



TIAGO ANDRÉ MENDES DE ALMEIDA RIBEIRO

TIAGO ALEXANDRE QUARESMA ALVES

Adji‒boto - Manual Técnico

Inteliêngia Artificial, Engenharia de Software

Ano letivo 2018 / 2019

Docentes

Professor, Joaquim Filipe

Professor, Hugo Silva

Engenheiro, Filipe Mariano

Índice

[1. Introdução 3](#_Toc532423361)

[2. Estrutura do Sistema 4](#_Toc532423362)

[Puzzle 4](#_Toc532423363)

[Procura 4](#_Toc532423364)

[Projeto 4](#_Toc532423365)

[Problemas 4](#_Toc532423366)

[3. Implementação técnica 5](#_Toc532423367)

[4. Algoritmos 6](#_Toc532423368)

[5. Limitações técnicas 7](#_Toc532423369)

# Introdução

Adji-boto é uma aplicação desenvolvida baseada num jogo em Common Lisp para a unidade curricular de Inteligência Artificial. Esta consiste num pequeno programa que tem como objetivo encontrar uma solução para um determinado problema, baseada num critério: menor custo ou caminho.

O problema e o algoritmo utilizado é escolhido pelo utilizador, de entre os disponíveis, sendo os problemas lidos de um ficheiro. No final da execução, o resultado mostrado ao utilizador no ecrã e gravado em ficheiro.

Em termos de jogo, neste caso é uma variante do Adji‒boto e Oware, especialmente concebida para este projeto, e o objetivo é encontrar um estado onde já não haja mais peças no tabuleiro.

# Estrutura do Sistema

Esta aplicação está dividade em três ficheiros *.lisp*, de modo a facilitar a manutenção e compreensão humana do programa: puzzle, procura e projeto. São ainda utilizados dois ficheiros *.dat* de *input* e *output* : problemas, resultados.

## Puzzle

Este ficheiro contém toda a informação relativa ao domínio de aplicação, neste caso o jogo. Neste encontram-se todas as funções de manipulação do tabuleiro de jogo, como por exemplo distruir peças pelas casas (allocate-pieces), validação de jogadas (is-move-validp), etc.

## Procura

Neste ficheiro encontram-se todas as funções, quer principais como de auxílio, associadas aos algoritmos de procura em espaço de estados. Aqui estão funções como comparação de elemento a e b para ordenação de lista, estrutura de dados (construtor e getters), funções de avaliação de eficiência, etc.

## Projeto

Por fim, o ficheiro projeto que tem tudo o resto, nomedamente funções de I/O, interação com o utilizador, inicialização da aplicação e funções de testes para facilitar o teste dos algoritmos.

## Problemas

Este ficheiro contém os problemas que o utilizador mais tarde escolhe para aplicar os algoritmos. Este está estruturado da seguinte forma: cada problema numa linha, cada um é composto por duas “listas” (de LISP), seperados por uma vírugla, sendo cada lista uma das linhas do tabuleiro. Este ficheiro é depois lido linha a linha e é criado uma lista geral que contém todas os problemas (listas).

Resultados

Este é o ficheiro produzido no final. Caso já exista este ficheiro na diretoria, apenas adiciona informação ao ficheiro, não substituí. Neste ficheiro estão as caracteristicas do problema que foi resolvido - profunidade máximo, algortimo escolhido, problema escolhido, heuristica utilizada e solução -, e resultados finais - Nós gerados, nós expandidos, penetrância, fator de ramificação, tempo de execução e o caminho do nó inicial até ao nó solução.

# Implementação técnica

Tirando o facto de ter sido utilizado um paradigma funcional devido ao LISP ser uma linguagem funcional, foram-nos impostas algumas limitações ténicas extra. A implementação deste projeto foi toda feita com foco na recursividade, não havendo nenhum ciclo existente no programa, tirando alguns casos exceptions onde optamos por não utilizar na mesma. Foi-nos também impedido o uso de sequênciação e funções com efeitos secundários, à exceção de aquando de interação com o utilizador, sendo que nestes casos era necessário sempre imprimir mensagens na consola ou ficheiro (com uso de format que é uma função com efeitos secundários) e executar um outro comando.

# Algoritmos

Nesta implementação do projeto, por vários motivos, apenas foram implementas os algortimos breadth first search (bfs), depth first search (dfs) e A\*. Independentemente, o principal objetivo deste projeto era o estudo do desempenho dos algoritmos a resolver os problemas fornecidos. Para esta medição, era utilizado o fator de ramificação média, penetrância, número de nós gerados e expandidos e o tempo de execução (em segundos) do algortimo.

* Penetrância: Dá uma boa percepção do número de nós, desnecessários à resolução do problema que foram gerados até se encontrar o nó objetivo.
* Número de nós gerados / expandidos: número de nós gerados / expandidos desde nó inicial até ao nó solução. Quantos menos expandir / gerar mais rapidamente encontra a solução, mais eficiente é.
* Tempo de execução: Este é bastante explicativo, quantos segundos passaram desde que o algoritmo começou até que acabou. Quanto menor este tempo, mais eficiente é. No entanto, esta métrica não é muita boa nalguns casos sendo que conta segundos e não milisegundos.

Numa pré-analise, pode-se assumir que o A\* irá ser o mais eficiente na maioria dos casos, sendo que este é um algoritmo informado e este jogo apresenta problemas de explusão combinatória. Ou seja, tendo em conta que resolver problemas de explusão combinatória é o propósito dos algoritmos informados, podemos afirmar que, em teoria, o A\* será mais indicado. Isto se a heurística utilizada para a assunção de custo for uma heurística indicada ao problema.

Para uma melhor análise, o grupo decidiu elaborar uma tabela comparativa onde estão listado os resultados da resolução dos problemas fornecidos pelo enunciado com os respetivos resultados:

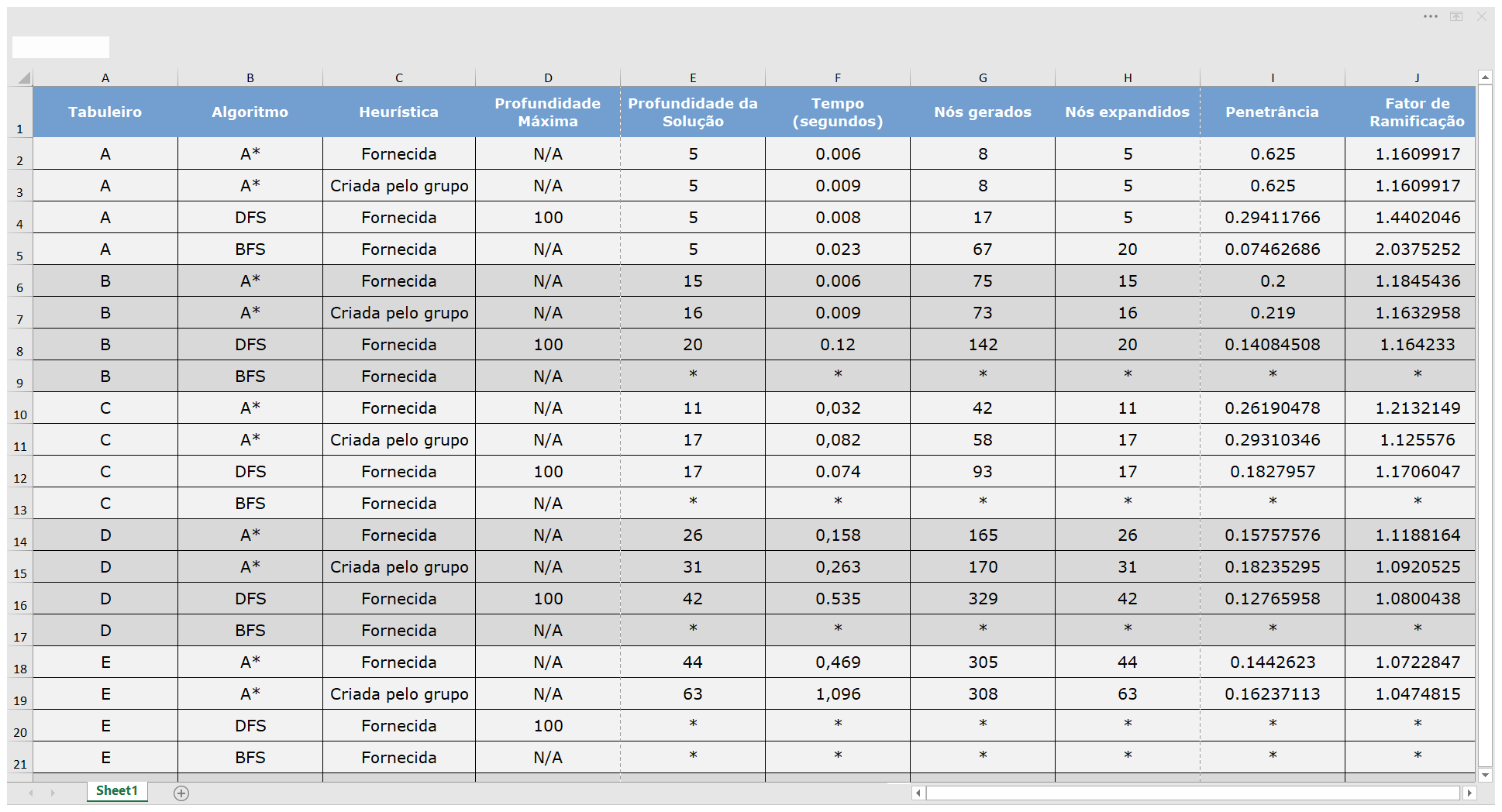


Figura 1 - Captura de ecrã do ficheiro de comparação de resultados

Embora não muito perceptível divo ao número de colunas existentes nesta captura, podemos concluir que de facto o BFS é o pior algoritmo dos três, com uma taxa de sucesso de 14%, tendo apenas conseguido chegar à solução num dos 7 problemas. Seguidamente temos o DFS, que embora um melhoramento significativo em relação ao BFS, com 4 dos 7 (cerca de 57%) problemas resolvidos. No entanto estes resultados são ainda instatisfatórios, sendo que é preciso um algoritmo que seja eficaz em pelo menos 90% dos casos. Nesta ótica, conseguímos verificar que de facto os algortimos informados, neste caso apenas o A\*, são muito melhores que os não informados, tendo conseguido resolver todos os problemas. E também observamos que não há apenas uma heurística possível para a resolução, embora a heurística fornecido no enunciado seja melhor que a calculada pelo grupo, ambas conseguem cumprir o seu objetivo.

A falha no BFS e DFS deve-se à explosão combinatória. Sendo que estes geram todos os sucessores possiveis sem quaisquer restrições, exaustam os recursos da máquina com muito mais rapídez, impendido-os de conseguir obter a solução. Esta exaustão de recursos apresentou-se como falta de memória (stack overflow) ou falta de memória heap disponibilizada pela versão gratuíta do IDE que o grupo utilizou, o LispWorks.

Para uma melhor análise, encontra-se anexado com este ficheiro o documento com a tabela de comparação de algortimos.

# Limitações técnicas

Uma das limitações mais evidentes foi de facto o IDE necessário à criação da aplicação. O LispWorks é um IDE, que embora tenha algumas ferramentas boas como o tracer, debugger, etc, carece várias funcionalidades e “quality of life improvements” que muitos outros IDEs já implementaram, tornando a nossa experiência enquanto programadores horrenda. Não falando no facto de só termos acesso à versão gratuída lmita memória *heap*, impedindo a execução completa de alguns algoritmos nalguns problemas e o número de vezes que o IDE fecha aleatóriamente numa tentativa de nos convencer a comprar uma versão acima.

A falta de tempo por parte do grupo para conseguir implementar os restantes algortimos e incapacidade, tanto por falta de conhecimento do contexto aplicacional como por tempo, da criação de uma segunda heurística, talvez mais admissível à resolução do problema também afetou o produto final.