Alexandre Norcia Medeiros - 10295583
Daniel Penna Chaves Bertazzo - 10349561
Miguel de Mattos Gardini - 10295728

Trabalho 1

SCC0230 - Inteligência Artificial
Prof. Alneu de Andrade Lopes
Aluna PAE: Fabiana Góes

Ciências de Computação
ICMC USP São Carlos
18/10/2019

1. Introdução

O primeiro trabalho da disciplina de Inteligência Artificial teve como proposta a implementação de quatro algoritmos de busca, com o objetivo de analisar e comparar o desempenho dos mesmos, neste caso, aplicado na procura pela saída de um labirinto 2D. Este projeto foi desenvolvido em **C++**, decisão tomada por unanimidade, devido à prévia experiência de todos os membros do grupo.

2. Descrição das implementações

As buscas foram implementadas todas em um único arquivo. Existe uma função "***main()***" comum para todos os algoritmos, os quais se tratam de métodos da classe "***Labirinto***". As chamadas para as buscas estão comentadas no código da "***main()***", basta descomentá-las para executar o método que desejar.

Também é possível realizar mais de uma busca na mesma execução do código, graças ao método que reseta o labirinto, sobre o qual falaremos mais à frente.

O código possui 3 classes, as quais serão explicadas abaixo:

2.1 Classe *Vértice*

Trata-se, simplesmente, de uma tupla, "***int x***" e "***int y***", as quais guardam as coordenadas de uma posição no labirinto. Esta classe é utilizada dentro da classe "***Caminho***".

2.2 Classe *Caminho*

Trata-se de um vetor de vértices, que guarda literalmente um caminho realizado por uma busca.

Possui, também, as seguintes variáveis:

- ***float h***: valor da função heurística (usada no algoritmo A*).
- ***vector<Vertice> c***: Vetor que armazena os vértices do caminho
- ***int tamanho***: Número de vértices do vetor acima.
- ***float peso***: Soma dos pesos dos vértices, resultando no peso total do caminho.

2.2.1 ***Push_back()***, ***Pop_back()*** e ***back()***

- ***Push_back()***: Adiciona um vértice no fim do vetor ***caminho***.
- ***Pop_back()***: Remove um vértice do fim do vetor ***caminho***.
- ***Back()***: Acessa o vértice ao fim do vetor ***caminho***, mas sem removê-lo.

2.2.2 ***Calcula_h()*** (utilizado pela busca A*)

Faz o cálculo da distância entre o último vértice do caminho e o objetivo do labirinto.

2.2.3 Sobrecarga dos operadores ‘<’ e ‘>’

Um caminho é maior (>) que o outro, se seu “***peso***” for maior, e vice-versa.

2.2.4 exibe()

Função que printa todos os vértices do vetor “***c***”.

2.3 Classe *Labirinto*

Esta é a classe mais fundamental de todo o código. É nela que se encontram os algoritmos para a realização das buscas implementadas no trabalho. Além de armazenar o labirinto em si.

Seus atributos são:

- ***int lin, col***: Quantidade de linhas e colunas do labirinto.
- ***int xi, yi***: Posição inicial da IA.
- ***int xf, yf***: Objetivo da IA.
- ***vector< vector<char> > m***: Matriz que representa o labirinto. Cada membro da posição ***m[i][j]*** guarda o estado da posição:
    - ‘*****’ : Posição livre e ainda não visitada pelo algoritmo.
    - ‘**-**’ : Posição inválida (“parede”).
    - ‘**+**’ : Posição já visitada pelo algoritmo.
    - ‘***a***’, ‘***b***’, ‘***c***’ ... , ‘***z***’: Posições do caminho final na mesma ordem em que foram visitadas (em ordem alfabética para facilitar a visualização).

2.3.1 le_entrada()

Função que lê a entrada e armazena a estrutura do labirinto, bem como as informações das variáveis acima.

2.3.2 exibe()

Função que printa a estrutura do labirinto. Foi usado o ANSI color code para imprimir cores diferentes para cada componente do labirinto (parede, origem, objetivo, caminho final, pontos visitados)

2.3.3 Buscas

Cada busca é um método da classe. Elas fazem o processamento em cima das estruturas da própria classe, como a matriz ***m*** e as outras variáveis. Segue abaixo os detalhes sobre a implementação de cada algoritmo:

2.3.3.1 Busca em profundidade

A busca em profundidade foi feita recursivamente, já que é bastante natural pensar no algoritmo de forma recursiva.

A estratégia está restrita à ordem definida de escolha do próximo passo, e a busca sempre é feita na mesma ordem, de forma relativamente pouco inteligente. Sua eficiência é totalmente depende da estrutura do labirinto, o que torna essa busca extremamente instável.

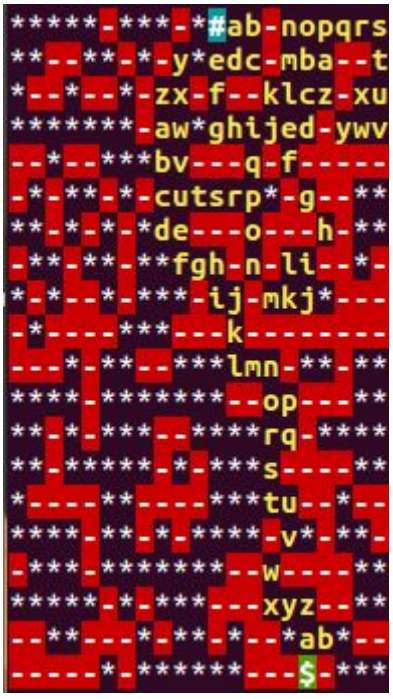
A ordem decidida por nós foi: *Cima* > *Direita* > *Baixo* > *Esquerda* > *Diagonal nordeste* > *Diagonal sudeste* > *Diagonal sudoeste* > *Diagonal noroeste*. Portanto, o caso ideal seria com o objetivo localizado na mesma coluna que a posição inicial, em uma linha mais acima, com nenhum obstáculo entre ambos.

A ideia da nossa implementação é seguir os seguintes passos:

- Verifica se a posição atual está fora do labirinto, está dentro de uma parede ou já foi visitada (retornando 0 em todos esses casos).
- Adiciona o vértice da posição atual no caminho.
- Retorna 1 caso a posição atual seja o objetivo
- Chamadas recursivas para todas as outras direções a partir da posição atual (seguindo a ordem explicada anteriormente), retornando 1 (caminho encontrado) a chamada recursiva tenha também retornado 1
- Remove o vértice atual do caminho e retorna 0 (caminho sem saída) se todas as chamadas recursivas tiverem retornado 0.

Comportamento da DFS em um labirinto vazio:

Comportamento da DFS em um labirinto gerado de forma aleatória:



2.3.3.2 Busca em largura

A busca em largura foi feita de forma iterativa, devido à necessidade de utilizar fila para armazenar os vértices visitados.

O algoritmo expande de forma praticamente igual em todas as direções, buscando a solução mais próxima. O problema, é que, quanto mais distante o objetivo, o número de vértices visitados cresce muito.

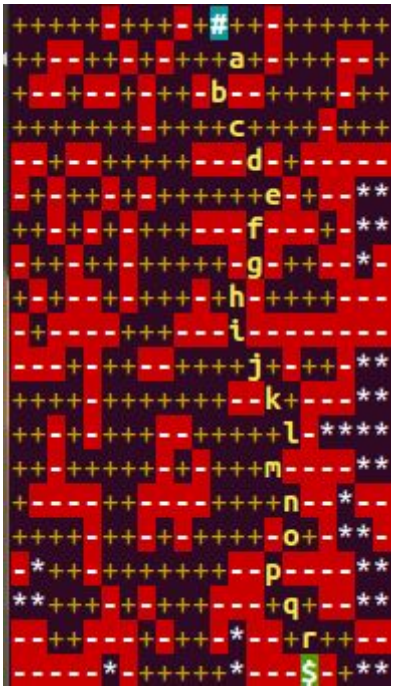
A ordem de enfileiramento dos visitados escolhida por nós foi: *Diagonal nordeste > Diagonal sudeste > Diagonal sudoeste > Diagonal noroeste > Cima > Direita > Baixo > Esquerda*.

Nossa implementação segue os seguintes passos:

- Cria uma fila '***queue<Caminho> q***' de caminhos
- Adiciona o vértice inicial na fila ***q***.
- Enquanto a fila ***q*** não estiver vazia:
    - Remove o caminho atual da fila
    - Verifica se o último membro do caminho atual da fila chegou no objetivo, saindo do *loop* se sim.
    - Adiciona todos os caminhos possíveis a partir da posição atual (+ 1 passo) na fila ***q***, marcando os novos vértices como visitados. Isso é feito na ordem de prioridade explicada anteriormente.
- Marca o caminho percorrido na estrutura do labirinto ('***this->m***').

Comportamento da BFS em um labirinto vazio:

Comportamento da BFS em um labirinto gerado de forma aleatória:



2.3.3.3 Busca Best-First Search

Essa busca também foi implementada iterativamente, pelo mesmo motivo da busca anterior.

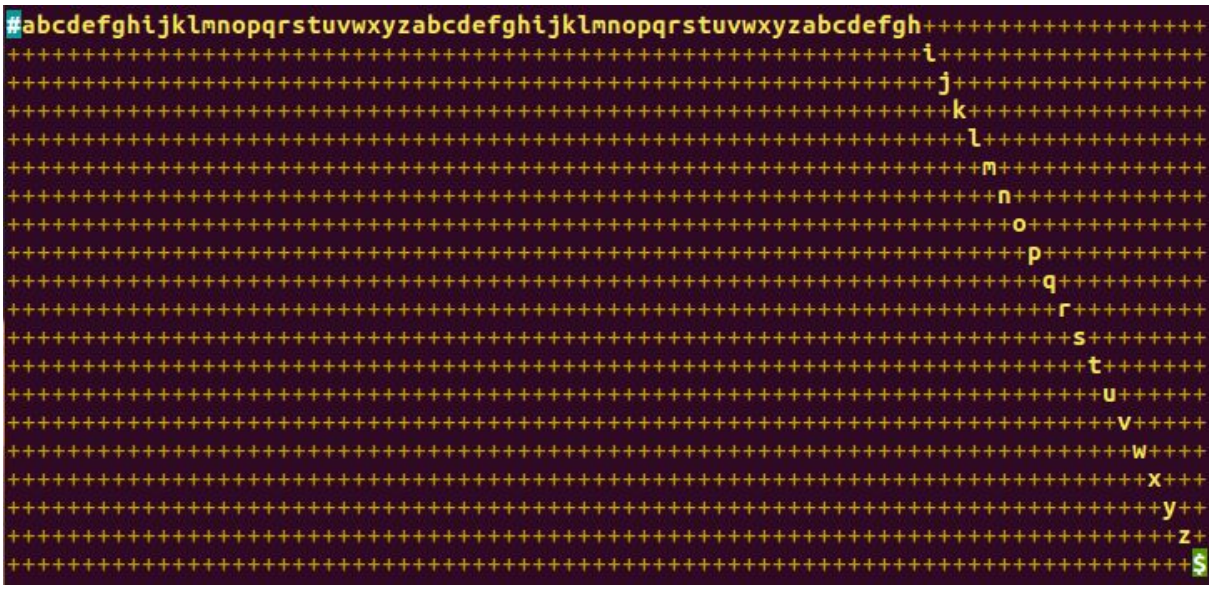
A grande diferença desse método é que ele prioriza a exploração pelo menor caminho até então. Porém, ele sempre explora o menor caminho atual, independentemente de quão perto esse caminho esteja do destino. Assim, ele visita muito mais vértices que a busca a*.

Neste caso, a ordem de enfileiramento se dá por meio da fila de prioridades, onde os menores caminhos possuem maior prioridade que os maiores.

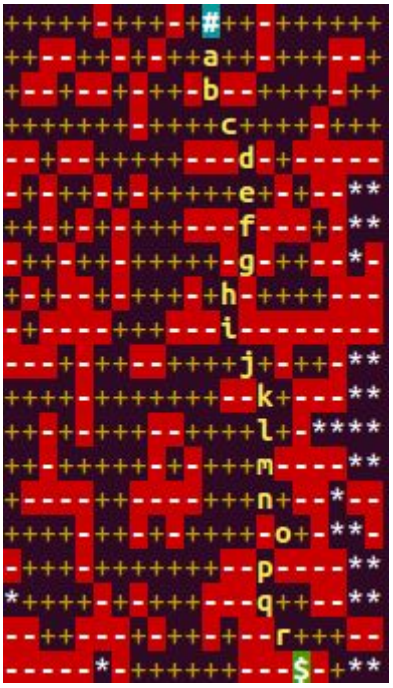
A ideia da nossa implementação é seguir os seguintes passos:

- Cria uma fila de prioridade '***priority_queue<Caminho, vector<Caminho>, greater<Caminho> > q***', onde o menor caminho fica no topo.
- Adiciona o vértice inicial na fila ***q***.
- Enquanto a fila ***q*** não estiver vazia:
    - Remove o caminho atual da fila
    - Verifica se o último membro do caminho atual da fila chegou no objetivo, saindo do *loop* se sim.
    - Adiciona todos os caminhos possíveis a partir da posição atual (+ 1 passo) na fila ***q***, marcando os novos vértices como visitados. A ordem dos comandos não faz diferença, uma vez que a fila ***q***, neste caso, é uma fila de prioridade.
- Marca o caminho percorrido na estrutura do labirinto ('***this->m***').

Comportamento da Best-First Search em um labirinto vazio:



Comportamento da Best-First Search em um labirinto gerado de forma aleatória:



2.3.3.4 Busca A*

Esta busca também foi feita, pelos mesmos motivos, de forma iterativa.

A busca A* pode ser vista como um aprimoramento da busca anterior. Neste caso, a prioridade do caminho não é mais apenas o menor caminhos de todos, mas também o caminho que está mais próximo do destino.

Assim, temos uma função f(x) que calcula a prioridade do caminho dada por:

f(x) = g(x) + h(x)

Onde g(x) é o tamanho atual do caminho e h(x) é uma das 3 seguintes heurísticas:

- h1(x) = Distância euclidiana do vértice atual até o objetivo.
- h2(x) = Distância euclidiana do vértice atual até o objetivo ao quadrado, o que faz com que o fator distância seja mais levado em consideração na equação de f(x).
- h3(x) = Distância *Manhattan* do vértice atual até o objetivo.

A escolha da heurística (h1, h2 ou h3) é decidida na chamada da função, passada como atributo (1, 2 ou 3, respectivamente).

Nossa implementação segue os seguintes passos:

- Cria uma função lambda '***f_comparacao***', que faz o cálculo da heurística explicada acima, para ser nossa f(x).
- Cria uma fila de prioridade '***priority_queue<Caminho, vector<Caminho>, decltype(f_comparacao)> q(f_comparacao)***', utilizando f(x) como função para determinar quais caminhos deverão vir primeiro.
- Adiciona o vértice inicial na fila ***q***.
- Enquanto a fila ***q*** não estiver vazia:

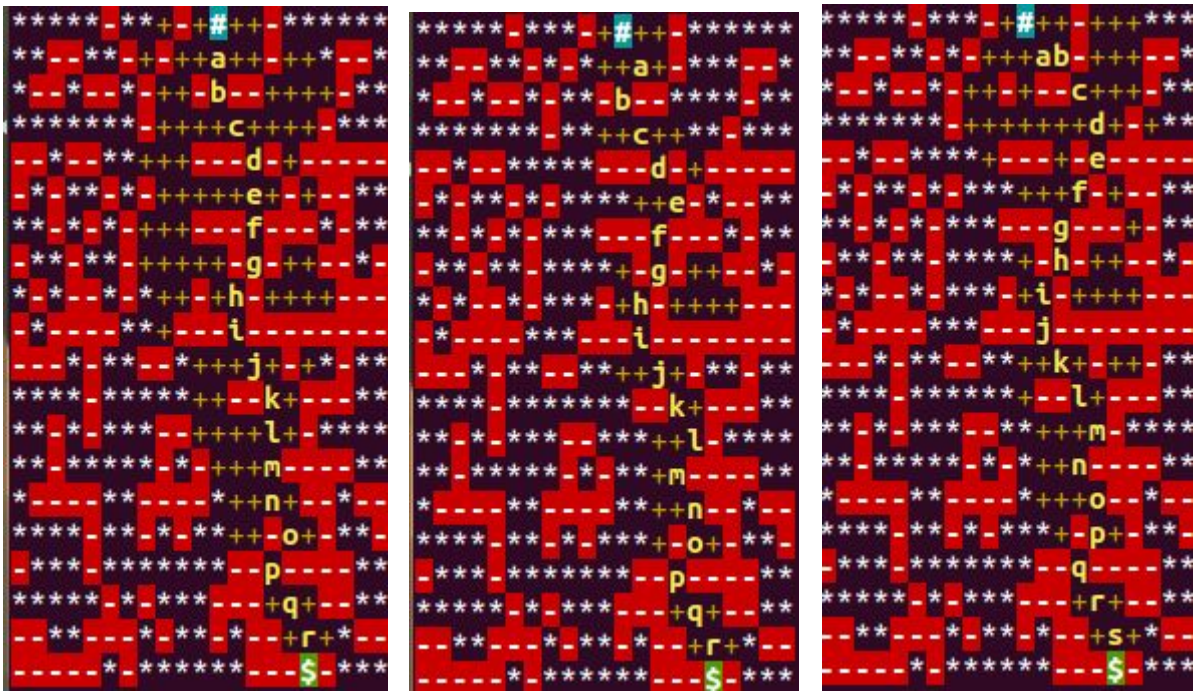
- Remove o caminho atual da fila
- Verifica se o último membro do caminho atual da fila chegou no objetivo, saindo do *loop* se sim.
- Adiciona todos os caminhos possíveis a partir da posição atual (+ 1 passo) na fila ***q***, marcando os novos vértices como visitados. A fila ***q*** ainda é uma fila de prioridade, mas, nesse caso, a prioridade é dada pela função lambda '***f_comparacao***'.

- Marca o caminho percorrido na estrutura do labirinto ('***this->m***').

Comportamento da A* em um labirinto vazio (heurística 1):

Comportamento da A* em um labirinto vazio (heurística 2 e 3):

Comportamento da A* em um labirinto gerado de forma aleatória (heurística 1, 2 e 3, respectivamente):



2.3.4 Geração de labirintos

A geração de labirintos para a realização das buscas foi implementada no arquivo "***gerador.cpp***". Com pequenas modificações neste código, é possível alterar a maneira como o número de linhas e colunas do labirinto será gerado. A princípio, eles são, também, gerados aleatoriamente a cada execução. Existem quatro maneiras distintas de gerar labirintos:

- **Labirintos linha:** Labirintos que alternam entre linhas livres (*) e linhas de parede (-). No caso das linhas de parede, apenas um ponto (aleatório) terá passagem, criando um caminho possível da origem (primeira linha) até o objetivo (última linha).
- **Labirintos coluna:** Labirintos que alternam entre colunas livres (*) e colunas de parede (-). No caso das colunas de parede, apenas um ponto (aleatório) terá passagem, criando um caminho possível da origem (primeira coluna) até o objetivo (última coluna).
- **Labirintos vazios:** Labirintos compostos apenas por passagens livres (*), usado para visualização do comportamento das buscas. Origem na posição [0, 0] e objetivo na posição [n_linhas - 1, n_colunas - 1], ou seja, em pontos opostos do labirinto.
- **Labirintos aleatórios:** Labirintos gerados de forma aleatória, onde há uma chance ***x*** de ocorrer passagem (*) e uma chance ***100 - x*** de ocorrer parede (-). A princípio, ***x*** está definido como 60.

3. Resultados

Aqui apresentaremos o desempenho dos algoritmos em labirintos (aleatórios) de tamanhos relativamente grandes, os quais não podem ser facilmente visualizados. Foram realizados testes com 3 tamanhos diferentes de labirinto, gerados. Os resultados podem ser conferidos abaixo:

3.1 Labirinto de tamanho 1000x200:

- **Busca em profundidade**: Não executa por falta de memória.

- **Busca em largura:**
    - Tamanho:1000
    - Peso: 1357.69
    - Tempo: 1.08215s

- **Busca best-first search**:
    - Tamanho:1001
    - Peso: 1142.9
    - Tempo: 1.38031s

- **A*** (com cada uma das heurísticas):
    - **h1** (distância euclidiana):
        - Tamanho:1004
        - Peso: 1143.83
        - Tempo: 1.21388s

    - **h2** (distância euclidiana ao quadrado):
        - Tamanho: 1080
        - Peso: 1347.4
        - Tempo: 0.038s

    - **h3** (distância manhattan):
        - Tamanho:1026
        - Peso: 1264.41
        - Tempo: 0.91527s

3.2 Labirinto de tamanho 2500 x 200:

- **Busca em profundidade**: Não executa por falta de memória.

- **Busca em largura**:
    - Tamanho:2500
    - Peso: 3390.85
    - Tempo: 6.31481s

- **Busca best-first search**:
    - Tamanho:2504
    - Peso: 2824.85
    - Tempo: 6.96418s

- **A*** (com cada uma das heurísticas):
    - **h1** (distância euclidiana):
        - Tamanho:2503
        - Peso: 2827.58
        - Tempo: 6.68329s

    - **h2** (distância euclidiana ao quadrado):
        - Tamanho:2607
        - Peso: 3233.15
        - Tempo: 0.184296

    - **h3** (distância manhattan):
        - Tamanho:2525
        - Peso: 3079.9
        - Tempo: 5.9555s

3.3 Labirinto tamanho 1000x1000:

- **Busca em profundidade**: Não executa por conta de memoria.

- **Busca em largura**:
    - Tamanho:1013
    - Peso: 1372.76
    - Tempo: 6.72095s

- **Busca best-first search**:
    - Tamanho:1019
    - Peso: 1234.63
    - Tempo: 8.10641s

- **A*** (com cada uma das heurísticas):
    - **h1** (distância euclidiana):
        - Tamanho:1040
        - Peso:1252.73
        - Tempo: 1.73668s

    - **h2** (distância euclidiana ao quadrado):
        - Tamanho:1076
        - Peso: 1383.16
        - Tempo: 0.032704s

    - **h3** (distância manhattan):
        - Tamanho:1112
        - Peso: 1401.77
        - Tempo: 0.828971s

4. Discussão sobre as heurísticas e desempenho dos algoritmos

4.1 Busca em profundidade

A busca em profundidade se mostrou pouco eficiente em vários casos, produzindo na grande maioria das vezes caminhos estranhos e desnecessariamente grandes.

Devido à sua total dependência da ordem das chamadas recursivas, ela sempre estará suscetível a andar em uma direção que não leva ao destino até atingir uma parede, adicionando estes vértices desnecessários ao caminho final. Portanto, o algoritmo também tende a andar "grudado" nas paredes.

Além disso, o seu caso ideal (de haver uma linha reta sem obstáculos entre a origem e o destino, na mesma direção onde ocorre a primeira chamada recursiva) é extremamente difícil de ocorrer em um labirinto aleatório.

A estratégia pode se mostrar útil em labirintos que possuam apenas 1 solução, pois o algoritmo é simples e de fácil implementação, e encontrará a mesma solução que todas as outras heurísticas.

Entretanto, em labirintos que possuam campos mais "abertos", o algoritmo muitas vezes se comporta de maneiras estranhas, como seguindo paredes ou andando em círculos em volta do objetivo.

4.2 Busca em largura

A busca em largura é capaz de encontrar caminhos de tamanho mínimo, o que já é uma vantagem sobre a busca em profundidade, analisada acima. Entretanto, ela ainda é um pouco limitada à ordem em que os vértices são empilhados no código. No caso da nossa implementação, por exemplo, as diagonais são empilhadas primeiro, o que faz com que a busca priorize andar em diagonais ao invés de andar para os lados dos quadrados. Isso pode ser facilmente notado ao analisar as imagens.

Além disso, o aumento da área do labirinto, juntamente com o aumento da distância entre a origem e o objetivo, impacta fortemente no desempenho deste algoritmo. Isso ocorre pois a busca em largura explora todas as direções na mesma velocidade, resultando em uma espécie de "círculo" que vai se expandindo ao redor do ponto inicial. Portanto, conforme o objetivo se distancia linearmente do vértice inicial (raio da circunferência aumenta), o espaço de busca e os vetores visitados pelo algoritmo cresce quadraticamente (πr^2).

4.3 Busca Best-First Search

Este algoritmo já é mais inteligente que a anterior, pois caminhos maiores têm menor prioridade na fila, evitando visitas desnecessárias em caminhos sinuosos que não levem para o objetivo. No entanto, essa característica não faz diferença (com relação à busca em largura) em labirintos sem obstáculos (campos abertos) ou quando a solução for o maior caminho possível. No segundo caso, isso ocorre por que ele busca todos os outros caminhos possíveis antes da solução.

De qualquer forma, a solução encontrada por este algoritmo é ótima, o que já é uma vantagem considerável, apesar dos problemas acima.

4.4 Busca A*

A busca A* é claramente mais complexa que as anteriores. Essa complexidade acaba por dar a ela uma inteligência superior e, dependendo da heurística e do problema a ser resolvido, resultados bem melhores também.

A grande vantagem deste algoritmo se encontra em sua heurística, a qual resulta o espaço da busca. Essa redução do espaço pode ser maior ou menor dependendo da heurística utilizada e do problema enfrentado.

Aqui podemos ver 3 heurísticas diferentes utilizadas. Inicialmente podemos comparar a distância euclidiana com peso normal e a distância euclidiana ao quadrado (sem tirar a raiz). A segunda heurística, naturalmente, dá muito mais ênfase em caminhos que estão mais

próximos do objetivo, o que impacta positivamente seu desempenho em labirintos onde a solução é um caminho com poucos obstáculos entre a origem e o destino. Um exemplo onde isso é facilmente observado é no exemplo do labirinto sem obstáculos (campo aberto), onde o desempenho dessa estratégia acaba sendo muito superior ao das outras buscas.

Em labirintos muito grandes, essa heurística da distância ao quadrado enviesa muito o caminho, o que faz com que o mesmo perca o resultado ótimo, porém, ele executa extremamente mais rápido, por explorar muito menos, com um resultado ainda considerável. A primeira heurística (distância com o peso normal) sempre mantém a solução ótima, o que dá a ela uma grande vantagem caso essa seja a prioridade do usuário.

Além das 2 heurísticas anteriores, existe uma terceira, que é a distância *Manhatan*. Essa heurística explora menos que a primeira (distância normal), mas tem um resultado bem parecido com o dela, o que pode ser uma vantagem. Entretanto, sua execução não é tão rápida quanto a da distância euclidiana ao quadrado nos casos com poucos obstáculos explicados anteriormente.

Comparação entre **A*** e **Best-first search**

Uma comparação interessante que pode ser feita entre os **dois algoritmos** é a influência que ambos sofrem pelo comprimento da solução e pela área do labirinto. Enquanto o primeiro é afetado apenas pelo tamanho do caminho ideal, o segundo é afetado por ambos os fatores. Por exemplo: apesar do comprimento da solução no labirinto vazio 1000x1000 ser menor que a solução no labirinto vazio 2500x200, o resultado do **Best-first search** em ambos os casos é o mesmo em questões de tempo de execução.

Isso ocorre por que a área do segundo labirinto é metade da área do primeiro, enquanto sua solução é aproximadamente o dobro do comprimento, resultando em desempenhos bastante próximos do **Best-first search**. Por outro lado, o **A*** tem um desempenho bem melhor no labirinto 1000x1000, já que a solução desse caso possui um comprimento bem menor (1000 vértices), então o algoritmo visita bem menos vértices, apenas os que estão bem próximos da diagonal do quadrado.

5. Instruções de compilação e execução

Para a compilação, qualquer compilador normal consegue compilar o arquivo, já que o código fonte está contido em um único arquivo. Utilizamos o g++ do linux:

- `g++ t1.cpp -o t1 -std=c++11`

Para compilar o gerador de labirintos, temos:

- `g++ gerador.cpp -o gerador`

A execução é feita conforme especificado no trabalho, precisando de uma entrada contendo a dimensão do labirinto, seguido do labirinto em caracteres. Também pode ser feita através do redirecionamento de um arquivo para a entrada padrão. Exemplo:

- `./t1 < entrada1.txt`

Para rodar o gerador, temos (dentro do código, também há instruções de uso):

- `./gerador [Mode] > out.txt`