Engenharia Informática

Inteligência Artificial

MONTA-CARGAS

Luís Matos

Nº 2151361

2151361@my.ipleiria.pt

Mayelson de Sousa

Nº 2151582

2151582@my.ipleiria.pt

**ABSTRACT**

No âmbito da cadeira de Inteligência Artificial foi proposto a elaboração de um programa que resolva puzzles do jogo Monta-Cargas, recorrendo a algoritmos de procura informados e não informados. Este artigo descreve a maneira como implementámos a resolução do jogo.

**ENVIRONMENT**

Este projeto foi um implementado no Netbeans e compilado para o Java 1.8. O Computador usado para os testes fui um Acer Aspire 5741ZG com um processador Pentium(R) CPU P6000 a 1.87GHz, 2 Cores, com 4.0 GB de RAM e com o Windows 10 64 bits.

**ALGORITHMS**

Neste projeto usámos os algoritmos de pesquisa para resolução de problemas, entre os quais:

Algoritmos Não Informados:

* Breadth First Search
* Depth First Search
* Depth Limited First Search
* Iterative Deepening Search
* Uniform cost search

Algoritmos Informados:

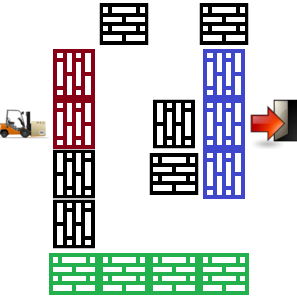
* A\* Search
* IDA\* Search
* Beam Search
* Greedy Best First Search

**MONTA-CARGAS**

O MONTA-CARGAS é baseado no Mouse um jogo para a mobile em que o objetivo consiste em descobrir a sequência de ações que permita deslocar um conjunto de caixas de modo a desobstruir o caminho entre o rato e a sua toca presenteando o jogador com a pontuação máxima caso este consiga descobrir a solução com o menor numero de ações possível.

No MONTA-CARGAS o conceito é o mesmo apenas substituímos o rato por um monta-cargas e a toca pela porta do armazém. A figura seguinte mostra uma possível configuração inicial do jogo. A saída encontra-se sempre do lado direito e o monta-cargas só se pode mover para a esquerda ou para a direita. O Armazém contém um conjunto de caixas que só se podem mover em linha reta dependendo da sua orientação (verticalmente ou horizontalmente).

Em cada estado só se pode mover uma peca do jogo para uma posição adjacente caso essa posição não esteja ocupada por outra peca, à exceção do monta-cargas, pois este pode mover-se para cima da porta terminando assim o jogo.



**Figura 1.** Monta-Cargas

# INTRODUCTION

Pretende-se implementar um programa que resolva os puzzles do Monta-Cargas, recorrendo aos algoritmos de pesquisa implementados nas aulas (algoritmos de pesquisa não informados e informados). O objetivo é usar estes algoritmos de forma a chegar à melhor solução para a resolução do problema. Pretende-se também formular heurísticas que permitam resolver os puzzles de uma maneira mais otimizada.

Para além da implementação dos algoritmos pedidos para a resolução do problema, pretende-se que seja também realizado um estudo comparativo do desempenho dos vários algoritmos de pesquisa bem como das heurísticas formuladas para a resolução dos puzzles.

No ponto 2 descrevemos a implementação do projeto, no ponto 3 comparamos o desempenho de todos os algoritmos implementados, no ponto 4 descrevemos e comparamos as heurísticas criadas e no ponto 5 temos a conclusão.

# IMPLEMENTATION

Como base para este projeto usamos o projeto do Puzzle-Eigth que foi desenvolvido nas aulas práticas. Este projeto já incluía a interface gráfica, os algoritmos de procura implementados e um agente que faz uso destes algoritmos para resolver o Puzzle-Eigth. Vamos agora explicar alterações e adaptações que fizemos para transformar este projeto no projeto proposto.

* 1. First Steps

Começamos por modificar a interface gráfica expandimos a grelha de 3x3 para 6x6 (este tamanho é agora definido no ficheiro do problema) e alterámos as imagens dos números para imagens de pecas do MONTA-CARGAS.

Em relação as classes existentes para resolver o problema decidimos fazer *Refracting* em todas as classes que tinham a ver com a resolução do Puzzle-Eight isto também serviu para ter uma ideia inicial de como estas classes interagiam umas com as outras.

* 1. New Classes
     1. Peca

Ao analisar o problema categorizamos todas as entidades do problema que se podem mover como uma peca, por isso criamos a classe abstrata Peca, esta classe tem apenas 4 atributos, linha, coluna, tamanho e uma lista de ações. Agora podemos mencionar o carro ou uma caixa como peca com posição (linha e coluna), tamanho e ações.

A posição de uma peca é obtida através da sua localização na matriz do estado já o seu tamanho e ações são atributos que dependem da peca em questão por isso definimos uma classe para cada peca que estende esta classe.

Cada peca apenas tem 2 ações, ou se pode mover para cima e para baixo (Verticalmente) ou para a direita e para a esquerda (Horizontalmente). As classes destas ações já existiam no projeto base.

* + 1. Posicao

Esta classe apenas possui dois inteiros que são o i (linha) e j (coluna). Esta classe fui criada para evitar ler a mesma peca da matriz caso esta já tenha sido criada.

Por exemplo imaginemos que uma linha da matriz é a seguinte [0 0 6 6 6 0], ao lermos o primeiro 6 já sabemos que nessa posição está uma Caixa com tamanho 3 na vertical e não precisamos de ler os próximos 2 números que se seguem por isso adicionamos essas próximas posições a uma lista de posições a passar quando estamos a ler a matriz.

Esta classe também podia ter sido usada como um atributo da classe Peca mas decidimos não o fazer para obtermos código mais limpo.

* 1. New Attributes and Methods on the State
     1. LinkedList<Peca> pecas;

Este atributo armazena todas as pecas do estado atual do problema e é preenchido quando analisamos a matriz que foi passada para construir o estado atual.

Como as pecas nunca são destruídas ou novas pecas são criadas criamos um construtor para o State que recebe uma lista deste tipo e copia essa lista para este atributo, assim não precisamos de analisar novamente a matriz caso já exista um estado anterior.

* + 1. Boolean canMove(Peca peca) & void move(Peca peca)

Para cada direção que uma peca se pode mexer existe o par de métodos canMove(Peca peca) e move(Peca peca) uma serve para verificar se a peca em questão se pode mover, isto é não há nenhuma peca a bloquear o movimento que se pretende fazer, e a outra para mover a peca receptivamente, por exemplo no caso da actionRight existem os métodos canMoveRight(Peca peca) e moveRight(Peca peca).

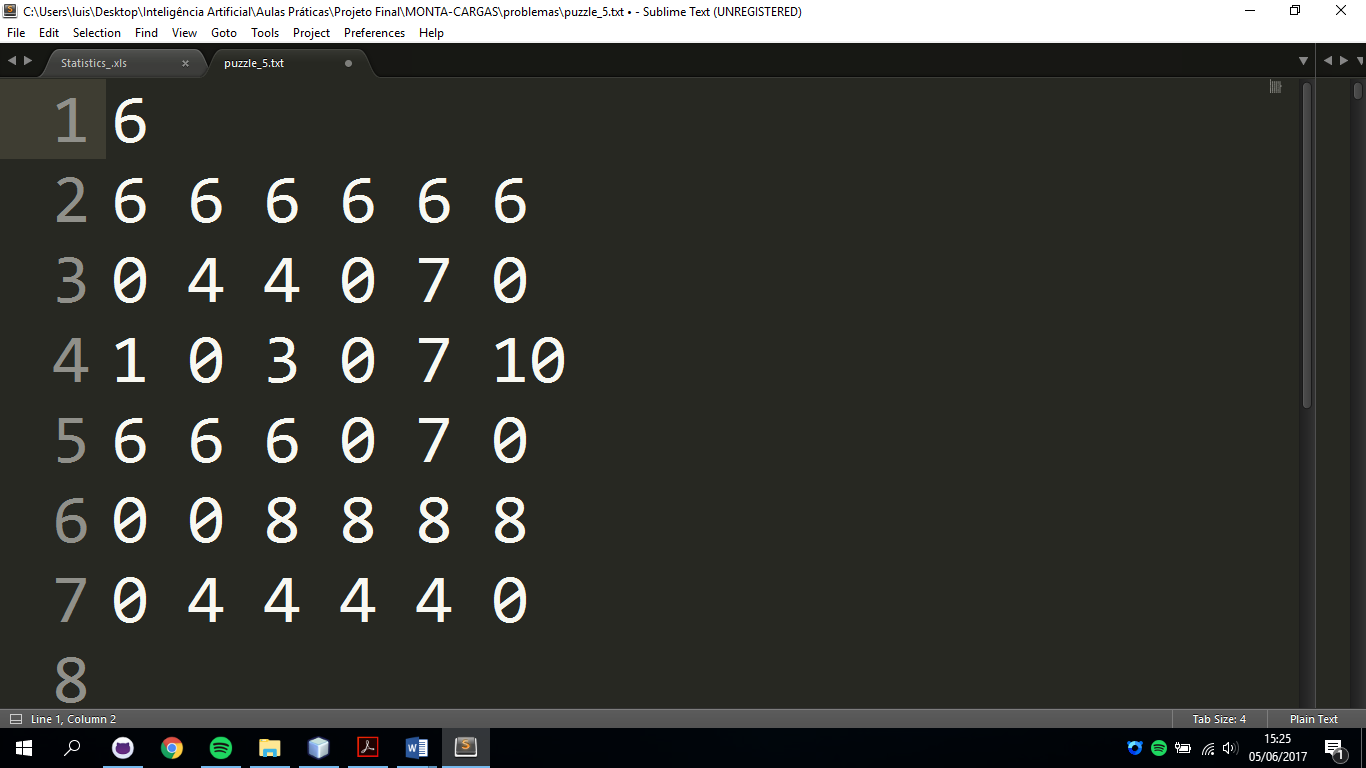
O caso do canMoveRight é um caso particular, pois se a peca for o carro este pode mover-se para a localização da porta.

* + 1. LinkedList<Action> getActions()

Este método é o responsável por devolver ao problema todas as ações que cada peca do estado atual detém.

* 1. State

Cada estado é definido por uma matriz de inteiros em que o numero 1 representa o carro e o numero 10 representa a porta, qualquer outro numero representa uma caixa especifica a esse número. A figura seguinte mostra o estado inicial do puzzle 5. E uma lista de pecas.



**Figura 2.** Estado inicial o puzzle 5.

* 1. Observations

Como já devem ter reparado a porta não é considerada uma peça. A porta esta apenas é mostrada na grelha e a sua posição é calculada consoante a coluna do carro. No caso do carro se encontrar na posição da porta significa que uma solução foi encontrada.

Todos os puzzles para este problema foram fornecidos pelos docentes desta disciplina.

# Performance

Neste capítulo vamos abordar os testes que fizemos com os diferentes algoritmos e comparar os resultados mais relevantes.

Estes testes foram realizados nos puzzle 1, que é o mais simples de todos.

* 1. Performance of Not Informed Search Algorithms

Através do gráfico acima, podemos observar que o algoritmo Depth first Search encontra a solução com um custo muito elevado, em contrapartida, os outros encontram soluções aceitáveis.

Neste gráfico o algoritmo Iterative Deepening Search explora uma quantidade muito elevada de nós, o que o torna muito lento.

No que diz respeito as fronteiras o algoritmo Iterative Deepening Search apesar de obter uma quantidade enorme de nos expandidos, ele encontra a solução com o tamanho da fronteira ligeiramente mais baixo do que os restantes resultados.

Como podemos observar pelos gráficos acima, os métodos de procura não informados mais promissores são o Uniform Cost e o Breadth First pois encontram a solução com menor custo e um número reduzido de nós explorados.

Através do gráfico acima, podemos observar que o algoritmo Depth First Search encontra a solução com muitos poucos nós gerados em relação aos outros, ao contrario do Iterative Deepening Seacrh que encontra a solução explorando uma quantidade enorme de nós.

Em relação ao tempo de execução também o algoritmo Depth First Search é o melhor de todos, ou seja, apresenta um tempo de execução mais baixo.

Vamos realçar que os algoritmos Depth First não encontraram a melhor solução, no entanto apresenta vantagens no que diz respeito ao tamanho da fronteira, o número de nós expandidos, o tempo de execução, e numerode nós gerados, em relação aos outros.

O Iterative Deepening é o algoritmo mais lento, ou seja, o tempo de execução é inaceitável, muito elevado, explora uma quantidade absurda de nós em comparação com os outros algoritmos.

* 1. Performance of Informed Search Algorithms

Nos algoritmos de procura informada todos eles encontram a solução com o menor custo, apesar de alguns terem um tempo de execução elevado, como é o caso do Greed Best First Search.

Iremos explicar as estatísticas obtidos para os diferentes gráficos, de modo a perceber qual dos algoritmos foi o melhor em relação ao problema que estamos a analisar.

A Heurística usada para estas comparações foi a “Distance to Door and Tiles Size In Front of Car”

Em relação ao numero de nós expandidos, através do gráfico a cima podemos ver que o Greedy Best Seacrh é o melhor, pois encontra a solução com o número de nos expandidos relativamente mais baixo do que os outros.

Deste gráfico, concluímos que o Beam Search é melhor, pois o tamanho máximo da fronteira é muito mais baixo do que os restantes algoritmos.

A partir deste gráfico, verificamos que todos os algoritmos apresentam boas soluções em relação ao números de nós gerados, no entanto observamos que o Greedy Best First Search é o melhor para este problema.

* 1. Procura Informada versus Procura Informada

Usamos um algoritmos de cada grupo de procura, (informados (A\* Search) e não informados (Depth First Search)) com a heurística “Distance to Door and Tiles Size In Front of Car”, para resolver o puzzle 8, e obtivemos os resultados que constam no gráfico a baixo.

Através do gráfico a cima, podemos observar que o A\*, apesar de ter um tempo de execução ligeiramente superior ao Depth First Search, apresenta um melhor número de nós expandidos, um tamanho da fronteira muito mais baixo, gera um poucos nós e um tamanho da fronteira também muito mais baixo.

# HEURISTICS

* 1. Analysis

A cada estado conseguimos obter várias informações que, futuramente ajudaram a desenvolver várias heurísticas. Ao olharmos com atenção, há 3 dados importantes e que devem ser trabalhados de modo a encontrar a heurística certa:

* Número de caixas entre o carro e a porta
* Distância do carro à porta
* O tamanho das peças
  1. Descrição das Heurísticas

Para comparação das heurísticas usamos o A\* Search.

* + 1. CarDistance

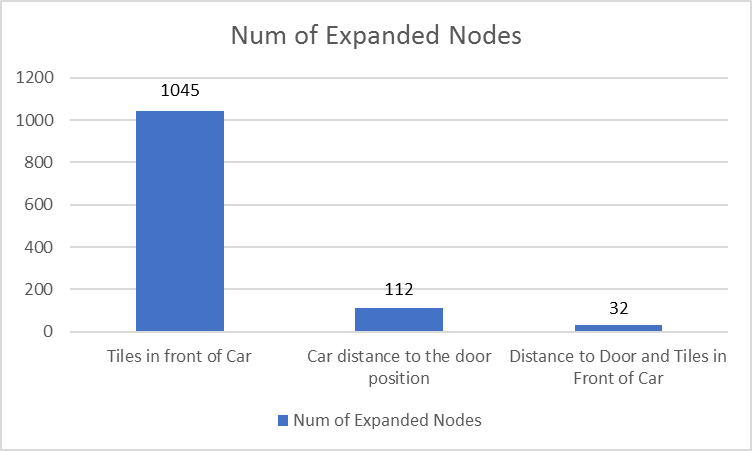
Esta é a heurística mais básica do nosso projeto, baseia-se no na distancia entre o carro e o veiculo, no estado atual. Ao analisar esta heurística vem trazer poucas vantagens resolução do problema pois haverão muitos estados em que a heurística será a mesma.

* + 1. TilesInFrontOfCar

Esta heurística também é muito básica, baseia-se apenas no numero de pecas que estão entre o carro e o veículo.

* + 1. CarDistancePlusTilesInFront

Esta heurística é apenas o conjunto das heurísticas faladas a cima, como podemos ver na figura de baixo esta combinação é essencial para o desenvolvimento de uma boa heurística.



* + 1. SizeOfTilesInFrontOfCar

Esta heurística é igual á anterior, mas em vez de contar apenas o numero de pecas que estão á frente do carro tem em conta o tamanho das pecas. Isto tem um grande impacto no numero de nos explorados como podemos ver no próximo gráfico.

* + 1. AllTilesInFrontOfCarIndividualy

Esta heurística faz parte da nossa heurística mais rápida e mais eficaz em termos de nós explorados, que será a ultima que iremos discutir.

Esta é provavelmente a heurística mais difícil de explicar. Como um dos criadores do projeto costumava jogar o jogo do Mouse a ideia foi tentar “replicar” parte do processo (sub)consciente que o levava a chegar à solução esse objetivo é o de mover as pecas maiores que se encontram na frente do carro para traz deste, desta forma apenas as caixas mais pequenas ficam na frente do carro e estas são as mais fáceis de mover para libertar o caminho, pois são as que bloqueiam menos pecas com um movimento.

Esta heurística acabou por ser um grande sucesso só por si por isso decidimos isolá-la para observar o seu comportamento individualmente e para a podermos explicar em pormenor.

* + 1. AllHeuristicTogether

Esta heurística é o apenas o conjunto de todas as heurísticas realizadas anteriormente. Pensamos que era bom ver como se comportavam juntas.

* + 1. AllHeuristicTogetherAjusted

Esta heurística é igual á anterior, mas com um pequeno ajuste para obter um balanceamento melhor entre elas. Esta a mais rápida e mais eficaz em termos de nós explorados que criamos, apesar de nem sempre encontrar a solução com o menor custo, mas o custo está sempre perto da melhor solução. O segredo para esta heurística já foi explicado no ponto 4.2.5 mas agora damos mais importância á distancia entre o carro e a porta e o numero e o tamanho de pecas que estão á frente do carro.

* 1. Gráficos

De seguida estão todos os gráficos que mostram a diferença entre a heurísticas usando o A\*.

# CONCLUSÃO

* 1. Algoritmos

Em conclusão, os algoritmos informados em relação aos algoritmos não informados são apresentam melhores resultados mais rápidos e fiáveis. No entanto, dos algoritmos informados, destacamos os Depth First Search, pois apresentam melhores resultados em relação aos outros, tornando-o também rápido e fiável.

Dos algoritmos informados destacamos o A\* pela sua forma de obter resultados fiáveis, rápidos na resolução dos problemas.

Os algoritmos de profundidade foram os piores, o número de estados expandidos e o tempo de execução é bastante elevado, acabando por ser impossível testar alguns dos problemas mais complexos.

* 1. Heurísticas

A heurística que se mostrou mais fiável foi a combinação de todas as heurísticas com ajuste. Os resultados obtidos foram muito surpreendentes. Gostávamos de realçar que para o puzzle 1 que têm um custo de solução de 7 o A\* apenas expande 7 nós.

* 1. Estatísticas

O utilizador também está livre de fazer as estatísticas que pretender por isso suportamos a criação de ficheiros *excel* com o relatório de cada heurística. Pretendemos facilitar comparação entre plataformas para isso no cabeçalho de cada ficheiro de estatística colocamos o processador, a memoria, o sistema operativo e a versão do java da plataforma em questão.

Todas as estatísticas finais que realizamos encontram-se juntamente com o projeto.

Columns on Last Page Should Be Made As Close As Possible to Equal Length