A5: Pesquisa aplicada à resolução do jogo Pukoban

Relatório Final

Inteligência Artificial

3º ano do Mestrado Integrado em Engenharia Informática e Computação

Elementos do Grupo:

Bruno Miguel Faustino Moreno – up201504781 – up201504781@fe.up.pt

Francisco Teixeira Lopes – ei11056 – ei11056@fe.up.pt

20 de Maio de 2018

Objectivo

O objectivo do trabalho incide sobre a resolução do jogo Pukoban, usando métodos de pesquisa. Para esse efeito será usado A* com algumas heurísticas diferentes, bem como, pesquisa de custo uniforme e pesquisa gulosa. O objetivo final sendo a comparação das performances dos vários algoritmos.

Descrição

O jogo Pukoban desenrola-se num mapa com objectos e movimentos em grelha. O objectivo é deslocar um conjunto de caixas de forma a ficarem sobre os campos marcados como destino, para esta deslocação, é possível tanto empurrar como puxar as caixas.

Como o jogo consiste num espaço em grelha, cada estado é representado pela posição dos vários elementos dinâmicos. Sendo que, os elementos estáticos, como paredes e formato do nível, não são guardados directamente nos estados mas entram na geração dos estados possíveis.

Sendo o principal algoritmo a implementar o A*, a função de transição será:

$$f*(n) = g(n) + h*(n)$$

Onde g(n) é o custo até ao estado actual e h*(n) é o custo estimado para chegar à solução a partir do estado actual. A heurística a implementar baseia-se na distância de cada objectivo à caixa mais próxima, na verdade, é o somatório das distâncias de cada objectivo à caixa mais próxima. A distância é calculada como sendo a distância na grelha actual, contando com obstáculos, e não apenas como sendo uma distância em linha recta. Além disso, é ainda considerado no cálculo da distância, o número de vezes que a caixa tem de mudar de direcção em 90 graus, pois isto implica o jogador ter de fazer no mínimo 2 jogadas.

$$\sum_{i=0}^{n} d(objectivo_i \text{ à } caixa\ mais\ próxima) + 2 * nCurvasNecessárias$$

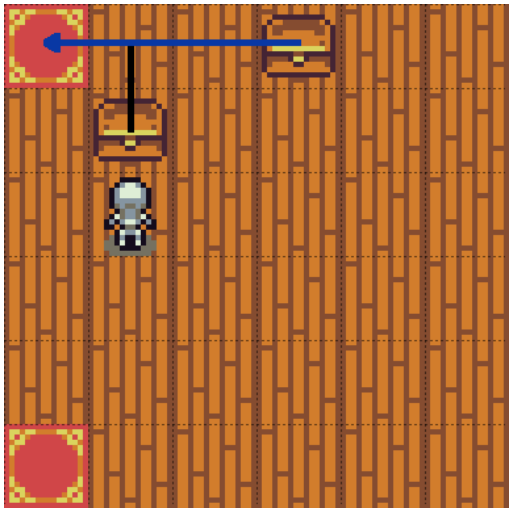

Figura 1 - exemplo estado de jogo com arte placeholder

A figura acima exemplifica a heurística para um objectivo específico, a rota a azul teria um peso de 3 já que a caixa não tem de mudar de direcção, por outro lado, a rota a preto teria um peso de 2 na distância, mas acrescia um peso de 2 pela necessidade de mudar uma vez de direcção, para um total de 4. A caixa mais afastada seria inicialmente favorecida devido a isto (peso de 3 vs peso de 4). Na verdade, estando o jogador mais perto da caixa com um peso superior, faria sentido para um jogador humano começar por essa caixa, contudo, para evitar o risco de sobreestimativa da heurística, não é considerada a posição do jogador em relação às caixas. Neste caso, seria a caixa mais próxima a jogada ideal mas existe uma infinidade de cenários em que a caixa mais próxima do jogador e do objectivo não é a jogada correta.

O algoritmo de pesquisa a utilizar será o A* com a função de transição e heurística enunciada acima. Os estados serão gerados baseando-se nas jogadas possíveis a partir do nó atual, verificando nós duplicados e considerando-os inválidos. Para um nível muito simples de Pukoban, o algoritmo comportaria-se da seguinte forma:

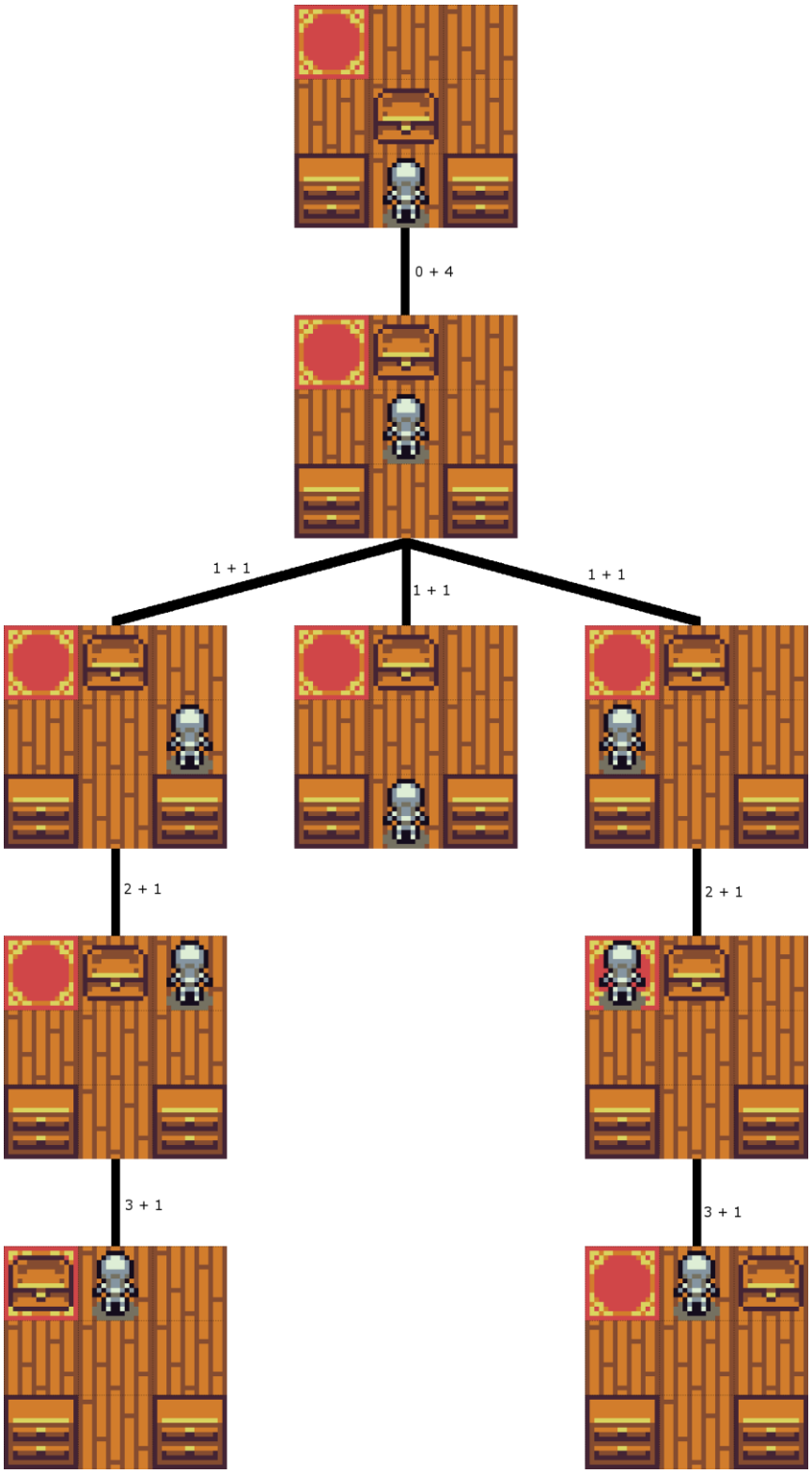

Figura 2 - aplicação do algoritmo

Imaginando que o algoritmo decidia prosseguir primeiro pelo ramo da direita, visto ambos os ramos terem o mesmo custo, chegaria a um estado em que a próxima jogada teria um custo heurístico de 2, pois a caixa afastou-se do objectivo, enquanto que no ramo esquerdo continuaria a 1, levando à eventual solução do nível. A heurística acaba por favorecer estados que aproximem as caixas do objectivo, sendo que, só expande os outros estados se estes primeiros não levarem a uma solução.

Desenvolvimento

Ferramentas/APIs utilizadas

A linguagem de programação utilizada foi Java em conjunto com a biblioteca LibGdx, a qual, facilita o desenvolvimento de jogos. O ambiente de desenvolvimento foi o IntelliJ IDEA pela facilidade em importar projetos que utilizam o sistema de gestão de dependências gradle.

Estrutura da Aplicação / Detalhes de implementação

A aplicação começa na classe AStar, responsável por correr a visualização no ecrã e por processar inputs do utilizador. Quando um mapa é carregado, a classe TiledHandler é responsável por encontrar os vários tipos de objetos desse mapa (paredes, jogador, objetivos, caixas). Depois de um mapa estar carregado, o utilizador pode correr um algoritmo à escolha, o que inicia a classe AStarAlgo responsável por correr o algoritmo A*. Dependendo do input do utilizador a função de custo varia para simular pesquisa de custo uniforme e pesquisa gulosa. Para guardar os estados possíveis é utilizada a classe MyVertex, a qual guarda a localização das caixas e do jogador para cada estado de jogo possível na pesquisa. Para uma análise mais detalhada, a entrega é acompanhada da respetiva documentação Javadoc.

```
// A* lists
private LinkedList<MyVertex> openList = new LinkedList<>();
private HashSet<MyVertex> closedList = new HashSet<>();
```

*Figura 3 - estruturas usadas para o algoritmo A**

A implementação do algoritmo A* é feita com uma LinkedList para a lista aberta e um HashSet para a lista fechada. A classe MyVertex faz override dos métodos necessários para o Java automaticamente impedir duplicados, o que faz com que a lista fechada apenas contenha estados únicos quando um elemento é inserido. A lista aberta seria melhor implementada com uma Fibonacci heap.

Experiências

A experiência realizada trata-se da comparação de tempos de execução para os variados métodos de pesquisa implementados nos diversos mapas. Para efeito de legenda, h1 significa a utilização da distância em grelha contando com obstáculos, h2 significa a adição das mudanças de direção necessárias no cálculo da heurística, h3 significa que as mudanças de direção adicionam um custo de 2 por mudança, que é o número mínimo de jogadas para mudar de direção. Além de A* utilizou-se uma função de custo que utiliza só g() e só h(), nos resultados denominam-se PCU (pesquisa custo uniforme) e PG (pesquisa gulosa) respetivamente.

Mapa	h1	h2	h3	PCU	PG
Level1	0.230s	0.380s	0.219s	0.150s	0.171
Level2	0.579s	0.862s	0.819s	0.545s	0.579s
Level3	0.500s	0.858s	0.864s	0.475s	0.500s
Level4	0.592s	1.053s	1.054s	0.565s	0.600s
Level5	0.565s	0.942s	0.930s	0.53s	0.541s
Level6	68.588s	76.905s	75.908s	62.39s	71.827s

Resultados obtidos num Intel Core i7-7700K @ 4.20GHz

Conclusões

Para alguma surpresa do grupo, o algoritmo A* não produziu os melhores resultados, mas sim, o algoritmo de pesquisa de custo uniforme. Tal, muito provavelmente, deve-se ao custo de computar h() para cada nó adicionado à lista aberta e também ao facto de não se usar a melhor estrutura para guardar a lista, tendo performance O(N) quando uma Fibonacci heap teria O(log N). Além disso, a consideração das mudanças de direção das caixas não melhorou o tempo de execução nos níveis utilizados. Num utro teste reparou-se que em mapas mais abertos essa heurística de consideração de mudança de direção era mais rápida.

Melhoramentos

Como mencionado na conclusão, o custo de computar h() parece ser uma das operações que torna o tempo de execução superior que o tempo de usar apenas g(). Para acelerar o processo, o mapa poderia conter informação pré computada do h(), o sistema de mapas usado para o trabalho é facílmente extensível para esse efeito e a heurística utilizada é propícia para o uso de informação pré computada. A implementação de pesquisa em largura e pesquisa em profundidade seria uma boa adição à comparação de resultados. Além disso, também como mencionado noutras secções, poderia-se ter implementado uma Fibonacci heap para a lista aberta.

Recursos

https://www.redblobgames.com/pathfinding/a-star/introduction.html (consultado em 19/05/2018)

IntelliJ IDEA / LibGDX

Bruno Miguel Faustino Moreno – 50%

Francisco Teixeira Lopes – 50%

Apêndice

Instalação

Instalar JDK 8+

Instalar Intellij IDEA 2018.1.3 (Community)

Para importar o projeto, no programa Intellij IDEA: File -> Open -> build.gradle -> clicar OK

Ctrl + Shift + Alt + S para fazer o setup do JDK a utilizar

Clique direito do rato sobre a classe DesktopLauncher e escolher “Run”, não vai funcionar mas cria a configuração por defeito

No menu de configurações, editar a configuração e escolher o working directory necessário para a configuração DesktopLauncher (“core/assets”)

Utilização

Seta para cima	- incrementa o nível (1 a 10)
Seta para baixo	- decrementa o nível (1 a 10)
Q	- corre o algoritmo A* com heurística de distância em grelha e consideração de obstáculos
W	- corre o algoritmo A* com a adição do número de mudanças de direção das caixas à heurística
E	- corre o algoritmo A* com as mudanças de direção das caixas a valer 2
R	- corre o algoritmo de pesquisa de custo uniforme
T	- corre o algoritmo de pesquisa gulosa
+	- aumenta a velocidade de apresentação da solução (pode ser necessário usar o + do numpad)
-	- diminuí a velocidade de apresentação da solução