

# **Postos de Vigia**

## **(Set Cover Problem)**

---

Trabalho 1

CC2006 – Inteligência Artificial

Diogo Barbosa

(up201805448)

## 1. Abordagem

O objetivo do trabalho é aplicar diversos algoritmos conhecidos da área da inteligência artificial ao problema dos Postos de Vigia. Para isso, decidiu-se envolver todo o trabalho à volta de uma ferramenta que, de forma interativa, responde a todas as questões.

Esta abordagem ao problema permitiu a criação de diversas camadas de abstração que tornaram o trabalho menos verboso e mais modular.

O ficheiro *README.md* contém instruções para a utilização desta ferramenta assim como informações sobre os ficheiros de input e de configurações.

## 2. *State, Approach* e execução

Os três elementos principais que resumem o funcionamento da ferramenta são o *State*, a *Approach* e a classe principal *PartitionProblem*.

### 2.1. *State*

O estado, representado na classe *State*, representa um estado arbitrário do problema. Isto é, um conjunto de vértices escolhidos, que consequentemente indicam o conjunto de vértices não escolhidos, retângulos cobertos e retângulos não cobertos.

A razão para o uso de dois *HashSets* em vez do *array de booleanos* para representar os vértices escolhidos e não escolhidos está em obter todos estes dados com complexidade  $O(1)$ . Isto porque, por exemplo, para obter a lista de vértices não escolhidos com o *array*, a complexidade temporal seria  $O(N)$ , pois teríamos de procurar todas as posições *false*. Além disso o uso da interface *Set* traz diversos métodos úteis como *addAll* e *removeAll* enquanto previne que existam estados duplicados dentro da estrutura de dados. O mesmo acontece com os retângulos cobertos e não cobertos.

O estado contém também métodos auxiliares que realizam operações que são necessárias durante a execução das *Approaches*, as quais serão introduzidas no ponto 2.2.

Dentro destes métodos estão incluídos:

- *expand* – Expande o estado atual e retorna uma lista de estados que provêm da seleção de um dos vértices não selecionados.

- *chooseVert* e *unchooseVert* – Retornam os estados resultantes da seleção ou remoção de um vértice.
- *isFinal* – Retorna um booleano que indica se a solução é ou não final (se todos os retângulos necessários estão cobertos)
- *getSolution* – Retorna um valor que indica quão boa é a solução. Quanto menor, melhor. Para as soluções finais, este valor corresponde ao número de vértices escolhidos. Para soluções não finais, é adicionado o número total de vértices mais um, de forma a fazer com que a pior solução final seja melhor do que ter 0 vértices escolhidos. Isto vai ser útil em algumas *Approaches*.

## 2.2. Approach

Esta é a classe que impulsiona a modularidade da ferramenta. É uma classe abstrata constituída por uma variável que conta o número de estados expandidos, para servir como métrica para comparação dos diferentes métodos, um estado *currentState* e o método abstrato *solve* que implementará a resolução específica de cada *approach*. Estas classes serão descritas no ponto 3 deste relatório. Juntamente com esta classe, foi criada uma classe *CLPApproach* para diferenciar as métricas.

## 2.3. Execução

O fluxo de execução da ferramenta é definido na classe *PartitionProblem*. Após ter sido selecionada a abordagem e o ficheiro de input, será gerada uma classe que estende a *Approach* e depois será executado o método *solve* antes de cada instância do problema. No final de cada instância será retornado um resultado e é dada a opção de exportar os vértices selecionados no estado final para um ficheiro de output.

A resolução instância a instância permite que só seja resolvida uma instância de um ficheiro muito grande de cada vez. Como estas podem levar muito tempo, isto permite ter uma interação mais satisfatória com o problema.

## 3. Implementações

Ao longo deste ponto vão ser resumidamente descritas as implementações da classe *Approach*, diretamente ligadas aos objetivos do trabalho.

### 3.1. Estratégias *Greedy*

#### 3.1.1. Mais Retângulos Cobertos

Nesta estratégia é usada uma heurística que se baseia no número de retângulos que queremos cobrir e que ainda não estão cobertos. Começa-se por um estado inicial vazio e a cada iteração, até chegar a uma solução final, escolhe-se o vizinho cujo valor da heurística mencionada seja menor. Em caso de empate, escolhe-se o primeiro.

#### 3.1.2. Retângulo Mais Difícil Primeiro

Esta estratégia segue o mesmo padrão da anterior. No entanto, o critério de seleção é diferente. A cada iteração, é criado um mapa cujas chaves são os retângulos que queremos cobrir e os valores são o número de estados vizinhos em que ficam cobertos. O retângulo cujo valor no mapa é menor é considerado o mais difícil, portanto escolhemos o estado que nos garante que este fica coberto.

### 3.2. BFS

A implementação desta *approach* segue o algoritmo BFS sem olhar a aspetos intrínsecos do problema. Esta é uma das vantagens de deixar esse trabalho para a classe *State*. Foram criadas duas variações para este algoritmo. Numa delas, paramos no primeiro estado final encontrado. Na outra, percorremos todo o espaço de resultados retornando a melhor solução encontrada. Nesta segunda é também adicionada a condição de que se encontrarmos uma solução com o valor de *getSolution* igual a  $\text{Math.ceil}(R/3)$  então não é necessário continuar a pesquisa.

### 3.3. DFS

Esta implementação é análoga ao BFS. É implementado um algoritmo DFS com ambas as variantes mencionadas no ponto anterior, assim como a condição de paragem.

### 3.4. IDDFS

É implementado o algoritmo *Iterative Deepening Depth First Search* com um valor K definido no ficheiro *config.properties*, que corresponde à altura máxima inicial do algoritmo. Esta *approach* é a execução iterada de um DFS com altura máxima igual a K, aumentando o K a cada iteração. Verificam-se melhores resultados com este método do que com DFS, por exemplo.

### 3.5. Branch and Bound

Esta implementação tira partido da heurística já referida anteriormente que corresponde ao número de “retângulos objetivo” que faltam cobrir. Esta heurística é admissível porque cada ramo da árvore de soluções tem custo 1 e esse custo está ligado à escolha de 1 vértice. Assim, o custo real nunca vai ser maior que o valor da heurística, já que é necessário pelo menos um vértice por retângulo de forma a cobri-los.

Nesta *approach* faz-se uma pesquisa a partir de uma *heap* cujo comparador se baseia na heurística associada (*branch*) e quando se encontra uma solução, limita-se a pesquisa a outras soluções não piores que essa (*bound*).

Foi também introduzida a condição de paragem referida no ponto 3.2.

### 3.6. A\*

O algoritmo A\* é análogo ao *Branch and Bound*. A única diferença encontra-se no cálculo do custo no comparador da *heap*. Em vez de se considerar apenas a distância estimada até à solução, soma-se a essa também o custo do estado atual. Esta pequena alteração trouxe resultados interessantes mencionados no ponto 5.

### 3.7. ILS

No *Iterated Local Search*, são repetidos K vezes (valor definido no ficheiro das propriedades) dois passos: *Local Search* e Perturbação. Durante a fase de local search, é usado um algoritmo para encontrar um mínimo local dentro do espaço de soluções. É aqui que se torna importante o desvio no valor de *getSolution* mencionado no ponto 2.1. Foi também implementada uma variante com randomização em que por vezes pode ser aceite uma solução errada com probabilidade definida nas propriedades. Assim, aumenta-se a chance de sair fora de um mínimo local para que na próxima iteração se encontre, possivelmente, o mínimo absoluto.

### 3.8. *Simulated Annealing*

Esta *approach* resume-se a duas funções: aceitação e perturbação. Inicia-se com um parâmetro  $T$  e com um *cooling rate*, definidos em *config.properties*. Enquanto  $T$  é maior que 1, vamos baixando  $T$  com o rácio *cooling rate* e a cada iteração é realizada uma perturbação na solução atual que depois é passada pela função de aceitação. Nesta função há duas possibilidades. Se a nova solução for melhor que a atualmente aceite, aceita-se sempre a nova. Caso contrário, pode ainda ser aceite dependendo de uma probabilidade baseada no quão afastada da solução atual está e do quão alto é o valor de  $T$  atualmente. Isto cria um sistema muito volátil inicialmente que vai aumentando a sua estabilidade ao longo do tempo, procurando não ficar preso em mínimos locais. Esta foi uma abordagem muito interessante pelo facto de chegar à solução ótima sem recorrer ao uso direto de heurísticas.

### 3.9. MAC – AC3

Entrando no domínio dos *Constraint Satisfaction Problems*, usou-se o algoritmo AC3 nesta abordagem.

Inicialmente, foram definidos os domínios. Para este problema específico, estes são conjuntos de vértices. Primeiro, é definido um vazio que irá conter os vértices da solução e depois é criado mais um domínio para cada “retângulo objetivo” que contém os vértices que cobrem esse mesmo retângulo.

Depois definem-se os arcos. Os arcos são ligações entre dois domínios. Neste caso concreto, os arcos são ligações entre o primeiro domínio mencionado e cada um dos segundos. Além disso, o arco contém uma função que indica se é consistente. Para este problema, um arco é consistente se o domínio da esquerda (solução) contiver pelo menos um elemento do da direita.

No início da execução, adicionam-se todos os arcos numa fila. Se este arco não for consistente, realizam-se as operações necessárias, alterando o domínio da esquerda, e volta-se a adicionar na fila todos os arcos que usam este domínio agora alterado. Isto é feito até a fila estar vazia.

## 4. CLP Approaches

Para responder aos objetivos do trabalho, as últimas duas implementações foram escritas na linguagem Prolog, mais especificamente com ECLiPSe CLP. Para as integrar na ferramenta foi usada a *Java-ECLiPSe Interface* e foi também criada a classe *DataConverter* que converte os dados da instância para um formato ideal que será lido pelo *ECLiPSe*. Os métodos de pesquisa para estas implementações podem ser configurados no ficheiro de propriedades.

### 4.1. Implementação no Problema Inicial

Para resolver o problema proposto, foi primeiro definido o modelo matemático:

Seja  $V$  o conjunto dos vértices,  $R$  o conjunto dos retângulos e  $V_r$  o conjunto dos vértices que vigiam  $r \in R$ .

Variáveis –  $x_v \in \{0, 1\}$  em que se  $x_v = 1$ , então o vértice  $v$  foi selecionado

Modelo – minimizar  $\sum_{v \in V} x_v$  sujeito a

$$\left\{ \begin{array}{l} \sum_{v \in V_r} x_v \geq 1 \text{ para todo o } r \in R \\ x_v \in \{0, 1\}, \text{ para } v \in V \end{array} \right.$$

Após ter o modelo definido, a resolução do problema baseou-se na criação da lista das variáveis, definição das restrições e chamada do predicado *search/6* com os parâmetros definidos em *config.properties*.

### 4.2. Implementação no Problema das Cores

Para abordar o problema das cores, primeiro decidiu-se separar os problemas, visto que no fundo são problemas diferentes. Após resolver o primeiro problema, definiu-se então um novo modelo:

Seja  $V'$  o conjunto dos vértices escolhidos e  $V'_r$  o conjunto dos vértices de  $V'$  que vigiam  $r \in R$ .

Variáveis –  $c_v \in \{0, \dots, \#V'\}$  que corresponde às cores escolhidas

Modelo – minimizar  $\sum_{v \in V} c_v$  sujeito a

$$\left\{ \begin{array}{l} c_v > 0 \text{ e são diferentes para } v \in V'_r \text{ e para todo o } r \in R \\ c_v = 0 \text{ para vértices de } V \text{ que não estão em } V' \end{array} \right.$$

A decisão de minimizar a soma dos  $c_v$  leva a que o predicado de minimização da biblioteca *branch\_and\_bound* tente usar sempre os identificadores de cores mais baixos. Isto leva também a minimizar o número de cores escolhidas, pois vai sempre ser dada prioridade à cor 1, por exemplo, e só mesmo em caso de não ser possível será tentada a cor 2 e assim sucessivamente.

## 5. Observações

Ao longo deste ponto, vamos analisar as respostas dadas pela ferramenta à primeira instância de três ficheiros incluídos no projeto: *data1.txt*, *data2.txt*, *data3.txt*. Vamos chamar estas instâncias de instância 1, 2 e 3 respetivamente. Além disso, vai-se também referindo modificações ou funcionalidades extra que poderiam melhorar este trabalho.

### 5.1. Algoritmos *Greedy*

Como previsto, as abordagens *greedy* apesar de rápidas podem retornar uma solução que não é a solução ótima. No primeiro algoritmo, encontramos as soluções ótimas nas instâncias 1 e 3, mas uma não ótima na instância 2. No segundo algoritmo, não é encontrada nenhuma solução ótima e o número de estados visitados é maior. Isto demonstra que a escolha desta segunda abordagem não foi a mais acertada. Poderia ser melhorada ou trocada por outra com resultados mais satisfatórios.

### 5.2. BFS

Com BFS é encontrada a solução ótima nas instâncias 1 e 2. No entanto, o algoritmo leva imenso tempo na terceira instância. Na variante em que procuramos a solução ótima, só é parado em tempo útil devido à condição de paragem definida no ponto 3.2.

### 5.3. DFS

Com DFS não é encontrada a solução ótima logo na instância 1. Isto seria de esperar visto que o algoritmo procura soluções em profundidade na árvore de pesquisa. Na variante em que se procura a solução ótima, esta abordagem é extremamente lenta até mesmo para instâncias pequenas.



## 5.4. IDDFS

Nesta abordagem, já conseguimos atingir a solução ótima na instância 1, ao contrário do DFS. No entanto, este algoritmo visita imensos estados antes de chegar a esta solução (363582 em comparação com 7670 no BFS). Isto poderia ser um pouco melhorado definindo em *config.properties* o K inicial deste algoritmo com um valor mais alto. O ideal para a instância em causa seria 4, já que sabemos que a solução nunca será melhor que 4 para 10 retângulos.

## 5.5. *Branch and Bound* e A\*

O *Branch and Bound* encontra em todas as instâncias a solução ótima, mas é muito explosivo também. Visita 193457 estados na instância 2.

Quando passamos para o A\*, é interessante ver o quanto uma pequena alteração melhora o algoritmo. Para a mesma instância 2, este algoritmo visita 66099 estados. Perto de um terço dos visitados em *Branch and Bound*.

## 5.6. *Iterated Local Search*

Chegou-se a resultados interessantes na instância 2.

Com os valores definidos por defeito para o passo de perturbação e com 500 iterações (também por defeito), nenhuma das variantes, com e sem randomização, chega à solução ótima na instância referida. No entanto, aumentando para 10000 iterações, a solução ótima é encontrada na variante com randomização. Isto mostra que com mais tentativas e mais randomização, consegue-se sair do mínimo local 5 e eventualmente descer até ao mínimo absoluto 4. A solução ótima é também atingida na instância 3.

## 5.7. *Simulated Annealing*

Como esta abordagem se baseia apenas na perturbação e aceitação com probabilidade, o número de estados visitados nesta abordagem é igual independentemente da instância do problema. Depende apenas dos parâmetros. Isso torna o *simulated annealing* um algoritmo aparentemente bom para usar em qualquer tipo de instância visto que encontrou com sucesso e de forma bastante eficaz as soluções ótimas em todas as 3 instâncias usadas em teste.

### 5.8. MAC – AC3

Este algoritmo, embora não encontre as soluções ótimas em nenhuma das 3 instâncias de teste, resolve-as muito rapidamente. Na instância 3, todas as outras abordagens levaram algum tempo a encontrar a solução enquanto que com AC3 foi praticamente imediato. Consegue-se observar através do número de arcos avaliados que o número de iterações é muito menor.

### 5.9. CLP

Nas implementações em *ECLiPSe CLP*, chegamos sempre à solução ótima, o que indica que os modelos foram bem definidos. No entanto, se fosse dada mais liberdade na ordenação dos vértices, por exemplo, poderiam ser experimentadas outras estratégias talvez mais eficientes.

## 6. Observações Finais

Este foi um trabalho que sem dúvida me ajudou a entender mais sobre a área e me fez aprender imenso. Tenho a certeza de que, com mais algum tempo, poderiam ser explorados outros pequenos detalhes de otimização nomeadamente na escolha de heurísticas e nos passos de perturbação que dariam resultados mais satisfatórios.