Desenvolvimento de um Agente autónomo para o jogo *Sokoban*

Preparado por: Francisca Barros (n 93102),
José Sousa (n 93019),
Margarida Martins (n 93169)

Disciplina: Inteligência Artificial

Docentes: Luís Seabra Lopes,
Diogo Gomes

Aveiro, 11 de dezembro de 2020

universidade de aveiro

Pesquisa

De forma a descobrir a solução para um nível é feita uma **pesquisa da posição das caixas até aos goals**, dentro dessa pesquisa, é utilizada outra árvore de pesquisa de forma a calcular os movimentos do **keeper até uma dada caixa**.

KeeperTree - Árvore de pesquisa (pesquisa um caminho da posição atual do keeper até a um dado target). Devolve as respetivas keys caso encontre solução.

KeeperNode - Descreve os nós constituintes da árvore de pesquisa relativa ao keeper. Cada nó contém o seu pai , a coordenada do keeper, uma string com as keys para o movimento e a heuristica.

Estado - Consiste no move do keeper e no último movimento que o keeper fez. Permite eliminar muitas iterações na pesquisa.

Tipo Pesquisa - Depende do argumento 'strat' passado ao método de search. Pode ser breath-first ou A*.

Actions - Calculadas a partir de uma biblioteca criada por nós (explicado mais à frente).

Sokoban Tree - Árvore de pesquisa das caixas até aos goals. . Calcula as ações para cada caixa e chama a pesquisa do keeper para que se calcule os moves do keeper até à dada caixa.

Node - Descreve os nós constituintes da árvore principal de pesquisa. Contém as posições das caixas, o seu pai, os moves acumulados até agora, a posição do keeper e a heurística.

Estado - Posição das caixas. Apenas muda se uma caixa mudar de posição.

Tipo Pesquisa - Greedy

Actions - Calculadas a partir de uma biblioteca criada por nós (explicado mais à frente).

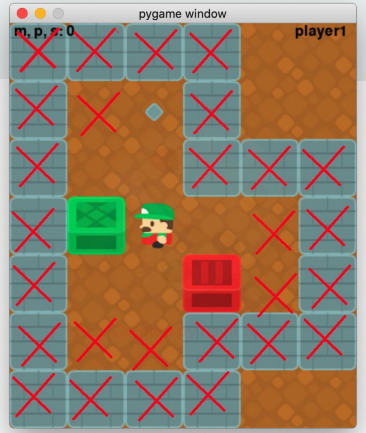
Util

- Biblioteca que ajuda à resolução de cada nível, com o uso de POO e DP.
- Criada para não tornar os ficheiros demasiado extensos. Mais fácil de perceber o código.
- Funcionalidades
  - Calcula Darklist - *i.e* simple deadlocks + paredes
  - Cálculo da heuristica
  - Assignment das caixas aos goals
  - Verificar se o nivel está completo
  - Cálculo das ações para cada nó da pesquisa do keeper até às caixas
  - Cálculo das ações para cada nó da pesquisa das caixas
  - Cálculo dos *freeze deadlocks*
  - Outros métodos utilizados internamente (ex. *get_tile* - qual o tipo de Tile na posição em argumento)
- *SokobanTree* e *KeeperTree* têm como atributo um objeto *Util*
  - Métodos invocados quando necessário (para auxílio da pesquisa)

- A *DarkList* torna-se um elemento fundamental para tornar a pesquisa mais eficiente
  - lista de todas as posições bloqueadas no nível (Paredes ou cantos)

- O cálculo dos *deadlocks* também é de extrema importância para possibilitar a resolução dos problemas propostos.

Deadlocks simple & freeze



Dark List Visualmente

No início de cada nível são calculadas as posições em que nunca queremos ter uma caixa - dark list - que consiste nas paredes e cantos. Estas posições 'más' obtidas são os **simple deadlocks + paredes**.

As actions das caixas são calculadas na biblioteca Util. O método **'possible_moves'** verifica para as quatro coordenadas circundantes se:

- A coordenada não pertence à dark list
- A coordenada não é uma BOX
- A coordenada oposta está livre (para o keeper poder fazer o push)
- A coordenada não foi já marcada como deadlock
- Caso este último não se verifique é efetuada a detecção de freeze deadlock.

Freeze deadlock (abaixo tem o caso na **horizontal**, o processo é o mesmo para a vertical):

- Tem parede do lado direito e esquerdo da caixa - caixa bloqueada no eixo do **x**.
- Tem um simple deadlock do lado direito e esquerdo da caixa - caixa bloqueada no eixo do **x**.
- Tem uma caixa do lado direito e esquerdo (uma destas tem de estar bloqueada)- caixa bloqueada no eixo do **x**.

Evitar **circular checks nos freeze deadlocks** (andar sempre a ver as mesmas posições):

As boxes que já foram verificadas são tratadas como posições inválidas e assim só precisamos de, na próxima caixa a ser verificada, ver o eixo oposto ao qual o freeze está de momento. Fazemos isto passando como argumento ao método se queremos que o check seja feito apenas horizontal ou vertical ou ambos.

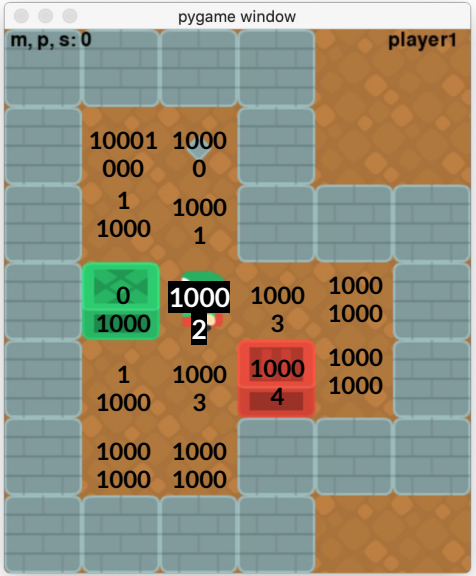
Heurística / Assignment

Heuristica para a pesquisa das caixas:

No início de cada nível, para cada goal é calculado o número de movimentos que seriam precisos para uma caixa em determinada posição chegar à posição do goal estando o mapa apenas com as paredes. Quando a partir de uma posição não se consegue chegar a um goal dá-se o valor de 1000 movimentos. Esta informação é guardada no dicionário ***distanceToGoal.***

Assignment para a pesquisa das caixas:

Na função para o cálculo da heurística de uma dada posição de caixas, ***heuristic_boxes***, o assignment caixa-goal é feito de uma forma "greedy", numa primeira fase é criada uma lista com todos os pares caixa-goal ordenada pela heurística (o valor correspondente no dicionário ***distanceToGoal***). Depois é atribuído um goal distinto a cada caixa por ordem crescente de heurística. No final, às caixas não atribuídas (devido ao facto de nalguns casos não ser possível a partir de uma posição chegar a um certo goal), é atribuído o goal mais próximo. A soma das distancia caixa - goal atribuído é a heurística.



Informação guardada no dicionário ***distanceToGoal***, para no nivel 1 do Sockoban

Heurística para a pesquisa do keeper

A heurística na pesquisa do keeper é a distância de Manhattan entre a posição atual e a pretendida Manhattan. Esta distância é calculada na função ***heuristic.***

Prunning/Backtracking

Na pesquisa das caixas:

O dicionário ***used_states***, é usado para guardar os estados das caixas (tuple com as atuais coordenadas de todas as caixas, que é a ***key*** do dicionário), e as respetivas posições do keeper (***set*** de coordenadas do keeper) ao longo da pesquisa.
Para cada movimento de caixa que o keeper consegue efetuar é verificado se o estado já existe no dicionário. Em caso afirmativo o novo nó **não** será expandido se o keeper a partir da sua posição atual conseguir aceder a pelo menos uma das posições do keeper armazenadas nesse estado.Desta forma reduz-se significamente o número de nós expandidos e consequentemente a pesquisa fica substancialmente mais rápida.
Inicialmente o ***used_states*** era uma lista onde o estado guardado consistia nas coordenadas de todas as caixas mais a posição do keeper. Desta forma o nivel 68 apresentava uma solução com perto de 5000 movimentos do keeper, passando após a otimização em cima a apresentar uma solução com menos de 300 movimentos.

Na pesquisa do keeper:

O set ***used_states_k*** é usado para guardar os estados do movimento do keeper (tuple com **posição atual do keeper** e **do último *move*** que ele executou). Para cada **action do keeper**, é verificado se ainda não foi usado este estado e, dependendo da ***estratégia*** de pesquisa definida, a lista de nodes abertos é expandida. Isto reduz o número de nós expandidos, o que leva a uma solução mais rápida e eficaz.
Inicialmente fazíamos apenas a verificação do **get_path** que apenas via se o keeper já tinha atingido o seu objetivo. Reparámos que o ciclo que percorre as actions estava a ser iterado demasiadas vezes (às vezes 40,000 vezes). Ao adicionarmos a **posição atual do keeper** e **último move do keeper** como uma forma de backtracking conseguimos restringir mais a procura e tornar o código muito mais eficiente.

Referências

https://baldur.iti.kit.edu/theses/SokobanPortfolio.pdf

https://docs.python.org/3/library/asyncio.html

http://sokobano.de/wiki/index.php?title=Main_Page

Módulos fornecidos nas aulas práticas de pesquisa.

Discussão do trabalho com outros colegas:

Rui Fernandes (92952)

Pedro Bastos (93150) e Eduardo Santos (93107)