores que 1, não se tem a garantia de que cada vértice será visitado uma única vez. Isso ocorre porque o valor de distância é superestimado em relação ao valor heurístico. Os vértices já visitados são marcados como CLOSED. Ao utilizar uma heurística inflada, não há mais a garantia de que os vértices que tem a distância menor serão visitados primeiro. Assim, pode acontecer de um vértice já visitado ter seu peso diminuído. Nesse caso ele teria que ser visitado mais uma vez. Como uma solução ótima não é necessária e para evitar a sobrecarga acarretada por múltiplas visitas no mesmo vértice, o vértice já visitado que tiver sua distância reduzida será marcado como inconsistente (INCONS). Entre duas execuções consecutivas de `ComputePath` todos os vértices marcados como INCONS são adicionados no

conjunto OPEN. Assim se um vértice já visitado tiver sua distância reduzida ele será visitado somente na próxima chamada.

O caminho mínimo pode ser recuperado a qualquer momento usando um método guloso, começando pelo vértice final e escolhendo o vértice subsequente que tem o menor custo. O caminho mínimo é recuperado ao repetir esse procedimento até atingir o vértice inicial.

2.2 Anytime Dynamic A*

Em ambientes dinâmicos — como um ambiente simulado de trânsito — é comum que as arestas do grafo tenham seu peso modificado com o tempo. Nesse caso, o caminho mínimo deve ser atualizado, em vez de recalculado. Ou seja, o grafo deve estar sempre atualizado. Além disso, queremos também o comportamento *anytime* para respostas rápidas subótimas. ARA* é capaz de gerenciar arestas com pesos que diminuem (que é o caso quando o valor de ε reduz), mas não suporta arestas com pesos aumentados.

O algoritmo Anytime Dynamic A* (AD*), que foi proposto originalmente por [Likhachev et al.], é uma modificação do ARA* que também permite aumento no peso das arestas.

Inicialmente o valor de ε é inicializado com um valor ε_0 definido previamente. Se nenhuma mudança é detectada no grafo, desse valor é subtraído δ e a função `ComputePath` (Figura 3) é chamada. Obtém-se então um comportamento *anytime* similar ao ARA*. Se uma ou mais arestas tem seu peso modificado o grafo deve ser atualizado, alterando-se os valores de distância dos vértices que podem alterar o caminho mínimo.

Para que o grafo possa ser atualizado corretamente, é guardado o valor de distância da iteração anterior de cada vértice u em $v(u)$. O conjunto em que serão escolhidos os vértices a serem expandidos, o conjunto OPEN, contém todos os vértices inconsistentes do grafo. Um vértice é dito inconsistente se a distância atual é diferente da distância anterior ($v(s) \neq g(s)$). Dependendo do tipo de mudança que uma aresta sofreu, os vértices podem se tornar inconsistentes ou sobreconsistentes.

Quando uma aresta (u,v) tem seu peso diminuído, o vértice v tornar-se **sobreconsistente**. A expansão desse vértice é similar a utilizada no ARA*. Neste caso são atualizados cada um dos vizinhos de v que tem sua distância diminuída passando por v. A nova distância é a distância de v somada com custo da aresta entre o vértice vizinho e v. Além disso é atualizado o valor de **bp** (backpointer) desses vizinhos para v. Esse valor indica quem é o sucessor de cada vértice e pode ser utilizado para recuperar o caminho mínimo.

O caso mais complicado é quando uma aresta (u,v) tem seu peso aumentado, nesse caso o vértice v é chamado **subconsistente**. O vértice v é atualizado e adicionado no conjunto OPEN. Os vizinhos de v afetados pela mudança são corrigidos. Se essa mudança fizer parte do caminho mínimo, todos os vértices subsequentes no caminho devem ser ajustados. A função de avaliação **Key** (Figura 2) garante que os próximos vértices a ser verificados sejam os do caminho mínimo. Key garante que os vértices que afetavam o caminho mínimo antes da mudança serão atualizados: os vértices subconsistentes são avaliados pela distância que tinham antes

da mudança. Assim se o vértice estava no caminho mínimo, então sua função de avaliação será a menor possível no conjunto OPEN (que só contém vértices inconsistentes). Além disso, só são corrigidos os vértices necessários, i.e. não são expandidos vértices desnecessariamente. A expansão ocorre até que o valor mínimo do conjunto OPEN seja maior do que o caminho já obtido até o vértice objetivo.

```
1  Procedimento Key(s)
2  se  v(s) ≥ g(s) então
3  |   retorna [g(s) + ε × h(s); g(s)] ;
4  senão
5  |   retorna [v(s) + ε × h(s); v(s)] ;
6  fim
```

Figura 2: função de avaliação do AD* [Likhachev et al.]

```
1  Procedimento ComputePath()
2  enquanto  key(s_goal) > min_{s∈OPEN}(key(s))  OU v(s_goal) < g(s_goal) faça
3  |   remova s com menor key(s) em OPEN ;
4  |   se  v(s) > g(s) então
5  |   |   v(s) = g(s); CLOSED = CLOSED ∪ {s} ;
6  |   |   para cada sucessor s' de s faça
7  |   |   |   se  s' nunca foi visitado então
8  |   |   |   |   v(s') = g(s') = ∞; bp(s') = null ;
9  |   |   |   se  g(s') > g(s) + c(s, s') então
10 |   |   |   |   bp(s') = s ;
11 |   |   |   |   g(s') = g(s) + c(s, s') ;
12 |   |   |   |   UpdateSetMembership(s');
13 |   |   |   fim
14 |   |   fim
15 |   senão
16 |   |   v(s') = g(s') = ∞ ;
17 |   |   UpdateSetMembership(s);
18 |   |   para cada sucessor s' de s faça
19 |   |   |   se  s' nunca foi visitado então
20 |   |   |   |   v(s') = g(s') = ∞; bp(s') = null ;
21 |   |   |   se  bp(s') = s então
22 |   |   |   |   bp(s') = argmin_{s''∈pred(s')} v(s'') + c(s'', s') ;
23 |   |   |   |   g(s') = v(bp(s')) + c(bp(s'), s') ;
24 |   |   |   |   UpdateSetMembership(s');
25 |   |   |   fim
26 |   |   fim
27 |   fim
28 fim
```

Figura 3: Função ComputePath do AD*[Likhachev et al.]

O mesmo problema de vértices visitados mais de uma vez no ARA* ocorre também no AD*. Para definir quais vértices são inconsistentes e devem ser visitados, atribui-se um conjunto para cada vértice. Após uma alteração em um vértice a função `UpdateSetMembership` (Figura 4) decide a qual conjunto cada vértice pertence. Se o vértice v for inconsistente ($v(v) \neq g(v)$) e v não tiver sido visitado na iteração corrente, então v deve ser adicionado no conjunto OPEN. Caso já tiver sido visitado deve ser adicionado em INCONS, para ser visitado somente na próxima iteração. Se v se tornou consistente e pertence ao conjunto OPEN ou INCONS, deve ser retirado do conjunto ao qual pertence.

```
1  Procedimento UpdateSetMembership(s)
2  se  v(s) ≠ g(s) então
3  |  se  s ∉ CLOSED então
4  |  |  insera/atualize s em OPEN com key(s) ;
5  |  senão se  s ∉ INCONS então
6  |  |  insira s em INCONS ;
7  |  fim
8  senão
9  |  se  s ∈ OPEN então
10 |  |  remova s de OPEN ;
11 |  senão se  s ∈ INCONS então
12 |  |  remova s de INCONS ;
13 |  fim
14 fim
```

Figura 4: Função Auxiliar do AD* [Likhachev et al.]

É importante ressaltar que também após uma ou mais mudanças é possível utilizar um valor de ε maior que 1. Assim o grafo é rapidamente atualizado. Eventualmente, com mais tempo de cálculo e com $\varepsilon = 1$, a solução encontrada é garantidamente ótima.

3 Teste Comparativo do Desempenho dos Algoritmos

Para testar ambos os algoritmos foi utilizado o grafo "USA-road-d.BAY" do DIMACS challenge [Dimacs]. Este grafo contém 321270 vértices e representa uma rede rodoviária em que as arestas representam as distâncias entre os vértices. Os algoritmos foram implementados utilizando a linguagem C++ e a biblioteca Boost [Boost] para representar os grafos. A implementação do A* utilizada no ambiente de teste também é a contida na biblioteca Boost. Todos os testes foram executados em um Core 2 Duo E4600 com 1 GB de memória.

3.1 ARA*

Para demonstrar como o ARA* encontra uma solução subótima em pouco tempo, compara-se na Figura 5 quantas vezes uma execução do ARA* é mais rápida que o A* para uma única execução com ε fixo. No eixo das abscissas tem-se o valor de multiplicação da heurística (ε) e no eixo das ordenadas temos a **aceleração**, que é quantas vezes o algoritmo ARA* é mais rápido que o A* (i.e., tempo A*/tempo ARA*). Os testes foram realizados da seguinte maneira: para cada valor de ε foi executado o algoritmo ARA* 100 vezes com diferentes vértices iniciais e finais escolhidos aleatoriamente. Os pontos no gráfico correspondem a média de aceleração dessas 100 execuções. As barras representam o intervalo de acelerações que ocorreram nas iterações, indo da menor razão até a maior (o melhor resultado encontrado).

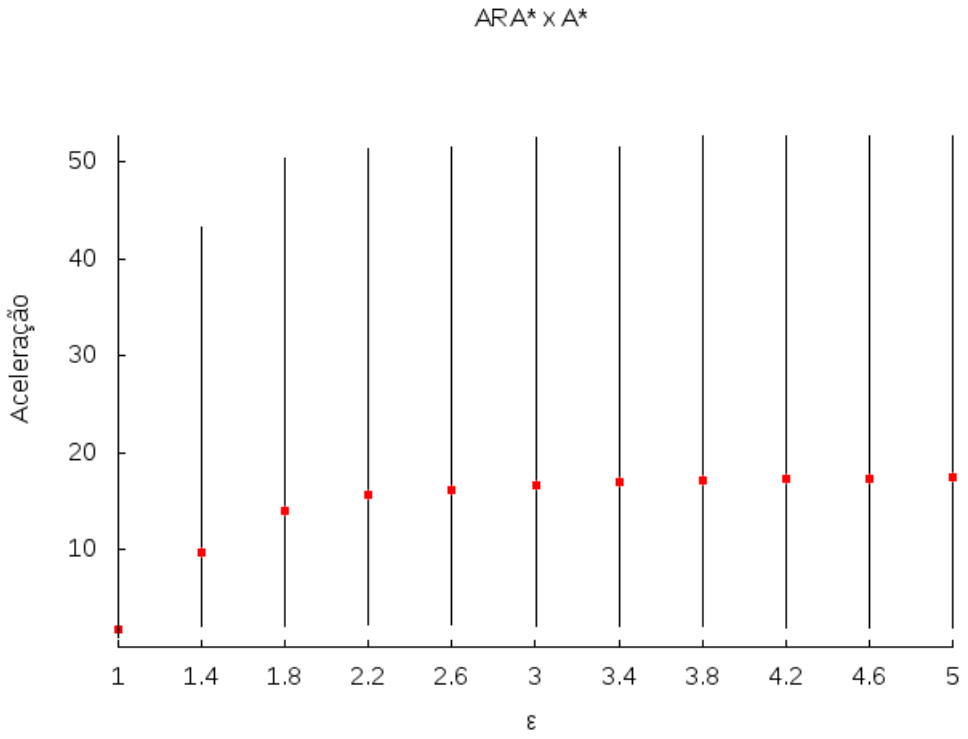

Figura 5: Teste computacional ARA*.

Nota-se que para o caso deste grafo, para ε maior que 3 já não temos mais um crescimento na aceleração significativo. O desempenho do algoritmo depende muito do grafo em que ele opera. Portanto, o valor de ε é um parâmetro e deve ser escolhido experimentalmente. Caso a qualidade da solução seja uma restrição do problema, pode-se escolher ε com base nessa informação. Por exemplo, se é necessária uma solução no máximo duas vezes pior que a solução ótima, então deve-se ter $\varepsilon = 2$.

ARA* mostrou uma aceleração máxima de 50 vezes sobre o A*. No caso médio o ganho é de mais de 10 vezes para ε maior que 1.8. Constata-se que o algoritmo é ideal para casos em que uma solução ótima não é necessária ou que o tempo de deliberação é muito curto.

3.2 AD*

Na Figura 6 pode-se observar graficamente como o AD* pode verificar menos vértices utilizando soluções anteriores. Cada imagem representa uma iteração do algoritmo, com os vértices inicial e final escolhidos aleatoriamente e indicados na figura através das marcas "start"e "goal", respectivamente. AD* é executado no grafo original. Mudanças aleatórias nos pesos das arestas são feitas em 1% das arestas do grafo. Os vértices em verde só foram verificados uma única vez (no grafo original), os vermelhos foram verificados e alterados depois das mudanças aleatórias. As imagens 2 e 4 são exemplos de mudanças que não tiveram um grande custo para que o caminho mínimo fosse recalculado, a maioria dos vértices não precisa ser revisitado nesse caso. Já as imagens 1 e 3 demonstram casos em que o custo de reutilizar soluções acaba não compensando. Nesses casos, o ideal seria descartar a solução e recalcular novamente o caminho.

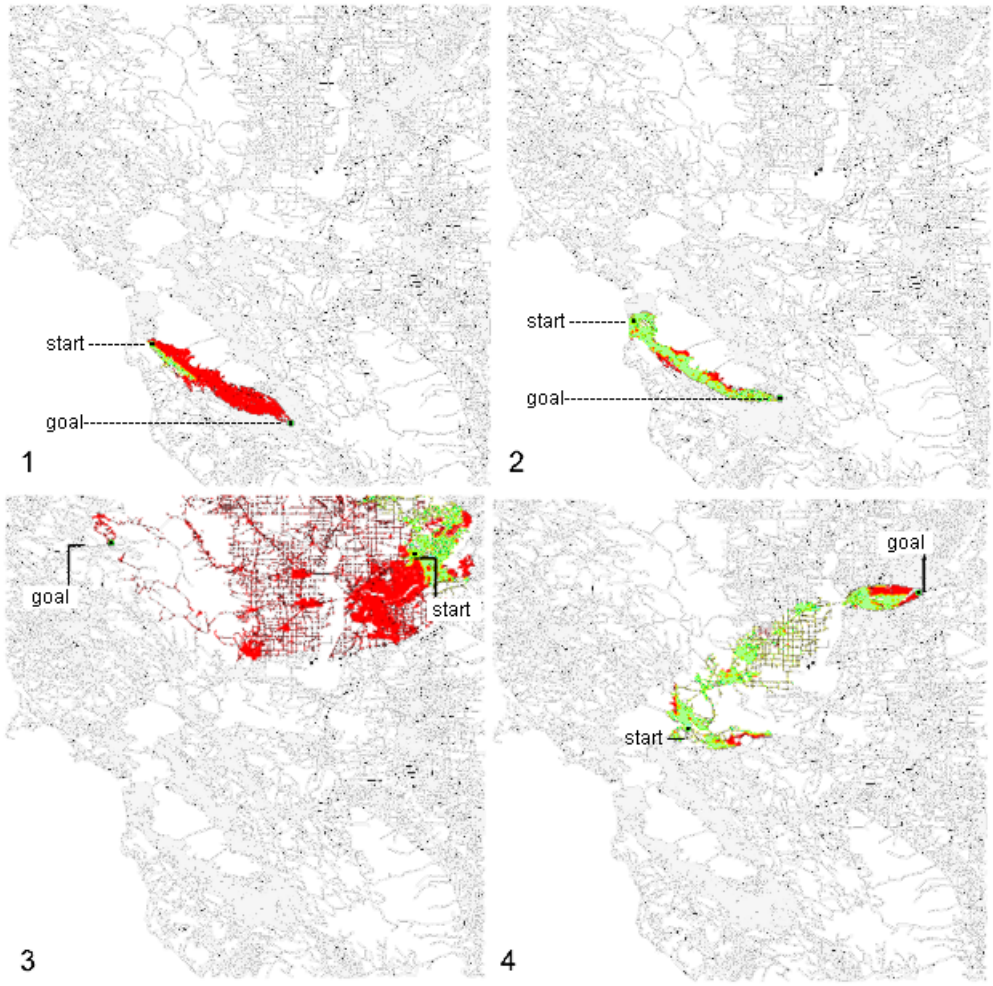

Figura 6: Vértices visitados em alguns exemplos de execuções do AD*.

Um segundo teste foi executado para comparar o desempenho do algoritmo AD* com o A*. O resultado encontra-se na Tabela 1. Nesse teste, a cada execução são selecionados vértices iniciais e finais diferentes aleatoriamente. Os algoritmos AD* e A* são executados no grafo original (testes indicados com o tipo *orig*). De-

pois foram executadas 16 iterações de mudanças aleatórias na região de interesse (a região com vértices verificados). A cada iteração uma aresta ou uma porcentagem da área de interesse tem o peso diminuídas (tipo *dec*) e então a operação inversa de aumentar o peso das arestas para o original (tipo *inc*), os valores correspondentes na tabela são a média dessas 16 execuções. *Mud* é a quantidade de mudanças, um valor ou uma porcentagem da região de interesse. A *aceleração* é a razão entre o tempo de execução médio da A* e o do AD*. Um ganho maior que um significa que o AD* teve um desempenho melhor que o A*. A tabela é dividida entre o A* e o AD*; para o A* temos o *tempo* médio de execução em milissegundos e a quantidade de *vértices* visitados; para o AD* temos o *tempo* médio de execução em milissegundos, a quantidade vértices visitados como subconsistentes (*sub*) — i.e. a visita dos vértices semelhante a do A*, em que é considerado se o custo de ter o caminho passando pela aresta visitada é menor do que o custo atual de seus vizinhos — e a quantidade vértices visitados como sobreconsistentes (*sobre*) — ou seja, o ajuste dos vértices para que o custo fique subconsistente depois de um incremento nas arestas.

			A*		AD*		
Tipo	*Mud*	*Aceleração*	*Tempo*(ms)	*Vértices*	*Tempo*(ms)	*Sub*	*Sobre*
orig	0	1,1	407,0	163099	375,1	163098	0
dec	1	100,8	202,3	81549	2,0	0	0
inc	1	105,4	204,9	81549	1,9	0	0
orig	0	1,1	18,0	590	16,1	589	0
dec	1%	6,8	6,8	295	1,0	0	0
inc	1%	6,8	6,9	295	1,0	0	0
orig	0	1,9	31,0	7086	16,1	7085	0
dec	2,5%	-	11,6	3543	-[1]	13	0
inc	2,5%	-	11,9	3543	-[1]	13	14
orig	0	0,9	204,0	82316	218,1	82318	0
dec	5%	1,5	106,4	41169	70,4	26375	40
inc	5%	0,7	101,6	41177	142,6	27133	27138
orig	0	1,1	203,0	84962	188,1	84961	0
dec	10%	1,2	119,2	42487	103,6	35624	744
inc	10%	0,6	119,2	42492	203,3	36296	36331
orig	0	1,2	78,0	25434	63,1	25433	0
dec	20%	1,2	40,0	12721	33,3	9562	374
inc	20%	0,7	38,2	12723	55,6	9569	9568
orig	0	1,1	140,0	44693	125,1	44691	0
dec	50%	0,9	67,3	22363	74,3	20859	598
inc	50%	0,5	66,4	22374	132,8	21101	21145

[1]valores insignificantes

Tabela 1: AD* avaliação

Nota-se que as mudanças incrementais levam a um tempo maior de execução, isso se deve ao custo de deixar o grafo sem vértices sobreconsistentes de um aumento no peso das arestas. Para 5% de mudanças incrementais na região de

interesse (cerca de 4000 arestas modificadas) já não temos vantagens em reutilizar as soluções anteriores. Porém, se tivermos mudanças decrementais, encontramos uma solução em um tempo menor que o A* para até 20% da região modificada. Ainda, no caso em que as mudanças são feitas fora da região de interesse, o custo para se obter o caminho mínimo se torna quase nulo.

4 Conclusão e Trabalhos Futuros

Neste artigo foram descritos dois algoritmos de caminho mínimo em grafos propostos nas literatura, e os mesmos testados sobre grafos rodoviários. ARA* mostrou uma aceleração de até 50 vezes de um algoritmo ótimo. O algoritmo oferece uma boa flexibilidade para o programador, que pode aumentar a quantidade de tempo disponível — diminuindo o tamanho de ε — para obter uma melhor garantia na solução encontrada. Utilizar o algoritmo AD*, para um número de mudanças na região de interesse maior que 5%, pode ser menos eficiente que recalcular o caminho mínimo com um algoritmo tradicional. Como possível foco de pesquisa, poderiam ser avaliados métodos de verificar eficientemente em quais casos vale a pena descartar a solução atual e recalcular todo o caminho mínimo.

O próximo passo da pesquisa é a integração dos algoritmos implementados no simulador ITSUMO [Silva et al.], simulador de tráfego microscópico baseado em autômatos celulares. Este simulador é capaz de simular diversos aspectos de uma rede de tráfego, como o comportamento do motorista, a coordenação de semáforos, e a predição de engarrafamentos. Devido ao aspecto dinâmico de um rede de tráfego, o AD* pode trazer grandes ganhos de desempenho na execução do simulador, em especial para grafos grandes.

Referências

[Boost] Boost C++ Libraries, http://www.boost.org

[Dimacs] DIMACS, http://www.dis.uniroma1.it/∼challenge9

[Hart et al.] Hart, P. E.; Nilsson, N. J.; Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Transactions on Systems Science and Cybernetics SSC4 4 (2): 100–107

[Silva et al.] Silva, Bruno Castro da; Junges, Robert;Oliveira, Denise and Bazzan, Ana L. C., 2006. ITSUMO: an Intelligent Transportation System for Urban Mobility. Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS. May. 1471-1472

[Likhachev et al.] Likhachev, M.; Ferguson, D.; Gordon, G.; Stentz, A.; Thrun, S. (2008). Anytime Search in Dynamic Graphs. Artificial Intelligence Journal (AIJ), 172 (14): 1613-1643

[Ramalingam et al.] Ramalingam, G.; Reps, T. 1996. An incremental algorithm for a generalization of the shortest-path problem. J. Algorithms 21, 2 (Sep. 1996), 267-305.

[Goldberg et al.] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. Better Landmarks Within Reach, WEA, 38-51, 2007.

[Köhler et al.] Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Fast Point-to-Point Shortest Path Computations with Arc-Flags. In: 9th DIMACS Implementation Challenge, 2006.

[Lauther] Ulrich Lauther. An Experimental Evaluation of Point-To-Point Shortest Path Calculation on Roadnetworks with Precalculated Edge-Flags. In: 9th DIMACS Implementation Challenge, 2006.

[Bauer e Wagner] Reinhard Bauer, and Dorothea Wagner. Batch Dynamic Single-Source Shortest-Path Algorithms: An Experimental Study. In: Proceedings of the 8th International Symposium on Experimental Algorithms (SEA'09), volume 5526 of Lecture Notes in Computer Science. Springer, June 2009. Joint work with Reinhard Bauer.