Trabalho Prático I
Inteligência Artificial (DCC028)

Hugo Araujo de Sousa
(2013007463)

Departamento de Ciência da Computação
Instituto de Ciências Exatas
Universidade Federal de Minas Gerais

1. Introdução

Em Inteligência Artificial, um tipo particular de agente baseado em objetivo é o **agente de resolução de problemas** [Russell e Norvig 2016]. Esses agentes possuem formulações bem definidas do problema a resolver e consideram como solução para esse problema uma sequência de ações a serem tomadas. Dados um estado inicial e um estado objetivo, esses agentes devem ser capazes de obter uma sequência de ações que os levem do inicial ao objetivo. Esse processo pode ser feito através de uma busca no espaço de estados.

O processo fundamental da busca no espaço de estados é o de analisar o estado atual do agente, obter os estados que podem ser atingidos a partir do estado atual e escolher um deles para seguir, deixando outras opções para mais tarde, caso a opção atual não leve a uma solução. Um exemplo de problema de busca em espaço de estados é o problema de achar uma rota em um determinado ambiente, em que cada posição no ambiente define um estado no espaço. Essas posições podem estar bloqueadas ou livres, permitindo ou não que o agente se movimente por elas, o que torna necessário que o agente seja capaz de decidir qual o melhor caminho a seguir e o que fazer caso encontre posições bloqueadas nesse caminho.

Nesse trabalho, serão implementados algoritmos de busca em espaço de estados, aplicados em um ambiente representado por um mapa de duas dimensões. Os algoritmos serão analisados e comparados, a fim de comparar os caminhos encontrados, o tempo de execução e o número de nodos expandidos no grafo de busca de cada mapa.

2. Implementação

Esta seção descreve como a foi feita a modelagem do problema de busca em espaço de estados e as escolhas de implementação realizadas.

2.1. Mapas

Cada mapa passado como entrada dos algoritmos é representado por uma matriz de caracteres. Caracteres "." indicam posições no mapa livres para serem atravessadas pelo agente. Já caracteres "@" indicam posições bloqueadas. Os agentes podem se mover em oito direções dentro dos mapas: cima, baixo, direita, esquerda, todas com o custo de 1, e as quatro diagonais, com o custo 1,5. É importante ressaltar que movimentos que levem o agente para fora do mapa de entrada ou que o façam atravessar diagonais em cantos bloqueados não são permitidos, isto é, não é possível seguir um caminho pela diagonal superior direta, por exemplo, caso as posições acima ou à direita do agente estejam bloqueadas.

2.2. Busca em Grafos

Ao todo foram implementados 4 algoritmos de busca em espaço de estados, descritos em detalhe na Seção 2.3. Como todos os algoritmos usam o mesmo arcabouço estrutural, com pequenas modificações, a modelagem de dados levou isso em consideração. Dessa forma, implementando todos as estruturas e algoritmos utilizando a linguagem *Python* 3.5, foi definida uma classe **GraphSearch**, responsável pela implementação da estrutura principal da busca em espaço de estados utilizando grafos. Para isso, também foi necessário implementar a classe **Node**, que representa um nodo no grafo de busca. Esse nodo guarda informações sobre a posição do mapa que representa, o custo do caminho até essa posição e o nodo anterior a ele no grafo.

A busca em grafos implementada neste trabalho utiliza a versão completa de busca, isto é, o agente evita estados já visitados. Para isso, são mantidas duas listas em memória: a lista **aberto**, que representa os nodos já visitados, porém, ainda não expandidos, e a lista **fechado**, que indica quais nodos já foram visitados e expandidos.

2.3. Algoritmos de Busca

Uma vez definida a estrutura principal dos algoritmos de busca, 4 algoritmos foram implementados. Eles são descritos a seguir.

- **Busca de aprofundamento iterativo (IDS):** Um dos tipos mais simples de busca em grafos é a busca em profundidade. Com ela, o algoritmo sempre prossegue para o nível mais profundo do grafo (as folhas). À medida que os nodos nos níveis mais profundos são expandidos, a busca volta ao nodo mais profundo que ainda tem sucessores não explorados.
  Um dos principais problema da busca em profundidade é que ela percorre os maiores caminhos da raiz do grafo até as folhas, quando a solução mais rasa (e de menor custo) pode não estar em um nível tão profundo do grafo de busca. A busca de aprofundamento iterativo resolve esse problema ao executar a busca de profundidade iterativamente, cada vez com um valor limite de profundidade. Essa busca foi implementada na classe **IDS**, utilizando uma pilha para implementar a lista "aberto".
- **Busca de custo uniforme:** Esse tipo de busca expande sempre, dentre os nodos na lista "aberto", aquele com o menor custo de caminho **g(n)**. Para isso, a lista "aberto" é implementada como uma fila de prioridade ordenada por g. Esse algoritmo está implementado na classe **UniformCost**.
- **Busca gulosa (best first search):** Esse algoritmo tenta sempre expandir o nodo que está mais próximo do objetivo, pensando que isso levaria a uma solução mais rapidamente. Para isso, a busca avalia os nodos usando simplesmente uma função heurística **h(n)**, que serve para ordenar a lista "aberto" como uma fila de prioridades. Para este trabalho, a heurística desse tipo de busca foi a *Manhattan*, que é calculada dessa forma:

$$dx = abs(node.x - goal.x)$$
$$dy = abs(node.y - goal.y)$$
$$h(n) = (dx + dy)$$

  O algoritmo de busca gulosa está implementado na classe **BestFirst**.

- **Busca A*:** A busca A* tenta minimizar o custo total estimado da solução. Para isso, avalia não só uma função heurística **h(n)**, mas também o custo para atingir o nodo **g(n)** em uma função **f(n) = g(n) + h(n)**, que ordena a lista "aberto" como uma fila de prioridades. Para este trabalho, o algoritmo A* foi implementado com duas possíveis opções de função heurística: a distância *Manhattan*, descrita acima, e a distância *Octile*, que é calculada da seguinte forma:

$$\begin{aligned} dx &= abs(node.x - goal.x) \\ dy &= abs(node.y - goal.y) \\ h(n) &= max(dx, dy) + 0.5 * min(dx, dy) \end{aligned}$$

O algoritmo de busca A* está implementado na classe **AStar**.

2.4. Otimalidade da busca A*

A otimalidade do algoritmo A* é dependente de algumas características da heurística utilizada. Observemos as duas características a seguir.

- **Heurística Admissível:** Uma heurística é admissível se nunca superestima o custo para atingir um nodo. Isto é, *f(n)* nunca superestima o custo verdadeiro de uma solução ao longo do caminho atual através de *n*.
- **Heurística Consistente:** Uma heurística é consistente se, para todo nodo *n* e todo sucessor *n'* de *n* gerado através de uma ação *a*, o custo estimado para atingir o estado objetivo a partir de *n* nunca é maior que o custo do passo até *n'* mais o custo estimado de se atingir o objetivo a partir de *n'*:

$$h(n) \leq c(n, a, n') + h(n')$$

De acordo com [Russell e Norvig 2016], a versão de busca em árvore do algoritmo A* é ótima caso a heurística utilizada seja admissível. Já a sua versão de busca em grafo é ótima se a heurística utilizada é consistente. Temos ainda que toda heurística consistente é admissível.

Para a distância *Manhattan*, vemos que é fácil provar sua inconsistência, já que:

- **Andando na horizontal ou vertical:** nesse caso, o custo da ação é 1 e o valor da heurística é reduzido ou aumentado em 1 unidade. Logo:

$$\begin{aligned} c(n, a, n') &= 1 \\ h(n') &= h(n) \pm 1 \\ &\text{Assim:} \\ h(n) \leq 1 + (h(n) + 1) &\Rightarrow h(n) \leq h(n) + 2 \text{ (Ok!)} \\ h(n) \leq 1 + (h(n) - 1) &\Rightarrow h(n) \leq h(n) \text{ (Ok!)} \end{aligned}$$

- **Andando nas diagonais:** nesse caso, o custo da ação é 1.5 e o valor da heurística é alterado em ± 2 ou 0. Logo:

$$c(n, a, n') = 1.5$$
$$h(n') = h(n) \pm 2 \text{ ou } h(n') = h(n)$$
Assim:
$$h(n) \leq 1.5 + h(n) \text{ (Ok!)}$$
$$h(n) \leq 1.5 + h(n) + 2 \Rightarrow h(n) \leq 3.5 + h(n) \text{ (Ok!)}$$
$$h(n) \leq 1.5 + h(n) - 2 \Rightarrow 0 \leq -0.5 \text{ Impossível!}$$

Logo, o algoritmo A* não é ótimo quando utilizando a heurística de distância de *Manhattan* em mapas 2D que permitem movimentos diagonais com custo 1.5.

Analisemos agora a heurística de distância *Octile*.

- **Andando na horizontal ou vertical:** o custo da ação é 1 e o valor da heurística é alterado em ± 0.5 ou ± 1. Logo:

$$c(n, a, n') = 1$$
$$h(n') = h(n) \pm 0.5 \text{ ou } h(n') = h(n) \pm 1$$
Assim:
$$h(n) \leq 1 + h(n) + 0.5 \Rightarrow h(n) \leq h(n) + 1.5 \text{ (Ok!)}$$
$$h(n) \leq 1 + h(n) - 0.5 \Rightarrow h(n) \leq h(n) + 0.5 \text{ (Ok!)}$$
$$h(n) \leq 1 + h(n) + 1 \Rightarrow h(n) \leq h(n) \leq h(n) + 2 \text{ (Ok!)}$$
$$h(n) \leq 1 + h(n) - 1 \Rightarrow h(n) \leq h(n) \text{ (Ok!)}$$

- **Andando nas diagonais:** o custo da ação é 1.5 e o valor heurística é alterado em ± 1.5 ou ± 0.5. Logo:

$$c(n, a, n') = 1.5$$
$$h(n') = h(n) \pm 0.5 \text{ ou } h(n') = h(n) \pm 1.5$$
Assim:
$$h(n) \leq 1.5 + h(n) + 0.5 \Rightarrow h(n) \leq h(n) + 2 \text{ (Ok!)}$$
$$h(n) \leq 1.5 + h(n) - 0.5 \Rightarrow h(n) \leq h(n) + 1 \text{ (Ok!)}$$
$$h(n) \leq 1.5 + h(n) + 1.5 \Rightarrow h(n) \leq h(n) \leq h(n) + 3 \text{ (Ok!)}$$
$$h(n) \leq 1.5 + h(n) - 1.5 \Rightarrow h(n) \leq h(n) \text{ (Ok!)}$$

Portanto, o algoritmo A* é ótimo quando utiliza a heurística de distância *Octile* em mapas 2D que permitem movimentos diagonais com custo 1.5.

2.5. Entrada, saída e execução

3. Experimentos

4. Considerações Finais

Referências

Russell, S. J. e Norvig, P. (2016). *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,.