Dynamic-A: A* para qualquer configuração de vizinhos

Guilherme Caulada[1], **Pedro Cacique**[1]

[1]Faculdade de Computação e Informática – Universidade Presbiteriana Mackenzie (UPM)
R. da Consolação, 930 - Consolação, São Paulo - SP, 01302-907 - Brazil

guistoppa1995@gmail.com, phcacique@gmail.com

Abstract. *The quest for the shortest path is one of the most frustrating problems for the gaming industry. There are several implementations of algorithms that solve this problem, such as Dijkstra's, bread first search, or depth first search, however the A* algorithm is the one that has the best solution for this type of problem. Since its creation this algorithm has received attention from researchers and developers who have created a large list of modifications to the algorithm applying different techniques to improve it. This article describes a modification to the algorithm that seeks to make it more flexible, so that it is able to handle dynamic objects during the calculation of the smallest path. At the beginning you will be presented with an overview of the shortest path search algorithms. Then the A* algorithm will be described in detail as the basis for the optimization presented. Finally, a series of situations where this type of algorithm could be used are presented and we come to a conclusion.*

Resumo. *A busca de menor caminho é um dos problemas mais frustrantes para a industria de jogos. Existem diversas implementações de algoritmos que resolvem esse problema, como o Dijkstra, busca em largura, ou busca em profundidade, entretanto o algoritmo A* é o que possui a melhor solução para este tipo de problema. Desde sua criação esse algoritmo recebeu atenção de pesquisadores e desenvolvedores que criaram uma grande lista de modificações para o algoritmo aplicando diferentes técnicas para aprimora-lo. Esse artigo descreve uma modificação para o algoritmo que busca buscando torna-lo mais flexível, de forma que ele seja capaz a tratar objetos dinâmicos durante o calculo de menor caminho. No inicio sera apresentada uma visão geral sobre algoritmos de busca de menor caminho. Em seguida, o algoritmo A* sera descrito em detalhes como base para a optimização apresentada. Finalmente, uma serie de situações aonde este tipo de algoritmo poderia ser utilizado são apresentadas e chegamos a uma conclusão.*

1. Introdução

A busca do menor caminho geralmente se refere a busca da menor rota entre um ponto inicial, e um ponto final. No nosso dia-a-dia esse tipo de problema aparece em situações mais simples como no transito ou quando vamos fazer compras, e situações mais complicadas, como robôs em uma fabrica ou jogos de computador. Segundo [Cui 2011], com o crescimento da industria de jogos, o problema da busca de menor caminho tem se tornado cada vez mais popular e frustrante. Jogos em tempo real geralmente possuem personagens que são enviados de um certo ponto do mapa para um diferente ponto para completar uma certa tarefa. O problema mais comum encontrado na busca de menor caminho em jogos

de computador é como desviar obstáculos e lidar com diferentes tipos de terreno. As primeiras soluções para busca de menor caminho em jogos de computadores, foram logos ultrapassados pelo crescimento da complexidade dos jogos produzidos pela industria.

Devido ao grande sucesso do algoritmo A* muitos desenvolvedores apostam em aumentar sua velocidade para satisfazer as necessidades de seu software. Grandes esforços tem sido feitos nos últimos anos para optimizar esse algoritmo e melhorar sua performance. Exemplos de otimização envolvem novas heurísticas, representações de mapa, estruturas de dados e redução do consumo de memoria [Stout 1997].

Neste projeto aproximamos o algoritmo A* de forma diferente, introduzindo uma otimização que busca aumentar a flexibilidade do algoritmo sem grandes impactos em sua performance. O método apresentado, chamado de Dynamic-A, move o calculo de heurística e de vizinhos do A* para cada uma das posições presentes no mapa, dessa maneira cada posição pode mudar seus vizinhos de forma dinâmica, durante o processamento do algoritmo. Cada posição possui em si as informações de como o algoritmo A* deve calcular os seus vizinhos, e esta informação so e acessada enquanto esta posição é analisada, portanto cada posição pode modificar seus vizinhos individualmente, dinamicamente, durante o processamento do menor caminho. Dynamic-A não depende do preprocessamento, não introduz nenhum impacto a performance do algoritmo e sempre encontra o menor caminho.

2. Algoritmo A*

O algoritmo A* é um algoritmo de busca genérico que pode ser utilizado para diversos problemas, a busca do menor caminho é um deles. Para encontrar o menor caminho o algoritmo A* repetidamente examina o as posições inexploradas que considera mais promissoras. Quando uma posição é explorada o algoritmo para se essa posição é o destino final; caso contrario, ele guarda os vizinhos daquela posição para serem explorados no futuro.

```
1. Adicione a posição inicial a lista de posições em aberto.
2. Repita os seguintes passos:
   a. Procure pela posição que possui o menor f na lista
      de posições em aberto. Defina essa posição como
      a posição atual.
   b. Mova essa posição para lista de posições fechadas.
   c. Para cada posição vizinha a posição atual.
      i. Se estiver na lista de posições fechadas, ignore.
     ii. Se não estiver na lista de posições em aberto
         adicione-a a lista. Defina a posição atual como
         parente dessa posição. Grave o valor de f, g e h
         dessa posição.
    iii. Se ja estiver na lista de posições em aberto,
         cheque se este não é um melhor caminho. Caso seja,
         mude sua posição parente para a posição atual e
         recalcule os valores de f e g.
   d. Pare quando
      i. Adicionar a posição final a lista de posições fechadas.
     ii. Falhar ao buscar a posição final e a lista de posições
         em aberto estiver vazia.
3. A lista de posições utilizadas para se mover da posição
   inicial a posição final é o menor caminho encontrado.
```

Figure 1. Pseudocódigo do A*.

Como padrão quando nos referimos ao A* o valor g de uma posição representa o custo exato do inicio até si, o valor h representa a estimativa de custo ate a posição final e o valor de f é a soma desses dois valores. A Fig.1 apresenta o algoritmo passo-a-passo.

O A* possui diversas propriedades que foram provadas por Hart, Nilsson e Raphael em 1968. A primeira propriedade é que o algoritmo A* garantidamente trás um caminho entre a posição final e inicial caso ele exista. A segunda, a qualidade do caminho resultante depende em grande parte da heurística escolhida. Para que essa heurística seja admissível ela deve sempre estimar um valor menor ou igual ao menor custo da posição que esta sendo analisada a final. E a terceira propriedade do A* é que ele faz um uso eficiente da função heurística escolhida para ser o método de busca que examina a menor quantidade de posições para encontrar o melhor resultado. Nenhum outro método utiliza uma função heurística para atingir o mesmo objetivo.

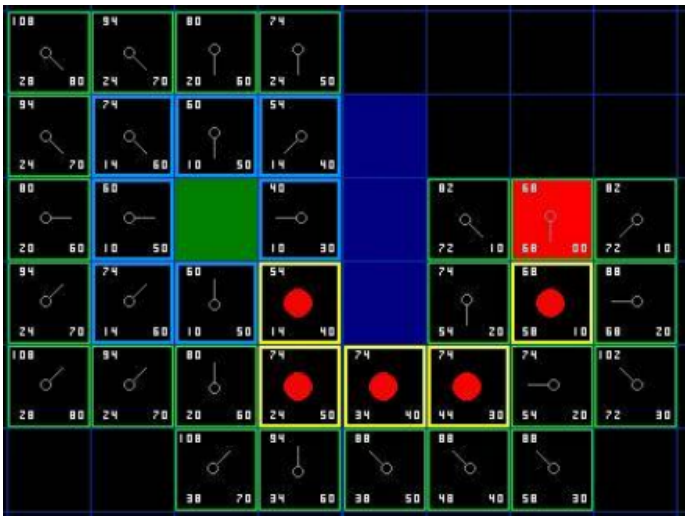

Figure 2. Exemplo do algoritmo A* em funcionamento denotando os valores de f no canto superior esquerdo, g no canto inferior esquerdo, e h no canto inferior direito [Lester 2005].

Apesar do A* ser a escolha mais popular para a busca de menor caminho em jogos de computador, como aplicar esse algoritmo em um jogo depende da sua natureza e representação do mundo virtual. Por exemplo em um jogo que possui uma grade retangular com 1000 posições verticais e 1000 horizontais, com um total de 1 milhão de posições possíveis, sera mais trabalhoso calcular um caminho, reduzindo o espaço de busca pode acelerar significantemente o algoritmo A* [Cui 2011]. A Fig.2 demonstra um exemplo do A* sendo executado em uma grade quadriculada.

3. Otimizações

As sessões a seguir apresentam otimizações possíveis para o algoritmo A* que foram aplicadas em conjunto com a otimização sugerida neste artigo.

3.1. Espaço de Busca

Em qualquer mapa virtual, os elementos presentes nele devem utilizar uma estrutura de dados para representar esse ambiente de forma que sejam capazes de calcular o caminho

para uma posição destino. Encontrar a melhor estrutura de dados para representar este ambiente é de grande importância para atingir uma performance aceitável para a busca de menor caminho. Como citado no exemplo anterior, um espaço de busca mais simples permite que o algoritmo execute mais rapidamente. A Fig.3 apresenta exemplos de representações possíveis para um mesmo mapa virtual.

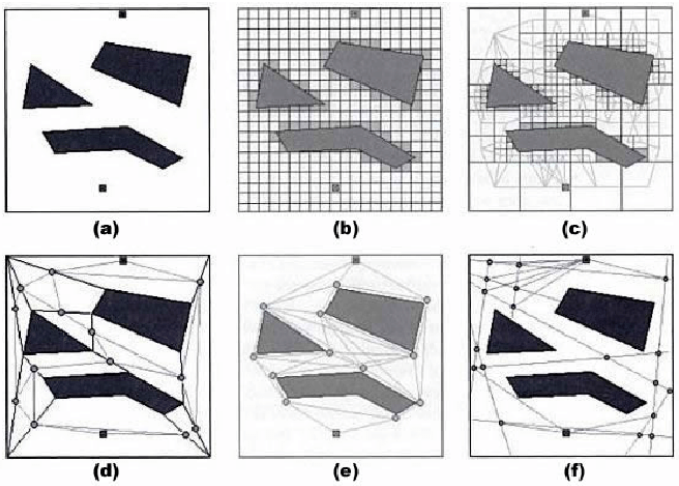

Figure 3. Cinco maneiras de se representar o espaço de busca [Stout 2000].

3.2. Função Heurística

O segredo para o sucesso do A* é que ele estende o algoritmo de Dijkstra introduzindo o uso da função heurística. O algoritmo de Dijkstra garantidamente encontra o menor caminho em um grafo conexo com elementos de diferentes pesos contanto que nenhum de seus pesos possua um valor negativo, entretanto ele não é eficiente pois todos os elementos do grafo devem ser analisados. O algoritmo A* melhora a eficiência computacional significantemente introduzindo o uso de heurística na tomada de decisões. Através da heurística invés de realizar uma busca extensiva em todos os elementos, apenas as posições que aparentam ser boas opções são analisadas. A função heurística utilizada no algoritmo A* estima o custo de uma posição qualquer até a posição destino. Caso o custo estimado seja exatamente igual ao custo real, então apenas as melhores posições são escolhidas e nada mais é analisado. Portanto, uma função heurística que estima o custo com precisão pode tornar o algoritmo muito mais rápido. Por outro lado, utilizar uma heurística que estima um custo um pouco acima do real normalmente resulta em uma busca mais rápida com um caminho aceitável [Steve 2000]. A Fig.4 apresenta o resultado do algoritmo de busca utilizando diferentes heurísticas para ultrapassar um obstaculo.

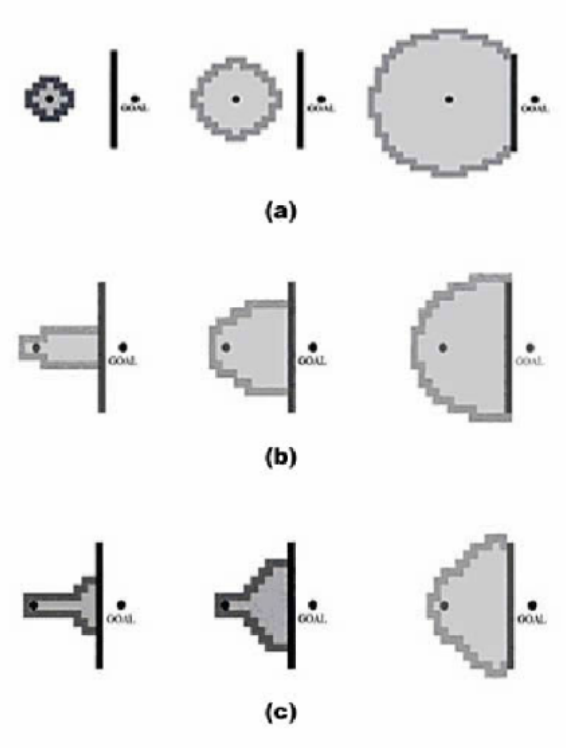

Figure 4. Comparação entre diferentes heurísticas [Steve 2000].

Quando a heurística estima um valor zero (como mostrado na Fig.4a), o algoritmo A* se torna semelhante ao algoritmo de Dijkstra. Quando a heurística utiliza a distancia euclideana ate o destino (mostrado na Fig.4b), apenas as posições que aparentam ser melhores opções são examinadas. Quando a heurística estima valores um pouco acima do valor real (como na Fig.4c), a busca foca nas posições mais próximas do destino final. Apesar de conseguirmos uma execução mais rápida estimando custos acima dos valores reais, quanto acima dos valores reais devem ser estas estimativas é um problema difícil, sem solução ate o momento.

3.3. Estrutura de Dados

Ao ser inicializada uma posição deve ser guardada em algum lugar para rápido acesso. Uma tabela de *hashes* pode ser considerada uma das melhores escolhas pois permite constante acesso aos dados e torna possível descobrir se uma certa posição esta na lista de posições abertas ou fechadas instantaneamente.

Uma fila com prioridades é a melhor maneira para implementarmos e manter as listas de posições, neste caso implementamos uma arvore *heap* binaria. Existem poucos trabalhos introduzindo novas formas de estruturar e manter as listas de posições abertas e fechadas, provavelmente introduzindo uma nova estrutura de dados para o A* pode ajudar a melhorar sua performance significativamente [Cui 2011].

4. Dynamic-A

Dynamic-A é uma modificação do algoritmo A* que torna sua execução mais flexível tornando possível a detecção de objetos dinâmicos durante o processamento do menor caminho, adaptando sua analise de acordo com as propriedades das posições que estão

sendo analisadas. As sessões a seguir descrevem o algoritmo Dynamic-A, como seu funcionamento difere do A* original e por que esse algoritmo pode ser considerado uma otimização. Dynamic-A não depende de uma linguagem especifica e pode ser adaptado para qualquer linguagem de programação orientada a objetos.

4.1. Classes

4.1.1. Node

A classe Node representa cada célula do mapa. Cada posição possível no espaço virtual deve ser atribuída a um Node, esse deve conter as informações das coordenadas individuais daquela posição, o peso, suas funções de vizinhos e heurísticas. Dessa forma quando o A* for analisar esta célula seus vizinhos poderão ser calculados dinamicamente com funções que acessam variáveis externas, seus vizinhos e heurística poderão ser atualizadas a qualquer momento, sendo consideradas apenas quando o algoritmo esta realizando a analise daquela célula.

4.1.2. Graph

Essa classe representa o mapa e as posições de todas as células e suas conexões. Cada grafo possui uma lista de nodes e uma tabela de quantas dimensões forem necessárias para mapear as posições das células do mapa utilizado. Essa classe garante acesso rápido e continuo as células que serão analisadas pelo algoritmo durante o calculo do menor caminho.

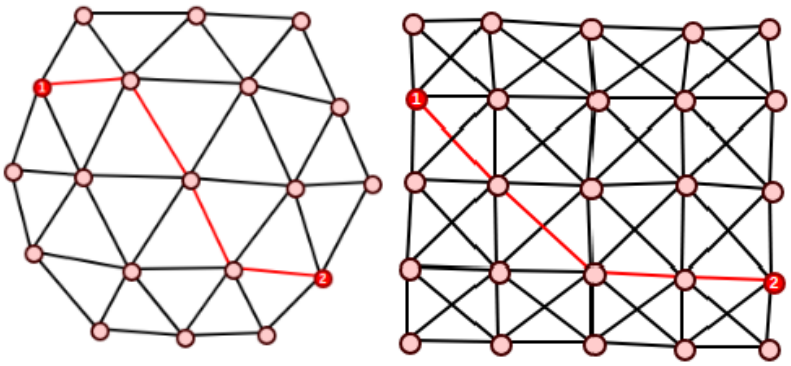

Figure 5. Uma grade hexagonal (a esquerda) e retangular (a direita) representadas no formato de um grafo.

4.1.3. BinaryHeap

A classe BinaryHeap implementa uma arvore binaria *heap*, ou de forma mais abstrata uma fila de prioridades. Essa fila guarda um certo grupo de objetos de acordo com sua chave (a prioridade), e possui um numero de operações para inserir um objeto novo, encontrar o elemento de menor prioridade e para deletar esse elemento [Brass 2008]. É possível substituir o foco no objeto de menor prioridade pelo de maior, e também criar uma *heap*

que realiza a busca por ambos. Na Fig.6 podemos visualizar essa estrutura em formato de arvore com seus índices de lista indicados entre colchetes.

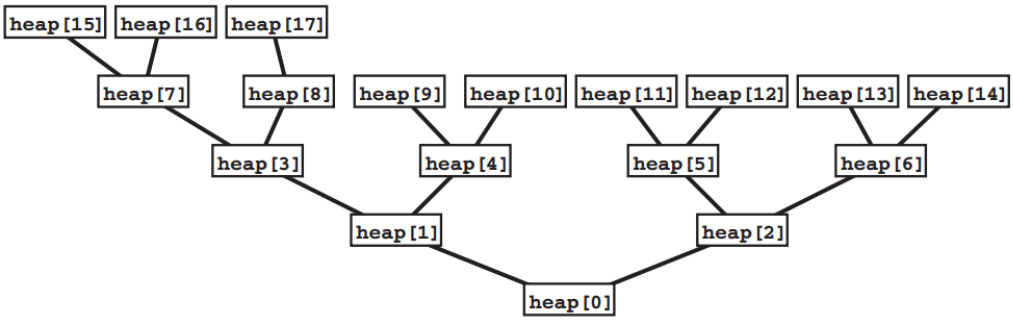

Figure 6. Representação de uma heap baseada em lista com seus elementos ordenados por prioridade [Brass 2008].

.

Nesse projeto criamos uma *heap* para servir como a lista de posições em aberto, essa lista sera organizada pelo valor de f de cada posição. Ao ser adicionado na lista um elemento sera reposicionado de acordo com o seu valor de f. Ao removermos um elemento da lista, seja ele o primeiro da lista, ou em qualquer outra posição, a lista sera reorganizada de acordo com os valores de f de cada Node utilizando funções para mover seus elementos dentro de sua estrutura.

4.1.4. Astar

Essa classe agrega as funções necessárias para a busca de menor caminho e tratamento de grafos, funções para limpar células, retornar o caminho do node atual ao inicial, marcar nodes como abertos ou fechados e inicializar grafos, definindo como zero os valores de f, g e h de todas as células de um grafo. A função de busca sera implementada de acordo com a definição da Fig.1, entretanto os vizinhos de cada célula e a heurística a ser utilizada sera definido dentro do node a ser analisado pelo algoritmo.

4.2. Implementação e Testes

Para implementar esse algoritmo a linguagem escolhida foi JavaScript, devido a sua flexibilidade e facilidade na criação e manipulação de novas estruturas e objetos. Para visualização do algoritmo utilizamos uma combinação de HTML, JavaScript e CSS para construir uma aplicação web capaz de executar diferentes testes no algoritmo. A linguagem HTML não é suficiente para a construção de aplicações web mais sofisticadas. Por isso HTML incorpora JavaScript para facilitar a criação de aplicações mais complexas. Basicamente JavaScript é uma linguagem suportada na maioria dos navegadores e na muitas vezes é utilizado para estabelecer interações através de interfaces visuais. Apesar de ser uma linguagem poderosa e com capacidade de orientação a objetos, ela é fácil de se aprender e entender, é uma linguagem recomendada para novos desenvolvedores por sua facilidade e simplicidade [Zeeshan 2014]. As próximas sessões descrevem os testes realizados a a partir dessa implementação.

4.2.1. Interface de Testes

Na interface de testes uma grade hexagonal e uma grade quadriculada representam nodes de diferentes propriedades, através de comandos é possível operar a interface para alterar a heurística e as funções de vizinhos de certas células, permitindo executar o algoritmo em diferentes condições para realizar uma analise de seus resultados.

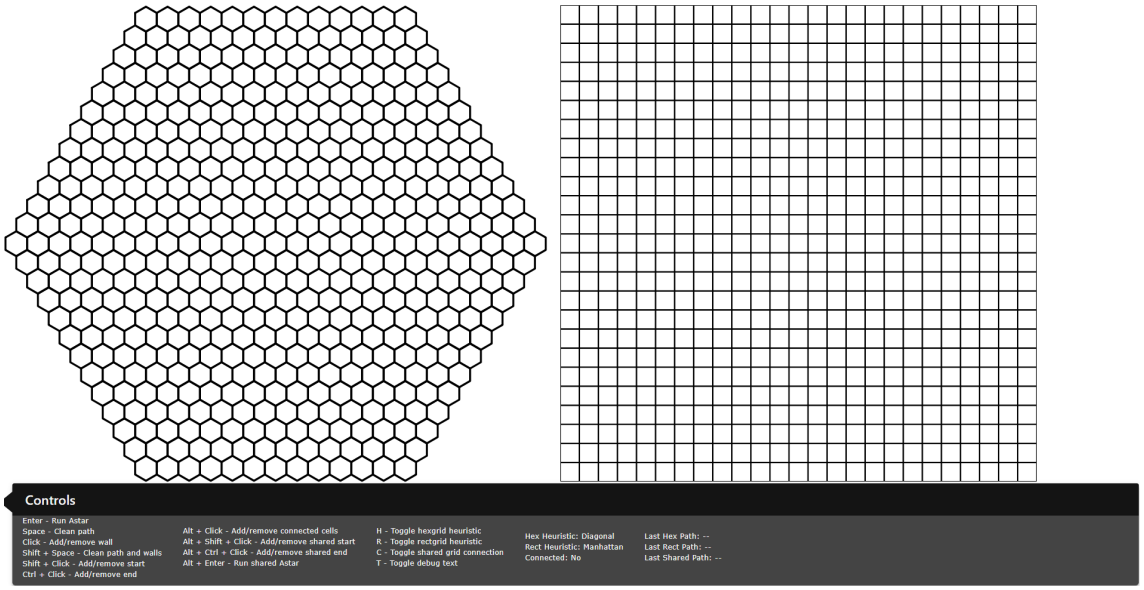

Figure 7. Interface de testes.

.

Na Fig.7 pode-se visualizar a interface utilizada. Em sua parte inferior existem opções para executar o algoritmo, remover a ultima execução, limpar células invalidas e modificar a heurística em cada grade. Clicando sobre as células é possível alterna-las entre validas e invalidas, uma célula invalida sera considerada como um obstaculo pelo algoritmo. Através de cliques também é possível definir o inicio e final da busca e conectar células, efetivamente modificando suas funções de vizinhos. No mesmo painel aonde são exibidos os controles também encontra-se estatísticas sobre a ultima execução.

4.2.2. Diferentes Propriedades

4.2.3. Diferentes Heurísticas

4.2.4. Adaptação

5. Aplicações

6. Conclusões

7. Agradecimentos

Agradeço ao professor Pedro Cacique assim como a todos os outros professores da FCI por seus ensinamentos.

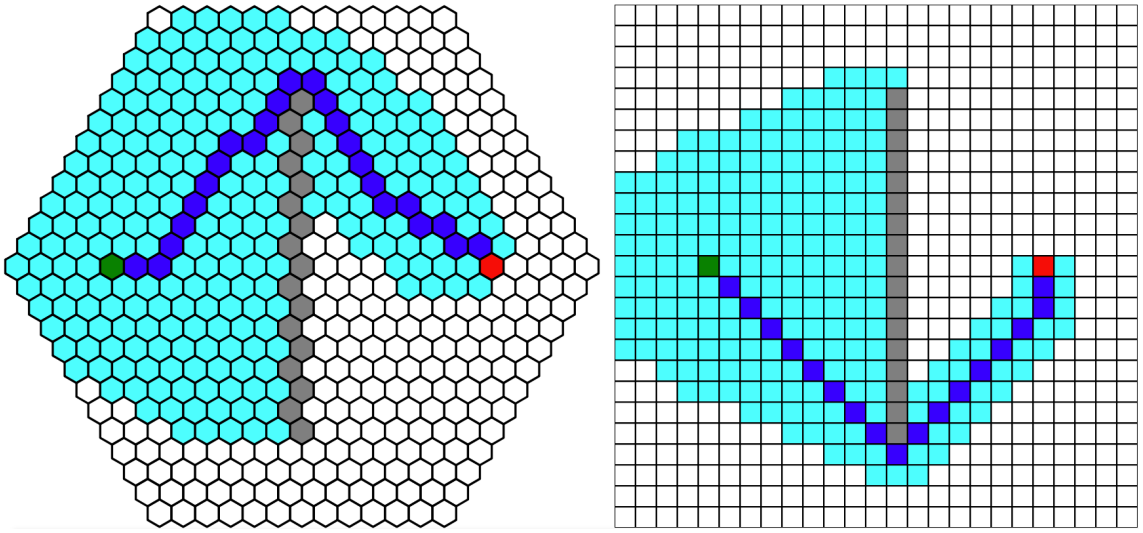

Figure 8. Demonstração do resultado gerado por diferentes propriedades.

.

References

Brass, P. (2008). Heaps. In *Advanced Data Structures*, pages 209–271. Cambridge University Press.

Cui, Xiao; Shi, H. (2011). A*-based pathfinding in modern computer games. *International Journal of Computer Science and Network Security*, 11(1):125–130.

Lester, P. (2005). A* pathfinding for beginners. pages 1–11.

Peter, Hart; Betram, R. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans.Syst.Sci.Cybernet*, 4(2):100–107.

Steve, R. (2000). A* speed optimizations. In DeLoura, M., editor, *Game Programming Gems*, pages 264–271. Charles River Media.

Stout, B. (1997). Smart moves: Intelligent pathfinding. *Game Developer Magazine*, pages 1–10.

Stout, B. (2000). The basics of a* for path planning. In DeLoura, M., editor, *Game Programming Gems*, pages 254–263. Charles River Media.

Zeeshan, A. (2014). Which one is better - javascript or jquery. *International Journal of Computer Science and Mobile Computing*, 3(6):193–207.

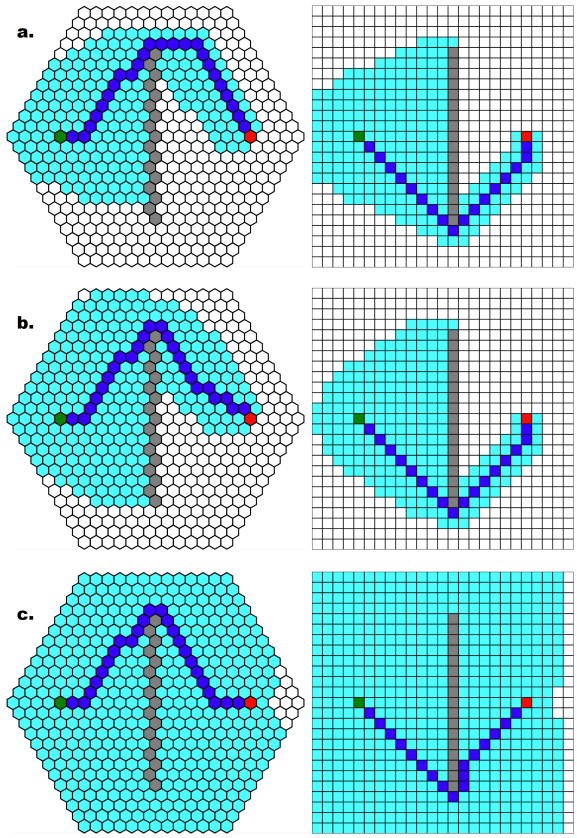

Figure 9. Comparação entre as heurísticas Manhattan (a), Diagonal (b) e sem heurística (c).

.

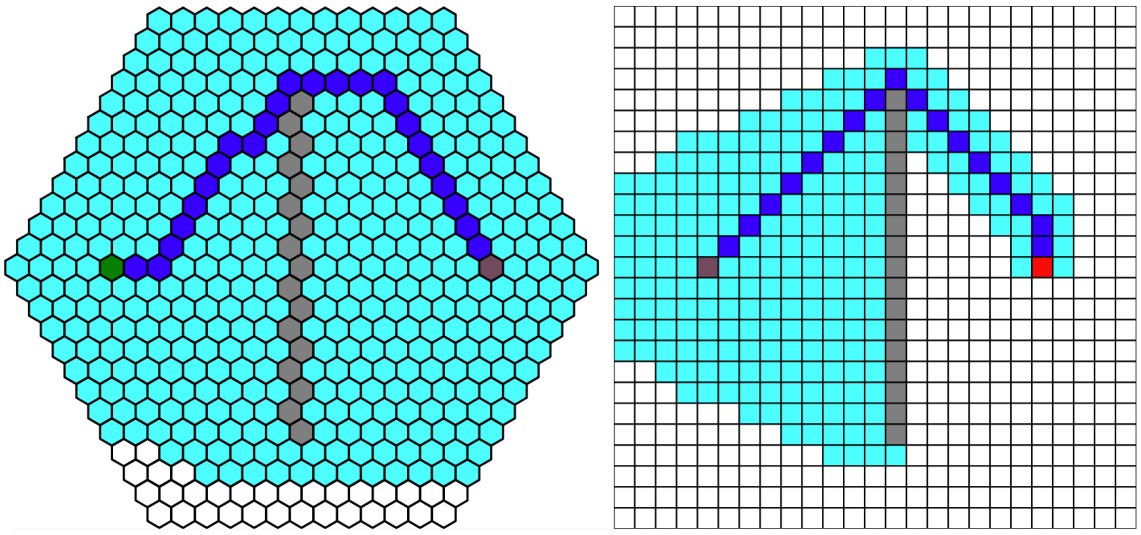

Figure 10. Demonstração do algoritmo se adaptando a diferentes propriedades.

.