Manual Técnico

Projeto de Inteligência Artificial

Blokus Uno - Primeira Fase

Autores:

Tiago Farinha (201802235)

Francisco Moura (201802033)

Docente:

Eng. Filipe Mariano

Índice do Manual

Introdução

Este manual técnico conterá os detalhes de implementação da primeira fase do projeto **Blokus Uno**, realizado no âmbito da disciplina de Inteligência Artificial.

Estrutura do Programa

Este projeto estará dividido em 3 ficheiros diferentes, cada um com um propósito distinto.

Estes três ficheiros são os seguintes:

- **projeto.lisp** - ficheiro responsável pela interação com o utilizador, leitura e escrita de ficheiros e estatísticas.
- **procura.lisp** - ficheiro responsável pelos algoritmos de procura, isto é, *Breadth-first Search* (BFS) e *Depth-first Search* (DFS), tal como funções auxiliares por estes usados.
- **puzzle.lisp** - ficheiro responsável pela adaptação do problema Blokus Uno aos algoritmos de procura genéricos.

Projeto.lisp

As funções desenvolvidas neste ficheiro são de interação com o utilizador, estatísticas dos algoritmos implementados e de leitura e escrita em ficheiros externos.

O programa inicia-se com a função **iniciar**. Esta chamará a função **menu-inicial** que imprimirá no ecrã o menu e posteriormente recolherá a escolha do utilizador.

```
(defun iniciar()
"Função que imprime o menu e espera pelo input do utilizador, redirecionando-o depois à 'página'
seguinte"
  (menu-inicial)
  (let ((comando (read)))
    (cond
      ((not (numberp comando)) (format t "Insira um número!~%") (iniciar))
      ((= comando 0) t)
      ((= comando 1) (executar-subcomando-problema))
      ((= comando 2) (ver-tabuleiros-comando))
      (t (format t "Comando Inválido! Insira 1 para resolver um problema ou 0 para sair da
aplicação...~%") (iniciar))
    )
  )
  (values)
)
```

A função **executar-subcomando-problema** é para recolher a escolha do tabuleiro a ser utilizado.

```
(defun executar-subcomando-problema ()
"Função que imprime o sub-menu e espera pelo input do utilizador, redirecionando-o depois à
'página' seguinte"
  (menu-escolha-problema)
  (let ((comando (read)))
    (cond
      ((not (numberp comando)) (format t "~%Insira um número!~%") (executar-subcomando-
problema))
      ((= comando 0) (iniciar))
      ((= comando 1) (executar-subcomando-algoritmo (first (ler-ficheiro-problemas)) comando))
      ((= comando 2) (executar-subcomando-algoritmo (second (ler-ficheiro-problemas)) comando))
      ((= comando 3) (executar-subcomando-algoritmo (third (ler-ficheiro-problemas)) comando))
```

```
      ((= comando 4) (executar-subcomando-algoritmo (fourth (ler-ficheiro-problemas)) comando))
      ((= comando 5) (executar-subcomando-algoritmo (fifth (ler-ficheiro-problemas)) comando))
      ((= comando 6) (executar-subcomando-algoritmo (tabuleiro-vazio) comando))
      (t (format t "~%Comando Inválido! Insira entre 1 e 6 para resolver um problema ou 0 para
sair da aplicação...~%") (executar-subcomando-problema))
    )
  )
  (values)
)
```

Foi criada a função **ver-tabuleiros-comando** que recolhe do utilizador o desafio que este quer ver e mostra usando a função **mostrar-tabuleiros** com o propósito de mostrar ao utilizador os problemas (tabuleiros) disponíveis.

```
(defun ver-tabuleiros-comando ()
"Função que imprime o layout do menu dos tabuleiros dos desafios e que permite voltar ao menu
inicial."
  (mostrar-tabuleiros)
  (let ((comando (read)))
    (cond
      ((not (numberp comando)) (format t "~%Insira um número!~%") (ver-tabuleiros-comando))
      ((= comando 0) (iniciar))
      (t (format t "~%Comando Inválido! Insira 0 se deseja voltar ao menu inicial...~%") (ver-
tabuleiros-comando))
    )
  )
  (values)
)
```

A função **executar-subcomando-algoritmo** será a responsável por recolher a escolha do algoritmo a usar na resolução do problema.

```
(defun executar-subcomando-algoritmo (problema &optional numeroProblema)
"Função que imprime o sub-menu e espera pelo input do utilizador, redirecionando-o depois à
'página' seguinte"
  (menu-escolha-algoritmo)
  (let ((comando (read)))
    (cond
      ((not (numberp comando)) (format t "~%Insira um número!~%") (executar-subcomando-algoritmo
problema))
      ((= comando 0) (executar-subcomando-problema))
      ((= comando 1) (let ((resultado (list (GET-INTERNAL-RUN-TIME) (bfs 'no-solucaop 'nos-
possiveis-todas-pecas (list (cria-no problema '(10 10 15)))) (GET-INTERNAL-RUN-TIME)
numeroProblema))) (progn (escrever-estatisticas resultado 'bfs) (iniciar))))
      ((= comando 2) (executar-escolha-profundidade problema numeroProblema))
      ((= comando 3) (let ((resultado (list (GET-INTERNAL-RUN-TIME) (dfs 'no-solucaop 'nos-
possiveis-todas-pecas 100 (list (cria-no problema '(10 10 15)))) (GET-INTERNAL-RUN-TIME)
numeroProblema))) (progn (escrever-estatisticas resultado 'dfs2) (iniciar))))
      (t (format t "~%Comando Inválido! Insira 1 ou 2 para resolver o desafio com o respetivo
algoritmo ou 0 para sair da aplicação...~%") (executar-subcomando-algoritmo problema))
    )
  )
  (values)
)
```

Depois do algoritmo escolhido tentar resolver o problema selecionado, as estatísticas dessa resolução serão guardadas pela função **escrever-estatisticas** no caminho *C:\Projeto IA\estatisticas.dat*

```
(defun escrever-estatisticas (resultado algoritmo)
"Função que escreve as estatistícas de desempenho da realização de um desafio com um algoritmo"
  (let ((tempoInicio (first resultado))
        (solucao (second resultado))
        (tempoFim (third resultado))
        (desafio (fourth resultado)))

          (with-open-file (ficheiro (make-pathname :host "c" :directory '(:absolute "Projeto
IA") :name "estatisticas" :type "dat") :direction :output :if-exists :append :if-does-not-exist
:create)
             (progn
               (cond
                ((= desafio 1) (format ficheiro "~%| Resolução do Tabuleiro A |"))
                ((= desafio 2) (format ficheiro "~%| Resolução do Tabuleiro B |"))
                ((= desafio 3) (format ficheiro "~%| Resolução do Tabuleiro C |"))
                ((= desafio 4) (format ficheiro "~%| Resolução do Tabuleiro D |"))
                ((= desafio 5) (format ficheiro "~%| Resolução do Tabuleiro E |"))
                ((= desafio 6) (format ficheiro "~%| Resolução do Tabuleiro F |"))
                (t (format ficheiro "~%* Resolução do Tabuleiro N/A"))
               )
               (cond
                ((equal algoritmo 'bfs) (format ficheiro "~%Algoritmo: Breadth-First Search
(Largura)"))
                ((equal algoritmo 'dfs) (format ficheiro "~%Algoritmo: Depth-First Search
(Profundidade)"))
                ((equal algoritmo 'dfs2) (format ficheiro "~%Algoritmo: Depth-First Search
Modificado (Profundidade)")))
               (format ficheiro "~%Número de nós gerados > ~a" (+ (second solucao) (third
solucao)))
               (format ficheiro "~%Número de nós expandidos > ~a" (second solucao))
               (format ficheiro "~%Penetrância > ~10f" (/ (no-profundidade (first solucao)) (+
(second solucao) (third solucao))))
               (if (equal algoritmo 'dfs)
                    (format ficheiro "~%Profundidade escolhida > ~a" (fifth resultado))
               )
               (format ficheiro "~%Tempo de Execução > ~a ms" (- tempoFim tempoInicio))
               (format ficheiro "~%Caminho até ao fim do jogo:~%~%")
               (formatar-solucao-ficheiro (solucaoProblema (first solucao)) ficheiro)
             )
          )
  )
)
```

Procura.lisp

Algoritmo Breadth-first Search

A função que implementa o algoritmo de procura em largura é o **BFS**. Este algoritmo procura pela solução na lista de nós abertos de forma recursiva, ao mesmo tempo que vai expandindo cada um dos nós em que entra sequencialmente, colocando-os no final da lista de nós abertos. Fará isto até encontrar a solução.

```
(defun bfs (funFimDoJogo funSucessores abertos &optional (fechados '()))
"Implementação do Algoritmo Breadth-First Search"
  (if (equal (length abertos) 0)
    nil
    (let ((n (push (pop abertos) fechados)))
      (let ((filhos (funcall funSucessores (first n))))
          (if (equal (funcall funFimDoJogo (first n)) t) (progn (formatar-solucao
(solucaoProblema (first n))) (list (first n) (length abertos) (length fechados))) (bfs
funFimDoJogo funSucessores (nconc abertos filhos) fechados))
        )
    )
  )
)
```

Algoritmo Depth-first Search

A função que implementa o algoritmo de procura em profundidade é o **DFS**. Este algoritmo procura pela solução no primeiro nó expandido da lista de nós abertos recursivamente, expandindo esse, colocando os seus sucessores no inicio da lista aberta e expandindo novamente o primeiro nó dessa lista. Fará isto até atingir a profundidade máxima pretendida. Caso a solução não seja encontrada até esse valor, repetirá o processo para o nó aberto seguinte que esteja na lista e não ultrapasse a profundidade máxima.

```
(defun dfs (funFimDoJogo funSucessores profMax abertos &optional (fechados '()))
"Implementação do Algoritmo Depth-First Search"
  (if (equal (length abertos) 0)
    nil
    (let ((n (push (pop abertos) fechados)))
      (let ((filhos (funcall funSucessores (first n))))
        (if (equal profMax (no-profundidade (first n)))
          (dfs  funFimDoJogo funSucessores profMax abertos fechados)
          (if (equal (funcall funFimDoJogo (first n)) t) (progn (formatar-solucao
(solucaoProblema (first n))) (list (first n) (length abertos) (length fechados))) (dfs
funFimDoJogo funSucessores profMax (nconc filhos abertos) fechados)
          )
        )
      )
    )
  )
)
```

Foram criadas várias outras funções auxiliares de apoio aos algoritmos.

A função **solucaoProblema** devolve o caminho da solução ao estado inicial do problema.

```
(defun solucaoProblema (noSolucao)
"Função que retorna a lista do caminho da solução."
  (cond
   ((equal (no-pai noSolucao) nil) (list (first noSolucao)))
   (t (append (solucaoProblema (no-pai noSolucao)) (list (first noSolucao))))
 )
)
```

As funções **formatar-solucao** e **formatar-solucao-ficheiro** fazem essencialmente o mesmo, diferindo em imprimiri a solução formatada para o ecrã ou para um ficheiro.

```
(defun formatar-solucao (solucao)
"Formata o caminho solução em forma de tabuleiros."
  (if (not (equal solucao nil))
    (progn (format t " _____~%")
           (format t "|                                       |~%")
           (format t "|      A B C D E F G H I J K L M N    |")
           (format t "~%")
           (let ((linha 0))
           (mapcar #'(lambda (atual) (progn (setf linha (1+ linha))
                                            (if (< linha 10) (format t "|   ~a ~a   |~%" linha
atual)
                                            (format t "|  ~a ~a   |~%" linha atual)))) (first
solucao)))
           (format t "|_____|~%")
           (format t "~%") (formatar-solucao (rest solucao))
           (values))
  )
)

(defun formatar-solucao-ficheiro (solucao ficheiro)
"Formata o caminho solução em forma de tabuleiros para um ficheiro."
  (if (not (equal solucao nil))
    (progn (format ficheiro " _____~%")
           (format ficheiro "|                                       |~%")
           (format ficheiro "|      A B C D E F G H I J K L M N    |")
           (format ficheiro "~%")
           (let ((linha 0))
           (mapcar #'(lambda (atual) (progn (setf linha (1+ linha))
                                            (if (< linha 10) (format ficheiro "|   ~a ~a   |~%"
linha atual)
                                            (format ficheiro "|  ~a ~a   |~%" linha atual))))
(first solucao)))
           (format ficheiro "|_____|~%")
           (format ficheiro "~%") (formatar-solucao-ficheiro (rest solucao) ficheiro))
  )
)
```

A função **no-solucaop** devolve verdadeiro caso o nó que recebe não tem mais sucessores possíveis, indicando que esta é uma solução.

```
(defun no-solucaop (no)
"Função que retorna t se o nó recebido por parâmetro é nó solução. Nil caso contrário."
  (if (equal (length (nos-possiveis-todas-pecas  no)) 0) t nil)
)
```

A função **nos-posicoes-possiveis** devolve uma lista com os nós sucessores de um nó para uma peça especifica.

```
(defun nos-posicoes-possiveis(no peca)
"Função que verifica e coloca as jogadas possíveis (em nós - (cria-no no)) numa lista a ser
usada, mais tarde, nos algoritmos."
    (if (> (cond
                ((equal peca 'peca-a) (first (nr-pecas no)))
                ((equal peca 'peca-b) (second (nr-pecas no)))
                ((or (equal peca 'peca-c-1) (equal peca 'peca-c-2)) (third (nr-pecas no)))) 0)

        (mapcar #'(lambda(sucessorAtual)
                            (cria-no sucessorAtual (decrementar-nr-pecas peca (nr-pecas no)) (1+
(no-profundidade no)) no)
                           )
                 (mapcar #'(lambda(coordAtual)
                             (verificar-peca-encaixa-e-colocar  (first coordAtual) (second
coordAtual) peca (no-estado no)))
                  (sucessores (no-estado no) 0 0 peca))
        )
               nil)
)
```

Puzzle.lisp

Este ficheiro contém as funções referentes às peças em si, validações dessas peças e modificações ao tabuleiro. É o ficheiro responsável por adaptar o problema aos algoritmos.

As funções **casa-vaziap** e **verifica-casas-vazias** são responsáveis por verificar se as peças não estão a ser inseridas em cima de outra peça.

```
(defun casa-vaziap (line row tray)
  "Função que recebe dois índices e o tabuleiro e verifica se a casa correspondente a essa
posição se encontra vazia, ou seja, igual a 0."
  (if (equal (celula line row tray) 0)
      T NIL
  )
)

(defun verifica-casas-vazias (tray positions)
  "Função que recebe o tabuleiro e uma lista de pares de índices linha e coluna e devolve uma
lista com T ou NIL caso se encontrem vazias."
  (mapcar #'(lambda (curr)
              (if (equal (casa-vaziap (first curr) (second curr) tray) T) T NIL)
            ) positions)
)
```

As funções **substituir-posicao** e **substituir** são responsáveis por colocar no tabuleiro uma peça, dada a coordenada inicial de uma peça.

```
(defun substituir-posicao (row lineList &optional (value 1))
  "Função que recebe um índice, uma lista e um valor (por default o valor é 1) e substitui pelo
valor pretendido nessa posição"
  (let ((currPos 0))
    (mapcar #'(lambda (curr)
                (if (and (= row currPos) (= curr 0)) (progn (setf currPos (1+ currPos)) value)
                                    (progn (setf currPos (1+ currPos)) curr)
                )
              )
   lineList)
  )
)

(defun substituir (line row tray &optional (value 1))
  "Função que recebe 2 índices, o tabuleiro e um valor (por default = 1) e retorna o tabuleiro
com a substituição pelo valor pretendido."
  (let ((currLine 0))
        (mapcar #'(lambda (curr)
                (if (= line currLine) (progn (setf currLine (1+ currLine)) (substituir-posicao
row (linha line tray) value))
                                      (progn (setf currLine (1+ currLine)) curr)
                )
              )
         tray)
  )
)
```

As principais funções deste ficheiro são **verificar-peca-encaixa-e-colocar** e **verificar-peca-encaixa**. A primeira chama a função **colocar-peca** caso se tenha verificado que a peça efetivamente pode ser colocada inteiramente nessa posição sem violar as regras do jogo.

```
(defun colocar-peca(peca line row tray)
"Função que chama uma das funções de colocação de uma peça"
  (cond
   ((equal peca 'peca-a) (peca-a line row tray))
   ((equal peca 'peca-b) (peca-b line row tray))
   ((equal peca 'peca-c-1) (peca-c-1 line row tray))
   ((equal peca 'peca-c-2) (peca-c-2 line row tray))
  )
)

(defun verificar-peca-encaixa-e-colocar (line row peca tray)
"Função que coloca a peça no tabuleiro, verificando se estas podem ser inseridas. Se todas as
validações forem positivas, adicionar peça ao tabuleiro. NIL caso contrário."
    (if (equal (verificar-peca-encaixa line row peca tray) nil)
        nil
        (colocar-peca peca line row tray)
    )
)

(defun verificar-peca-encaixa (line row peca tray)
"Função que coloca a peça no tabuleiro, verificando se estas podem ser inseridas."
  (let ((piece (peca-casas-ocupadas line row peca)))
```

```
    (if (not (member nil (mapcar #'(lambda (atual)
                (if
                  (and
                        (not (equal (celula (first atual) (1- (second atual)) tray) 1))
                        (not (equal (celula (first atual) (1+ (second atual)) tray) 1))
                        (not (equal (celula (1- (first atual)) (second atual) tray) 1))
                        (not (equal (celula (1+ (first atual)) (second atual) tray) 1))
                        (verificar-dentro-dos-limites (first atual)(second atual))
                        (casa-vaziap (first atual) (second atual) tray))
                  t nil
                )
              )piece)))
        (if (or (member t (mapcar #'(lambda (atual)
                (if
                  (and
                        (or (equal (celula (1- (first atual)) (1- (second atual)) tray) 1)
                            (equal (celula (1+ (first atual)) (1- (second atual)) tray) 1)
                            (equal (celula (1- (first atual)) (1+ (second atual)) tray) 1)
                            (equal (celula (1+ (first atual)) (1+ (second atual)) tray) 1)

                        )
                   )
                   t nil
                )
                ) piece))
      (member t (mapcar #'(lambda (atual)
                (if (and (equal (first atual) 0) (equal (second atual) 0))
                  t nil
                )
              ) piece)))
            t
)
     )
  )
)
```

Limitações

O IDE fornecido limita seriamente a capacidade do BFS e DFS, sendo que estes apenas resolvem o primeiro tabuleiro. Caso seja usado um número superior a 100, o DFS consegue resolver todos os tabuleiros sem qualquer problema de memória. Decidimos então criar um DFS modificado que apenas chama o DFS com 100 de profundidade afim de facilitar a resolução de todos os tabuleiros.

Não fomos capazes de implementar nem o algoritmo A* nem o bónus devido a falta de tempo. Não foram implementadas heurísticas pois não foram criados algoritmos para as usarem.

Não conseguimos calcular o fator de ramificação.

Resultados

Tabuleiro	Algoritmo	Nós Gerados	Nós Expandidos	Penetrância	Tempo de execução

Tabuleiro	Algoritmo	Nós Gerados	Nós Expandidos	Penetrância	Tempo de execução
A	BFS	34	23	0.05882353	16ms
A	DFS (d=5)	158	2	0.012658228	79ms
A	DFS (d=100)	26	17	0.30769232	16ms
B	DFS (d=100)	93	79	0.13978495	31ms
C	DFS (d=100)	117	101	0.12820514	16ms
D	DFS (d=100)	164	146	0.103658535	16ms
E	DFS (d=100)	159	136	0.13836478	31ms
F	DFS (d=100)	312	283	0.08974359	32ms