Manual Técnico

Projeto realizado por: Francisco Zacarias nº190221105

Índice:

1 Introdução

Este manual apresenta um desenvolvimento mais técnico sobre o projeto. Vão ser apresentadas as estruturas de dados desenvolvidas para o funcionamento do projeto, descrição dos algoritmos de procura implementados, uma análise de cada problema com os algoritmos e heurísticas implementados. Este manual assume que o leitor tem conhecimentos básicos do *syntax* da linguagem *Lisp*, assim como conhecimento das regras do jogo do Quatro. O escopo deste projeto apenas inclui apenas a utilização de algoritmos de procura em espaço de estados para encontrar a solução mais rápida a um dado estado do problema. Sendo assim, esta implementação não permite que o jogo seja propriamente jogado.

1.1 Ficheiros *Lisp*

O projeto está separado em 3 ficheiros lisp, um ficheiro para importar os problemas a serem testados e um ficheiro onde serão escritas as estatísticas de cada execução de um algoritmo.

- Ficheiros .lisp
  - procura.lisp - Inclui uma implementação genérica dos algoritmos de procura em espaço de estados implementados. Neste projeto foram implementados o Breadth-first-search, Depth-first-search e A*.
  - puzzle.lisp - Inclui código relacionado com problema, I.e. funções especificas ao domínio de aplicação, assim como operadores e heurísticas.
  - projeto.lisp - Inclui o código relacionado com a interação com o utilizador, assim como toda a escrita e leitura de ficheiros.

1.2 Ficheiros *Dat*

O projeto contém também 2 ficheiros ".dat" que guardam dados relativos aos projeto.

- problemas.dat - Ficheiro que contém os problemas que serão posteriormente importados para o projeto. Cada problema deve estar escrito de forma independente do outro (não devem estar encapsulados dentro de uma lista maior) separados através de um separador legal. A aplicação apenas efetua operações de *Read* neste ficheiro.
- log.dat - Ficheiro onde serão escritos todos os dados estatisticos relativos a cada execução de um algoritmo sobre um problema. Estas estatísticas serão especificadas e analisadas no capítulo 4 Análise dos problemas. A aplicação apenas efetua operações de *Write* neste ficheiro.

2 Implementação do problema

Neste capítulo será feita uma visão global das decisões de implementação tomadas, tanto na forma como as estruturas de dadas foram estruturadas, como nas funções auxiliares mais relevantes para o funcionamento do jogo.

2.1 Estrutura de dados

O jogo do Quatro é constituído por um tabuleiro de dimensões 4 x 4 e pecas que podem estar dentro do mesmo, ou fora (na chamada reserva). Para representar um tabuleiro vazio, onde todas as peças estão na reserva, iriamos utilizar a seguinte estrutura de dados:

```
(
    ; Tabuleiro
    (
        (0 0 0 0)
        (0 0 0 0)
        (0 0 0 0)
        (0 0 0 0)
    )
    ; Reserva
    '(
        (branca quadrada alta cheia)
        (branca quadrada alta oca)
        (branca quadrada baixa oca)
        (branca quadrada baixa cheia)
        (preta quadrada alta cheia)
        (preta quadrada alta oca)
        (preta quadrada baixa oca)
        (preta quadrada baixa cheia)
        (branca redonda alta cheia)
        (branca redonda alta oca)
        (branca redonda baixa cheia)
        (branca redonda baixa oca)
        (preta redonda alta cheia)
        (preta redonda alta oca)
        (preta redonda baixa cheia)
        (preta redonda baixa oca)
```

```
    )
)
```

Aqui identificamos uma lista que contém duas sublistas.

- A primeira sublista representa o tabuleiro do jogo, quatro listas com quatro posições, onde a posição 0 representa uma célula vazia.
- A segunda sublista representa as peças na reserva, i.e. peças que ainda não foram jogadas.

Esta será a estrutura utilizada para representar um estado de um problema no ficheiro problemas.dat. Uma vez carregados para o projeto, esta sofre pequenas alterações. É acrescentado, no fim, a profundidade, o valor da heurística e o nó pai.

3 Algoritmos de procura

Neste projeto foram implementados 3 algoritmos de procura em espaço de estados, 2 algoritmos não informados e um algoritmo informado. Entende-se por algoritmo informado um algoritmo que utiliza uma função de heurística para estimar a proximidade de um nó até ao nó solução.

3.1 Procura não informada

Foram implementados 2 algoritmos de procura não informada. O *Breadth-first-search*, ou algoritmo de procura em largura, e o *Depth-first-search*, ou algoritmo de procura em profundidade. Estes algoritmos são bastante similares na medida em que ambos percorrem todos os nós até chegar à solução. A diferença chave é que o *BFS* exausta cada nível de profundidade antes de avançar para o nível seguinte, enquanto o *DFS* dá prioridade aos nós mais fundos, exaustando os níveis de profundidade superiores face a qualquer outro nó.

3.1.1 Implementação do *BFS* e *DFS*

Como foi explicado anteriormente, existe uma diferença significativa na navegação dos algoritmos na árvore de espaço de estados, porém esta diferença em termos de código é minima, o que significa que foi apenas desenvolvido um algoritmo, uma implementação genérica de um algoritmo de procura, cuja única diferença é a ordenação dos sucessores com a lista de abertos. Vejamos a implementação do projeto:

```
(defun procura-nao-informada (algoritmo solucao sucessores operador profundidade
abertos fechados)
"Executa o algoritmo de procura nao informada bfs ou dfs (passado por parametro)"
    ; Identificar o tipo de algoritmo
    (let    ((abertos-algoritmo
                (ecase algoritmo
                    (bfs 'abertos-bfs)
                    (dfs 'abertos-dfs)
                ))
            (elemento (car abertos)))
        (cond
            ((null abertos) nil)
            ; Se o nó atual existe em fechados, ignoramos esta iteração
            ((no-existep elemento fechados) (procura-nao-informada algoritmo
```

```
solucao sucessores operador profundidade (cdr abertos) fechados))
            (t
                (let* (
                        ; Identificamos os sucessores do nó atual (agora
expandido)
                        (nos-sucessores (funcall sucessores elemento operador
algoritmo profundidade))
                        ; Verificamos se algum dos seus sucessores é a solução
                        (solucoes (remove nil (mapcar #'(lambda (sucessor)
                                (if (equal t (funcall solucao (tabuleiro
sucessor))) sucessor nil)
                            )nos-sucessores)))
                    )
                    (if (null solucoes)
                        ; Se não encontrar solução, é feita uma chamada recursiva
ao algoritmo, removendo o nó atual da lista de abertos e adicionando-o a fechados
                        (procura-nao-informada algoritmo solucao sucessores
operador profundidade
                                    (funcall abertos-algoritmo (cdr abertos) nos-
sucessores) (cons elemento fechados))
                        ; Em caso de solução, apresentamos a primeira solução
encontrada, assim como quantos nós foram expandidos e quantos foram gerados, pelo
algoritmo.
                        (list
                            (car solucoes)
                            (+ (length fechados) 1) ; expandidos, fechados + 1
porque o no elemento nao foi adicionado
                            (+ (+ (- (length abertos) 1) (length nos-sucessores))
(+ (length fechados) 1)) ; gerados, abertos + sucessores porque  os sucessores
gerados nesta iteracao nao foram contados
                        )
                    )
                )
            )
        )
    )
)
```

O algoritmo está devidamente comentado para este manual, mas passo a explicar mais detalhadamente. Este algoritmo, sendo uma abstração do BFS e DFS, recebe por argumento qual o algoritmo que deve ser executado. Na primeira instrução *let*, vemos a seleção do algoritmo. Esta seleção está a escolher uma de duas funções existentes:

```
(defun abertos-bfs (abertos sucessores)
"Devolve a juncao da lista abertos e sucessores, conforme o algoritmo de pesuqisa
em largura"
    (append abertos sucessores)
    ; Sendo que o BFS é uma pesquisa em largura, damos prioridade aos nós que
foram gerados primeiro, e só depois os sucessores
)
(defun abertos-dfs (abertos sucessores)
```

```
"Devolve a juncao da lista abertos e sucessores, conforme o algoritmo de pesuqisa
em profundidade"
    (append sucessores abertos)
    ; Sendo o DFS uma pesquisa em profundidade, damos prioridade aos nós
sucessores, que serão, por natureza, de profundidade maior
)
```

Estas funções são responsáveis por organizar a ordem pela qual a árvore é percorrida. Portanto, a abstração feita do algoritmo geral, revolve sobre estas duas funções que devolvem a ordem da lista de abertos que será aplicada na chamada recursiva da função. O corpo do algoritmo é o seguinte:

```
- Identificar algoritmo de procura
- Retirar primeiro elemento de abertos
    - Se abertos vazio, retorna nil
    - Se o elemento retirado existe em fechados, é feita uma chamada recursiva,
ignorando esta iteração.
- Gerar sucessores
- Verificar se algum sucessor é solução
    - Se SIM, devolve a primeira solução encontrada
    - SENÃO, chamada recursiva à função, em que o argumento abertos é a função de
ordenação do algoritmo escolhido, e adiciona-se o elemento atual aos fechados.
```

3.2 Procura informada

Foi implementado apenas um algoritmo de procura informada, A*, ou a estrela. Este algoritmo designa-se de "informado" devido à chamada de uma função de heurística para cada nó, que visa a estimar o quão perto o dado nó está de ser o nó solução. Esta estimativa nunca deverá ultrapassar o custo real do nó até à solução.

3.2.1 Implementação do A*

Este algoritmo segue a estrutura do algoritmo de procura ordenada, onde a diferença está na função do custo pela qual os nós são ordenados. Aqui, os nós de abertos são ordenados com base na função de avaliação:

```
f(n) = g(n) + h(n)
```

onde g(n) é o custo do nó n, e h(n) é o valor heurístico do nó n, obtido pela utilização da função de heurística implementada.

A implementação do algoritmo A* é a seguinte:

```
(defun a-estrela (algoritmo solucao sucessores operador profundidade heuristica
abertos fechados)
"Executa o algoritmo de procura informada a-estrela"
    (let ((elemento (car abertos)))
        (cond
            ((null abertos) nil)
```

```
            ; Se o nó atual existe em fechados, ignoramos esta iteração
            ((no-existep elemento fechados) (a-estrela algoritmo solucao
sucessores operador profundidade heuristica (cdr abertos) fechados))
            (t
                (let* (
                        ; Identificamos os sucessores do nó atual (agora
expandido)
                        (nos-sucessores (funcall sucessores elemento operador
algoritmo profundidade heuristica))
                        ; Verificamos se algum dos seus sucessores é a solução
                        (solucoes (remove nil (mapcar #'(lambda (sucessor)
                                (if (equal t (funcall solucao (tabuleiro
sucessor))) sucessor nil)
                                )nos-sucessores)))
                    )
                    (if (null solucoes)
                        (a-estrela algoritmo solucao sucessores operador
profundidade heuristica
                                    (abertos-a-estrela (cdr abertos) nos-
sucessores) (cons elemento fechados))
                        (list
                            (car solucoes)
                            (+ (length fechados) 1) ; expandidos, fechados + 1
porque o no elemento desta iteracao nao foi adicionado mas foi expandido
                            (+ (+ (- (length abertos) 1) (length nos-sucessores))
(+ (length fechados) 1)) ; gerados, abertos + sucessores porque  os sucessores
gerados nesta iteracao nao foram contados
                        )
                    )
                )
            )
        )
    )
)
```

A diferença surge na aplicação da função `abertos-a-estrela`, que insere cada sucessor em abertos por ordem do seu valor heurístico. Vejamos esta função:

```
(defun abertos-a-estrela (abertos sucessores)
"Insere, de forma ordenada, os sucessores em abertos.
Devolve a lista resultante."
    (cond
        ; Quando não houverem mais sucessores, devolve abertos
        ((null sucessores) abertos)
        ; Insere o primeiro elemento de sucessores em abertos, ordenado pelo
retorno da função no-custo
        (t (abertos-a-estrela (insere-ordenado (car sucessores) abertos 'no-custo)
(cdr sucessores)))
    )
)
```

Esta função recursiva vai, para cada sucessor, inserir na lista de abertos um sucessor de forma ordenada com base na função `no-custo`. Assim, a lista retornada contém os sucessores e abertos ordenados dos mais prováveis de estarem perto do nó solução até aos menos prováveis.

3.3 Heurísticas

Heurísticas são funções, especificas ao domínio do problema, cujo objetivo é avaliar o grau de interesse de um dado nó, isto é, o quão perto um nó se poderá encontrar de uma solução. Estas funções nem sempre resultam, na medida em que não existe uma fórmula para o seu desenvolvimento, pelo que requerem um conhecimento aprofundado do problema.

Para este projeto foram implementadas duas heurísticas. Estas vão ser explicadas neste capítulo, mas apenas mais tarde durante a análise dos problemas, é que os seus resultados serão comparados. Deste ponto em diante, vou referir-me às heurísticas como heurística base, para a heurística dada no enunciado do projeto, e heurística nova para a heurística implementada por mim.

3.3.1 Heurística base

Esta primeira heurística é a heurística pedida no enunciado do projeto. Em código é designada pela função `heuristica-base` que é chamada com o argumento que representa um jogo. A fórmula é seguinte:

```
h(x) = 4 - p(x)
```

Onde p(x) representa o numero máximo de peças com características comuns já alinhadas nas direções válidas para a solução.

3.3.2 Heurística nova

Esta segunda heurística é uma implementação mais genérica do que a anterior. Enquanto a anterior procura o numero máximo de peças com características comuns já alinhadas, esta heurística apenas procura, entre as 10 linhas válidas, quantas destas linhas têm 2 ou 3 peças, independentemente das suas características.

```
h(x) = 10 - p(x)
```

Onde p(x) representa a função descrita.

4 Análise de execuções dos algoritmos

O ficheiro logs.dat contém todos os valores indicados nesta tabela. Todos os algoritmos e heurísticas foram executados para cada um dos 6 problemas e os resultados são os apresentados nas seguintes tabelas. Para os algoritmos de procura informados, a heurística utilizada encontra-se entre parêntesis, após o nome.

Ainda antes de apresentar as tabelas de execução, queria deixar dois comentários relativos aos dados observados.

- O BFS no problema 6 atinge o limite do heap size para a versão grátis do LispWorks. Isto faz sentido porque, certamente que a versão grátis impõe uma limitação de memória que impede o BFS de percorrer os nós até encontrar a solução, visto ter que percorrer necessariamente os primeiros 3 níveis de profundidade.
- Alguns tempos de execução apresentam apenas 0ms. O algoritmo foi corrido várias vezes mas o resultado foi sempre 0, por isso vou assumir daqui em diante que estes algoritmos ocorrem quase instantaneamente.

BFS	**1**	**2**	**3**	**4**	**5**	**6**
Expandidos	1	2	11	149	283	X
Gerados	17	42	722	14241	53959	X
Penetrância	0.05882353	0.04761905	0.002770083	2.1065936E-4	5.5597768E-5	X
Ramificação	16.998291	5.999756	26.373291	23.895264	37.44507	X
Tempo de Execução	0ms	0ms	16ms	47ms	312ms	X

DFS	**1**	**2**	**3**	**4**	**5**	**6**
Expandidos	1	2	3	4	4	4
Gerados	17	42	195	363	735	847
Penetrância	0.05882353	0.04761905	0.015384615	0.011019284	0.005442177	0.00472255
Ramificação	16.998291	5.999756	5.4260254	4.071045	4.925537	5.1086426
Tempo de Execução	0ms	0ms	0ms	0ms	0ms	0ms

A* (Base)	**1**	**2**	**3**	**4**	**5**	**6**
Expandidos	1	2	2	149	115	4
Gerados	17	42	146	14241	19519	847
Penetrância	0.05882353	0.04761905	0.01369863	2.1065936E-4	1.536964E-4	0.00472255
Ramificação	16.998291	5.999756	11.590576	23.895264	26.58081	5.1086426
Tempo de Execução	0ms	0ms	0ms	1422ms	422ms	16ms

A* (Nova)	**1**	**2**	**3**	**4**	**5**	**6**
Expandidos	1	2	2	4	4	260
Gerados	17	42	146	363	735	58366
Penetrância	0.05882353	0.04761905	0.01369863	0.011019284	0.005442177	8.566631E-5
Ramificação	16.998291	5.999756	11.590576	4.071045	4.925537	8.758545

A* (Nova)	1	2	3	4	5	6
Tempo de Execução	0ms	0ms	0ms	0ms	16ms	531ms

Entre os algoritmos de procura não informados, o Depth-First-Search apresentou consistentemente um fator de ramificação inferior ao Breadth-First-Search, dado que gera consideravelmente menos nós. O algoritmo A*, até ao problema 3 inclusive, tem uma melhor performance que os anteriores. A partir daqui, começam a aparecer discrepâncias nos dados estatísticos segundo heurísticas diferentes. A heurística base, dada no enunciado do projeto, foi consideravelmente mais rápida a resolver o problema 6, onde gerou e expandiu significativamente menos nós. Já a heurística nova, apresentou resultados melhores para o problema 4 e 5, tanto em termos de tempo de execução, como em nós gerados e consequentemente fatores de ramificação inferiores.

5 Limitações técnicas

Este projeto tem várias limitações que não devem ser desvalorizadas para o funcionamento correto do projeto.

- Não são feitas verificações aos inputs. Isto é, qualquer input do utilizador não é validado e será corrido pelo programa, o que poderá despoletar um erro no projeto. É, no entanto, apresenta a lista de opções válidas, sempre que é pedido input do utilizador.
- Quando qualquer algoritmo é executado pelo utilizador, é necessário definir qual a heurística, mesmo que esta seja irrelevante para o algoritmo escolhido.
- Executar o algoritmo a-estrela no problema 4 causa um stack overflow, onde é necessário utilizar fazer uma chamada à função `(continue)` 5 vezes, diretamente no REPL para o algoritmo prosseguir.
- O algoritmo A* não faz a verificação de nós com custo mais baixo em abertos e fechados.
- Explicado no capítulo anterior, o BFS não resolve o problema 6, onde eu estimo que seja devido às limitações de memória da versão gratuita LispWorks.
- Definir a profundidade máxima, em certos problemas faz com que o algoritmo demore demasiado tempo a ser resolvido, tendo que ser necessário reiniciar o LispWorks.