SOKOBAN

Maria Inês Seabra Rocha NMEC: 93320 (LEI)
Pedro Gonçalo da Silva Abreu NMEC: 93240 (MIECT)

IA+IIA Dezembro 2020

DESCRIÇÃO DO ALGORITMO

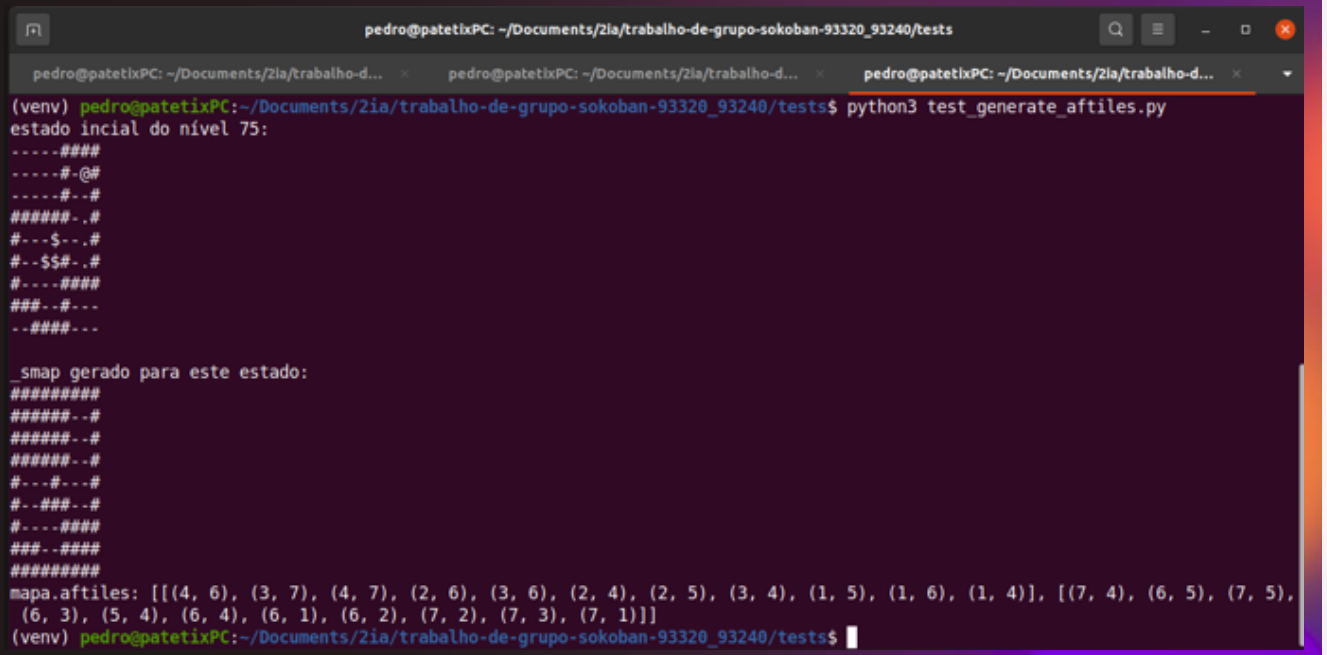
Para este projeto, decidimos procurar primeiro pela sequência de movimentos das caixas que nos leva à resolução de cada nível e, de seguida, gerar o caminho que o *keeper* tem que percorrer para as efetuar, partindo do seu estado inicial. Assim implementámos duas árvores de pesquisa, utilizando o módulo tree_search das aulas práticas.

O módulo push.py implementa a classe que estende SearcDomain. Neste módulo, o *state* do problema é uma instância de SearchPath que guarda um mapa e uma lista com os *pushes* aplicados até aquele momento, e a *action* é um *push,* caraterizado por um tuplo, em que o primeiro elemento são as coordenadas da caixa a empurrar, e o segundo a direção do movimento da caixa.

Na segunda treeSearch (treeSearchKeeper), o *state* é são as coordenadas do keeper na instância de pesquisa e a *action* é a direção em que este se pode movimentar. O módulo path.py implementa a classe que estende o SearchDomain.

Em ambas as árvores optámos pela pesquisa a* com uma inserção ordenada dos novos estados criados nos self.open_nodes, pois consideramos esta pesquisa a mais eficiente na maioria dos casos.

Fases de Desenvolvimento



Inicialmente, implementámos uma searchTree com base no exemplo dos guiões práticos que procurava pela sequência correta de movimentos das caixas , fazendo uso de *brute force* para verificar se o keeper poderia deslocar-se até à tile que lhe permitisse empurrar a caixa. Assim, se existisse um caminho, guardávamos numa stack os mapas já atualizados que teríamos de continuar a explorar até encontrar aquele que correspondesse à solução. No entanto, esta não se revelou uma técnica eficaz, pois uma pesquisa exaustiva dos caminhos intermédios em estados que não serviam para a solução final era muito demorada, permtindo apenas atingir o nível 19.

Posto isto, acrescentámos o atributo _smap (lista que conté m listas de tiles adjacentes entre si - cf. Img.1) para melhorar a pesquisa do caminho do keeper entre os vários movimentos das caixas. Neste atributo, transformámos as caixa em paredes e os goals e o keeper em chão. Através desta representação conseguimos perceber se o keeper pode chegar à posição-destino para empurrar uma caixa, verificando se esta posição-destino deste pertence à lista de adjencent_tiles da posição do keeper no momento. Note-se que o atributo _smap é constantemente atualizado quando movimentamos uma caixa.

Pela imagem vemos que é gerada uma lista com duas listas que correspondem às tiles adjacentes à esquerda e à direita do mapa do conjunto de caixas neste caso for a do sítio que bloqueia a movimentação do keeper, respetivamente.

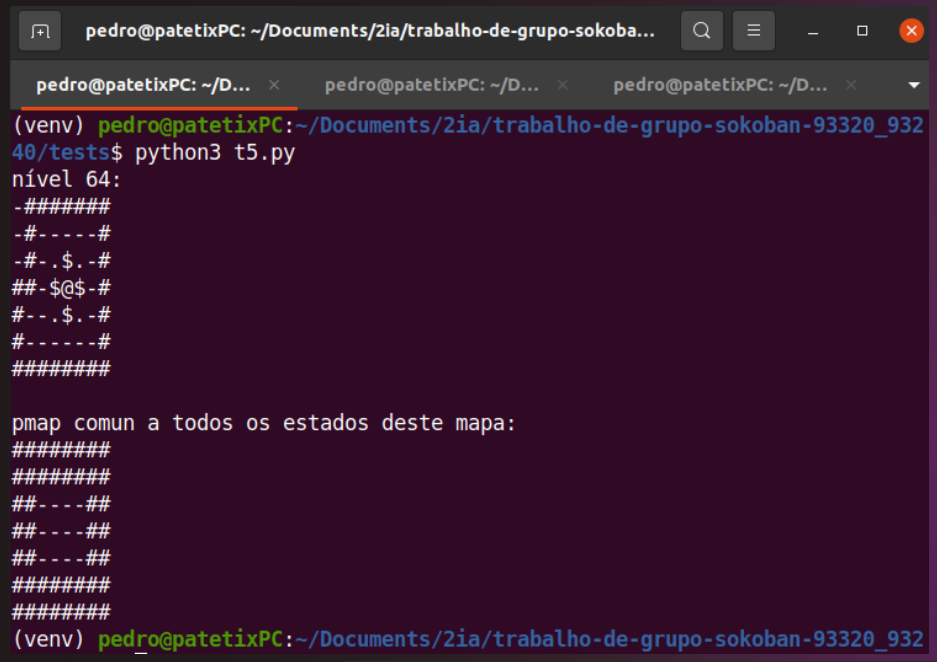
Fases de Desenvolvimento

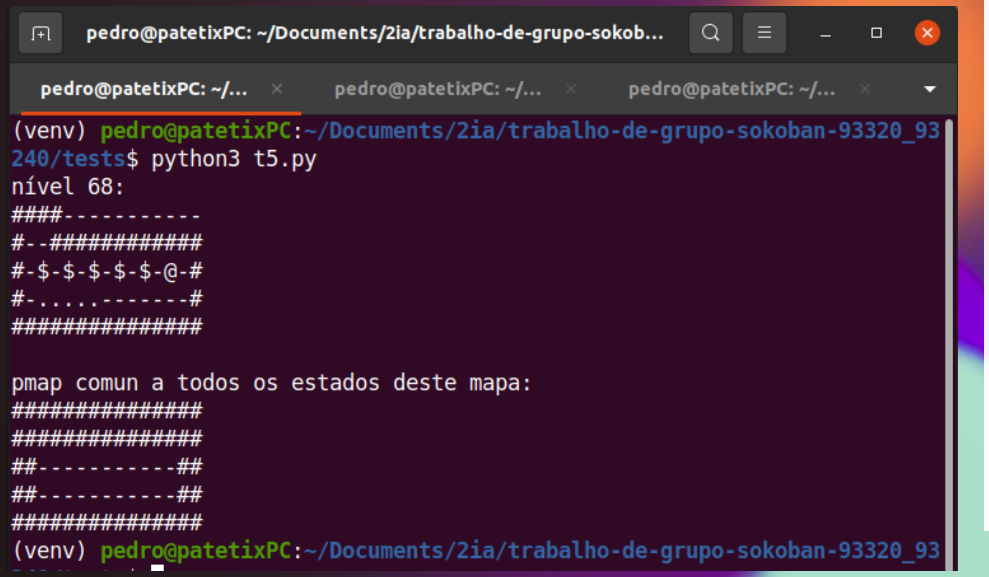
Logo após a implementação do _smap, percebemos que era fundamental ter uma estrutura com todos os estado anteriores, tendo, portanto, escolhido um *set*, denominado self.states e colocado nos construtores das classes push.py e path.py, para evitar estados repetidos.
Em ambas as fases anteriores, tínhamos como heurística em path.py (classe relacionada com a pesquisa dos caminhos do keeper) a hipótenusa entre as posições inicial e final do keeper e em push.py, um algoritmo mais complexo que soma a combinção de menor valor da distância de manhanttan para um par caixa-"emptyGoal". Em termos de custo no primeiro caso era 1 e no segundo era o número de empty_goals.
Nestas condições alcançámos o nível 68.

Numa terceira fase, melhorámos a forma como inseríamos os nós na lista de self.open_nodes na class tree_search, pois a cada nova inserção ordenávamos tudo, incluindo o que já estava ordenado. Para tal, descobrimos que o python tem um módulo bisect que, após redefinição das condições de comparação para ordenar (tendo matido o algoritmo a* -> cost+heuristic), insere os novos nós na posiçao correta.

Melhoramos, por tentativa erro, a heurística em push.py, de modo a aumentar em estados avançados o peso da heuristica face ao custo. Então concluímos que a melhor heurística para nós seria à heuristica anterior somar o número de goals livres e elevar tudo ao quadrado. Para além disso, ajustámos ainda o custo para 1 em push.py.

Fases de Desenvolvimento





Por fim, criámos o atributo _pmap. Este é igual ao mapa inicial, mas são retiradas algumas das posições que levam a deadlocks. Neste mapa são representadas como paredes zonas que permitem diversos movimentos das caixas, sendo que todos eles conduzem a deadlocks e como chão o restante mapa. Sem o _pmap, demorávamos um pouco mais a excluir estas situações e, apesar de só funcionar num nivel mais "exterior", esta alteração teve algum impacto. Podemos ver nas imagens em cima (muito impacto), poucos estados por explorar e em baixo (menor impacto).

Com todas estas alterações estávamos no nível 105 (no servidor do professor) e 112 (no nosso computador).

Para tentar melhorar um pouco mais a solução e ganhar algum tempo para concluir o nivel, fomos analisar o código e reparámos que corriamos a função result() (push.py) duas vezes, uma na função actions e outra na função search() (search_tree.py), o que era desvantajoso. Para contornar esta situação, tivémos que alterar um pouco a searchTree dos "pushes" face à original então separámos a original, que usamos apenas para encontrar os caminhos do keeper e a nova versão que recebe da função actions() já a lista de novos estados.

Com esta melhoria conseguimos chegar ao nível 112 (professor) e 117 (nosso computador).

Conclusão

Concluímos que estamos satisfeitos com os resultados alcançados, ainda que um pouco desiludidos, pois mesmo tendo feito algumas melhorias, no final, acabámos por não chegar ao mesmo número de níveis no nosso computador e no serivdor do professor, por falta de rapidez da nossa solução.
Achamos que houve evolução no projeto e uma constante tentativa de melhorar os algoritmos, no entanto, neste nível mais avançado percebemos que alguns deadlocks terão ficado por eliminar e teríamos, provavelmente, de fazê-lo recursivamente.

Teríamos, possivelmente, de melhorar a função generate_tiles(), tornando-a também recursiva pois, atualmente, ainda tem muitos ciclos.

Como valorização do projecto, destacamos a crição do _smap, o algoritmo que identica as tiles adjacentes num determinado mapa (função generate_aftiles em mapa.py), que serviu para identificar as tiles que o keeper pode alcançar, e a criação do _pmap que contribuí para reduzir o nível de estados criados na pesquisa.

Tentamos dividir o trabalho ao máximo, tendo estado muitas horas em sessões colaborativas no VisualStudio Code a trabalhar no código. Assim sendo, consideramos que a nota deve ser dividida igualmente pelos dois elementos.