1 Introdução

Tivemos como objetivo desse trabalho a elaboração e comparação de desempenho de quatro diferentes algoritmos de busca - sendo duas buscas cegas e duas buscas informadas.

- As buscas cegas são:
  - Busca em Profundidade;
  - Busca em Largura.
- As buscas informadas são:
  - Best-first Search;
  - A*.

Para isso foram utilizados tabuleiros tais como foram descritos na especificação do trabalho com uma origem, um destino e obstáculos distribuídos, os quais serão gerados a partir de um script de autoria própria. As métricas utilizadas para comparar os algoritmos são: tempo de execução, quantidade de casas no caminho encontrado, quantidade de casas analisadas (um vizinho no caso das buscas cegas e casas consideradas nas buscas informadas), quantidade de casas visitadas durante a execução e tamanho do caminho encontrado. O código foi desenvolvido em Python 3, sendo necessária a instalação do pacote celluloid (utilizado pelo grupo para gerar animações com a execução dos algoritmos). Para isso, basta executar o seguinte comando no terminal:

```
sudo pip3 install celluloid
```

2 Tabuleiros

Foram criadas funções para criação, leitura e escrita de tabuleiros que serão usados no presente trabalho. Para a manipulação e visualização, mostrou-se mais fácil utilizar matrizes numéricas para representar os objetos sobre o tabuleiro, enquanto nos foi requisitado o uso de representação com caracteres para as casas na leitura e escrita dos tabuleiros. Desta forma, estabeleceu-se o código apresentado pela tabela 2, implementado no dicionário `str2n` contido no arquivo `utils.py`.

Significado	Código de caracteres	Código numérico
Casa livre	*	0
Parede ou obstáculo	-	4
Casa objetivo	$	3
Casa de início da busca	#	2
Marcações temporárias	(Nenhum)	<1

Table 1: Tabela de conversão entre os diferentes códigos utilizados no projeto.

2.1 Criação dos tabuleiros

Criou-se algoritmo para a geração automática de tabuleiros, com suas paredes e os pontos de início e fim da busca. Visto que se deseja aparência semelhante à exposta na descrição do projeto (Figura **??**), não seria possível a geração completamente aleatória das paredes, e fez-se necessário desenvolver a estratégia descrita adiante.

2.1.1 Visão geral

O processo de gerar tabuleiros é executado pelo script `gen_boards.py`, mais especificamente e em seu mais alto nível pela função `gen_board` implementada no referido *script*, acontecendo da seguinte maneira:

1. É criado um tabuleiro (matrix) em branco (preenchido com zeros), a partir das dimensões informadas com argumento;

2. São sorteados dois pontos aleatoriamente para servirem de início e fim da busca. São sorteadas quantas vezes forem preciso até que tenham distância Manhattan entre si maior que um comprimento heuristicamente definido como a soma das dimensões do tabuleiro sobre 2;

3. São construídas as paredes, como melhor explicado posteriormente.

[código da função]

2.1.2 Orquestração da contrução das paredes

As peças mais importantes dese processo são as duas funções `build_walls` e `random_walk`, a primeira sendo de nível superior. O algoritmo de criação das paredes segue o seguinte raciocínio, coordenado pela `build_walls`:

1. Uma casa aleatória do tabuleiro (chamada *seed* no código) é sorteada por meio da função `seeds_gen`;

2. É desenhada uma parede a partir dessa casa com a função random walk;

3. O processo é repetido `nseeds` vezes, um dos parâmetro sda função `build_wals`, definido empiricamente por padrão como um décimo da área do tabuleiro.

[código da função]

2.1.3 Construção de cada parede

A partir de cada semente (casa aleatória do tabuleiro) fornecida à função `random_walk` pela função `build_wals`, será traçada (ou pelo menos tentar-se-á traçar) uma nova parede. O papel da `random_walk` é então "andar" pelas casas do tabuleiro marcando-do com o símbolo que designará aqueles quadrados como um novo obstáculo. A `random_walk` pede como argumento duas funções essenciais, a `end_func` e a `turn_func`, que devem receber o comprimento do caminho traçado e retornar um valor booleao. O processo de traçado executado pela `random_walk` é então esclarecido a seguir:

1. A partir da posição inicial (argumento `start`), verifica-se quais são os deslocametos unitários possíveis a partir `start`, isto é, que não levarão a casas ocupadas por algum obstáculo, que levarão a casas marcadas com 0, e sorteia-se um desses "passos" (tuplas no formato (1,0), (-1, 1), (0, -1), etc.). Os passos podem ser restritos aos ortogonais (baixo, cima, direita, esquerda) definindo como `True` o parâmetro `orth`;

2. Se não houver passo possível, a parede não é criada;

3. Caso contrário desloca-se a posição para posição + passo e marca-se essa casa como parede (o número marcado é dado pelo argumento `trail`). As outras casas do tabuleiro referentes aos outros passos possíveis não escolhidos são também marcadas com algum número menor que 1 (0.1 no caso), para que não sejam ocupadas em iterações posteriores e mantenham as paredes separadas entre si;

4. Esse processo de deslocamento e marcação prossegue, avançando com o mesmo passo sorteado, na mesma direção, até que:

   (a) É encontrado um obstáculo (casa do tabuleiro com valor não nulo) à frente na direção escolhida atual;

   (b) A função `turn_func` retorne `True`, caso em que a direção (passo) será sorteada novamente, ou;

   (c) A função `end_func` retorne `True`, caso em que a criação da parede será finalizada.

As funções `end_func` e `turn_func` são uma boa forma de controlar a dinâmica da criação de paredes. Se esses argumentos da função `random_walk` são providos a ela como `floats` entre 0 e 1, a `random_walk` os substitui por funções que retornam `True` com a probabilidade representada pelos `floats` fornecidos.

Outra possibilidade criada, é fornecer um inteiro como argumento `len` para a `random_walk`, caso em que `end_func` se torna função que retorna `True` se a distância traçada for maior que o inteiro fornecido. Nesse caso, o inteiro representaria um comprimento máximo para a parede, de forma que ela seria finalizada por colisão com uma casa não vazia ou por atingir esse comprimento máximo.

Para os experimentos são usadas `turn_func = 0.2` e `end_func = 0`, de forma que há sempre um quinto de probabilidade de virar, e a parede será desenhada até que se encontre um obstáculo.

3 Algoritmos de busca

Todos os algoritmos de busca possuem o mesmo cabeçalho:

```
def search(board: list, origin: tuple,
           target: tuple, camera: Camera = None) -> list:
```

em que `board` é o tabuleiro, `origin` é a tupla da casa de início, `target` é a tupla da casa de destino e `camera` é utilizada somente na criação de imagens GIF para visualização do caminho tomado pelo algoritmo.

3.1 Busca cega

Nessa modalidade de busca, o algoritmo não faz uso de nenhuma informação sobrea localização de casa-alvo. Os tipos implementados são descritos a seguir.

Figure 1: Exemplo de tabuleiro 80x80 criado com o algoritmo descrito. A casa vermelha é a posição de início da busca, e a laranja é a casa-objetivo.

3.1.1 Busca em profundidade

3.1.2 Busca em largura

3.2 Busca informada

Nos algoritmos de busca informada, utiliza-se o conhecimento das coordenadas do alvo para guiar a procura. Contudo, a presença e localização dos obstáculos não é conhecida, e faz-se necessário que o algoritmo determine os melhores caminhos alternativos de desvio.

Como nas buscas cegas, cria-se uma lista de casas a serem visitadas conforme se explora o tabuleiro. Contudo, a diferença essencial é a forma como os elementos são retirados dessa lista: A cada casa com coordenadas `pos` que se pretende adicionar à lista, utiliza-se uma função chamada `f(pos)`, que, em posse do conhecimento da posição da casa-objetivo, fornece um valor peso para a nova casa explorada. Esses pesos são interpretados como o quanto será favorável explorar aquele casa ou não, de forma que ao ser inserida a casa na lista, cria-se naturalmente uma fila de prioridades em função da ordem desses valores peso.

Essa função `f(pos)` ainda se desdobra em dois termos:

```
f(pos) = g(pos) + h(pos),
```

sendo que `g(pos)` retorna a distância de `pos` até a casa inicial, isto é, o quanto teríamos nos deslocado durante a busca desde o início caso estivéssemos em `pos`; e `f(pos)` retorna uma estimativa da distância entre `pos` e o alvo.

O caso mais geral desse tipo de busca, que abrange todas as buscas que usam a função `f(pos)` como descrito é a busca A* (pronunciada "A estrela"). No caso específico em que `g(pos)` seja 0, o algoritmo é chamado, *algoritmo de Dijkstra*, não utilizado no presente trabalho. Se, por sua vez, `g(pos)` seja definida como 0, o algoritmo é chamado busca *best-first*.

Sobre o cálculo das distâncias, além da norma euclideana mais canonicamente utilizada em problemas do tipo, faz-se experimentações com uma função norma diferentemente pensada, aqui chamada *distância trapezoidal*, justamente por se comportar da forma apresentada na figura 3.2.

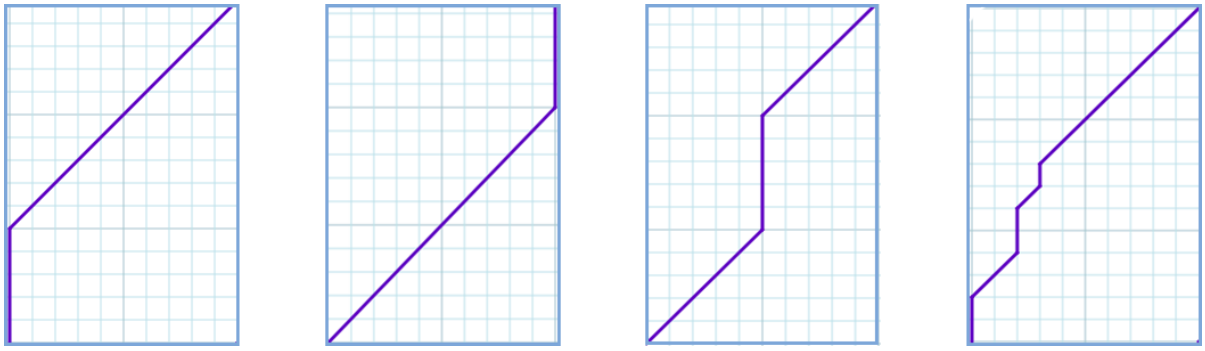

Figure 2: Várias representações da mesma distância trapezoidal entre pontos em vértices opostos do retângulo.

Dada sua representação, se os pontos que se distam formam um retângulo de lados a e b e assumindo ab sem perda de generalidade, temos que a norma trapezoidal será dada por $\sqrt{2}a + (a - b)$, ou em código:

```
def trapezoidal_dist(pos: tuple, target: tuple) -> float:
    """ Distancia trapezoidal de pos a target """
    if pos not in trapezoidal_dist.values:
        a = abs(pos[0] - target[0])
        b = abs(pos[1] - target[1])
        d = abs(a-b)
        trapezoidal_dist.values[pos] = sqrt(2) * min(a, b) + d
    return trapezoidal_dist.values[pos]
```

Em que se armazena dinamicamente os resultados para futuras consultas.

É possível provar, embora fuja do escopo do presente trabalho que, dadas as restrições de movimento impostas, a distância trapezoidal é a mínima distância percorrida pelos algoritmos de busca, de forma a fazer com que a heurística assuma o máximo valor ainda ótimista para as trajétorias de busca, favorecendo tempo de processamento sem deixar de garantir que o mehor caminho seja retornado na busca A*.

3.2.1 Busca *best-first*

Esse algoritmo de busca é implementado no arquivo `best_first.py`, e, como antes dito, é caracterizada pelo uso de heurística que considera somente a distância estimada ao alvo a partir da casa sendo visitada. Assim sendo, não há garantia de retorno do caminho ótimo, pois não se leva em conta o peso do caminho da posição inicial à casa visitada.

A função `search` no arquivo, responsável por executar a busca, utiliza um set com as casas visitadas, para se certificar de que não entre em loop visitando a mesma casa mais de uma vez.

A lista `queue` é justamente a responsável por guardar os camihos percorridos. Cada vez que uma casa é visitada, ela é substituída pela concatenação entre ela e seus filhos, de forma que cada item de `queue` seja uma tupla com o caminho todo percorrido até uma determinada casa e o valor de peso retornado pela função heurística. As funções `heappop` e `heappush` do módulo `heap` são responsáveis por inserir e retirar os caminhos da lista mantendo sua ordem de prioridades.

A identificação de casas filhas disponíveis é feita pela função `available_moves`, contida no arquivo `utils.py`.

3.2.2 Busca A*

Como antes mencionado, a busca por novas células através do algoritmo A* é feita utilizando dois cálculos: uma função `g` que determina o custo do caminho da origem até a posição atual, e outra função `h` (heurística) que determina um custo estimado otimista do caminho da posição atual até o destino. Estamos interessados em uma função `f` tal que `f = g + h`.

O cálculo de g está implementado no arquivo `a_star.py`, na função `calc_g(pos1, pos2)`, sendo `pos2` a posição que se deseja explorar e `pos1` o nó pai de `pos2` no tabuleiro. Inicialmente, a função `calc_g()` determina se `pos1` e `pos2` diferem em apenas uma dimensão ou em ambas (variável `dist`): caso `dist` seja igual a 1 sabemos que o passo foi dado em uma mesma dimensão, portanto o peso do passo será igual a 1; caso contrário (`dist` igual a 2) sabemos que o passo foi dado em uma diagonal, então o peso do passo será igual a $\sqrt{2}$. Por fim, obtemos o valor de `g` para `pos2` somando o valor anterior com o valor de `g` já calculado anteriormente para `pos1`, e armazenamos esse valor em uma estrutura de dicionário caso já não o possua ou o novo valor seja menor que o anterior.

A cálculo de `h` pode ser obtido através de duas funções (heurísticas) diferentes, ambas implementadas no arquivo `util.py`: distância euclidiana (função `euclidian_dist(pos, target)`) e distância trapezoidal (função `trapezoidal_dist(pos, target)`). `euclidian_dist` é a distância euclidiana (linha reta) entre `pos` e `target`, `trapezoidal_dist` simula a distância que seria de fato percorrida caso não houvessem obstáculos, portanto segue o formato de um trapézio. Por conta disso, a primeira função é mais otimista do que a segunda e, portanto, espera-se que a primeira seja mais lenta.

A busca A* está implementada no arquivo `a_star.py`, na função `search(board, origin, target)`, onde `board` é o tabuleiro (labirinto), `origin` é a posição inicial e `target` é a posição destino. Dentro dessa função está declarada a função `calc_path(parents)`, cujo objetivo é calcular o caminho

percorrido desde `target` até `origin`, utilizando o dicionário `parents` que foi gerado durante a busca e representa a posição pai de cada posição visitada.

Na busca em si utilizamos 5 estruturas de dados: `open_list` é uma fila de prioridades que guarda as posições que foram analisadas porém ainda não foram totalmente processadas (ou seja, seus filhos não foram analisados), o parâmetro de ordenação da fila de prioridades é o valor de `f` de cada posição; `closed_list` é uma estrutura do tipo set que guarda as posições já totalmente processadas, ele é utilizado para verificar se a próxima posição a ser analisada ainda não foi processada (caso tenha sido, essa posição é ignorada), o tipo set foi utilizado para otimizar a busca e inserção, cujas complexidades são O(1) (constantes); `parents` é um dicionário (par chave-valor) onde as chaves são posições do tabuleiro e o valor é o respectivo pai da posição na chave; `calc_g.values` é um dicionário pertencente à função `calc_g` e guarda os cálculos do menor `g` para cada posição `pos2` do tabuleiro, partindo de `pos1`; `trapezoidal_dist.values` pertence à função `trapezoidal_dist` e guarda os cálculos da heurística `h` em distância trapezoidal para cada posição pos do tabuleiro até o destino. No loop principal da busca, o algoritmo checa de `open_list` está vazio, retira a próxima posição de `open_list` (com menor valor de f), checa se a posição é o destino (caso for, encerra a busca e calcula o caminho),