Instituto Superior de Engenharia de Lisboa

Documento Síntese 2

INTELIGÊNCIA ARTIFICIAL PARA SISTEMA AUTÓNOMOS

Ana Sofia Oliveira | 39275

Índice

Introdução

No âmbito da unidade curricular de Inteligência Artificial de Agentes Autónomos, foi desenvolvido o relatório em análise, com foco em desenvolvimentos de arquiteturas de agentes reativos e métodos de pesquisa em espaço de estados.

Durante o decorrer do relatório iram ser explicados termos inerentes à temáticas acima indicadas, nomeadamente arquitetura de agentes reativos, pesquisa em espaço de estados, métodos de pesquisa em espaço de estados, entre outros conceitos essenciais para o desenvolver da matéria.

Todos estes aspetos constituem o caso prático 2 da unidade curricular que consiste na resolução de um problema de trajetórias entre localidades e de um problema de resolução de um puzzle 3x3. Durante o decorrer deste relatório iram também ser abordados comuns e específicos da implementação e desenvolvimento prático.

Enquadramento Teórico

AGENTES REATIVOS

Agentes que com base num conjunto de regras, Reações, geram uma ação em função de estímulos obtidos do ambiente. As reações são caracterizadas por um par estimulo – resposta, ou seja, para um dado estimulo percecionado pelo agente devolvem uma ação a ser aplicada no ambiente (Figura 1).

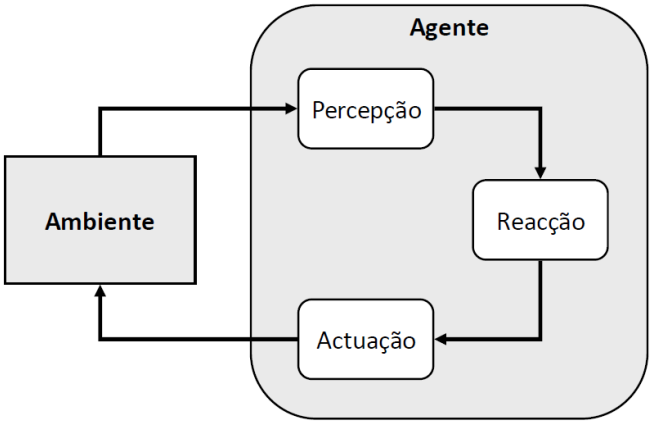


Figura 1 - Arquitetura de Agentes Reativos

PESQUISA EM ESPAÇO DE ESTADOS

O espaço de estados é o conjunto dos estados e de todas as transições eles, representando todas as possíveis evoluções de um dado sistema. Ou seja, assumindo o problema do planeador de trajetos, o espaço de estados vai corresponder à representação de todas as localidades e das ligações elas.

Desta forma o espaço de estado é caracterizado por:

- **Estado** – Conjunto das características que representam o domínio do problema num dado instante de tempo.
- **Transições** – Representam a aplicação de um dado operador num estado atual, de forma a gerar um estado sucessor.

Um espaço de estados pode ser representado de duas formas, através de um grafo de espaço de estados ou através de uma árvore de procura.

Grafo

Representação gráfica dos estados disponíveis do problema e transições possíveis de cada estado.

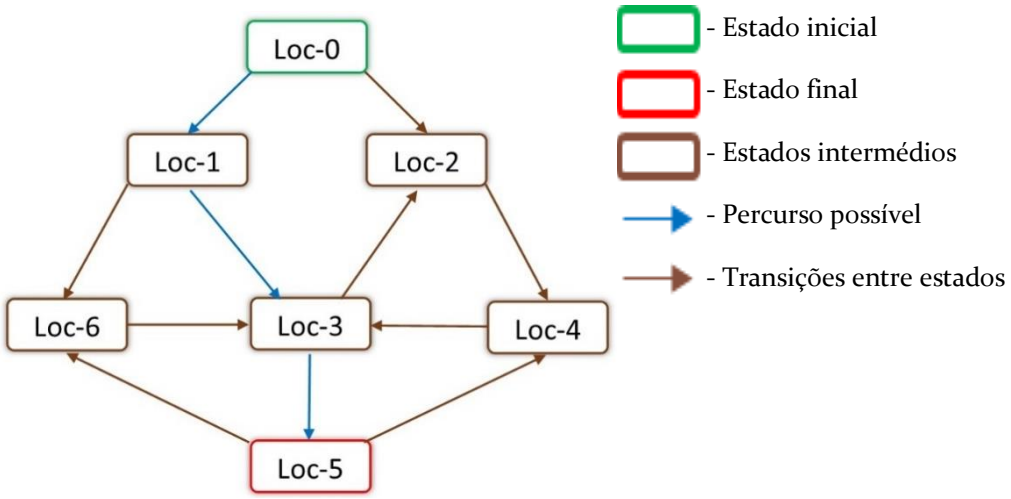


Figura 2 - Exemplo de Espaço de Estados para o Problema do Planeador de Trajetos

Árvore de Procura

Encadeamento de nós gerados numa procura sob o espaço de dados. Baseado numa representação gráfica em forma de árvore.

A árvore de procura é caracterizada por:

- **Nós** – Estados possíveis.
- **Operadores** – Ação que gera o Nó Sucessor.
- **Nós antecessores** – Nó "Pai" do Nó corrente.

Em cada nó, são guardadas estas três características, e assim, é possível obter todo o caminho do estado final ao estado inicial, que será o nosso resultado.

MÉTODOS DE PROCURA

Iniciando a procura no primeiro nó, é aplicado o processo de expansão, que consiste na aplicação dos operadores gerando novos nós.

Este processo é aplicado a todos os nós que ainda não foram expandidos (representados na fronteira de exploração) até se obter o nó objetivo, terminando a procura.

O processo de procura baseia-se, de forma cíclica, em 2 passos:

1. Extrair e avaliar-se o primeiro nó da lista é o objetivo;

   Se for, termina-se a procura;
2. Se não for, aplicar o processo de expansão ao nó sendo acrescentados os novos gerados à fronteira de exploração;

Pode ser implementado através do algoritmo da figura seguinte:

```
function GRAPH-SEARCH(problem, fringe) returns a solution, or failure

  closed ← an empty set
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
     if EMPTY?( fringe) then return failure
     node ← REMOVE-FIRST( fringe)
     if GOAL-TEST[ problem](STATE[node]) then return SOLUTION(node)
     if STATE[node] is not in closed then
        add STATE[node] to closed
        fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
```

Figure 3.19 The general graph-search algorithm. The set *closed* can be implemented with a hash table to allow efficient checking for repeated states. This algorithm assumes that the first path to a state *s* is the cheapest (see text).

[Russel & Norvig, 2003]

Procura Não Informada

Caracterizada por não ter conhecimento de informação do domínio do problema.

Há duas suportes da pesquisa não informada. A primeira recorrendo à análise de todos os nós sucessores do nó corrente e a segunda recorrendo à análise de um dos nós sucessores do nó corrente.

- ### Procura em Largura

  Caracteriza-se pela análise de todos os nós do nível N antes da análise dos nós do nível N+1.

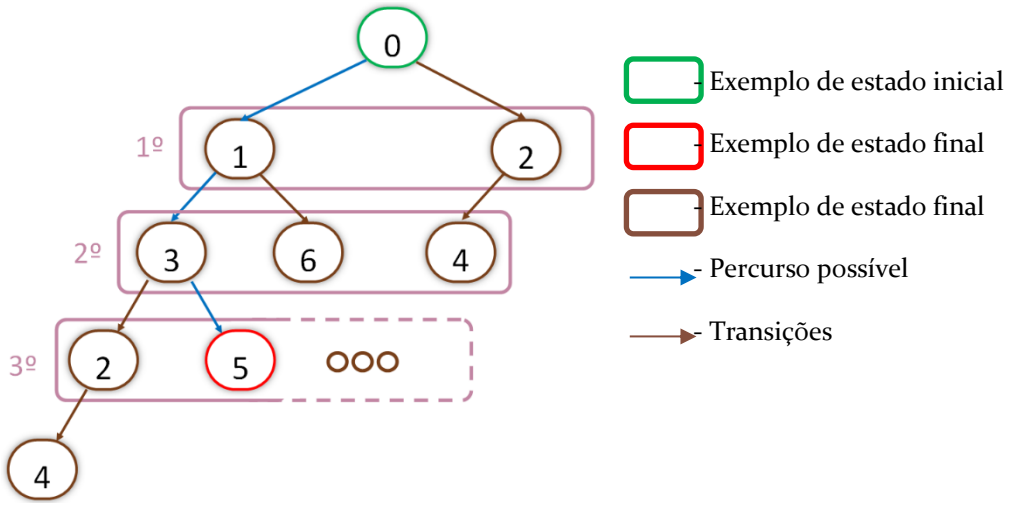
  Garante que sempre que esteja a ser analisado um nó com uma dada profundidade, todos os nós existentes nas profundidades anteriores já foram expandidos e analisados. Desta forma os nós gerados terão de ser inseridos no final da fronteira de exploração para sejam sempre explorados os nós mais antigos.

Pode ser representado através do algoritmo da figura seguinte:

```
function BREADTH-FIRST-SEARCH(problem) returns a solution or failure
  return GENERAL-SEARCH(problem,ENQUEUE-AT-END)
```

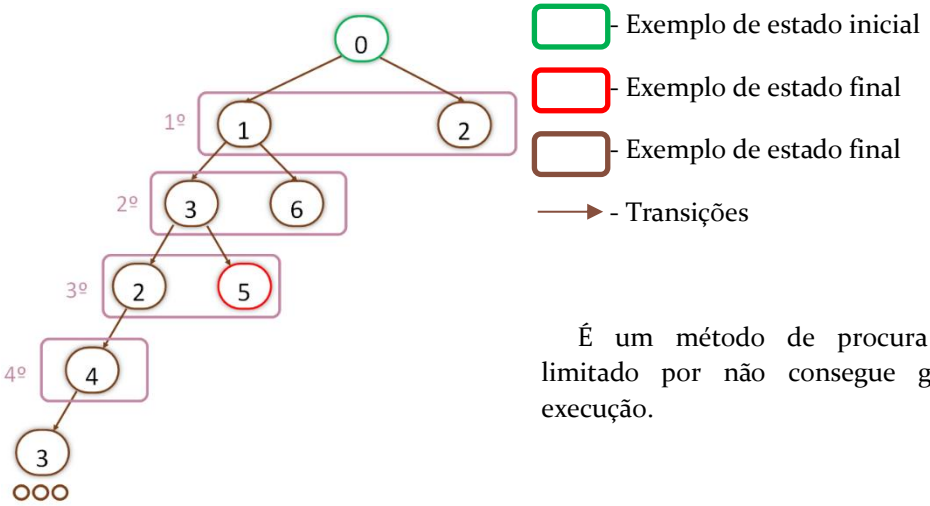
É um método de procura completo pois avalia todos os níveis de profundidade na procura garantindo assim que só para quando encontra uma solução. Por este mesmo motivo é também ótimo pois vai sempre encontrar a solução garantindo que esta se encontra no menor nível de profundidade.

Assumindo que todos os nós têm o mesmo número de sucessores, fator de ramificação (r), e representado o nível de do nó (n) podemos dizer que a complexidade temporal e espacial são na ordem de r^n.



- Procura em Profundidade

A procura em profundidade é outro método de exploração do grafo de espaço de estados que consiste em aplicar o processo de expansão ao nó mais recente. Sendo assim, os nós gerados pela expansão, são acrescentados ao início da lista dos nós de fronteira. critério de exploração, é a maior profundidade.



É um método de procura relativamente limitado por não consegue gerir ciclos de execução.

Não é um método de procura completo pois caso exista um ciclo na procura este fica em "loop" nunca encontrando solução. Por este mesmo motivo também não é um método ótimo.

Relativamente à complexidade temporal é na ordem de r^n. E em termos de complexidade espacial está na ordem de r*n.

- Procura em Profundidade Limitada

De forma a evitar os constrangimentos apresentados acima, em relação à procura em profundidade, foi desenvolvido este método de procura limitando a pesquisa em profundidade a um nível de profundidade máximo. Atingindo este limite é incrementado o valor de profundidade máximo por um valor de incremento de procura e é feita novamente a pesquisa em profundidade até este novo limite máximo de procura.

O seu algoritmo consiste na chamada repetitiva do algoritmo de pesquisa em profundidade limitada para valores de profundidade crescentes.

Desta forma este método é completo, pois deixam de poder ocorrer ciclos na pesquisa. E é ótimo se considerarmos o custo de cada transição o mesmo.

A complexidade espacial é de ordem r*n, e a complexidade temporal é de ordem r^n.

- Procura de Custo Uniforme

O método de procura de custo uniforme é uma transformação do método de procura em largura, no entanto escolhe analisar o nó na fronteira de exploração cujo custo é menor. Desta forma a fronteira deverá estar ordenada.

Assumindo que o custo de um caminho não pode diminuir à medida que vamos explorando o espaço de estados, podemos afirmar que também este método é completo e ótimo, pois garante que encontramos sempre solução e que esta solução tem associada o menor custo possível.

Relativamente à complexidade temporal e à complexidade temporal este método de procura apresenta as mesmas características que o método de procura em largura, na ordem de r^n.

Procura Heurística (Informada)

Pesquisa com base na informação do domínio do problema. A esta informação denominamos heurística (*h(n)* ou *f(n)*).

Consideramos uma heurística admissível quando esta é otimista, ou seja, a estimativa do custo desde o nó atual até ao nó objetivo, é menor do que custo efetivo mínimo total da solução. A melhor heurística, é a que menos restrições elimina.

- ### Procura Sôfrega

  Consiste na escolha do primeiro nó da fronteira de exploração que apresenta ser o mais promissor de acordo com a função h(n), estimativa do custo até à solução. A fronteira de exploração tem então de se manter ordenada de acordo com os valores de h(n).

  Não é um método completo e por sua vez também não é ótimo, pois basta que na sua execução hajam ciclos para que não encontre qualquer solução.

  A complexidade temporal, bem como a complexidade espacial são na ordem de r^n.

- ### Procura A*

  Neste método de procura é usada como função heurística o conhecimento do futuro (h(n)) e do passado (g(n)). Desta forma obtemos a função f(n) como a soma entre o passado e o futuro do domínio do problema.

  Desta forma, e garantido uma heurística admissível, o método de procura é tanto completo como ótimo.

Como escolher o melhor método de procura? E as suas limitações

As características que devemos ter em conta quando analisamos os métodos de procura são a complexidade temporal (número de nós expandidos) e complexidade espacial (número máximo de nós na fronteira de exploração).

Devemos também ter em conta se o método de procura é completo, ou seja, encontra sempre solução. E se é ótimo, ou seja, sempre que encontre solução esta solução é a melhor.

Assim sendo construímos a seguinte tabela que evidencia todos estes aspetos:

Método de Procura	**Complexidade Temporal**	**Complexidade Espacial**	**Ótimo**	**Completo**
Largura	$O(b^d)$	$O(b^d)$	Sim	Sim
Profundidade	$O(b^m)$	O(bm)	Não	Não
Profundidade Iterativa	$O(b^d)$	O(bd)	Sim	Sim
Custo Uniforme	$O(b^d)$	$O(b^d)$	Sim	Sim
Sôfrega	$O(b^d)$	$O(b^d)$	Não	Não

MEMÓRIA DE PROCURA

A memória de procura, é criada para evitar ciclos, criando uma noção dos nós que estão na fronteira de exploração, ou uma lista de nós que já foram explorados para que não seja necessário voltar a explorar os mesmos nós, evitando desperdício de recursos repetindo processos que já não são necessários, nem gasto de memória de coisas que já são conhecidas (nós iguais). Dependendo do método de procura, os novos nós gerados pelas transições, são colocados no início ou no fim da lista dos nós da fronteira.

FIFO –"First In, First Out"

Memória FIFO (primeiro a entrar, é o primeiro a sair), cumpre com a ordem de acrescentar os nós ao final da lista, permitindo que sejam explorados primeiro, os nós mais antigos. Exemplo: método de procura em largura.

LIFO – "Last in, First Out"

Memória LIFO (último a entrar, é o primeiro a sair), cumpre com a ordem de acrescentar os nós ao início da lista, permitindo que sejam explorados primeiro, os nós mais recentes. Exemplo: método de procura em profundidade.

Enquadramento Prático

PACKAGE "PEE"

Classe Solucao

Estende a classe Java Iterable, para que possa a solução ser iterada no ciclo.

Tem os métodos getDimensão() e getCusto() que iram ser definidos nas classes que implementem esta classe.

Classe PassoSolucao

É uma interface que define os métodos getEstado(), getOperador() e getCusto().

Classe Procura

Interface que define o método resolver. Este método pode ser chamada com os argumentos problema e profMax, ou apenas com o argumento problema.

Package "modprob"

- Estado

  Classe abstrata com o método equals() para que possa comparar dois estados e identificar se o estado em análise é o estado objetivo do problema.

  Contém também a classe hashCode() com o propósito de devolver o identificador da instancia da classe. Esta método não é definido nesta classe pois a sua implementação deverá ser feita no domínio do problema em análise.

- Operador

  Interface que define os contratos para os métodos aplicar(), que recebe e retorna um estado, e custo(), que recebe o estado atual e o estado sucessor e retornando um double.

- Problema

  Classe abstrata que tem como varáveis o estado inicial e os operadores inerentes ao problema. O construtor recebe um estado inicial e um conjunto de operadores possíveis a serem aplicados.

  Tem implementados os métodos get para as variáveis privadas da classe e define a função objetivo(), que por ser do domínio do problema em análise não é implementada nesta classe.

- ProblemaHeur

  Define o problema heurístico e estende a classe Problema.

  O construtor recebe o estado inicial e os operadores a serem aplicados porém este irá passar estes argumentos ao construtor da classe "mãe" Problema

  Esta classe ainda tem um método abstrato heurística() que recebe um estado e retornando um valor double.

Package “mecproc”

- No

  Contém variáveis privadas representativas de um nó, nomeadamente estado, antecessor, operador, profundidade e custo.

  O construtor recebe o estado em análise, o operador que lhe deu origem e os estado antecessor. E define ainda os métodos get para todas estas variáveis privadas da classe

- MecanismoProcura

  Contém dois métodos resolver. O primeiro chama o segundo passando como parâmetros o problema que recebe e um valor inteiro para a profundidade máxima dado por Integer.MAX_VALUE, que é o maior inteiro possível em Java. O segundo método resolver limpa- a memória de procura e de seguida obtém o nó inicial do problema, inserindo-o na memória de procura. De seguida verifica para cada nó da memória se o seu estado é o objetivo do problema, retornando a solução caso seja, ou caso contrário expandindo o nó em análise.

  Tem também o método expandir(), que recebe o nó a ser expandido. Neste método é obtido o estado referente ao nó em análise e aplicados todos os operadores para este estado. Sempre que exista um estado sucessor é inserido o nó correspondente na memória de procura.

  É definido posteriormente o método gerarSolucao(), que tem como argumento um nó final que, quando desconstruídos os seus antecessores ,retorna solução do problema.

  Também existe um outro método designado expandir que recebe um nó e aplica a cada estado desse nó um operador do problema, verificando depois se existe um estado sucessor e caso exista coloca-o na memória de procura. Existem ainda os métodos get que fornecem a informação necessária para as outras classes.

- Percurso

  Implementa a Classe Solucao de forma a ser um objeto iterável e que represente o percurso desde o nó raiz até ao objetivo.

  Tem uma variável percurso do tipo LinkedList de PassoSolucao, que irá guardar todos os passos desde o nó inicial ao nó final.

  Redefine o método getDimensao(), devolvendo a dimensão total do percurso, e o método getCusto(), fazendo o cálculo do custo acumulado desde o nó raiz até ao nó objetivo. Tem também o método juntarNo() que irá juntar sempre o próximo nó ao início da variável percurso.

- ProcuraHeur

  Interface que define o contrato para os métodos resolver.

- Packagem “mem”

– **MemoriaFIFO**

Classe que estende de MemoriaProcura e cujo construtor chama o construtor da classe herdada, através do método super, passando-lhe como argumento uma nova fronteira do tipo LinkedList.

– **MemoriaLIFO**

Idêntica à classe MemoriaFIFO, porém será passada uma fronteira do tipo LinkedList com as características de uma LifoQueue.

– **MemoriaPrioridade**

Classe que estende de MemoriaProcura e cujo construtor recebe um comparador passando ao construtor da classe herdada uma nova fronteira do tipo PiorityQueue que tem como parâmetros o valor 1 e o comparador.

– **MemoriaProcura**

No construtor, que recebe qual a fronteira desejada por cada tipo de memória.

Define o método limpar(), que limpa a lista da fronteira e dos nós explorados, e o método inserir(). O método inserir recebe um nó e verifica se o nó não se encontra na memória e se o seu custo é inferior a todos os nós que lá se encontram. Caso as condições descritas se verifiquem o nó é colocado na fronteira, bem como nos nós explorados mais o seu estado.

É também definido, nesta classe, o método remover(), que remove e retorna o primeiro nó da fronteira, o método fronteiraVazia(), que retorna true ou false consoante a fronteira esteja vazia e ainda os métodos get de forma a aceder às variáveis privadas da classe.

Package “larg”

- ProcuraLarg

Representa a procura em largura.

Estende de MecanismoProcura e apenas contém o método inserirMemoria() que retorna uma nova memória FIFO.

Package “Prof”

- ProcuraProf

Representa a procura em profundidade.

Estende de MecanismoProcura e apenas se encontra definido o método inserirMemoria() que retorna uma nova memória LIFO.

- ProcuraProfIter

  Representa o método de procura em profundidade iterativa.

  Estende de ProcuraProf e tem dois métodos procurar que retornam uma Solucao, recebendo os parâmetros problema, profundidade máxima e incremento de profundidade. É então criada uma solução conforme se vai percorrendo a árvore até se chegar ao limite e quando existe mesmo uma solução esta é retornada. Caso não haja solução é retornado null. Cada vez que se tenta gerar uma solução é chamado o método procurar com os parâmetros problema e profundidade máxima da classe MecanismoProcura.

Package "melhorprim"

- ProcuraMelhorPrim

  Retrata a procura por melhor primeiro que estende de MecanismoProcura e implementa a classe Java Comparator. Contém o método inserirMemoria que retorna uma nova memória por prioridade e o método compare que recebe dois nós e os compara utilizando Double.compare em que nos argumentos é chamado o método F que recebe um nó e retorna um double.

- ProcuraCustoUnif

  Representa a procura por custo uniforme que deriva da classe ProcuraMelhorPrim e que implementa apenas o método, obrigado pela classe de onde herda os atributos, designado F que recebe um nó e retorna o custo do mesmo.

- ProcuraSofrega

  Retrata a procura sôfrega e estende da classe ProcuraMelhorPrim e tem o método obrigatório F que recebe um nó e retorna a distância de Manhattan chamando o método heurística e passando como argumento o estado do nó.

- ProcuraAA

  Representa a procura A* e estende de ProcuraMelhorPrim, tendo o método F que recebe um nó e retorna a soma do custo deste com a distância de Manhattan obtida com o método heurisitca que recebe como parâmetro o estado do nó.

PACKAGE “RESPUZZLE”

resPuzzle

É nesta classe que se encontra o método main onde se chama todas as classes anteriores necessárias, por forma a ser conseguido atingir-se o objectivo, ou seja iniciar um puzzle numa configuração inicial percorrendo uma árvore com vários nós, aplicando os operadores, até ser obtida a configuração final. Para além deste método ainda se encontram os métodos mostrarConfiguracao, que percorre cada pertencente à solução e mostra qual a configuração do puzzle presente, e o definirOperadores, que cria um array de operadores com os quatro movimentos possíveis a serem aplicados nos puzzles (Cima, Baixo, Esquerda e Direita).

Package “modpro”

- ### EstadoPuzzle

  Estende de Estado e tem um constructor que recebe um puzzle e guarda-o. Tem os métodos hasCode, que retorna qual o identificador da instância do puzzle recebido, toString que imprime o puzzle e getPuzzle que retorna o puzzle.

- ### ProblemaPuzzle

  Extende de ProblemaHeur e contém um constructor que recebe o puzzle inicial, o puzzle final e os operadores e chama o constructor da classe ProblemaHeur com o super passando um novo EstadoPuzzle, correspondente ao puzzle inicial, e os operadores. Após chamado este constructor é criado um EstadoPuzzle referente ao puzzle final.

  Outros métodos pertencentes a esta classe são o método objectivo, onde é verificado se o EstadoPuzzle recebido é igual ao estado do puzzle final e o método heuristica que recebe um EstadoPuzzle, de onde se obtém o seu puzzle com getPuzzle() para ser possível verificar qual a distância de Manhattan entre este e o puzzle final.

- ### OperadorMoverPosVazia

  Implementa a interface Operador e tem um constructor que recebe um movimento de puzzle e o guarda. Contém o método aplicar que recebe um EstadoPuzzle, de onde se obtém o seu puzzle através do getPuzzle() e se aplica o movimento, caso o movimento seja bem sucedido é retornado um novo puzzle com o movimento aplicado.

  Visto o custo ser dado por valor 1 por cada movimento efectuado, então o método custo apenas retorna uma variável FINAL que tem valor igual a 1.

Conclusão

No término deste caso prático conseguimos identificar a arquitetura e comportamento de um agente reativo, identificando também seis métodos de procura em espaço de estados:

- Procura em Largura;
- Procura em Profundidade;
- Procura em Profundidade Iterativa;
- Procura de Custo Uniforme;
- Procura Sôfrega;
- Procura A Asteristo (AA).

Conseguimos também caracterizar estas procuras nos seus métodos de análise: completo, ótimo, complexidade temporal e complexidades espacial.

Desta forma, foi possível implementar a resolução para os problemas expostos no decorrer do relatório e efetuar as devidas análises mediante o problema em análise.

Bibliografia

- [Russel & Norvig, 2003] S. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, 2nd Edition, Prentice Hall, 2003.
- [Anabela Simões & Ernesto Costa] Inteligência Artificial, 2008.
- [Sérgio Guerreiro] Introdução à Engenharia de Software, 2015.