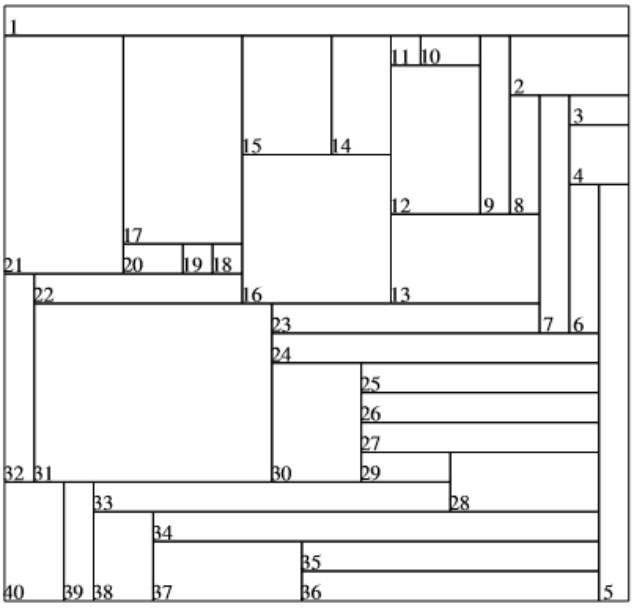
Faculdade de Ciências da Universidade do Porto

Vigilância de Partições Retangulares

Inteligência Artificial 2020



Ana Beatriz Saldanha - up201506293

Diana D'Egas - up201604621

Manuel Sá - up201805273

1.O Problema

1.1. Vigilância de Partições Retangulares

Dado n instâncias ao programa e a sua respetiva partição R composta por retângulos, pretende-se colocar guardas nos vértices desses mesmos, de modo a que todos os retângulos pertencentes à partição R sejam vigiados por uma guarda. Qualquer ponto pode ser considerado uma guarda no início da resolução, e o objetivo do programa é o retorno do número mínimo de guardas possível, com a condição que todos os retângulos estejam a ser vigiados. Com a estrutura do problema em mente, é necessária a criação de sistemas de pesquisa de modo a obter a informação necessária para a conclusão esperada, assim como as estruturas que guardam essa mesma informação. De seguida é preciso encontrar uma solução ótima, ou seja, a resposta mais eficiente que, neste caso é o número mínimo de guardas.

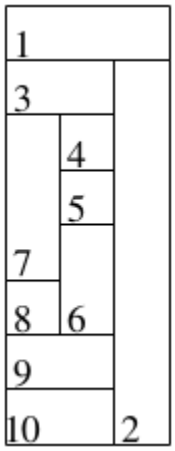


Fig 1. Exemplo de uma instância e a sua partição. Cada número representa um retângulo da partição.

2. Estratégias de pesquisa

2.1. Descrição da Implementação geral

Para as primeiras questões, a linguagem usada para o desenvolvimento foi Java, de modo a facilitar a manipulação de objetos e a implementação dos algoritmos de pesquisa.

Inicialmente começámos por definir as estruturas necessárias de acordo com o problema proposto, para que, deste modo criássemos várias classes que guardassem os dados de input.

A classe Point contém informações das quais fazem parte as coordenadas, um id que identifica o ponto num array global de pontos, um array de retângulos a que um ponto pertence, e ainda duas flags usadas posteriormente nos problemas de pesquisas que vão informar se aquele ponto é um guarda e se está visitado.

De forma idêntica construímos a classe Rect, que a cada retângulo atribui um id, bem como um array de pontos que constituem o mesmo. A *flag* relativa ao estado, visitado ou não, também será usada para a pesquisa.

Por fim, e antes de falar sobre cada classe dos algoritmos pormenorizadamente, referir a classe NodePoint, usada na construção de uma estrutura de dados relativa a uma árvore que armazena a nossa informação, os nós que representam retângulos e pontos.

De seguida para cada algoritmo específico é apresentada a nossa implementação, após uma introdução teórica sobre cada uma.

2.2. Pesquisas não informadas

Algoritmos de pesquisa não informada não têm informação sobre o domínio do problema. Estas pesquisas limitam-se a verificar se o estado atual é a solução ou não, ou seja, só distinguem o estado final de todos os outros estados não finais. Estas estratégias não encontram sempre a solução ótima, podem até mesmo não encontrar a solução. Servem de base a outras pesquisas mais “inteligentes” que veremos mais à frente.

Pesquisas não informadas abordadas no trabalho: Pesquisa em Profundidade (DFS), Pesquisa em Largura (BFS) e Pesquisa Iterativa Limitada em Profundidade (IDFS), Pesquisa Greedy (no nosso caso).

2.2.1. Pesquisa em profundidade (DFS)

Como funciona?

Como o próprio nome indica esta pesquisa realiza-se em profundidade, isto significa que a pesquisa vai começar num nó raiz e a partir daí vai expandir todos os nós de maior profundidade até que uma das seguintes situações aconteça:

(1) encontrar solução e deste modo devolve-a;

(2) encontrar nós sem filhos, nesta situação a pesquisa retrocede, backtracking, tentando expandir o nó em maior profundidade ainda não expandido.

Notar que, esta estratégia não é completa nem ótima.

Quando se usa?

Usamos este tipo de pesquisa quando queremos encontrar todas as soluções, isto é, problemas com mais que uma solução.

Complexidade temporal: $O(b^m)$

Complexidade espacial: $O(b \times m)$

b- Fator ramificação

m-Profundidade máxima

Implementação DFS

Relativamente à implementação deste algoritmo, consideramos a classe Pesquisas onde está o método correspondente à pesquisa DFS. Usámos pilhas, estruturas do tipo LIFO (Last-in-First-out) recorrendo às operações necessárias tais como push para cada novo "pai" ser adicionado à pilha e pop para que os elementos sejam retirados na ordem pretendida. Vamos gerando filhos com o método gerarFilhos definido na classe NodePoint, verificando soluções (verificaSolucao – classe Pesquisas) em profundidade, isto é, se encontrar uma solução, a partir do nó em que estamos percorremos o caminho inverso, adicionando à lista de soluções.

2.2.2. Pesquisa em Largura (BFS)

Como funciona?

A pesquisa em largura inicia-se ao nível da raiz, d=0, onde todos os nós são expandidos nesse nível, em seguida todos os nós do nível d+1 são expandidos e assim sucessivamente até encontrar a solução, garantindo sempre que todos os nós com menor profundidade são expandidos primeiro.

Este tipo de pesquisa é implementado com filas que garantem a ordem dos próximos nós a ser visitados.

Quando se usa?

Esta pesquisa é usada especialmente quando queremos encontrar uma solução ótima, isto é, o menor caminho possível de um nó inicial (raiz) até ao nó objetivo.

Complexidade temporal: $O(b^d)$

Complexidade espacial: $O(b^d)$

b- Fator ramificação

d-Profundidade da solução

Implementação BFS

A implementação para este algoritmo também se encontra na classe Pesquisas e é relativamente parecida com a implementação do DFS, com a diferença que este algoritmo expande os filhos em largura, isto é, para cada filho da raiz vamos gerar os seus filhos (gerarFilhos – classe NodePoint) para todos o nós desse nível. A ordem de visita dos nós é garantida, uma vez que usamos filas, que são estruturas do tipo FIFO (First-in-First-out), o que garante que o primeiro nó a entrar é o primeiro a sair e, desse modo, conseguimos efetuar a pesquisa na ordem pretendida.

2.2.3. Pesquisa Iterativa Limitada em Profundidade (IDFS)

Como funciona?

Esta estratégia de pesquisa é uma espécie de DFS, mas controlado, quer isto dizer que existe um limite definido para a expansão de nós até chegar à solução ou ao nível máximo imposto. O IDFS é uma estratégia ótima e completa.

Quando se usa?

Método de pesquisa é utilizado, quando a árvore apresenta grande profundidade e a solução é desconhecida. Incrementa o limite em cada iteração até chegar ao máximo.

Complexidade temporal: $O(b^d)$

Complexidade espacial: $O(b \times d)$

b- Fator ramificação

d-Profundidade da solução

Implementação IDFS

Sendo este algoritmo uma variação do DFS, apenas definimos um nível que incrementa a cada iteração enquanto não encontra a solução, chamando assim o método do DFS enquanto a condição não se verifica.

2.3. O que é uma heurística?

Quando falamos em pesquisas informadas, como vamos ver mais à frente, é necessário referir a importância das heurísticas. A pesquisa por heurísticas é uma pesquisa em que para um determinado problema, nos dá noção da proximidade a que estamos da solução. Isto significa que apenas se foca no problema específico de encontrar o objetivo final. Podem tornar a escolha do caminho do nó inicial ao final fácil e rápida, mas não podemos garantir que vá encontrar uma solução ótima para o programa.

2.4.1. Pesquisa Greedy

Como funciona?

Este algoritmo, realiza a melhor escolha no momento, toma decisões com base nas informações disponíveis na iteração corrente, sem ter em consideração as suas consequências. É semelhante à busca em profundidade pois, vai sempre na mesma direção do caminho da árvore para encontrar a solução, mas pode mudar de direção dependendo do custo dos nós ainda não explorados da árvore. É uma pesquisa simples e de fácil implementação, mas não garante necessariamente o melhor caminho.

Quando se usa?

Esta pesquisa deve ser utilizada quando se quer garantir uma menor complexidade espacial e temporal, caso contrário a imprecisão do cálculo dos custos pode torná-la numa busca infinita que não chega sequer a encontrar solução.

Complexidade temporal: $O(b^m)$
Complexidade espacial: $O(b^m)$

b- Fator ramificação

m-Profundidade máxima

Implementação Greedy

Num construtor criamos uma priority queue que tem como objetivo ordenar todos os pontos da partição por ordem do número de retângulos não visitados. Assim ao remover os pontos da fila, chamamos a função guarda, que torna a flag de guarda daquele ponto a true, caso o ponto possa ser guarda. Visto que todos os retângulos aos quais aquele ponto pertence são visitados, todos os restantes pontos desse retângulo também passam a ser visitados. Um ponto pode ser uma guarda em duas condições: ainda não foi visitado, ou foi visitado e tem pelo menos um retângulo adjacente a esse ponto ainda não visitado, a Greedy vai percorrer todos os pontos para verificar se podem ser guardas. No entanto, este algoritmo não retorna a solução ótima, visto que apenas escolhe a melhor opção no momento em que se encontra.

2.4. Pesquisas Informadas

Dada a definição de heurística, podemos agora explicar melhor este tipo de estratégias. Ao contrário da pesquisa não informada, estas pesquisas têm conhecimento geral do domínio do problema, o que as torna de modo geral mais eficazes. Também devido ao uso de métodos heurísticos, a expansão feita ao nível dos nós é mais correta, uma vez que existe uma avaliação relativa por parte desta função aos nós.

2.4.2. Pesquisa A*

Como funciona?

Um dos algoritmos mais populares, que faz combinações de pesquisa para obter um caminho até à solução. O A* faz uma pesquisa na árvore, expandido sempre o nó com o menor caminho possível, calculado pela heurística. Vai sempre encontrar uma solução ótima caso esta exista, se e só se a função heurística não superestimar o custo real da melhor solução. Este algoritmo é bastante eficiente.

Complexidade temporal: $O(b^d)$
Complexidade espacial: $\boldsymbol{O(b^d)}$

b- Fator ramificação

d-Profundidade da solução

Implementação A*

Na implementação deste algoritmo, criámos uma função Comparator que vai fazer a comparação entre o número de retângulos visitados por cada ponto a partir da função heurística. Criámos uma priority queue, na qual chamamos o comparator que vai ordenar a fila de forma decrescente, primeiramente os pontos com mais retângulos visitados e de seguida os pontos com menos retângulos visitados. Gerámos os nós filhos da raiz, que vão ficar guardados numa lista. Apenas serão gerados os filhos do nó que têm melhor valor heurístico (menor custo). As guardas são colocadas em todos os nós escolhidos.

2.4.3. Pesquisa Branch and Bound

Como funciona?

Este algoritmo é usado para encontrar soluções ótimas, geralmente em problemas de otimização combinatória. Consiste numa enumeração sistemática de todas as soluções candidatas, das quais se descarta as que são obviamente impossíveis. Pode ser implementado de maneira semelhante ao DFS, mas implementa o uso de heurísticas encontrando os caminhos mais curtos. O caminho com o menor custo é guardado e posteriormente, retornado quando a pesquisa termina.

Quando se usa?

Esta pesquisa combina a economia de espaço do DFS com heurísticas. O algoritmo branch and bound é utilizado quando temos vários caminhos e queremos encontrar apenas um caminho ideal.

Implementação Branch and Bound

Em relação à implementação deste algoritmo, ele funciona de forma idêntica a um DFS, ou seja, usamos uma pilha para o armazenamento de nós gerados ao longo da execução, que vai influenciar na forma como posteriormente fazemos a expansão de cada um. Deste modo, temos uma pesquisa em profundidade que combinada com uma heurística definida no método na classe do BNB, considera que os primeiros nós a serem visitados terão menor custo. Mais especificamente, a nossa heurística calcula o nível de um determinado nó e soma-lhe os retângulos cobertos até ao momento por esse mesmo nó. Deste modo obtemos a solução otimizada ao longo das iterações.

2.5. Pesquisa local iterada

2.5.1. Hill-Climbing

Hill-Climbing é um algoritmo de pesquisa local, que progride com o objetivo de alcançar a melhor solução do problema. Podemos comparar este tipo de pesquisa à subida até ao topo de uma montanha, onde a base é a solução não ótima e o topo é a solução ótima. Funciona de forma semelhante a uma pesquisa greedy, pois faz a melhor escolha no momento, não tendo em consideração as consequências subjacentes.

2.5.1.1. Hill-Climbing sem aleatoriedade

O Hill-Climbing sem aleatoriedade, utiliza um processo de melhoria iterativa**.** Este tipo de algoritmo utiliza a avaliação individual do estado do nó vizinho, através de uma função heurística e, posteriormente, seleciona aquele que otimiza o custo atual e define-o como estado atual**.** O método só termina quando, já não é possível atingir nenhuma melhoria significativa, foi efetuado um número fixo de iterações ou um objetivo foi alcançado. Esta técnica pode encontrar máximos locais, e, nesse caso não vai existir um próximo estado para uma solução melhor.

2.5.1.2. Hill-Climbing com aleatoriedade

O Hill-Climbing com aleatoriedade funciona de forma semelhante. A única diferença é o início da pesquisa. Ao invés de começar numa lista de pontos em que nenhum é guarda, começa numa lista onde pontos são considerados guardas aleatoriamente.

Implementação Hill-Climbing com e sem aleatoriedade

No ficheiro Ils.java foi criada a classe Ils. Esta classe vai ser responsável pela pesquisa local não aleatória, e também pela aleatória. Implementámos uma priority queue, com uma heurística que calcula o "custo" de cada solução possível. O custo de cada solução é determinado pela soma do número de guardas, somado ao número de retângulos por visitar, o qual está a ser multiplicado pela soma entre o dobro do número de retângulos mais três, ou seja, g+f(2n+3). Gerando a árvore onde os filhos são todos os pontos que não são pais naquele ramo, estes são inseridos na priority queue por ondem decrescente da heurística, ou seja o primeiro elemento da fila é sempre o de menor custo. Quando nenhum filho tiver um custo menor do que o pai, quer dizer que aquele pai é a melhor solução. A única diferença entre a implementação do Hill-Climbing sem aleatoriedade e com aleatoriedade é a raiz da árvore gerada. Enquanto a primeira é gerada vazia, a segunda raiz é um conjunto aleatório de pontos já assumidos como solução.

2.6.Pesquisa Simulated Annealing

Como funciona?

Este algoritmo é inspirado em estruturas de sistemas biológicos, sendo que é mais indicado para um espaço de pesquisa discreto. O grande objetivo é, usando uma técnica probabilística, aproximar a solução ótima global de uma função. A escolha de uma solução baseia-se nas estratégias greedy onde é feita a seleção da mesma, sem verificar os estados seguintes. A ideia é escolher um mínimo local onde a função de custo será menor que a dos seus vizinhos. Sendo x um nó e x* a vizinhança de x, x é um mínimo local se: f(x)<=f(x*), onde f é a função de custo.

A escolha do próximo nó a ser visitado é aleatória. Se o movimento random beneficiar, a solução é aceite caso contrário o movimento aceite é o que tiver probabilidade inferior a 1.

Quando se usa?

Usado para otimizar soluções.

Implementação Simulated Annealing

O ficheiro Annealing correspondente à implementação deste algoritmo, iniciámos a pesquisa com uma solução aleatória e a partir desta, gerámos os filhos aleatoriamente, com uma variável gerada pelo Random que representa o id do ponto, calculámos a heurística através da função que retorna o custo de determinado nó segundo a formula: g(n) + f(n) * (2 *R+ 3), a partir deste ponto calculámos a energia do sistema, isto é, a função heurística do pai menos a do filho, se essa energia for negativa ou a probabilidade dada for maior que um número aleatório entre 0 e 1, adicionamos como solução o nó seguinte, caso contrário repetimos o processo decrementando o T.

3. Resultados

Para a obtenção dos resultados utilizámos os seguintes inputs:

```
1
10
1 5 0 7 2 7 3 7 3 8 0 8
2 9 2 0 3 0 3 7 2 7 2 6 2 5 2 4 2 2 2 1
3 5 0 6 1 6 2 6 2 7 0 7
4 4 1 5 2 5 2 6 1 6
5 4 1 4 2 4 2 5 1 5
6 5 1 2 2 2 2 4 1 4 1 3
7 6 0 3 1 3 1 4 1 5 1 6 0 6
8 4 0 2 1 2 1 3 0 3
9 5 0 1 2 1 2 2 1 2 0 2
10 4 0 0 2 0 2 1 0 1
```

Input 1

```
1
10
1 9 3 0 4 0 4 7 3 7 3 6 3 5 3 3 3 2 3 1
2 4 0 6 3 6 3 7 0 7
3 5 0 5 1 5 3 5 3 6 0 6
4 5 1 3 3 3 3 5 1 5 1 4
5 4 0 4 1 4 1 5 0 5
6 4 0 3 1 3 1 4 0 4
7 6 0 2 2 2 3 2 3 3 1 3 0 3
8 4 2 1 3 1 3 2 2 2
9 4 0 1 2 1 2 2 0 2
10 5 0 0 3 0 3 1 2 1 0 1
```

Input 2

Para o Input 1 obtivemos os seguintes resultados:

	TEMPO(ns)	MEMORIA(Mb)	SOLUÇÃO	PROFUNDIDADE
GREEDY	52058047	2.1734	NÃO ÓTIMA	5
BFS	131412714	6.5344	ÓTIMA	4
DFS	87037932	6.6535	NÃO ÓTIMA	5
IDFS	41879571	3.6534	ÓTIMA	4
A*	3353972	2.1734	ÓTIMA	4
BNB	98834150	8.6534	ÓTIMA	4
HILL C.	1970621	2.1734	ÓTIMA	4
HILL C. RAND	28527162	3.1734	ÓTIMA	4
SIMULATED. AN	139462245	3.9734	ÓTIMA	4

Para o Input 2 obtivemos os seguintes resultados:

	TEMPO(ns)	MEMORIA(Mb)	SOLUÇÃO	PROFUNDIDADE
GREEDY	358351	2.1734	NÃO ÓTIMA	5
BFS	164451854	6.9743	ÓTIMA	4
DFS	3237862	2.1734	NÃO ÓTIMA	5
IDFS	55690215	3.6534	ÓTIMA	4
A*	1407000	2.1734	ÓTIMA	4
BNB	123413238	10.1734	ÓTIMA	4
HILL C.	2687895	2.1734	ÓTIMA	4
HILL C. RAND	13119925	2.6534	ÓTIMA	4
SIMULATED. AN	134835308	3.6914	ÓTIMA	4

3.1. Conclusão de resultados

De entre todos os algoritmos, os únicos que não têm solução ótima são o DFS e o Greedy. Dos algoritmos de pesquisa não informada, conseguimos destacar o Greedy e o IDFS como os que gastam menos memória, isto é, precisam de gerar menos nós para obter uma solução. Constatámos ainda que o BFS gasta bastante memória, uma vez que tem de expandir todos os nós no mesmo nível e, em consequência demora mais tempo, apesar de encontrar a solução mais acima na árvore em relação ao DFS, que demora menos tempo porque existe a possibilidade de uma solução não ótima ser encontrada primeiro no DFS que uma solução no BFS (que neste caso será sempre ótima visto que pesquisa em largura). O nosso DFS retorna a primeira solução que encontra, apesar de reduzida, existe a probabilidade de este ser mais eficiente e ótimo que o BFS, se a primeira solução que encontrar for a ótima.

Nas pesquisas informadas, o algoritmo A* é o melhor, uma vez que demora menos tempo, ocupa menos memória, pois, gera apenas os nós segundo a heurística e consegue alcançar a solução ótima em menos tempo em relação ao Branch and Bound por exemplo, que, no nosso caso precisa de gerar mais nós.

Nas pesquisas locais, para as duas variantes do Hill-Climbing, podemos concluir que o aleatório, por começar com uma solução aleatória não ótima, faz as iterações seguintes serem afetadas por esse ponto de partida, tornando a pesquisa não ótima. No nosso caso o Hill-Climbing sem aleatoriedade, usa uma heurística que torna a seleção dos próximos nós sempre melhor em relação ao anterior, otimizando a cada iteração a solução.

Por fim, o Simulated Annealing começa também com uma solução aleatória e vai escolhendo soluções melhores, combinando uma função de probabilidade com a função heurística.

4. Aplicação de Programação por Restrições (CLP)

4.1 Modelo matemático do problema

Um modelo matemático pode ser definido por três componentes: variáveis, domínios e restrições. Face ao problema Vigilância de Retângulos, decidimos definir o modelo matemático da seguinte forma:

As variáveis são as seguintes:

X -Indicador se um ponto i é guarda (1 true, 0 false);

Y -Indicador se um ponto pertence a um retângulo j (1 true, 0 false);

Z -Indicador se um retângulo está a ser vigiado por um guarda (1 true, 0 false);

Nota: i (representa um ponto), $i \in [1 , 2n + 2]$;

j (representa um retângulo), $j \in [1 , n]$;

n (número total de retângulos), $n \in N$

Os domínios do modelo são os seguintes:

$X_i \in \{0,1\}$

$Y_{i,j} \in \{0,1\}$

$Z_j \in \{0,1\}$

Os domínios funcionam da seguinte forma:
Todos eles representam valores de verdade para cada variável (1,0).

$\sum_{k=1}^{n} y_{ik} \leq 3;$

$z_j \geq x_{i^*} y_{ij};$

$\sum_{k=1}^{n} z_k == n$

As restrições funcionam da seguinte forma:

O primeiro somatório diz que o número máximo de retângulos adjacentes em cada ponto é três. A segunda restrição diz o valor de verdade em relação à existência de uma guarda naquele retângulo j (verificando se o ponto i é guarda e se o ponto i pertence ao retângulo j). Há que realçar o facto de o retângulo poder ser vigiado por outro ponto, daí a restrição ter o símbolo “>=”. A terceira restrição verifica se a soma dos retângulos visitados é igual ao número total de retângulos.

4.2. Implementação do modelo matemático

Apesar de não termos conseguido implementar o modelo em código, pelo qual não fará parte deste trabalho, criámos uma teoria que pensamos ser correta para a criação de um programa.

A ideia consistia no seguinte: Criar uma função para cada restrição, a qual iriamos chamar a cada criação de um nó. Nas classes que possuímos no programa, temos guardada toda a informação necessária para passar para as restrições (os retângulos visitados daquele ponto, a flagGuard e os pontos adjacentes a um retângulo). A ideia seria eliminar as opções de guardas com base nas restrições à medida que progredíssemos na árvore, através da pesquisa DFS. Quando a terceira restrição fosse verdadeira, retornaríamos esse caminho como solução.

5. Nota

As ideias para a parte 4 foram debatidas com o grupo 42.

6. Bibliografia

1. Martins AM. Optimização Geométrica em Problemas de Visibilidade: Soluções Metaheurísticas e Exactas. Disponível em:

http://sweet.ua.pt/leslie/AMM_PhD.pdf (acedido em 13/05/2020).

2. Bealdung. Example of Hill Climbing. Disponível em:

https://www.baeldung.com/java-hill-climbing-algorithm?fbclid=IwAR2DT8os2ZKQOPbYZApvF3s_AgLNVsJi5FY6U_bSr2DJfyl_D9lOTxzrbpw (acedido em 09/05/2020).

3. JavaTpoint. Hill Climbing Algorithm in Artificial Intelligence. Disponível em:

https://www.javatpoint.com/hill-climbing-algorithm-in-ai (acedido em: 09/05/2020).

4. GeeksforGeeks. Introduction to Hill Climbing | Artificial Intelligence. Disponível em:

https://www.geeksforgeeks.org/introduction-hill-climbing-artificial-intelligence/ (acedido em: 09/05/2020)

5. Borba L, Ritt M. A heuristic and a branch-and-bound algorithm for the AssemblyLine Worker Assignment and Balancing Problem. Computers & Operations Research. 2014; 45: 87-96.

6. Artificial Intelligence Foundations of Computational Agents. Branch and Bound. Disponível em:

https://artint.info/html/ArtInt_63.html?fbclid=IwAR3b7mF5XKydGS1_cM_BktHbz0OtBFuFKUkb6CVCyVr6DG4rJV3wO1FQBug#dfbb-search-fig (acedido em 11/05/2020)

7. Hackerearth. Basic of Greedy Algorithms. Disponível em:

https://www.hackerearth.com/practice/algorithms/greedy/basics-of-greedy-algorithms/tutorial/ (acedido em 12/05/2020)

8. DCC. Estratégias informadas de Busca. Disponível em:

https://www.dcc.fc.up.pt/~ines/aulas/1819/IA/buscas_informadas_new.pdf (acedido em

5/05/2020)

9. DCC. Estratégias não informadas de Procura/Busca. Disponível em:

https://www.dcc.fc.up.pt/~ines/aulas/1819/IA/buscas_nao_informadas.pdf (acedido em 5/05/2020)