Universidade de São Paulo
Escola de engenharia de São Carlos

Trabalho de Buscas

SCC0630 – Inteligência Artificial - Trabalho 03

Nome: Nilo Conrado Messias Alves Cangerana
Número USP: 9805362

1.Introdução

O trabalho consiste na implementação de cinco algoritmos de busca, muito utilizados em aplicações de Inteligência Artificial. O objetivo dos algoritmos é encontrar um caminho possível dentro de um labirinto, que é representado por uma matriz. A matriz do labirinto é passada como entrada para o programa através de um arquivo de texto. Assim, o programa pode executar um dos cinco algoritmos pedidos, retornando um possível caminho que foi encontrado, tempo de execução e a matriz de entrada com indicação de todos as posições exploradas pelo algoritmo.

O trabalho foi desenvolvido em Java, versão 8, com utilização da IDE Eclipse, Sistema Operacional Windows 10.

O projeto possui 8 classes:

-Programa.java: contém a função main que executa o programa e gera um menu de execução.

-Manip_arquivos.java: contém o método para leitura do arquivo de entrada e gerar a matriz com os dados na memória.

-Manip_matriz.java: contém metódos para manipular dados de matrizes, como recuperar valor dado uma linha e coluna, verificar valor de linhas e colunas adjacentes e avançar linhas e colunas adjacentes, dentro outros.

-BuscaProfundidade.java: possui o método para realizar a busca em profundidade.

-BuscaLargura.java: possui o método para realizar a busca em largura.

-BestFirstSearch.java: possui o método para realizar a busca best-first search.

-AEstrela.java possui o método para realizar a busca A*.

-HillClimbing.java: possui o método para realizar a busca com o Hill Climbing.

2.Instruções para compilação e execução

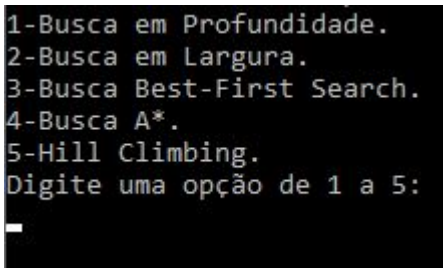
Para executar o programa, coloque o arquivo trabalho3.jar e os arquivos de texto de entrada em uma mesma pasta:

No prompt de comandos do Windows(cmd), entre no diretório da pasta em que estão os arquivos e digite o comando:

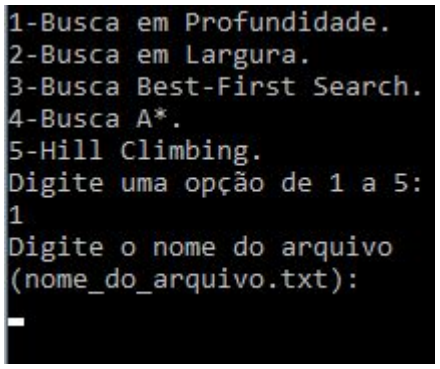
java -jar trabalho3.jar

```
\Trabalho 3>java -jar trabalho3.jar
```

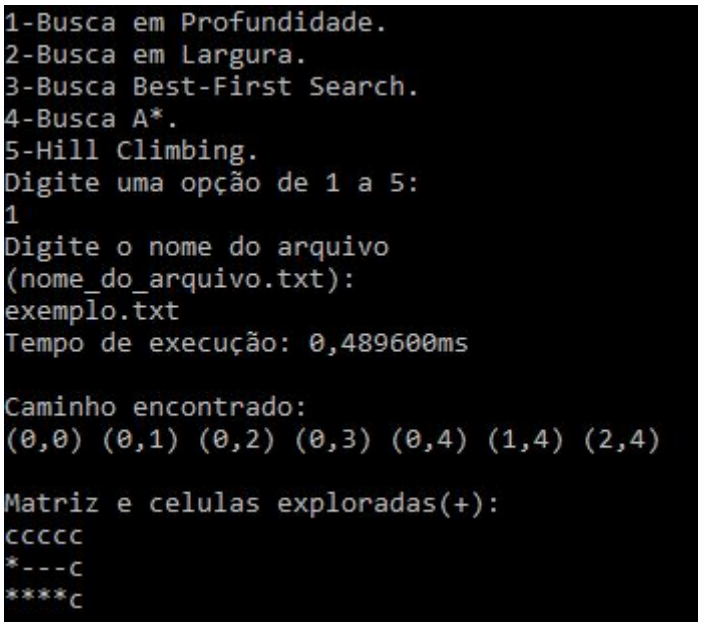
Assim que o comando for executado, o programa inicia com o menu indicado abaixo:



Para executar o programa, basta digitar um número de 1 a 5, referente ao algoritmo em questão. Em seguida, o nome do arquivo é pedido:



O arquivo deve estar na mesma pasta que o trabalho3.jar e então, o nome dele deve ser digitado. Depois disso, são retornado as coordenadas do caminho encontrado, o tempo decorrido para encontrar a solução e a matriz de entrada, onde o símbolo ‘c’ representa o caminho escolhido pelo algoritmo e o símbolo ‘+’ representa as posições que foram processadas pelo algoritmo e não necessariamente pertencem ao caminho. Neste exemplo, os símbolos ‘c’ e ‘+’ estão nas mesmas posições, dado que o caminho escolhido contém as posições processadas pelo algoritmo.



3.Busca em Profundidade

A implementação consiste da construção de uma lista onde são armazenadas as coordenadas do caminho, sempre que o programa avança uma posição, a posição é armazenada na lista e sempre que retorna uma posição, a posição é retirada da lista. No fim, a lista contém somente as coordenadas do caminho encontrado.

O algoritmo inicia na posição inicial do labirinto indicada por '#' e, a partir daí, checa a posição ao norte da atual. Se o norte for uma posição disponível para passar('*' ou '$'), o algoritmo avança uma posição para o norte e ela é armazenada na lista. Se o norte não for uma posição disponível para passar('-' ou fim da matriz), o algoritmo checa a posição à direita da atual.

Se a direita for uma posição disponível para passar, ele avança para a direita e ela é armazenada na lista. Se a direita não for uma posição disponível, o algoritmo checa a posição ao sul da atual.

Se o sul for uma posição disponível para passar, ele avança para o sul e a coordenada é armazenada na lista. Se o sul não for uma posição disponível, o algoritmo checa a posição à esquerda da atual.

Se a esquerda for uma posição disponível para passar, ele avança para a esquerda e ela é armazenada na lista. Se a esquerda não for uma posição disponível, o algoritmo não pode checar mais nenhuma posição.

Quando o algoritmo não possui mais uma posição para checar(atingiu um caminho sem saída), ele retorna posição por posição, por onde ele passou, removendo esses elementos da lista de coordenadas. Ele retorna até alguma posição na qual ele pode avançar ou para o norte, ou para a direita, ou para o sul ou para a esquerda e continua avançando a partir deste ponto.

O algoritmo fica em repetição até a posição atual for a mesma que a posição final do labirinto indicada pelo símbolo ('$'). As coordenadas por onde o algoritmo passou são marcadas com um símbolo '+' na matriz para facilitar a visualização das posições que foram exploradas por ele. Assim, o caminho pode ser retornado através das coordenadas armazenadas na lista anteriormente. As posições pertencentes ao caminho são marcadas com um símbolo 'c'.

4.Busca em Largura

A implementação da busca em largura consiste da construção de uma estrutura de árvore que armazena todas as coordenadas das posições que já foram exploradas e a construção de uma estrutura de fila na qual são armazenadas as coordenadas das posições que devem ser exploradas ainda.

O algoritmo inicia na posição inicial do labirinto, armazenando-a na fila. A partir daí, checa a posição ao norte da atual. Se o norte for uma posição disponível para passar, o algoritmo armazena a coordenada ao norte na fila e checa a posição à direita da atual.

Se a direita for uma posição disponível para passar, o algoritmo armazena a coordenada da direita na fila e checa a posição ao sul da atual.

Se o sul for uma posição disponível para passar, o algoritmo armazena a coordenada ao sul na fila e checa a posição à esquerda da atual.

Se a esquerda for uma posição disponível para passar, o algoritmo armazena a coordenada da esquerda na fila.

Depois da checagem das quatro direções, a coordenada atual é marcada como explorada com o símbolo '+', é armazenada na árvore, e é removida da fila. O próximo elemento da fila, que agora está na cabeça da fila, é marcado como elemento atual e então é processado da mesma forma.

O algoritmo fica em loop, sempre processando a posição que está na cabeça da fila, até que a posição final é encontrada. Quando o fim do labirinto é encontrado, é realizado um backtracking na árvore para retornar o caminho do inicio ao fim.

5. Busca Best-First Search

A busca Best-First Search(BFS) ocorre da mesma maneira que a busca em largura, através do processamento das posições que estão na cabeça da fila. No entanto, é implementada uma heurística capaz de reduzir a quantidade de posições que são processadas na busca em largura, chegando ao resultado de forma mais eficiente.

A heurística escolhida para implementação foi a Manhattan Distance. Diferente da distância euclidiana, na qual a distância entre dois pontos é calculada através do teorema de pitágoras, a Manhattan Distance é calculada como a distância horizontal somada a distância vertical entre dois pontos. Como o movimento sobre a matriz é feita somente na horizontal e vertical, a menor distância entre dois pontos pode ser calculada dessa forma. A fórmula utilizada é indicada a seguir:

$$d = |x_2 - x_1| + |y_2 - y_1|$$

Onde:

(x_1, y_1) representa um ponto P1 no plano.

(x_2, y_2) representa um ponto P2 no plano.

d representa a distância entre P1 e P2.

Para o algoritmo BFS, a manhattan distance é calculada para todos os elementos que estão na fila. No cálculo, é utilizada a posição do ponto na matriz e a posição do fim do labirinto indicada por '$'. A posição que possui a menor distância calculada é colocada no início da fila, para ser processada em seguida. Com isso, a coordenada do início da fila é explorada, garantindo que o final seja encontrado

mais rapidamente através do processamento de posições que estão cada vez mais próximas do fim.

6. Busca A*

A implementação da busca A* ocorre da mesma maneira que a busca em largura através do processamento das posições que estão na cabeça da fila. No entanto, para cada elemento, é calculada a distância(g) entre o dado elemento e o início do labirinto e a distância entre o elemento e o final do labirinto(h). Tais distâncias são calculadas com a fórmula da manhattan distance e são somadas, obtendo o valor de custo(f). Esse valor representa o custo do ínicio até um certo elemento mais o custo desse elemento até o fim do labirinto. A fórmula é mostrada a seguir:

$$f = g + h$$

O algoritmo garante que o caminho encontrado seja o menor possível, dado que ele procura o menor custo(menor valor de f) entre os elementos que estão na fila e, ao encontrar o menor, o algoritmo o coloca no início da fila para ser processado em seguida.

Assim, para cada elemento que está na cabeça da fila, ocorre o mesmo procedimento da busca em largura, é verificado o norte, a direita, o sul e a esquerda. As posições possíveis de passagem são colocadas na fila, com seus custos calculados e ordenados. Caso algum elemento possua um custo f igual a outro, o algoritmo prioriza os elementos que possuem o menor valor de h, que é a distância do dado elemento até o final. Todos os elementos processados são armazenados numa estrutura de árvore e, através do backtracking na árvore, o caminho encontrado é retornado.

7. Hill Climbing

O algoritmo Hill Climbing é implementado da mesma maneira que a busca em largura na qual uma estrutura de árvore e uma fila são geradas. No entanto, o algoritmo prioriza sempre as posições que são vizinhas do elemento que está sendo analisado no momento e as posições mais próximas do objetivo, utilizando a heurística de manhattan distance, ou seja, a busca se expande sempre em uma direção até não conseguir progredir mais.

O algoritmo inicia no ponto inicial do labirinto, checa as posições do norte, direita, sul e esquerda, calcula qual a menor distância até o ponto final do labirinto e armazena essas posições na fila. A posição inicial é retirada da fila e armazenada na árvore. A posição que está na cabeça da fila é explorada da mesma forma e assim por diante. A cabeça da fila é sempre atualizada com o elemento de menor distância até o objetivo final, dado que esse elemento é sempre vizinho do elemento

anteriormente processado, garantindo que o algoritmo ande sempre na mesma direção, diferente da busca em largura que expande em todas direções à procura da solução. Quando todo o caminho é explorado e não se achou a solução, o algoritmo seleciona outro ponto com menor distância na fila para continuar. O algoritmo continua até encontrar o final do labirinto.

Esse método não garante que o caminho encontrado seja o menor caminho, dado que ele possui o problema de encontrar um vizinho que é máximo local e não o global, achando uma solução possível, porém que não é otimizada.

O caminho é retornado da árvore, através do backtracking pela árvore da posição final até a posição inicial.

8. Resultados

Abaixo são mostrados resultados dos algoritmos para sete exemplos distintos. Os exemplos utilizados estão na pasta do trabalho e são os arquivos e1.txt, e2.txt, e3.txt, e4.txt, e5.txt, e6.txt, e7.txt.

1) **Busca em Profundidade**
   - **1.1)e1.txt**
     Tempo de Execução = 2.162200ms
   - **1.2)e2.txt**
     Tempo de Execução = 2.332200ms
   - **1.3)e3.txt**
     Tempo de Execução = 6.310400ms
   - **1.4)e4.txt**
     Tempo de Execução = 2.999600ms
   - **1.5)e5.txt**
     Tempo de Execução = 5.342000ms
   - **1.6)e6.txt**
     Tempo de Execução = 1.426900ms
   - **1.7)e7.txt**
     Tempo de Execução = 3.602600ms

     Média = 3.453700ms

2) **Busca em Largura**
   - **2.1)e1.txt**
     Tempo de Execução = 4.132700ms
   - **2.2)e2.txt**
     Tempo de Execução = 2.810500ms
   - **2.3)e3.txt**
     Tempo de Execução = 6.486300ms
   - **2.4)e4.txt**

Tempo de Execução = 6.266400ms

2.5)e5.txt

Tempo de Execução = 6.330100ms

2.6)e6.txt

Tempo de Execução = 6.533100ms

2.7)e7.txt

Tempo de Execução = 7.343700ms

Média = 5.700400ms

3) **Busca Best-First Search**

3.1)e1.txt

Tempo de Execução = 3.293000ms

3.2)e2.txt

Tempo de Execução = 1.559400ms

3.3)e3.txt

Tempo de Execução = 4.197000ms

3.4)e4.txt

Tempo de Execução = 2.951500ms

3.5)e5.txt

Tempo de Execução = 2.930200ms

3.6)e6.txt

Tempo de Execução = 2.988800ms

3.7)e7.txt

Tempo de Execução = 7.921400ms

Média = 3.691614ms

4) **Busca A***

4.1)e1.txt

Tempo de Execução = 3.532400ms

4.2)e2.txt

Tempo de Execução = 1.791800ms

4.3)e3.txt

Tempo de Execução = 4.343400ms

4.4)e4.txt

Tempo de Execução = 3.950900ms

4.5)e5.txt

Tempo de Execução = 3.783200ms

4.6)e6.txt

Tempo de Execução = 6.817700ms

4.7)e7.txt

Tempo de Execução = 9.695000ms

Média = 4.844914ms

5) **Hill Climbing**

5.1)e1.txt

Tempo de Execução = 3.214600ms

5.2)e2.txt

Tempo de Execução = 1.604400ms

5.3)e3.txt

Tempo de Execução = 4.316700ms

5.4)e4.txt

Tempo de Execução = 3.891500ms

5.5)e5.txt

Tempo de Execução = 3.588800ms

5.6)e6.txt

Tempo de Execução = 4.291300ms

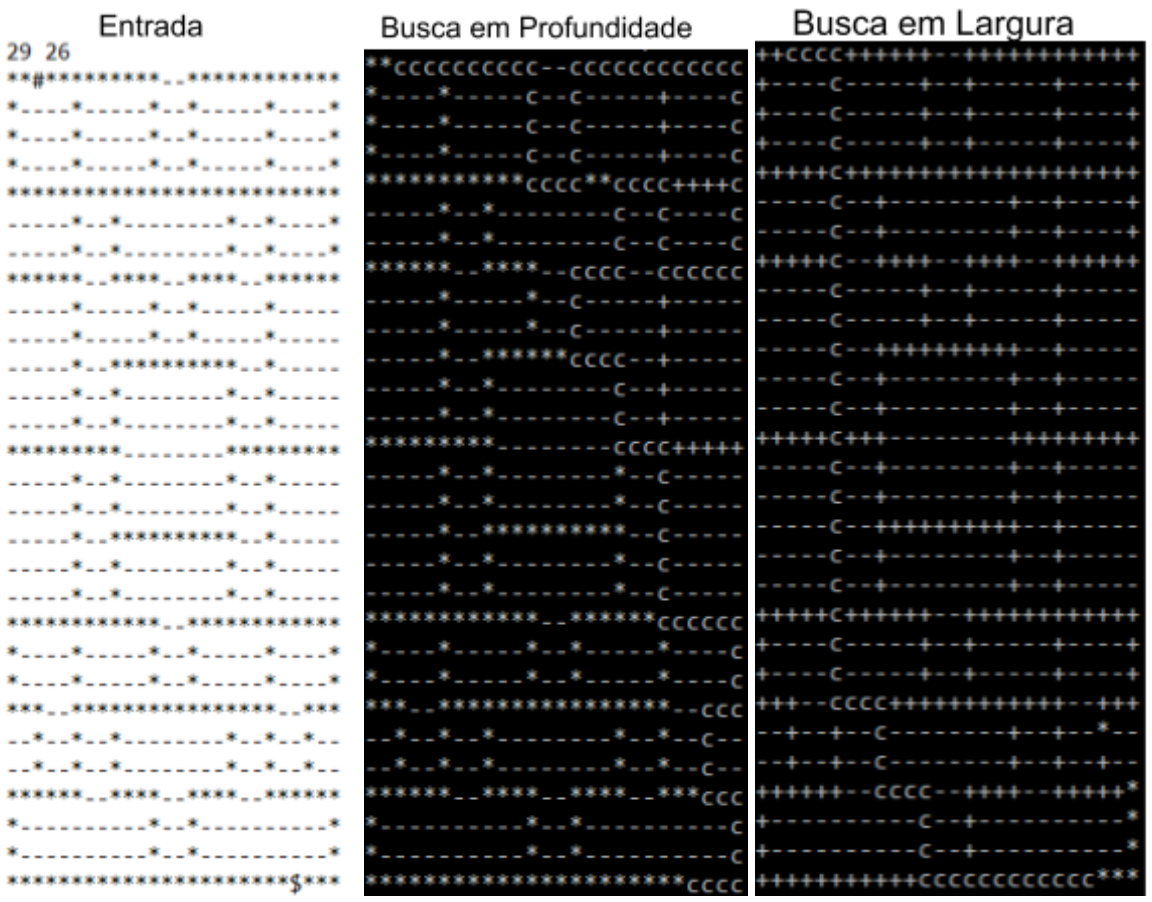
5.7)e7.txt

Tempo de Execução = 10.79620ms

Média = 4.529071ms

9. Exemplos de Execução

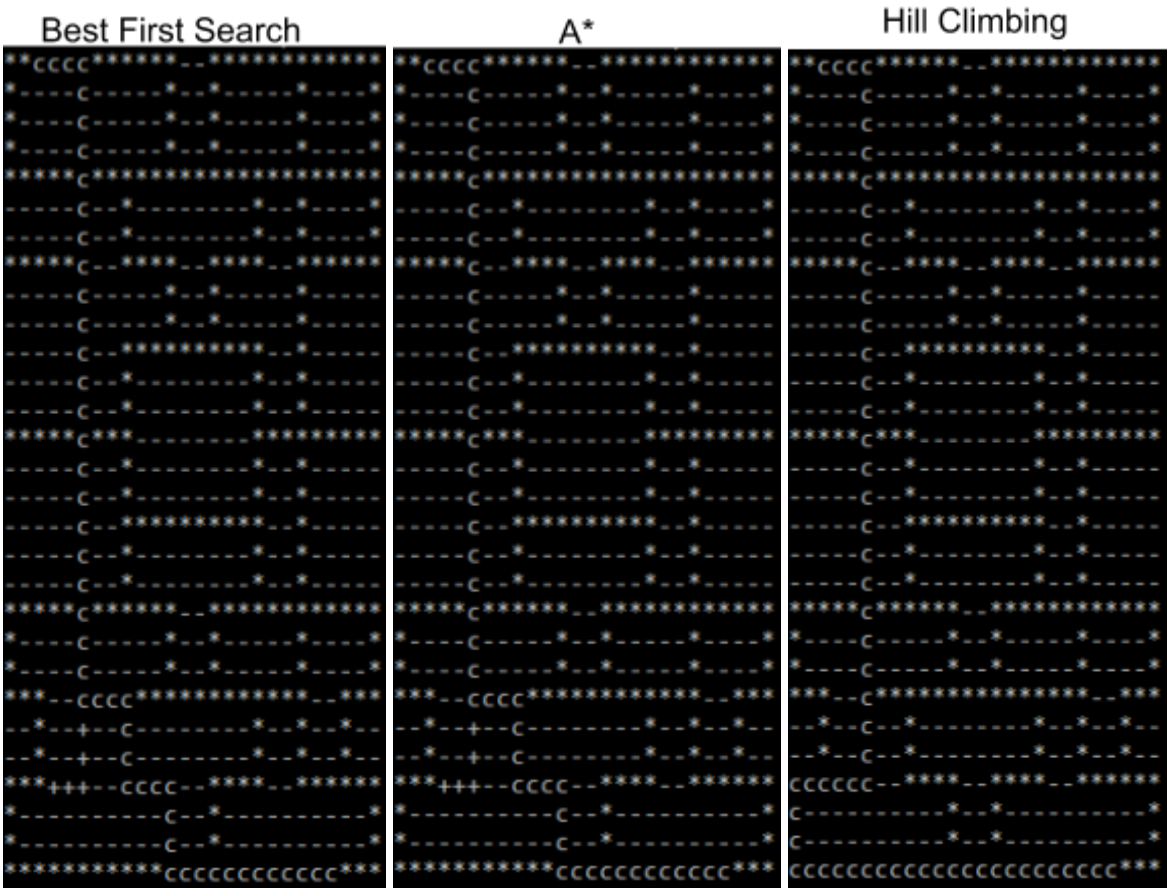
O símbolo ‘+’ nas matrizes retornadas pelos algoritmos representa as posições exploradas por cada algoritmo e não o caminho tomado. O símbolo ‘c’ representa o caminho escolhido:



Caminhos Encontrados por cada algoritmo:

Busca em Profundidade:
(0,2) (0,3) (0,4) (0,5) (0,6) (0,7) (0,8) (0,9) (0,10) (0,11) (1,11) (2,11) (3,11) (4,11) (4,12) (4,13) (4,14) (3,14) (2,14) (1,14) (0,14) (0,15) (0,16) (0,17) (0,18) (0,19) (0,20) (0,21) (0,22) (0,23) (0,24) (0,25) (1,25) (2,25) (3,25) (4,25) (5,25) (6,25) (7,25) (7,24) (7,23) (7,22) (7,21) (7,20) (6,20) (5,20) (4,20) (4,19) (4,18) (4,17) (5,17) (6,17) (7,17) (7,16) (7,15) (7,14) (8,14) (9,14) (10,14) (10,15) (10,16) (10,17) (11,17) (12,17) (13,17) (13,18) (13,19) (13,20) (14,20) (15,20) (16,20) (17,20) (18,20) (19,20) (19,21) (19,22) (19,23) (19,24) (19,25) (20,25) (21,25) (22,25) (22,24) (22,23) (23,23) (24,23) (25,23) (25,24) (25,25) (26,25) (27,25) (28,25) (28,24) (28,23) (28,22)

Busca em Largura:
(0,2) (0,3) (0,4) (0,5) (1,5) (2,5) (3,5) (4,5) (5,5) (6,5) (7,5) (8,5) (9,5) (10,5) (11,5) (12,5) (13,5) (14,5) (15,5) (16,5) (17,5) (18,5) (19,5) (20,5) (21,5) (22,5) (22,6) (22,7) (22,8) (23,8) (24,8) (25,8) (25,9) (25,10) (25,11) (26,11) (27,11) (28,11) (28,12) (28,13) (28,14) (28,15) (28,16) (28,17) (28,18) (28,19) (28,20) (28,21) (28,22)



Caminhos Encontrados por cada algoritmo:

Best-First Search
(0,2) (0,3) (0,4) (0,5) (1,5) (2,5) (3,5) (4,5) (5,5) (6,5) (7,5) (8,5) (9,5) (10,5) (11,5) (12,5) (13,5) (14,5) (15,5) (16,5) (17,5) (18,5) (19,5) (20,5) (21,5) (22,5) (22,6) (22,7) (22,8) (23,8) (24,8) (25,8) (25,9) (25,10) (25,11) (26,11) (27,11) (28,11) (28,12) (28,13) (28,14) (28,15) (28,16) (28,17) (28,18) (28,19) (28,20) (28,21) (28,22)

Busca A*
(0,2) (0,3) (0,4) (0,5) (1,5) (2,5) (3,5) (4,5) (5,5) (6,5) (7,5) (8,5) (9,5) (10,5) (11,5) (12,5) (13,5) (14,5) (15,5) (16,5) (17,5) (18,5) (19,5) (20,5) (21,5) (22,5) (22,6) (22,7) (22,8) (23,8) (24,8) (25,8) (25,9) (25,10) (25,11) (26,11) (27,11) (28,11) (28,12) (28,13) (28,14) (28,15) (28,16) (28,17) (28,18) (28,19) (28,20) (28,21) (28,22)

Hill Climbing
(0,2) (0,3) (0,4) (0,5) (1,5) (2,5) (3,5) (4,5) (5,5) (6,5) (7,5) (8,5) (9,5) (10,5) (11,5) (12,5) (13,5) (14,5) (15,5) (16,5) (17,5) (18,5) (19,5) (20,5) (21,5) (22,5) (23,5) (24,5) (25,5) (25,4) (25,3) (25,2) (25,1) (25,0) (26,0) (27,0) (28,0) (28,1) (28,2) (28,3) (28,4) (28,5) (28,6) (28,7) (28,8) (28,9) (28,10) (28,11) (28,12) (28,13) (28,14) (28,15) (28,16) (28,17) (28,18) (28,19) (28,20) (28,21) (28,22)