Mini-Project #1

José Senart 77199, Miguel Condesso 79160
Instituto Superior Técnico
Artificial Inteligence and Decision Systems

October 2016

Resumo

Neste mini-projecto pretende-se estudar algoritmos de procura em grafo. Para isso implementar-se-á um algoritmo de procura informado,A*, e um não informado, Uniform-Cost (daqui para), aplicados a um problema cedido pelo docente. O projecto estará dividido em duas partes: o algoritmo de procura e as classes referentes ao problema.

1 Representação do Problema

Para a interface entre o algoritmo de procura e a implementação do problema foram definidas duas classes *Problem* e *Node*. Foram também definidas, como auxílio a operações nestas primeiras classes, as classes *Cask*, *Edge* e *Stack*.

1.1 Classe *Problem*

É nesta classe que se armazenam todas as variáveis referentes à definição de um problema, sendo estas os *casks* relevantes para o problema, as *stacks* e as *edges* presentes, além das informações sobre o estado *goal*. É aqui que estão definidas as regras para a expansão de nodos e para o cálculo dos custos associados a cada passo.

1.2 Classe *Node*

Esta classe é dependente do problema. É nela que estão armazenadas as informações necessárias para univocamente definir um estado do problema: o local onde se encontra o carro bem como a disposição actual dos *casks* nas *stacks*. Tanto no estado inicial como no final, o veículo encontra-se na posição 'EXIT'. Inicialmente, as *casks* encontram-se como definido no ficheiro mapa. No estado *goal*, o carro tem o *goal cask* (**GC**), não importando a disposição dos restantes *casks*.

1.3 Classes *Cask*, *Stack* e *Edge*

Classes que materializam os objectos descritos no enunciado, de variáveis cedidas nos ficheiros de mapa.

A classe *Cask* contém o *ID*, o peso e o comprimento de um *cask*. *Stack* cede informação sobre a localização da *stack*, o seu comprimento, e os *casks* que lá estão. Por fim, *Edge* armazena

2 Algoritmo de Procura

Para o algoritmo de procura escolheu-se utilizar o $A^\star$ já que é, por construção, tanto completo como óptimo. Para se efectuar uma procura não informada, simplesmente atribuiu-se à função heurística o valor zero, obtendo-se uma procura dita *Uniform-Cost* (UC), também completa e óptima. O mesmo não poderia ser garantido para outros algoritmos de procura não informada, visto que o custo não garantidamente é uma função estritamente crescente da profundidade: um nodo mais profundo pode ter um custo mais baixo que outro mais superficial, desde que num ramo diferente. As complexidades temporal e espacial são de ordem $O(b^d)$ em que b é o *branching factor* efectivo e d é a profundidade da solução.

Para a implementação precisou-se de uma *Priority Queue*, construindo-se a classe *PriorityQ*.

O funcionamento do algoritmo é semelhante ao explicado no livro da cadeira [1], embora a implementação tenha sido ligeiramente diferente, seguindo-se a descrita em [2]. A partir de um nodo, expandem-se os nodos que correspondem às posições adjacentes à actual e, caso o carro se encontre numa *Stack*, os correspondentes à operação de *load* ou *unload* de um *Cask*, conforme seja possível

Utilizaram-se as listas: *open_list*, *priority queue* de onde se retira o próximo nodo a expandir, *costs*, que na verdade é um dicionário da forma {nodo: custo até nodo}, e *isChild*, também um dicionário, agora da forma {nodo: nodo pai}. A *open_list* é apenas útil na procura. Já os dicionários serão utilizados *a posteriori*, para reconstruir o caminho encontrado.

2.1 Funções Heurísticas

Para a construção da função heurística para um dado nodo, considerou-se um problema relaxado, cujo objectivo consiste na procura do caminho óptimo da posição deste até uma determinada *Stack*. Dividiu-se o problema em duas fases: quando o veículo não tem o **GC** e pretende-se chegar à *Stack* onde este se encontra; quando já o tem e se pretende retornar à posição inicial *EXIT*. Para obter os caminhos óptimos dos possíveis nodos a expandir até estas duas posições, efectuam-se procuras não informadas, preenchendo-se dois dicionários com a forma {nodo: custo até posição}.

Estas procuras são feitas com base numa nova classe, *RelaxProb*, cujo método de expansão de nodos retorna apenas os nodos que correspondem ao deslocamento para uma posição adjacente. Definiu-se uma nova função de custo, que considera apenas o comprimento da *Edge* que liga estes nodos.

Quando se pretende calcular o valor da função heurística para um dado nodo, $h(n)$, em primeiro lugar, identifica-se a fase do processo em que se encontra, verificando-se então se já existe uma entrada no dicionário respectivo correspondente a n. Caso contrário, efectua-se uma procura, para complementar o dicionário.

Se o veículo já tiver o **GC**, $h(n)$ calcula-se através da distância de n à *EXIT*, ponderada pelo peso do *goal Cask*. Caso contrário, $h(n)$ obtém-se através da soma da distância de n até ao *goal Stack*, com a distância do *goal Stack* à *EXIT*, de novo com a mesma ponderação.

Na 2ª função heurística, $h_2(n)$, utilizada, a este $h(n)$, é adicionado o custo de transportar todos os *casks* que, na *goal Stack*, se encontrem sobre o *goal Cask* para a posição mais próxima.

2.1.1 Consistência

A função $h_2(n)$ domina $h(n)$, uma vez que tem um valor sempre igual ou superior ao desta. Assim, caso se prove a admissibilidade ou mesmo a consistência de $h_2(n)$, fica provada a de $h(n)$. Uma função é admissível caso não sobrestime, para nenhum nodo, o custo até ao objectivo. Pode provar-se que uma heurística é admissível caso seja consistente, o que corresponde à condição:

$$h_2(n) \leq c(n, a, n') + h_2(n') \quad (1)$$

Caso no nodo n o veículo já tenha o *goal cask*, a heurística corresponde ao custo até *EXIT*, tendo-se precisamente $h_2(n) = c(n, a, n') + h_2(n')$ nesta situação.

Caso isto ainda não se verifique, $h_2(n)$ tem três parcelas:

- o custo do carro se deslocar até à *goalStack* - corresponde ao caso mínimo, não se tendo em conta a possibilidade de estar a transportar um outro *cask* e de ter de o depositar numa *stack*;
- caso hajam *casks* sobre o *goal cask*, o custo de transportá-los, em simultâneo, até à posição mais próxima e voltar - estimativa grosseira e que, no máximo é igual ao custo mínimo desta operação (se só houver um *cask* e se este puder ser colocado numa *stack* que esteja à distância mínima da *goal stack*);
- por fim, o custo de transportar o *goal cask* da *goal Stack* até *EXIT* - corresponde precisamente ao custo desta operação.

O custo de fazer *load* e *unload* de *casks* não éarn contemplado, pelo que, nestes casos, ter-se-á, $h_2(n) = h_2(n')$. Assim,

2.2 Comparação de Resultados

A comparação dos resultados cedidos pela UC e pela $A^\star$, sendo ambas óptimas e completas, reduz-se à comparação das complexidades temporal e espacial. Para tal recorreu-se ao código em *testes.py* que efectua procuras de todos os *casks* admissíveis em todos os ficheiros mapa disponíveis. Este guarda os tempos e espaços empreendidos, como função da profundidade das soluções. O mapa estudado foi uma variação do ficheiro *s1.dat*, para simplicidade. Fizeram-se os tamanhos dos *casks* ser 1 e adicionaram-se 4 *casks* à *stack* onde estavam os primeiros. Procedeu-se a resolver o problema para cada um dos *casks*. Começou-se por comparar as complexidades espaciais (figura 1). Ajustaram-se os números de nodos gerados com cada procura a expressões do tipo $y = a * b^{cx}$.

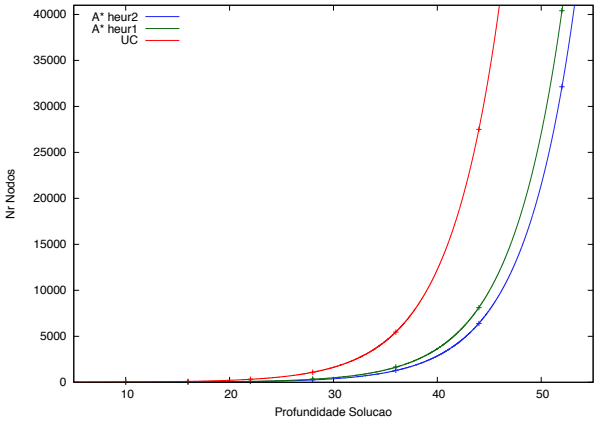

Figura 1: *Comparação das complexidades espaciais dos 3 algoritmos.*

Nota-se uma redução de complexidade espacial da procura UC para a $A^\star$. Isto é consequência de um corte no *branching factor*, que se obteve nos três casos sendo $b = 1.262$ para a UC, $b = 1.231$ para o $A^\star$ com a Heurística 1 e $b = 1.226$ com a Heurística 2.

Um *branching factor* mais pequeno vai-nos dar, em princípio uma complexidade temporal mais baixa. Utilizando o mesmo conjunto de dados pode-se corroborar a anterior afirmação com mais dois ajustes à mesma expressão representados na figura 2.

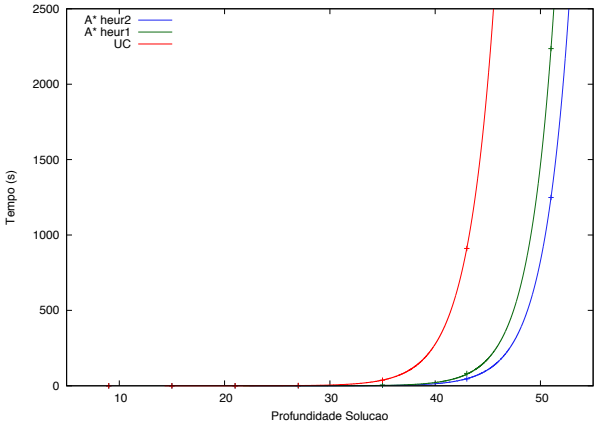

Figura 2: *Comparação das complexidades temporais dos 3 algoritmos.*

Calculam-se outra vez os *branching factors* obtendo $b = 1.17$, $b = 1.16$ e $b = 1.15$ respectivamente. Estes valores são todos aproximados, já que o número de nodos expandidos e o tempo despendido são exponenciais apenas num regime assimptótico. Ainda assim corrobora-se que se tem $b_{UC} > b_{H_1} > b_{H_2}$, concluindo que a implementação de uma heurística aumenta a qualidade do algoritmo de procura.

Referências

[1] Stuart J. Russel and Peter Norvig. *Artificial Inteligence, a Modern Aproach*. PearsonEducation,Inc., Upper Saddle River, New Jersey, 2010.

[2] `http://www.redblobgames.com/pathfinding/a-star/implementation.html#orgheadline8`