INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

ÁREA DEPARTAMENTAMENTAL DE ENGENHARIA DE ELECTRÓNICA E TELECOMUNICAÇÕES E DE COMPUTADORES

Licenciatura em Engenharia Informática e Multimédia

Inteligência Artificial para Sistemas Autónomos

Semestre de Verão 19/20

Síntese do 2º Trabalho Prático

Turma: 41D

Data: 26 maio 2020

Docente: Luís Morgado

Nome: Luís Fonseca

Número: 45125

Introdução

O segundo trabalho de IASA consiste em implementar e desenvolver um mecanismo de raciocínio automático com base em procura em espaços de estado. Este tipo de raciocínio foi aplicado na resolução automática de puzzles.

Raciocínio automático consiste num raciocínio prospetivo, onde é realizada uma antecipação dos acontecimentos, sendo seguidas instruções que nos levam ao destino desejado.

Quando falamos deste tipo de raciocínio, temos de ter em conta duas condições fundamentais, sendo elas a exploração e avaliação de opções.

- Em relação à exploração de opções, existe um raciocínio prévio (antecipa a situação) e uma simulação do mundo (interna);
- Em relação à avaliação de opções, existem duas condições de decisão, que são o custo e a utilidade;

Em suma, este raciocínio permite que os computadores se auto-programem, ou seja, resolvam problemas por eles próprios.

Paradigma simbólico

Este paradigma consiste num ciclo lógico que é a perceção/deliberação e ação. O que isto quer dizer é que a primeira coisa que fazemos é percebermo-nos da situação ao criamos um modelo, planearmos o que iremos fazer, quando agimos e ao existir uma ação motora por parte do nosso grupo.

Puzzle

O puzzle é um tabuleiro, de dimensões 3x3, constituído por peças deslizantes numeradas de 1 a 8 à exceção da peça vazia. As regras são as enumeradas abaixo:

- Só é permitido movimentar uma peça de cada vez;
- As peças apenas se podem movimentar numa posição vazia do tabuleiro;
- Cada movimento só pode efetuar uma deslocação;
- Estes movimentos não podem ser efetuados na diagonal;

O objetivo deste puzzle é chegar a uma determinada configuração em que todos os números do puzzle se encontram na sequência desejada.

Para a resolução automática deste puzzle, são sempre fornecidas duas configurações, uma inicial e outra final.

Raciocínio através de Procura

Num raciocínio através de procura existem termos fundamentais como situação, ação e procura de solução a considerar. A situação é representada como um estado sendo que num problema existe um espaço de estados. Quando a ação gera a transformação de um estado e tem um operador, vai ser responsável pela transição de estado. Sobre a procura de solução ou também conhecido como raciocínio, o mesmo consiste em fazer uma representação interna considerando um estado num espaço de estados e faz uma procura no mesmo.

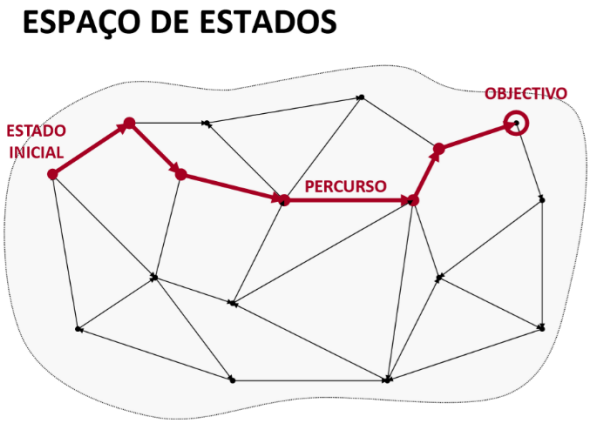


Figura 1 – possível percurso de um estado, com o seu estado inicial e a solução

Processo de Procura

Existindo um espaço de estados, existem diversas formas de efetuar a procura. A isso denominamos de processo de procura. O processo de procuta é o "protocolo" para a qual se explora, procura e encontra o estado num grafo de espaço de estados. Num grafo de espaço de estados, existe o estado antecessor que como o nome indica, refere-se ao estado anterior ao atual e a através da aplicação "avança-se" na árvore.

Conceitos a mencionar sobre Procura

- Estado: o que define a situação a tratar;
- Operador: responsável por efetuar transições de estados tendo associado um custo;
- Problema: é representado por um estado inicial, tendo um objetivo/solução e um tipo de operadores com significado do mesmo. No caso do puzzle, os operados serão movimentos, o estado inicial a configuração inicial e a solução a configuração pretendida;
- Solução: representa o objetivo;

Diferentes métodos de Procura

Quando se trata de definir a maneira de efetuar, existem duas hipóteses:

- Explorar primeiro os nós mais recentes;
- Explorar os mais antigos nós primeiro;

Em baixo, é possível de ver uma representação gráfica de ambas as situações em estudo:

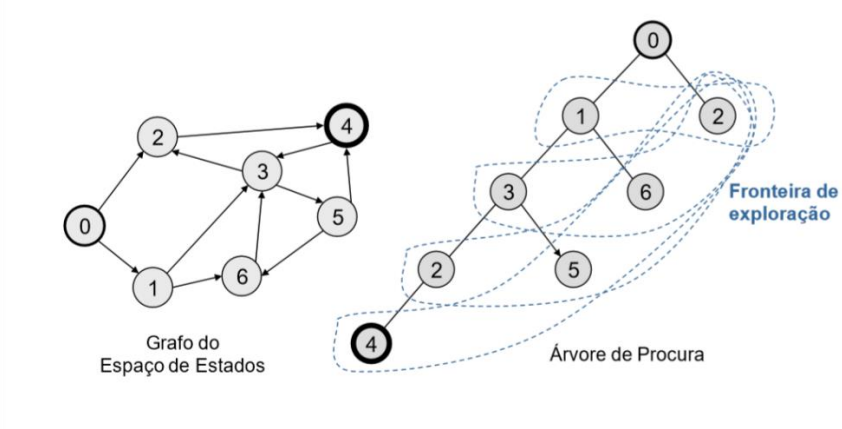


Figura 2 - exploração dos estados, sendo como cenário explorar os nós mais recentes primeiro

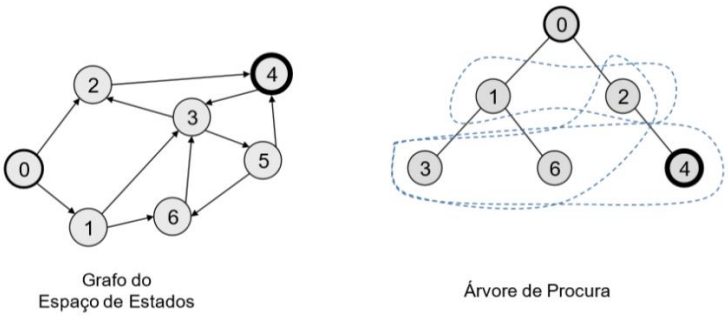


Figura 3 - exploração dos estados, sendo como cenário explorar os nós mais antigos primeiro

Resumo sobre os métodos de procura

Quando se fala em métodos de procura, devemos ter em conta os seguintes conceitos:

- Completo: garante que encontrará a solução se existir;
- Ótimo: garante que a melhor solução é encontrada;
- Complexidade: tempo necessário e espaço necessário;

Em baixo, segue-se uma tabela com as diferentes procuras, com os conceitos abordados em cima:

COMPLEXIDADE COMPUTACIONAL

Método de Procura	Tempo	Espaço	Óptimo	Completo
Profundidade	$O(b^m)$	$O(bm)$	Não	Não
Largura	$O(b^d)$	$O(b^d)$	Sim	Sim
Custo Uniforme	$O(b^{[C^*/\varepsilon]})$	$O(b^{[C^*/\varepsilon]})$	Sim	Sim
Profundidade Limitada	$O(b^l)$	$O(bl)$	Não	Não
Profundidade Iterativa	$O(b^d)$	$O(bd)$	Sim	Sim
Bidireccional	$O(b^{d/2})$	$O(b^{d/2})$	Sim	Sim

b – factor de ramificação
d – dimensão da solução
m – profundidade da árvore de procura
l – limite de profundidade
C^* – Custo da solução óptima
ε – Custo mínimo de uma transição de estado ($\varepsilon > 0$)

Figura 4 - tabela com as procuras estudadas, cada procura tem o seu tempo, espaço, ótimo e completo associado

Tipo de Procura - Profundidade

Este tipo de procura consiste em explorar primeiro os nós com maior profundidade. Este tipo de procura contém uma complexidade espacial de O(bm) e uma complexidade temporal de O(b^m), sendo b o fato de ramificação e d a profundidade da procura. A principal desvantagem deste tipo de procura é o facto de não poder encontrar solução.

Tipo de Procura - Largura

Este tipo de procura explora primeiro os nós com menor profundidade, tem complexidade espacial e temporal de O(b^d). A principal desvantagem desta procura é o uso extensivo de memória.

Tipo de Procura – Profundidade Limitada

Este tipo de procura limita a procura a uma dada profundidade máxima. Tem uma complexidade espacial de O(bd) e complexidade temporal de O(b^d).

Tipo de Procura – Procura em grafos com ciclos

Existem dois tipos importantes em procuras com ciclos. São eles os nós aberto (gerados, mas não explorados) e os nós fechados (expandidos).

Tipo de Procura – Procura Melhor Primeiro

Neste tipo de procura faz-se uma avaliação do estado e ordena-se a fronteira de exploração por ordem crescente de f(n).

Tipo de Procura – Procura Custo Uniforme

Neste tipo de procura explora-se primeiro os caminhos com menor custo. Na figura em baixo, pode-se visualizar um exemplo de uma procura com custo uniforme

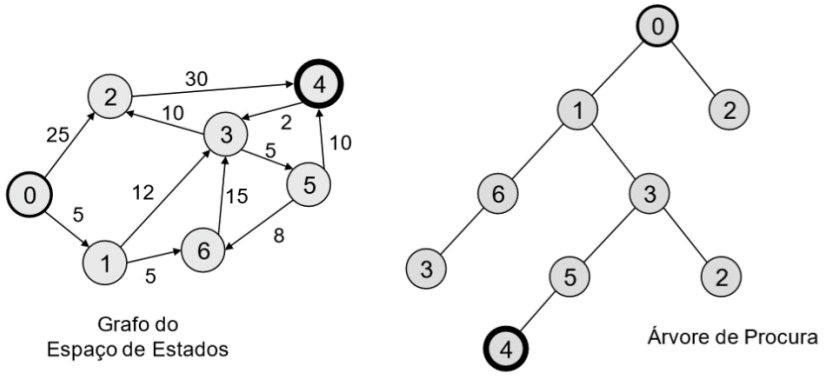


Figura 5 - exemplo de uma procura com custo uniforme

Tipo de Procura - Procura em Espaços de Estados

Neste tipo de procura, podemos ter uma procura não informada e uma procura informada.

- Uma procura não informada consiste em ordenar a fronteira de exploração sem qualquer tipo de controlo de procura, ou seja, sem tirar partido do conhecimento do domínio do problema. Este tipo de procura é, portanto, uma procura não guiada(exaustiva);
- Uma procura informa, é o oposto da anterior, ou seja, é uma procura guiada que tira partido do conhecimento do domínio do problema;

Função Heurística H(n)

Esta função representa uma estimativa do custo do percurso desde o nó (nó inicial) até ao nó objetivo (que pode ser também a solução). Este reflete o conhecimento que tem acerca do domínio do problema para tornar a procura guiada. A função heurística depende apenas do estado (associado a n) e do objetivo.

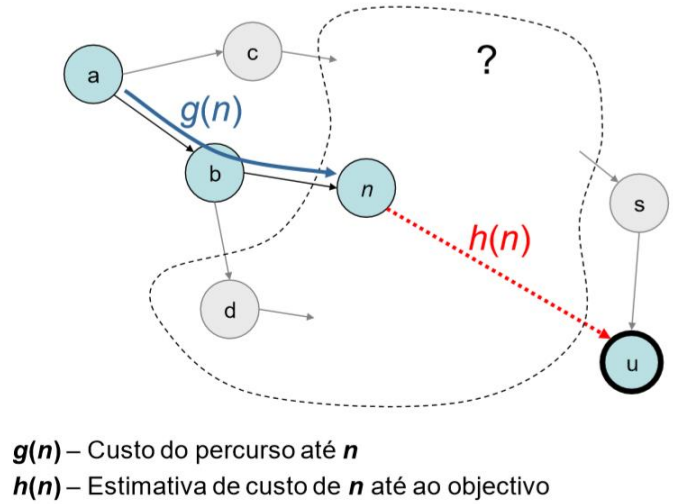


Figura 6 - método de procura informada, com a funcao heuristica aplicada

Tipo de Procura – Procura Melhor-Primeiro (Best-First)

Este tipo de procura é tipo de procura que através da função f(n) efetua uma avaliação de cada nó que é gerado (sendo que f(n) representa uma avaliação do custo da avaliação). Nesta procura a fronteira de exploração é ordenada por ordem crescente de f(n).

Existem 3 variantes dentro da procura melhor-primeiro sendo essas as seguintes:

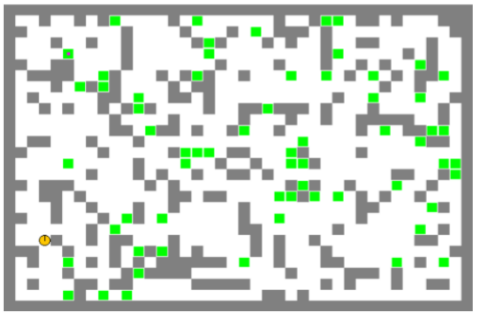
- Procura custo uniforme(g(n)): não tira partido de conhecimento do domínio do problema expresso através da função h(n);
- Procura Sôfrega (Greedy Search): não tem conta o custo do percurso explorado, faz uma minimização do custo local e tem soluções sub-ótimas;
- Procura A*(heurística admissível): faz uma minimização do custo global;

*Tipo de Procura – Procura A**

Na procura A*, a heurística é admissível sendo que uma heurística admissível é otimista e a estimativa de custo é sempre inferior ou igual ao custo efetivo mínimo para um determinado nó objetivo.

Este tipo de procura é completo e ótimo.

A distância euclidiana corresponde a retirar as restrições (não existe movimentação através de obstáculos e não existe movimentação na diagonal).



h_1 – Distância Euclidiana

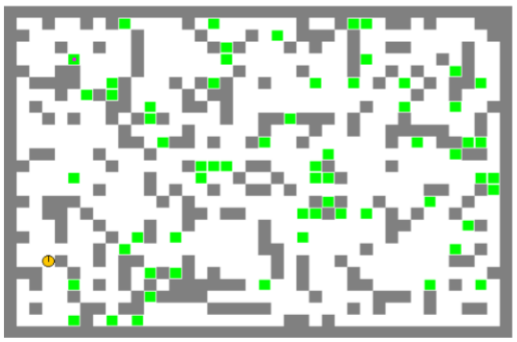
$$h_1(n) = \sqrt{(x_n - x_{obj})^2 + (y_n - y_{obj})^2}$$

Admissível?

SIM

Figura 7 - procura A, usando distância Euclidiana

Também é possível associar esta procura com a distância de Manhattan. Esta distância corresponde em retirar a restrição (não existe movimentação de obstáculos).



h_2 – Distância de Manhattan

$$h_2(n) = |x_n - x_{obj}| + |y_n - y_{obj}|$$

Admissível?

- SIM : Se não forem possíveis movimentos diagonais
- NÃO : Caso contrário

Figura 8 - procura A, com a distância de Manhattan associada*

Nota: se os nós já visitados não forem eliminados a heurística pode não ser consistente.

Na Implementação…

Sobre a implementação deste trabalho, o que é de notar é o termo de comparação de objetos utilizados, o *hashcode*. Este método permite verificar se dois objetos são iguais ou não.

Todas as procuras aqui mencionados foram implementadas também, cada uma com uma implementação diferente da outra.

No caso da procuraAA, apenas foi retornado um método de nome *f(No no)*, que consiste em aplicar o método f, passando-lhe um nó. Este método retorno o problema da heurística, sendo necessário ir buscar o estado atual do nó, e somar o seu respetivo custo. Em baixo é possível de visualizar o método criado para esta procura.

```
@Override
protected double f(No no) {
	return problema.heuristica(no.getEstado()) + no.getCusto();
}
```

No caso da procura com custo uniforme, foi também implementado o método *f(No no)*. Mas neste caso iremos apenas retornar o custo do nó, seguindo a definição da procura acima já explicada. A implementação deste método é a seguinte:

```
@Override
protected double f(No no) {
	return no.getCusto();
}
```

No caso da procura por Heurística, será uma interface, com dois métodos de nome *resolver*, que permitem resolver a solução do nosso problema. A diferença entre estes dois métodos é que um deles resolve o problema e o outro resolve também o problema, mas com uma determinada profundidade máxima.

No caso da procura melhor primeiro, iremos ter dois métodos, que são o método *iniciarMemoria()* que como o próprio nome indica, permite inicializar a memória, é retornado uma *MemoriaPrioridade*, viste que consiste numa procura com prioridade, e o método *compare(No o1, No o2)*, que permite compara dois nós. O código implementado para esta classe é o seguinte:

```
	@Override
	protected MemoriaProcura iniciarMemoria(){
		return new MemoriaPrioridade(this);
	}

	@Override
	public int compare(No o1, No o2){
		return Double.compare(f(o1), f(o2));
	}
```

A Procura Sôfrega (outro nome para Greedy Search) foi implementado o método *f(No no)* que também consiste em retornar o problema heurística, sendo necessário ir buscar o estado atual.

```
	@Override
	protected double f(No no) {
		return problema.heuristica(no.getEstado());
	}
```

Em relação à Procura Iterativa, foi necessário fazer a implementação do método *resolve (Problema problema, int profMax)* que consiste em resolver uma solução, dado um problema, com uma profundidade máxima. Para isso iremos percorrer a nossa profundidade máxima, e de seguida iremos chamar a nossa solução. Caso a solução não esteja vazia, é retornada essa respetiva solução.
O código implementado para este método é o seguinte:

```
	public Solucao resolver(Problema problema, int profMax){
		for(int profMaxIt = getIncProf(); profMaxIt<=profMax;
profMaxIt+=getIncProf()){
			Solucao solucao = super.resolver(problema,profMax);
			if(solucao != null){
				return solucao;
			}
		}
		return null;
	}
```

A procura em Largura, apenas irá retornar a memoriaFIFO no seu construtor.

Por último a procura em profundidade, apenas irá retornar uma memóriaLIFO no construtor.

Com as procuras tratados, e o correto funcionamento dos nós, iremos passar para a implementação do puzzle, em Java, usando os conhecimentos aqui estudados. Para a implementação do puzzle foram implementadas quatro classes, classes essas que tem o nome de:

- EstadoPuzzle: que consiste em retornar um estado no puzzle;
- OperadorPuzzle: consiste em aplicar um operador no puzzle, ou seja, uma jogada, quando existe um espaço vazio no puzzle;
- ProblemaPuzzle: que permite aplicar o problema heurística ao nosso puzzle
- PlaneadorPuzzle: que consiste em resolver o puzzle, passando uma configuração inicial e outra final(classe esta que servirá como classe "main" para correr o nosso puzzle).

Para a classe "EstadoPuzzle", como já foi referido, foi apenas retornado o estado do puzzle atual.

A classe "OperadorPuzzle" é onde irá ser aplicada uma jogada no nosso puzzle. Para isso foram implementados os métodos *aplicar (Estado estado)* e *custo (Estado estado, Estado estadoSuc)*. O método *custo* apenas retorna o custo de uma jogada, entre um estado inicial e um estado final. O método *aplicar* permite aplicar uma jogada quando existe um espaço vazio dentro do nosso puzzle. A implementação destes métodos pode ser vista em baixo:

```
	@Override
	public Estado aplicar(Estado estado) {
		Puzzle puzzle, newPuzzle;
		if(estado instanceof EstadoPuzzle) {
			puzzle = ((EstadoPuzzle) estado).getPuzzle();
			newPuzzle = puzzle.movimentar(movimento);
			if(newPuzzle != null) {
				return new EstadoPuzzle(newPuzzle);
			}
		}
		return null;
	}

	@Override
	public float custo(Estado estado, Estado estadoSuc) {
		return custoJogada;
	}
```

A classe “ProblemaPuzzle” permite aplicar o problema de heurística. Para isso, no construtor é necessário passar no construtor o estado inicial (neste caso será o nosso puzzle inicial) e o estado final (que será o nosso puzzle final). De seguida foram implementados os métodos *heurisitica(Estado estado)* que permite aplicar o problema da heurística, e o método *objectivo(Estado estado)* que permite verificar quando deixam de existir pecas no puzzle. A implementação para estes métodos foi a seguinte:

```
	public ProblemaPuzzle(Puzzle puzzleInicial,Puzzle puzzleFinal,Operador[]
operadores){
		super(new EstadoPuzzle(puzzleInicial), operadores);
		this.puzzleFinal = new EstadoPuzzle(puzzleFinal);
	}
	@Override

	public double heuristica(Estado estado) {
	EstadoPuzzle estadoAtualPuzzle = (EstadoPuzzle) estado;
	doubledistManPuzzle =
estadoAtualPuzzle.getPuzzle().distManhattan(puzzleFinal.getPuzzle());
		return distManPuzzle;
	}


	@Override
	public boolean objectivo(Estado estado) {
		return estado.equals(puzzleFinal) ? true : false;
	}
```

Por fim a classe “PlaneadorPuzzle” que permite correr o nosso puzzle, passando as diferentes configurações que foram pedidas. Apenas foi implementado o método *showPuzzle(Solucao solução)*_que consiste em mostrar o puzzle, coma respetiva solução.

```
	private static void showPuzzle(Solucao solucao) {
		System.out.println();
		for(PassoSolucao s: solucao) {
			System.out.println(s.getEstado());
			System.out.println();
		}
	}
```

Segue-se um exemplo de código, neste caso apenas será mostrado o resultado, demostrando o correto funcionamento.

Nota: o valor da complexidade temporal e espacial e o custo irá ser mostrado num ficheiro excel, para as diferentes procuras.

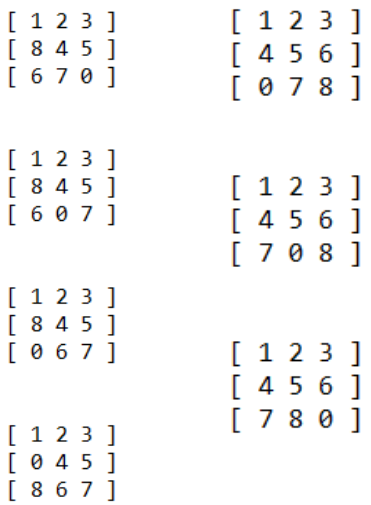


Figura 9 - configuração inicial(figura à esquerda) e final do puzzle(figura à direita)

De seguida, é apresentada uma tabela criada em excel, com o custo e a complexidade espacial e temporal, para cada tipo de procura:

Nota: este documento, pode ser editado e visto na pasta “Mod”, juntamente com a entrega do trabalho;

	Configuração A			Configuração B		
Procura	Custo	Complexidade espacial	Complexidade temporal	Custo	Complexidade espacial	Complexidade temporal
Profundidade	34366	25546	36591	60178	40336	74053
Largura	14	2125	3543	26	10465	163687
Profundidade Iterativo	34366	25546	36591	60178	40336	74053
Custo Uniforme	14	2059	3160	26	11113	163394
Sofrega	18	55	85	62	198	275
A*	14	53	83	26	1302	2440

Figura 10 - tabela com os valores do custo e da complexidade temporal e espacial

Conclusões

Neste trabalho prático foi implementada toda a teórica que foi explicada neste documento. A implementação deste trabalho consistiu em primeiro lugar estudar a teoria, compreender os seus conceitos e após isso implementar o código.

Bibliografia

Slides fornecidos pelo docente Luís Morgado