Sumário: Na realização deste projeto, são utlizados dois métodos de procura de diferente natureza. O *Uninformed Search Method* e *Informed Search Method*. Sendo que para o *Uninformed Search Method* utiliza-se o método *Uniform Cost Search* (UCS) e para o *Informed Search Method* utiliza-se o método A^* *Search*. O principal objetivo do projeto passa por agendar os lançamentos dos componentes, a fim de montar a Estação Espacial Internacional, com o menor custo possível. Tem também como objetivo o desenvolvimento de código em *Python* (versão 3).

1 - Introdução Teórica O UCS é um método de procura que origina um resultado ótimo para qualquer passo da função de custo. A cada passo, este método expande sempre o nó que apresenta menor $g(n)$, onde $g(n)$ é o *path cost* até chegar a esse nó. É possível expandir desta foma, porque as "fronteiras"são guardadas numa fila de prioridade, oredenadas por g. O UCS expande os nós de acordo com os custos óptimos, sendo que este método não tem em conta o número de passos que um caminho tem mas sim o custo total destes."O método só é completo quando o custo dos passos é superior a uma constante de valor bastante pequeno ϵ". [1] O pior caso deste algoritmo deve-se à complexidade de tempo e custo, $O(b^{1+[\frac{C^*}{\epsilon}]})$. Isto acontece porque o UCS pode explorar grandes àrvores de passos pequenos antes de explorar caminhos que incluam passos maiores e úteis. Tendo-se assim $f(n) = g(n)$.

A^* *Search* é completo e óptimo, tendo um algoritmo semelhante ao UCS. "Este método que avalia os nós, combinando $g(n)$ (custo do caminho até chegar a um certo nó) e $h(n)$, o custo de chegar desse nó até ao objectivo. Tendo-se assim $f(n) = g(n) + h(n)$

Dado que $g(n)$ dá o custo do caminho ao nó n, e $h(n)$ é o custo estimado do caminho com menor custo desde n até ao objetivo, temos que $f(n)$ é o custo estimado da solução mais barata passando por n". [1]

2 - Representação do Problema

Abstraction: De modo a abstrair alguma da complexidade do problema, este foi pensado da seguinte forma: inicialmente, não havendo ainda nenhum lançamento a ser carregado, o algoritmo irá escolher sempre como primeira acção um lançamento. Posteriormente, irá escolher um módulo de cada vez, de modo a que os módulos escolhidos encaixem naqueles que já estão em órbita ou no lançamento e, que não ultrapassem a capacidade de carga do mesmo. Apenas será escolhido um novo lançamento se já não existirem módulos que não excedam a capacidade do lançamento actual. Quando se trata de escolher o novo lançamento, apenas iremos escolher aqueles que irão ocorrer após a data de lançamento do anterior.

O problema pode então ser descrito por:

- ***State***: o estado vai conter os módulos que já se encontram em órbita ou no laçamento a ser carregado e, os laçamentos que já foram efectuados ou o laçamento que está a ser carregado. Para além disso, de modo a facilitar a implementação de algumas restrições ao problema, nomeadamente a capacidade máxima do lançamento, incluiu-se no estado o valor actual da carga que está a ser carregada no lançamento.
- ***Initial State***: o estado inicial é um estado em que não há lançamentos efectuados nem módulos em órbita. A carga que está a ser carregada é, consequentemente, igual a 0.
- ***Actions***: as acções irão ser apenas "escolher um módulo"ou "escolher um laçamento", com as restrições indicadas na ***asbtraction***.
- ***Transition Model***: o estado resultante de escolher um módulo irá ser um estado com as condições do estado anterior, mais o módulo escolhido, somando também o seu peso à carga actual do lançamento. O estado resultante de escolher um lançamento irá ser um estado com as condições do estado anterior, mais o novo lançamento escolhido, inicializando a carga actual do lançamento a 0
- ***Goal test***: O objectivo é ter todos os módulos em órbita (no estado).
- ***Path cost***: O custo depende do lançamento. O custo de escolher um módulo é dado pelo produto entre o seu peso e o custo variável do lançamento associado. O custo de escolher um

laçamento é dado pelo seu custo fixo.

3 - Dominio Dependente e Independente

No algoritmo desenvolvido, é bem explícita a distinção de dominios. Foi desenvolvido um ficheiro denominado *"domainDependent.py"* que irá conter todas as funções que são dependentes do problema em questão. Foi também criado um ficheiro *"domainIndependent.py"* que contempla todas as funções gerais do algoritmo de busca e as funções que é necessário desenvolver (indicando o formato do que é retornado das mesmas) de modo a que a formulação de um outro problema possa ser também resolvida pelo algoritmo de busca.

Importa salientar que, uma vez que a formulação do problema permite que exista repetição de estados, implementou-se um ***General Graph-Search*** com as modificações necessárias para que possa ser utilizado *Uniform Graph-Search* ou *A* Search* (verificação do *goal state* apenas após expandir o *node*, de modo a evitar soluções que se encontrem num *suboptimal path*).

4 - Heurística

Uma vez que foi utilizado um algoritmo de *Graph-Search*, de modo a garantir que A* é óptimo, a heuristica tem de ser **consistente**. Uma heuristica diz-se consistente se, para cada *node* n e para cada sucessor n' gerado pela acção a, o custo estimado para atingir o objectivo a partir de n não for superior ao custo associado quando geramos n' (*step cost*) mais o custo estimado para atingir o objectivo a partir de n':

$$h(n) \leq c(n, a, n') + h(n') \tag{1}$$

A nossa heuristica foi gerada com base em *relaxed problems*: "*the cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem*". O nosso *relaxed problem* irá ser então: carregar todos os módulos que ainda faltam no lançamento actual - deste modo, estamos a retirar do problema a restrição da carga máxima dos lançamentos. Assim, o valor da heuristica h irá ser o peso dos módulos a multiplicar pelo valor variável do lançamento.

Podemos verificar facilmente que a heuristica obdece à regra *triangle inequality*, logo é consistente, garantindo assim uma solução óptima com o A* Graph Search.

5 - Performance e Comparação de Algoritmos

		Cost Search		*Effective Branching Factor*	
Ficheiro	d	UCS	A^*	UCS	A^*
Trivial2	4	37	8	2.13	1.49
Trivial1	6	33	24	1.51	1.41
Simple	14	659	217	1.47	1.33
Simetry	15	3093	2045	1.60	1.55
Mir2	11	1561	296	1.81	1.52
Mir	12	28652	8450	2.24	2

Facilmente se verifica que o *Uniform-cost Search* tem maior dificuldade em resolver os exemplos onde os lançamentos têm maior capacidade e levam mais módulos. Isto deve-se ao facto de o algoritmo adicionar módulos um a um. Esta análise permitiu optar pela implementação da heuristica, descrita anteriormente, no algoritmo A^*. Verifica-se que este algoritmo tem uma melhor performance, com um bom *effective branch factor* (idealmente $b^*_{OPT} = 1$) excepto no exemplo do *mir.txt* - que utiliza lançamentos com custos muito próximos, ou mesmo iguais em alguns casos, o que leva o algoritmo a gerar uma maior quantidade de *nodes*. Pode-se verificar esse efeito no *mir2.txt*, onde se retiraram os lançamentos com valores iguais e se ajustaram os custos.

Referências

[1] Stuart Russell, Peter Norvig, $Artificial Intelligence$*A Modern Approach*, Pearson Education, Inc., New Jersey, 3rd Edition, 2010.