

Introdução

- ❖ Utilizámos 4 ficheiros:
 - ❖ **dominio.py**
 - ❖ **tree_search.py**
 - ❖ **student.py**
 - ❖ **common2.py**
- ❖ Foram também utilizados outros ficheiros de testes que foram realizados ao longo do projeto para verificar se se ia concretizando o pretendido, para efeitos de degub.

Trabalho realizado por:

Rafael Amorim, 98197

Tiago Alves, 104110

Domínio

- ❖ É constituído por cinco funções:
 - ❖ `__init__()`: Inicializa o domínio com um mapa;
 - ❖ `actions()`: Retorna as ações possíveis de fazer num mapa;
 - ❖ `result()`: Retorna o estado (mapa) consoante o resultado do movimento de uma peça no mapa fornecido;
 - ❖ `heuristic()`: Retorna o resultado da heurística selecionada. (Será falado mais à frente)
 - ❖ `satisfies()`: Retorna se já foi chegada à solução
- ❖ É utilizado pela árvore de pesquisa para obter as diferentes ações, resultados, melhorar a mesma com heurísticas e para encontrar a solução.

Search

- ❖ Criou-se um set para **guardar os estados visitados** de forma a que não seja repetida a mesma pesquisa, desnecessariamente. No final de cada iteração é guardado o novo estado que foi visitado;
- ❖ Ao começar a pesquisa retira-se o **1º elemento da lista de nós abertos**, começando pela raiz e enquanto o estado do nó não for igual à solução continua-se a pesquisa;
- ❖ Se chegar à solução devolve a **lista de ações a realizar** para chegar ao objetivo;
- ❖ Se a pesquisa continuou, cria-se uma **nova lista de nós abertos**. São analisadas **todas as ações possíveis** e os seus resultados(novos estados) no mapa anterior. Se estes novos estados não estiverem nos estados já visitados serão então produzidos os novos nós e guardados na lista criada;
- ❖ Na próxima iteração é selecionado o **próximo nó** a analisar tendo em conta a **estratégia de pesquisa** escolhida. Para o caso da pesquisa gulosa, por exemplo, os nós são ordenados pela heurística;
- ❖ Voltamos a verificar se ganhou e assim sucessivamente.



Student

- ❖ No ficheiro student.py é feito o envio dos movimentos de cada carro para o servidor. Sendo assim é necessário agrupar tudo neste ficheiro de modo a que seja possível obter as chaves (ou movimentos) a enviar para o servidor.
- ❖ Primeiro é feita a pesquisa da solução tendo o mapa inicial, obtendo os movimentos que cada peça tem que fazer.
- ❖ Para que se possa utilizar os movimentos da solução é necessário verificar se o cursor se encontra a selecionar o carro correto.
- ❖ Para tal foram criadas várias funções:
 - ❖ `vaiPoCarro()`: retorna um movimento único de forma a que o cursor fique cada vez mais próximo do carro e eventualmente chegue ao carro (pela ponta mais próxima deste);
 - ❖ `cursorOnCar()`: verifica se o cursor já se encontra sobre o carro;
 - ❖ `carSelected()`: verifica se o carro já se encontra selecionado;
 - ❖ `entrada()`: reúne todas as funções anteriores de forma a saber se pode enviar o próximo passo da solução da pesquisa, se precisa de movimentar o cursor para o carro ou selecionar o carro.

Heurísticas



- ❖ Neste projeto usamos três heurísticas, tendo experimentado cada uma à medida que íamos avançando. Estas realizam o seu cálculo a cada novo estado (mapa) e permitem ordenar a ordem de pesquisa por estados que tenham heurística menor, facilitando assim a procura da solução.
- ❖ 1^a heurística: Representa a distância do carro vermelho até à saída.

Nível 55 tem 1.508990 segundos
Nível 56 tem 1.362006 segundos
Nível 57 tem 0.365000 segundos
Total: 28.944583 segundos
- ❖ 2^a heurística: A função desta heurística retorna uma lista com os carros a bloquear o caminho para a saída que é usada na 3^a heurística; Para obter o valor desta heurística basta obter o comprimento da lista.

Nível 55 tem 1.158999 segundos
Nível 56 tem 1.038996 segundos
Nível 57 tem 0.377010 segundos
Total: 24.746966 segundos
- ❖ 3^a heurística: Representa o número de carros que bloqueiam o carro A e quais destes também estão bloqueados. Por cada carro na lista de carros a bloquear A se este estiver bloqueado representa 1 ponto nesta heurística, caso contrário, representa 2.

Nível 55 tem 1.352995 segundos
Nível 56 tem 1.196004 segundos
Nível 57 tem 0.383000 segundos
Total: 26.733677 segundos
- ❖ Concluímos que a 2^a heurística foi a que nos forneceu melhores resultados.

Testes/ Resultados

- ❖ Para os testes foram criados vários programas, alguns com testes mais simples para verificar o resultado de algumas funções:
 - ❖ `test_domain.py`: Testa o resultado de algumas funções do domínio como a `actions()` e o `result()`;
 - ❖ `test_tree_search.py`: Testa o resultado da árvore de pesquisa.
- ❖ Para testes posteriores tendo em vista melhorias de tempo foram criados dois ficheiros:
 - ❖ `testes.py`: Indica o tempo de procura do resultado da pesquisa de um só nível;
 - ❖ `testes2.py`: Indica o tempo de procura do resultado da pesquisa para todos os níveis e soma no final;
 - ❖ `testesBenchmark.py`: Variação do `testes2.py` para registo dos testes em ficheiro;
 - ❖ Os três ficheiros acima permitiram melhorar os tempos da árvore de pesquisa, analisando as mudanças que foram sendo feitas e o seu impacto nos tempos;
 - ❖ Exemplos que melhoraram o nosso tempo:
 - ❖ Sem heurísticas a procura seria muito mais lenta e de entre as feitas também foi possível ver qual era a mais eficaz, como é indicado no slide anterior;
 - ❖ Transição na árvore de pesquisa de objetos do tipo `mapa` para `string`;
 - ❖ Utilização de `set` em vez de `lista` para guardar os estados visitados (`set` não tem repetição de elementos);
 - ❖ Um exemplo fora da árvore poderá ser, no `student`, o envio do cursor para a ponta mais próxima do carro.