Introdução à Inteligência Artificial

Sokoban

1. Introdução

O presente trabalho tem como princial objetivo a aplicação de métodos de procura que promovam soluções para níveis do Sokoban. Para isso, ter-se-á em consideração os três passos fundamentais de Modelação, Implementação e Experimentação.

2. Meta 1 - “Modelação e Implementação”

Na primeira meta foi disponibilizado o “Unity Package”, ferramenta que incluía a implementação do método de pesquisa em largura. Desmo modo, foi possível compreender e testar as opções tomadas na modelação do problema.

Num segundo momento, através do recurso ao teste e validação das implementações, procedemos à implementação dos métodos de pesquisa cega, pesquisa em profundidade limitada e aprofundamento progressivo. No seguimento, procedemos à implementação dos métodos de pesquisa informada, pesquisa sôfrega e A*. Neste caso desenvolvendo uma heurística fornecida pelo enunciado, em que a estimativa do custo de transição de um estado para outro foi o número de caixas que não se encontravam numa posição objetivo. Assim sendo, começamos por analisar a package fornecida e fazer uma modelação do problema.

De seguida criámos novos scripts de Unity para a implementação dos algoritmos de pesquisa pedidos e implmentado a heurística fornecida. Por fim, procedemos aos testes e à validação das implementações dos diferentes algoritmos, através do recurso aos mapas fornecidos.

3. Meta Final - “Meta Refinamento e Experimentação”

O objetivo da meta final foi o desenvolvimento de heurísticas capazes de permitir melhorias de performance em relação aos métodos de pesquisa cega.

Por esse motivo, começamos por pesquisar, lendo artigos e teses densenvolvidas sobre o Sokoban, de maneira a conseguir uma melhor ideia no modo

de desenvolver as heurísticas. Após a primeira fase de pesquisa, editamos os scripts de pesquisa informada, acrescentando novas heurísticas.

4. Modelação do Problema

• O que é um estado?

Neste problema um estado é caracterizado pelas posições das caixas e do jogador no campo. Cada um deles pode estar numa posição legal do campo.

• Qual o estado inicial?

O estado inicial é o estado que contém o jogador e as caixas na sua posição inicial, tal como lida no mapa.

• Como se define um estado final?

O estado final é o estado em que todas as caixas se encontram em posições objetivo.

• Quais são os operadores de mudança de estado?

Os operadores de mudança de estado são todas as ações possíveis, que podem ser aplicadas no estado. No caso do Sokoban, os operadores são possível movimento do jogador para cima, baixo, esquerda ou direita. Existem algumas restrições como, por exemplo, a impossibilidade de atravessar paredes e de sobrepôr caixas.

• Qual a natureza da solução pretendida?

A solução é a série de ações que levam do estado inicial ao estado final. Procura-se uma solução rápida, não necessariamente ótima, mas que resolva o puzzle.

• Qual o custo associado a cada movimento?

O custo de cada acção é um e o jogador pode apenas realizar uma ação de cada vez.

5. Implementação dos algoritmos

5.1 Pesquisa em Profundidade Limitada

Este algoritmo é muito semelhante ao algoritmo de pesquisa em profundidade, mas resolve um dos principais do Depth First Search (DFS), que é a sua incapacidade de lidar com caminhos infinitos. Tal como o DFS, o Limited Depth Search, partindo da raiz, vai expandindo um nó e escolhe um dos seus sucessores que, por sua vez, se expande, - continuando o processo até que encontre a solução ou, no caso de profundidade limitada, até que chegue ao limite máximo de profundidade fixado. Neste último caso, continua o processo com um irmão do último nó analisado, caso exista, ou volta ao nível anterior para continuar o processo.

Para fazer a implementação deste algoritmo baseamos o algoritmo no Breadth First Search fornecido no “Unity Package”. As alterações feitas implicam guardar os nós numa pilha em vez de numa fila. A diferença mais importante nesta alteração de estrutura de dados é que ao inserir numa fila inserimos os nós no fim da fila, enquanto que numa pilha inserimos os nós na posição inicial. Outra alteração, e a mais óbvia, é que antes de adicionar o nó ao grupo de nós visitados, verifica-se se não excede o limite máximo de procura.

Uma nota a fazer em relação a esta implementação é o uso de um HashMap closedSet que guarda os nós visitados. Ao testar o algoritmo deparamo-nos com alguns problemas. No mapa 01, ao estabelecermos o limite máximo como cinco, encontra-se uma solução, mas se o limite for seis já não se encontra.

5.2 Aprofundamento Progressivo

Baseado também nos algoritmos de pesquisa em profundidade e em profundidade limitada, este algoritmo tenta resolver a questão de não se saber

antecipadamente o valor do limite máximo com que se pode encontrar a solução. Como tal, alteramos o algoritmo de maneira a fazer variar o limite de zero a infinito. Assim, o algoritmo consiste na chamada repetida do algoritmo de procura limitada para valores crescentes do limite máximo.

5.3 Pesquisa Sôfrega

Na procura sôfrega o princípio consiste em escolher o nó, na fronteira da árvore de procura, que aparenta ser o mais promissor de acordo com o valor estimado por **h(n)**. Assim, o algoritmo limita-se a manter a fronteira da árvore ordenada pelos valores da heurística, escolhendo sempre o nó com o valor mais baixo, isto é, o que está supostamente mais perto da solução.

Para a implementação deste algoritmo pensamos primeiro em utilizar uma SortedList, mas surgiam problemas na implementação, pois não há a possibilidade de ter chaves duplicadas. Por esse motivo, decidimos tentar uma PriorityQueue, uma fila ordenada. Utilizamos uma implementação de PriorityQueue encontrada online, que trouxe bons resultados. A priority queue recebe os nós da árvore e o valor para ordenar, neste caso **h(n)**, e procede ao ordenamento.

5.4 A*

O algoritmo A* junta as ideias do algoritmo de Pesquisa Sôfrega, mencionado em cima, e do algoritmo de Pesquisa Custo Uniforme, no qual utilizamos o custo para chegar do nó inicial ao nó corrente **n** - dado por uma função **g(n)**, para fazer as nossas opções. O A* tenta escolher a cada instante o melhor caminho passando pelo nó **n**, utilizando para tal a função **f(n) = g(n) + h(n).**

A sua implementação difere da Pesquisa Sôfrega unicamente no valor com que os nós são adicionados à PriorityQueue. Enquanto que na Pesquisa Sôfrega os nós são ordenados por **h(n)**, em A* são ordenados por **f(n).**

5.5 Deadlocks

Um aspeto importante na otimização da resolução de níveis Sokoban é a deteção de deadlocks, isto é, estados em que é impossível encontrar uma solução.

Para tentar otimizar as nossas implementações, tentamos fazer a deteção de um tipo simples de deadlocks, que ocorre sempre que uma caixa vai para um canto do mapa. Para tal implementamos dois métodos, um que corre antes do algoritmo e que faz a deteção, marcando as posições dos cantos num HashMap, e outro que corre de lado com o algoritmo que verifica se os estados sucessores são um estado de deadlock. Infelizmente, a implementação do segundo método não apresentou os resultados esperados e, como tal, decidimos focar-nos noutros aspetos do projeto.

6. Heurística

6.1. O que é heurística?

A procura heurística, ou pesquisa informada, é uma procura que utiliza informação e conhecimento sobre o problema para fazer uma pesquisa mais eficiente e eficaz. A heurística é algo que procura estimar o custo do caminho do nó corrente até ao nó solução.

6.2. Heurística Inicial

A primeira heurística adicionada foi a do enunciado. Nesta heurística deve-se admitir que a estimativa do custo de transição de um estado **s** até ao estado final é igual ao número de caixas que não estão numa posição destino.

Implementamos a heurística fazendo uso da função disponibilizada IsGoal, que verifica se o número de posições objetivo ocupadas é igual ao número total de posições objetivo. Adaptamos o método para devolver o número de posições objetivo a preencher, como é pedido na heurística.

6.3. Heurística B

Como primeira abordagem ao desenvolvimento de uma heurística decidimos implementar uma que fosse relativamente simples e mais fiável do que a inicialmente dada. Como tal, decidimos criar uma heurística simples que calculasse a distância Euclidiana entre cada caixa e a posição objetivo mais próxima. Esta heurística cria

uma estimativa mais real do custo para chegar ao estado final mas ainda não tem em conta o movimento do jogador.

A implementação é muito simples. Apenas é necessário iterar cada posição objetivo por cada caixa e calcular a distância mínima através do método Distance da classe Vector2- No final, soma-se ao resultado que é retornado.

6.4. Heurística C

A segunda heurística implementada calcula a distância de Manhattan entre o jogador e a caixa mais próxima dele, mas também entre as caixas e as posições objetivo. A distância de Manhattan é a distância entre dois pontos medida ao longo dos eixos, como numa grelha. Esta heurística pretende arranjar uma estimativa aproximada à distância necessária para resolver o puzzle, tendo em conta tanto os passos necessários para preencher as posições objetivo, como também a deslocação do player para a caixa mais próxima. É uma estimativa mais real relativamente à Heurística B e a Heurística Inicial, mas que sobrestima a distância necessária.

A implementação desta heurística é simples. Para cada caixa no mapa iteramos pelas posições objetivo e adicionamos ao valor da heurística a distância a cada caixa. De seguida, calculamos a distância do jogador à caixa e verificamos se é a distância mínima. No fim devolvemos a soma de todas as distâncias das caixas às posições objetivo com a menor distância do jogador a uma caixa.

6.5. Heurística D

A heurística D é uma variante da heurística anterior mas é mais real. Em vez de somar a distância de todas as caixas até todas as posições objetivo, soma-se a distância de todas as caixas à posição objetivo mais próxima. Apesar de não ser necessariamente correta a assunção de que a caixa mais próxima é a certa, esta

heurística permite devolver um resultado mais aproximado do custo real para chegar ao estado final.

A implementação é simples e é praticamente igual à da Heurística C, mas acrescenta uma pequena verificação para saber qual a posição objetivo mais próxima da caixa.

6.6. Heurística Zero

Para observarmos e testarmos o comportamento, desenvolvemos uma heurística que devolve sempre zero.

6.7. Admissibilidade

Uma heurística é admissível se nunca sobrestimar o custo real do caminho que passa por **n**, isto é, **h(n) <= h_real(n)**. Como tal, a Heurística Inicial, a B e a D são admissíveis. A estimativa que fazem do custo é sempre igual ou superior ao custo real, por oposição à Heurística D que o sobrestima.

7. Testes e Experimentação

Nesta fase, foi pedido que se testasse os algoritmos implementados e as heurísticas desenvolvidas, considerando mapas de diferentes graus de complexidade e analisando os algoritmos em diferentes aspetos: Sucesso, Discriminação, Tempo e Memória. Para isso, encontramos alguns mapas usados para teste e fizemos uma seleção deles.

Corremos todos os algoritmos com todas as diferentes heurísticas, quando aplicáveis, para todos os mapas - de maneira a podermos verificar e comparar a sua performance. Podemos comparar os resultados entre eles e fazer uma avaliação tanto das vantagens de alguns algoritmos, como também comparar as diferentes heurísticas e, ainda, ver o impacto que têm nos algoritmos implementados.

É evidente a vantagem de algumas das heurísticas implementadas sobre outras e sobre os métodos de pesquisa cega. Também podemos perceber pelos resultados que os algoritmos de pesquisa informada, quando usados com a Heurística Zero, têm um comportamento igual ao da Pesquisa em Largura.

Algo a notar também foi a incapacidade dos testes resolverem o mapa 04 que tinha dimensões muito elevadas comparativamente aos outros mapas.

Os melhores resultados foram obtidos combinando a Heurística D com o algoritmo de pesquisa informada, Pesquisa Sôfrega.

7.1. Breadth First Search

O estudo teórico deste algoritmo indica que ele apresenta complexidade temporal e espacial exponencial. Isto indica-nos que para problemas que gerem poucos nós, e com um factor de ramificação baixo, é eficiente. Contudo, para problemas mais complexos já não é ideal.

Como podemos perceber pelos testes realizados, de facto, quando a complexidade dos problemas é muito simples, como por exemplo no mapa 01, o algoritmo tem um comportamento bastante bom, mas à medida que a complexidade aumenta o seu uso começa a ser impraticável, tanto em questões temporais como espaciais.

7.2. Pesquisa em Profundidade Limitada

A pesquisa em profundidade limitada teoricamente apresenta também uma complexidade temporal exponencial, mas ao contrário da pesquisa em largura, a sua complexidade espacial é linear com o limite máximo.

É fácil comprovar através dos resultados dos testes que os valores obtidos vão de acordo com o estudo teórico da complexidade. Dos algoritmos de pesquisa cega foi o que obteve os melhores resultados, mas existe o problema de não sabermos o limite a aplicar.

7.3. Aprofundamento Progressivo

A complexidade espacial deste algoritmo é a da procura em profundidade, ou seja, linear. No entanto, há uma sobrecarga temporal motivada pelo facto de vários nós serem analisados mais do que uma vez. Com o estudo teórico podemos ver que a sobrecarga temporal para valores elevados é mais favorável, no entanto a complexidade mantém-se exponencial.

Este algoritmo foi o que apresentou os piores resultados de todos os algoritmos implementados e testados. É útil para problemas de menor complexidade, em que é desconhecido o limite, mas em que o custo espacial e temporal fica demasiado grande à medida que a complexidade aumenta.

7.4. Pesquisa Sôfrega

Pelo estudo deste algoritmo percebemos que os nós vão sendo expandidos num modo híbrido entre uma procura em profundidade e em largura. Assim, no pior caso, todos os nós têm de ser expandidos e visitados - o que significa uma complexidade temporal exponencial. Como a fronteira tem de ser mantida toda em memória temos também no limite uma complexidade espacial exponencial.

A seleção da função heurística é de extrema importância para analisar a performance deste e de outros algoritmos de pesquisa informada. Como podemos verificar pelos testes efetuados com diferentes heurísticas, este algoritmo tanto pode apresentar resultados positivos como negativos. No último caso, o algoritmo comporta-se do mesmo modo que o algoritmo de pesquisa em largura, quando usa a heurística que devolve sempre zero. Por outro lado, o resultado é positivo quando se utiliza a Heurística D, verificando-se, neste caso, os melhores resultados de entre todos os algoritmos. O algoritmo não é discriminador, logo pode encontrar uma solução de forma mais rápida e menos custosa do que os outros algoritmos, mas não garante que seja a solução mais económica.

7.6. A*

O estudo deste algoritmo diz-nos que a sua complexidade depende em parte da qualidade da função heurística, mas no geral o número de nós nas fronteira definida pelo valor de **f** cresce exponencialmente e, como todos esses nós são conservados em memória, a sua complexidade espacial é também exponencial.

Como referido no algoritmo anterior a heurística é de principal importância na análise deste algoritmo. Podemos verificar pelos testes efetuados que, tal como o algoritmo de pesquisa sôfrega, apresenta resultados iguais ao algoritmo de pesquisa em largura quando a função heurística devolve sempre zero. Contudo, ao contrário do algoritmo de pesquisa sôfrega, o algoritmo A* é um algoritmo completo e ótimo, isto é,

discrimina todas as soluções, encontrando a melhor. Como tal, este algoritmo tem um custo, tanto temporal como espacial, mais elevado que a Pesquisa Sôfrega, mas apresenta sempre a solução mais económica.

8. Conclusão

As conclusões que tiramos depois da realização deste trabalho são relacionadas com as forças e as fraquezas inerentes às diferentes abordagens ao problema e também com o desenvolvimento de heurísticas.

Com este trabalho conseguimos compreender as vantagens e desvantagens dos diferentes algoritmos e a importância de uma função heurística bem desenvolvida para conseguir atingir os melhores resultados.

Chegamos à conclusão que simular inteligência humana é muito difícil, o que é fácil de compreender ao analisar os nossos resultados e no decorrer da execução do projeto.

João Craveiro	**2013136429**
João Faria	**2013136446**
João Vieira	**2013136370**