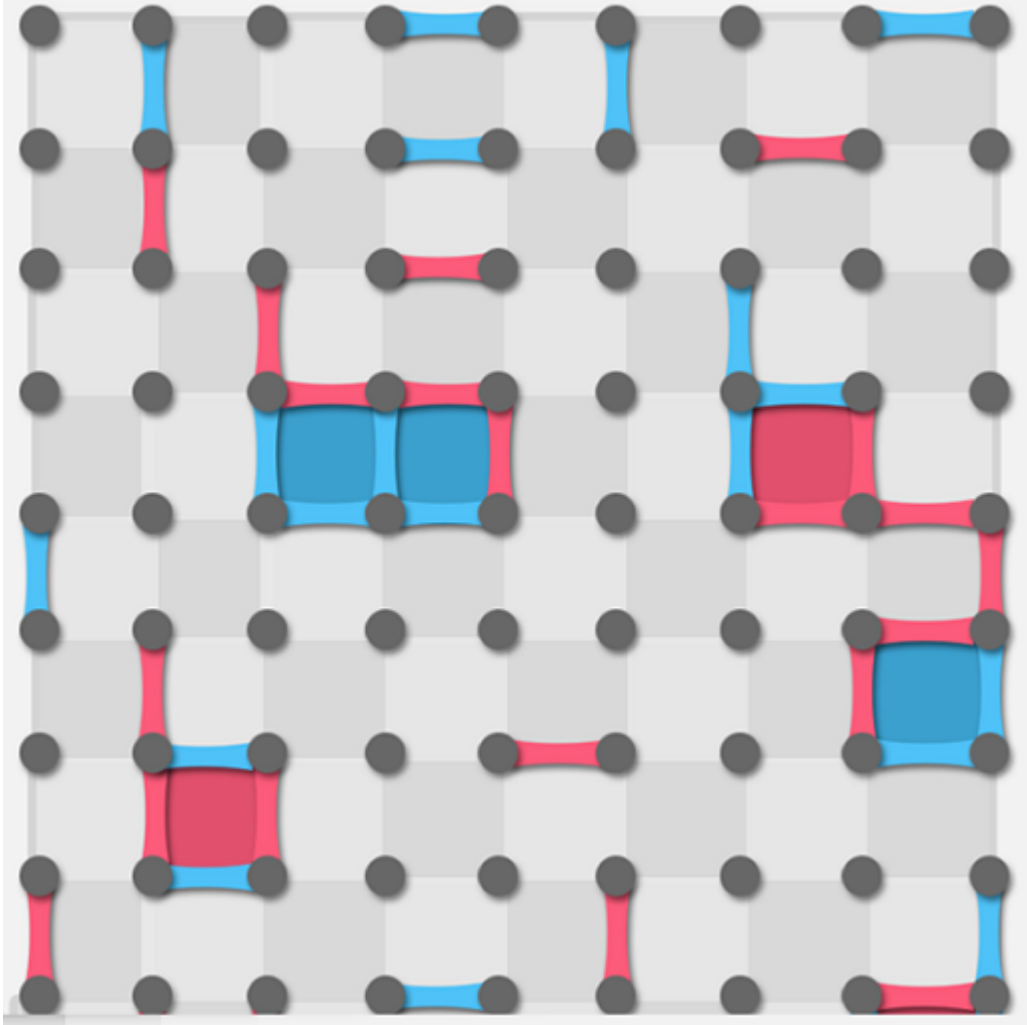
Manual de Utilizador

Dots & Boxes



IPS ESTS - Licenciatura Engenharia Informatica - 2022/2023

Inteligência Artificial
Docente: Joaquim Filipe

Trabalho realizado por:

Nome: Daniel Baptista - Nº:202001990
Nome: Rafael Silva - Nº:202001553

1. Arquitetura do sistema

Módulo de Procura - procura.lisp

É o módulo da aplicação onde estão localizadas todas as funções usadas pelos algoritmos e a implementação dos proprios algoritmos e este módulo pode ser usado genericamente por outros módulos.

O Módulo de Procura têm como Objetivo

Ser utilizado como biblioteca generica onde estão presentes os três algoritmos implementados (**BFS, DFS e A***), ou seja, poderá ser utilizado noutro problema qualquer que necessite o uso de um destes algoritmos.

Conexões do Módulo de Procura

- projeto.lisp;
- puzzle.lisp;

Relacionamento entre as conexões do Módulo de Procura

- Relacionamento deste módulo(procura.lisp) com o projeto.lisp é de modo a poder se utilizar os algoritmos de pesquisa em espaço de estados.
- Relacionamento deste módulo(procura.lisp) com o puzzle.lisp é de modo a poder buscar funções especificas.

Módulo da Interface com utilizador - projeto.lisp

É o módulo da aplicação em que é criada a interface para a interação com o utilizador e ainda a leitura e escrita de feicheiros externos.

O Módulo da Interface com utilizar têm como Objetivo

Ser utilizado para fazer a ligação entre o programa e o utilizador através de uma interface gráfica.

Conexões do Módulo da Interface com utilizador

- puzzle.lisp;
- procura.lisp;
- problemas.dat;
- solucoes.dat;
- log.dat;

Relacionamento entre as conexões do Módulo da Interface com utilizador

- O Relacionamento deste módulo(projeto.lisp) com o *puzzle.lisp* é de modo a obter as funções especificas do dominio do jogo "**Dots and Boxes**"
- O Relacionamento deste módulo(projeto.lisp) com a *procura.lisp* é de modo a poder executar as funções dos algoritmos implementados neste projeto.

- O Relacionamento deste módulo(projeto.lisp) com o *problemas.dat* é de modo a obter os tabuleiros de jogo disponibilizados pelos docentes.
- O Relacionamento deste módulo(projeto.lisp) com o *solucoes.dat* é de modo a saber quantas caixas são precisas para a solução.
- O Relacionamento deste módulo(projeto.lisp) com o *log.dat* é de modo a guardar as estatisticas das soluções realizadas para poder fazer comparações entre as mesmas.

Módulo do Problema - puzzle.lisp

É o módulo da aplicação onde existe uma resolução especifica de um problema em relação ao jogo "**Dots and Boxes**".

O Módulo do Problema têm como Objetivo

Ser utilizado para resolver o problema especifico ao utilizar funções especificas para gerar e avaliar nós do jogo "**Dots and Boxes**".

Conexões do Módulo do Problema

- projeto.lisp

Relacionamento entre as conexões do Módulo do Problema

*O Relacionamento deste módulo(projeto.lisp) com o *projeto.lisp* é de modo a disponibilizar as funções especificas do problema do jogo.

2. Entidades e sua implementação

- ***no*** - entidade que representa o ponto do tabuleiro de jogo.
- ***no-estado*** - entidade que representa o estado em que se encontra o ponto do tabuleiro de jogo.
- ***sucessores*** - entidade que representa todos os sucessores de um nó atual.

3. Algoritmos e sua implementação

Algoritmo BFS

É um algoritmo que faz a procura de cima para baixo e de nivel a nivel até que seja encontrada a solução final ou a lista de abertos ficar vazia e neste caso significa que não existe uma solução possível. Visto isto conseguimos concluir que o algoritmo chega sempre a uma solução possivel caso esta exista e tambem consegue garantir que esta solução seja a solução ótima do problema.
A complexidade algoritmica é de $O(b^d)$, em que **d** é a profundidade máxima e **b** o fator de ramificação médio.

Possivel forma de implementação do BFS

1. Nó inicial => **Abertos**.
2. Se **Abertos** vazia falha.
3. Remove o primeiro nó de **Abertos** (n) e coloca-o em **Fechados**.
4. Expande o nó n. Coloca os sucessores no fim dos **Abertos**, colondo ponteiros para n.

5. Se algum dos sucessores é um nó objectivo sai, e dá a solução, caso contrario volta a refazer os passos todos apartir do passo 2.

Características do BFS

- Assume-se que o nó inicial não é um nó objectivo.
- Este encontra sempre a solução com o caminho mais pequeno, ou seja, encontra sempre a solução ótima.
- Este termina com falha se não existir solução num grafo finito ou se o grafo for infinito este irá ficar num loop infinito não havendo um termino do mesmo.

Implementação feita no projeto

```
;;procura na largura
(defun bfs (no f-objetivo f-sucessores operadores &optional num-solucao abertos
fechados)
  (let* ((novos-fechados (cons no fechados))
         (novos-abertos (abertos-bfs abertos novos-fechados (funcall f-sucessores
no operadores 'bfs nil)))
         (no-solucao (remove nil (mapcar (lambda (no)
                                           (cond((funcall f-objetivo no num-
solucao) no)
                                                (t NIL)))
                                         novos-abertos))))
    (setf *abertos* novos-abertos)
    (setf *fechados* novos-fechados)
    (cond ((null novos-abertos) NIL)
          ((funcall f-objetivo no num-solucao) no)
          ((not (null no-solucao)) (cond ((/= (length no-solucao) 1) (car no-
solucao))
                                         (t (car no-solucao))))
          (t (bfs (car *abertos*) f-objetivo f-sucessores operadores num-solucao
(cdr *abertos*) *fechados*)))))
```

Algoritmo DFS

É um algoritmo que faz a procura de por ramos percorrendo-os verticalmente, neste algoritmo temos que definir a profundidade que pretendemos que este faça a pesquisa e dentro de um ramo se chegar ao fim e não tiver encontrado nenhuma solução vai percorrer o outro ramo seguinte até que encontre a solução. Visto isto pode-se concluir que se este não encontrar a solução no primero ramo que faça a pesquisa poderá tornar-se menos eficiente do que o algortimo BFS.
A complexidade algoritmica é de $O(b^d)$, em que **d** é a profundidade máxima e **b** o fator de ramificação médio.

Possivel forma de implementação do DFS

1. Nó inicial => **Abertos**.
2. Se **Abertos** vazia falha.

3. Remove o primeiro nó de **Abertos** (n) e coloca-o em **Fechados**.
4. Se a profundidade de n é maior do que d (profundidade máxima) volta para o passo 2.
5. Expande o nó n. Coloca os sucessores no início dos **Abertos**, colocando ponteiros para n.
6. Se algum dos sucessores é um nó objectivo termina e dá a solução, se não volta a fazer os passos todos apartir do passo 2.

Características do DFS

- Defini-se um nível de profundidade máximo e apartir do mesmo não são gerados mais nós.

Implementação feita no projeto

```
;; procura na profundidade
(defun dfs (no f-objetivo f-sucessores operadores profundidade &optional num-
solucao abertos fechados)
  (let* ((novos-sucessores-e-fechados (abertos-dfs abertos (cons no fechados)
(funcall f-sucessores no operadores 'dfs nil 0 profundidade)))
         (novos-abertos (car novos-sucessores-e-fechados))
         (novos-fechados (car (cdr novos-sucessores-e-fechados)))
         (no-solucao (remove nil (mapcar (lambda (no)
                                           (cond((funcall f-objetivo no num-
solucao) no)
                                                (t NIL)))
                                         novos-abertos))))
    (setf *abertos* novos-abertos)
    (setf *fechados* novos-fechados)
    (cond ((null novos-abertos) NIL)
          ((funcall f-objetivo no num-solucao) no)
          ((not (null no-solucao)) (cond ((/= (length no-solucao) 1) (car no-
solucao))
                                         (t (car no-solucao))))
          (t (dfs (car *abertos*) f-objetivo f-sucessores operadores profundidade
num-solucao (cdr *abertos*) *fechados*)))))
```

Algoritmo A*

É um algoritmo que se diz "*informado*" ao contrário dos outros dois algoritmos mencionados, este algoritmo usa metodos heurísticos para encontrar uma solução ótima da maneira mais eficiente possível.
Posto isto precisa-se de calcular que ramo iremos pesquisar para posteriormente o expandir, usando este método heurístico: $f(n)=g(n)+h(n)$
Em que f é o custo total, g o custo real e h o custo heuristico que indica quanto falta para chegar ao nó final.
A complexidade algoritmica é de $O(b^d)$, em que **d** é a profundidade máxima e **b** o fator de ramificação médio.

Possivel forma de implementação do A*

1. Nó inicial => **Abertos**. Faz $f(s)=0$.

2. Se **Abertos** vazia falha.
3. Remove o nó de **Abertos** (n) com menor custo (f) e coloca-o em **Fechados**.
4. Expande o nó n e calcula o $f()$ de cada um dos sucessores.
5. Coloca os sucessores que ainda não existem em **Abertos** nem em **Fechados** na lista de **Abertos**, por ordem de $f()$ colocando ponteiros para n.
6. Se algum dos sucessor for um nó objectivo termina e devolve a solução.
7. Associa aos sucessores já em **Abertos** ou **Fechados** o menor dos valores de f (existente ou agora calculado), coloca nos **Abertos** os sucessores que estavam em **Fechados** cujos valores de $f()$ baixaram, redirecionando para n os ponteiros de todos os nós cujos valores de $f()$ baixaram.
8. Volta a fazer todos os passos apartir do passo 2.

Características do A*

- Utilização de métodos heurísticos para determinar o próximo nó a ser expandido
- É possivel encontrar a solução ótima caso a heuristica usada seja admissível
- Pode-se concluir teoricamente que será o algoritmo mais eficiente do que os outros 2 mencionados.

Implementação feita no projeto

```
;;procura informada
(defun a* (no f-solucao f-sucessores operadores f-heuristica &optional num-solucao
abertos fechados)
  (let* ((novos-abertos-fechados (abertos-e-fechados-a* abertos (cons no fechados)
(funcall f-sucessores no operadores 'a* f-heuristica num-solucao)))
         (novos-abertos (car novos-abertos-fechados))
         (novos-fechados (car (cdr novos-abertos-fechados)))
         (no-solucao (cond((funcall f-solucao (car novos-abertos) num-solucao)
(car novos-abertos))
                          (t NIL))))
    (setf *abertos* novos-abertos)
    (setf *fechados* novos-fechados)
    (cond ((null novos-abertos) NIL)
          ((funcall f-solucao no num-solucao) no)
          ((not (null no-solucao)) no-solucao)
          (t (a* (car *abertos*) f-solucao f-sucessores operadores f-heuristica
num-solucao (cdr *abertos*) *fechados*)))))
```

4. Descrição das opções tomadas

Não houve grandes decisões a serem tomadas pois concretamente recorremos aos slides para fazer a implementação dos algoritmos e fomos progressivamente implementando funcionalidades devido aos laboratorios disponibilizados na cadeira, por isso nunca chegamos a ter mais do que uma decisãoa tomar.

5. Limitações técnicas e ideias para desenvolvimento futuro

Refactoring de modo a obter um maior nivel de abstração. Possiveis melhoramentos no desempenho dos algoritmos de procura.