“Rush Hour”

TPG - Inteligência Artificial

Rafael Gonçalves (102534)
André Butuc (103530)

universidade de aveiro

Arquitetura do projeto

Numa primeira fase do projeto, entendemos que deveríamos dividir o desenvolvimento do agente inteligente em dois módulos lógicos:

- **tree_search.py** - onde implementámos e testámos os algoritmos de pesquisa em árvore.
- **student.py** - que transmite ao servidor os movimentos do cursor necessários para resolver o puzzle.

tree_search.py

Classes de suporte

Matrix: é um nó da árvore, que guarda um estado do puzzle, sendo 2 dos seus atributos:

- grid: guarda o estado do puzzle. Representa uma matriz (lista 2D), porém é armazenada eficientemente, na forma de string.
- pieces: dicionário que mapeia cada peça do puzzle para a sua posição atual, na forma de um tuplo (minx, maxx, miny, maxy), que representa as suas fronteiras.

MatrixForGreedy: é uma subclasse de Matrix, que sobrescreve o método __lt__, de modo a que a comparação de dois nós seja feita apenas com base na heurística.

MatrixForAStar: é uma subclasse de Matrix, que sobrescreve o método __lt__, de modo a que a comparação de dois nós seja feita com base no custo total (custo acumulado do caminho do cursor + heurística).

AI: classe não instanciável, que disponibiliza métodos estáticos auxiliares.

SearchTree: representa a árvore de pesquisa, permitindo a exploração e expansão de nós.

- *search()*: não tem em conta custos ou heurísticas (**pesquisa em profundidade** e **em largura**).
- *search2()*: tem em conta a profundidade do nó (custo sem conhecimento da posição do cursor) e a sua heurística. É utilizada pela **pesquisa gulosa**, que ordena os nós, apenas pela heurística, sem atender ao custo supracitado, cujo intuito é a minimização da profundidade da árvore.
- *search3()*: tem em conta apenas o custo acumulado do caminho do cursor (**pesquisa uniforme).**
- *search4()*: tem em conta o custo acumulado do caminho do cursor e a sua heurística (**pesquisa A***).

Otimizações

- Quando um nó é criado, herda alguns dados do nó pai. Aplicando o padrão de software **Flyweight**, é possível reduzir, não só a memória utilizada, como o tempo de execução, uma vez que se promove a **reutilização de objetos** (cópias por referência e não por valor), para atributos constantes.

- Numa primeira versão do search, a estrutura grid visited era uma lista. Após discutirmos estratégias de otimização, com o grupo "102536_102778", passámos a utilizar um conjunto (**set**), o que reduziu a complexidade da operação de *lookup* de O(n) para O(1).

- Tendo por base outras UCs, recorremos a uma **min-heap**, para guardar os nós por explorar. Com o módulo **heapq**, nativo do Python, garantimos a ordenação dos open nodes, após cada inserção, não sendo necessária qualquer operação ulterior, com vista a obter o valor mínimo.

- **HEURÍSTICA UTILIZADA**: nº de peças a bloquear o caminho da peça A até à saída.

student.py

Em cada iteração do seu ciclo principal, o agente capta o estado do puzzle, através dos seus **sensores**:

- o próprio código de deteção de alteração da grid;
- *detectCrazy* - função que deteta a ocorrência de um “crazy car”;
- *detectStuck* - função que deteta a ocorrência de uma anomalia que não foi identificada como “crazy car”, mas impediu a realização com sucesso do comando enviado para o jogo.

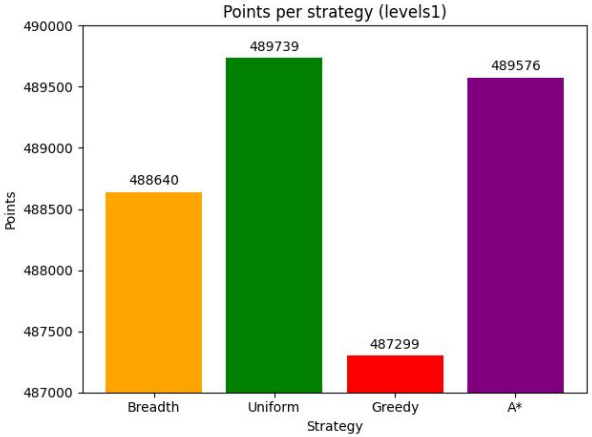
De seguida, o agente escolhe a **estratégia** de resolução do problema, consoante o tamanho da grid:

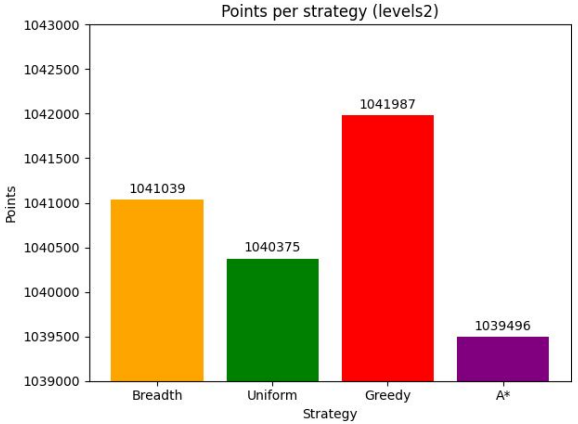
- pesquisa gulosa, quando a grid tem dimensão superior a 6x6;
- pesquisa uniforme, quando a grid tem dimensão igual ou inferior a 6x6;

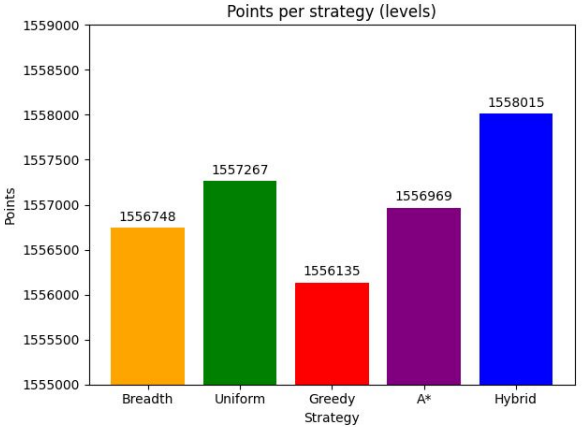
A solução do tree_search é convertida em movimentos do cursor, por intermédio da função *moveCursor*. Esta simula o comportamento do cursor, ao **selecionar uma peça a partir da sua fronteira mais próxima** (está otimizada, nesse sentido).

O tempo que demora a encontrar a solução e a convertê-la é cronometrado, para assegurar a **sincronização com o servidor**. Quando esse tempo total é maior do que a taxa de aceitação de comandos por parte do servidor, são enviados comandos vazios.

Resultados







levels1: conjunto de níveis de dimensão 6x6, fornecido antes da entrega preliminar.
levels2: subconjunto do *levels*, que inclui apenas níveis de dimensão 8x8.
levels: conjunto de níveis de dimensão 6x6 e 8x8, fornecido depois da entrega preliminar.

A partir dos resultados, podemos concluir que:

- a pesquisa **uniforme** é a mais indicada para níveis de dimensão reduzida (<= 6x6). É mais lenta que a pesquisa em largura, mas as suas soluções minimizam os movimentos do cursor.
- a pesquisa **gulosa** é a mais indicada para níveis maiores (8x8).
- a pesquisa **híbrida** (uniforme para <= 6x6 e gulosa para > 6x6) é a melhor solução final.

MAIS GRÁFICOS NA PASTA *GRAPHS* DO REPOSITÓRIO
(regressões lineares com tempos, nº de nós expandidos e nº de peças movimentadas)