# PROJETO DE VIGILÂNCIA DE PARTIÇÕES RETANGULARES

PROJETO DE INTELIGÊNCIA ARTIFICIAL

Guilherme Pereira, up 201809622
Nuno Taveira, up201809591

Cadeira Inteligência Artificial

Professora Ana Paula Tomás

MAIO, 2020

FACULDADE DE CIÊNCIAS DA UNIVERSIDADE DO PORTO

U.PORTO
FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

# Índice

# INTRODUÇÃO

No âmbito cadeira de inteligência artificial, lecionada pela professora Ana Paula Tomás, foi proposto realizar um projeto que consiste num problema, com o objetivo de vigiar partições retangulares, com aplicação de diversos algoritmos, de forma, a perceber melhor as diferenças e semelhanças das aplicações dos algoritmos no mesmo.

Os problemas realizados, ao longo do trabalho, foram realizados separadamente. Os problemas Greedy, BFS, DFS, IDS, SA e CP-MAC com AC3 foram realizados pelo aluno Guilherme Pereira (GP). E os restantes problemas, A*, B&B e Hill Climbing, foram realizados pelo aluno Nuno Taveira (NT).para cada aplicação dos algoritmos vão ser descritos os principais focos da implementação e detalhes que sejam importantes fazer referência.

Antes de abordar as estratégias, em todas as implementações feitas por GP , foi utilizado uma classe Coor. Esta classe é utilizada na sua totalidade em todas as implementações em java e representa os vértices do problema (apresenta Override de forma a poder ser utilizado corretamente nas Hashtables), existe ainda algumas funções que são repetidas e que são utilizadas em diversas implementações. Há ainda a opção para permitir diminuir o espaço de pesquisa em subconjuntos , mas que se encontra comentado no main (Linhas 96-97) de forma a poder aceitar mais do que uma instância.

Ao longo do relatório vai ser utilizado a definição Atinge Solução Ótima (ASO) , que é utilizado para classificar se um algoritmo atinge solução ótima para o número de retângulos dado. Aqui a solução ótima são soluções com menor número possível de guardas, seja n = número de retângulos , uma solução ótima é aquela que apresenta n/3 guardas , arredondado para cima.

## 1. GREEDY STRATEGY

Nesta implementação, foi utilizado uma lista de listas (LinkedList<LinkedList> list_list) de forma a representar os retângulos do problema (seja, por exemplo, o numero de retângulos dez, então os índices da list_list vão desde zero até nove) e consequentemente os vértices de cada retângulo , tendo em conta que apenas utilizamos esta lista no inicio como forma de inicializar as nossas estrutura de dados. Como estrutura de dados, é utilizado duas Hashtables, uma que guarda o id do retângulo e os vértices correspondentes (id_Coor) e outra que para cada vértice indica a sua ocorrência nos retângulos ainda não visitados (coor_Ocorr).

Assim, após ser lido o input e ser guardados os dados nas estruturas correspondentes, vai ser guardado numa lista (LinkedList<Coor> guards) os vértices de acordo com o numero de ocorrências que apresentam (dado pela Hashtable coor_Ocorr), sendo que no topo da lista vai ser retido os vértices de maior ocorrência e no fim da lista os vértices de menor ocorrência. Com isto tudo, a resolução do problema torna-se simples pois apenas temos de retirar os vértices do topo da lista até cobrir os retângulos todos (tendo em conta que cada vez que retiramos um vértice e assumimos o vértice como guarda , temos de atualizar as estruturas de dados já que o espaço de pesquisa é reduzido de cada vez).

**Testes:**

- 10 retângulos – ASO
- 20 retângulos – ASO
- 30 retângulos – Não ASO (+2 guardas) *1
- 40 retângulos – Não ASO (+3 guardas) *1
- 50 retângulos – Não ASO (+2 guardas) *1

*1 – Sensivelmente, dependendo no input os resultados podem variar entre +(1,..,3) guardas.

## 2. BFS

No BFS, é necessário explorar todas as subárvores existentes. Logo, de forma a manter os dados consistentes e conseguir explorar tudo. Utiliza-se duas estruturas de dados estáticas, uma para guardar as ocorrências de cada vértice ao longo da pesquisa (Hashtable<Coor,Hashtable> sv_Ocorr) e um para indicar os retângulos cobertos (Hashtable<Coor,boolean[]> sv_Arr). Alem disso, utiliza-se ainda uma outra variável estática para guardar os guardas que foram colocados (Hashtable<Coor,LinkedList> sv_Solut). Isto tudo, é feito de forma a manter a pesquisa progressiva, sendo que sempre que colocamos/encontramos um guarda novo , atualizamos nas Hashtables respetivas com a informação do pai (guarda anterior).

Nesta implementação, existe algumas funções dedicadas importantes, gerirViz(), str_State(), updateStates() e checkNewStates(). A função, gerirViz(), utiliza como argumento o Hashtable<Coor,Integer> coor_Ocorr e o boolean arr_Rec[] , que são usados para representar o estado de pesquisa em que o BFS se encontra (arr_Rec é um array de booleanos que indica os retângulos que ainda estão por vigiar). Com tudo, esta função gera os filhos/vizinhos a partir de um certo estado. Os filhos que esta função retorna , são aqueles que pertencem aos retângulos com menor número de vértices e que apresentam maior número de ocorrência (existe casos em que consideramos vértices comparáveis, isso ocorre quando os vértices escolhidos apresentam o mesmo número de ocorrências e os retângulos que estes vigiam não são iguais, ou seja, todos distintos). No entanto quando começamos a pesquisa por BFS, começamos no str_State() (Start State) que vai gerar (a partir do gerirViz()) os candidatos iniciais. Se apenas for retornado um vértice como candidato, iniciamos a pesquisa por esse vértice , mas se gerarmos mais do que um vértice (N vértices) , pela definição do BFS , temos de percorrer todas as subárvores e, portanto, iniciamos a pesquisa (N vezes) a partir de cada vértice gerado.

Na função BFS, as funções updateStates() e checkNewStates() a partir da estrutura de dados dada, no caso do updateStates() dado um vértice C (Coor c), esta função atualiza os dados assumindo C como guarda, enquanto que a função

checkNewStates() a partir do gerirViz() vai gerar os novos vértices , adiciona-los à queue e copiar a informação sobre a pesquisa para a estrutura de dados respetiva (Hashtables) mas com uma nova chave (novos vértices gerados).

**Testes:**

- 10 retângulos – ASO
- 20 retângulos – Não ASO (encontra soluções com 7-10 guardas)
- 30 retângulos – Não ASO (encontra soluções com 11-14 guardas)
- 40 retângulos – Não ASO (encontra soluções com 15-20 guardas)
- 50 retângulos – Não ASO (encontra soluções com 17-27 guardas)

## 3. DFS

Em contraste ao BFS, o DFS devido a sua caracterização perante algoritmo, não explora todas as subárvores existentes da solução. Contudo na implementação feita mantem se as mesmas funções (gerirViz(),updateStates(),etc) , mas com algumas modificações ligeiras. Na função str_State(), ao gerar os vizinhos invés de iniciar a pesquisa a partir de cada vértice (como no BFS) todos os vértices são colocados na stack , pois ao retirar um vértice da stack os próximos estados vão ser igualmente adicionados na stack.

**Testes:**

- 10 retângulos – ASO
- 20 retângulos – ASO *1
- 30 retângulos – ASO *1
- 40 retângulos – Não ASO *2
- 50 retângulos – Não ASO (encontra soluções com 17-23 guardas)

*1 – Poderá existir casos com +1/+2 guardas.

*2 – Maior parte dos resultados encontram soluções com 16 guardas, mas existe casos em que encontra soluções com um número de guardas entre 17-18.

## 4. IDS

No Iterative Deepening Search, a implementação assemelha-se ao DFS mas com algumas alterações de modo a manter a estrutura do algoritmo. No IDS , foi adicionado uma variável estática MaxDepth que consiste num contador da profundidade máxima (começa em zero) de forma a que quando atingirmos a profundidade máxima é incrementado o contador e recomeçamos a pesquisa. De forma a manter a pesquisa progressiva, até atingir uma solução, a chamada da função IDS() é feita na função str_State() em que sempre que for retornado uma lista vazia (Linha 435 e 457) é feito parcialmente um reset das estruturas de dados e é recomeçado a pesquisa.

**Testes:**

- 10 retângulos – ASO
- 20 retângulos – ASO *1
- 30 retângulos – ASO *2
- 40 retângulos – Não ASO (encontra soluções com 15-18 guardas)
- 50 retângulos – Não ASO (encontra soluções com 20-22 guardas)

Nota : Existe um bug para a resolução do problema para cobrir apenas um subconjunto dos retângulos.

*1 – Existe casos em que se encontra soluções com +2 guardas.
*2 – Maioritariamente encontra resultados com +1 guarda, mas existe instâncias que encontra com +2 guardas.

# 5. A* (A STAR), BRANCH AND BOUND AND HILL CLIMBING (LOCAL SEARCH)

Cada programa contém as classes:

- Rectangle_<Nome_do_Algoritmo> (Represent um Retângulo);
- Vertice_<Nome_do_Algoritmo> (Representa um Vertice);
- State_<Nome_do_Algoritmo> (Representa um estado de pesquisa);
- Field_<Nome_do_Algoritmo> (Usado somente para representar o campo);
- Guardas_<Nome_do_Algoritmo> (Onde contém o main).

A razão de, em todos os programas as mesmas classes terem nomes diferentes, é porque poderia ser necessário efetuar uma alteração num método de uma classe específica do algoritmo e deste modo era possível efetuar sem efeitos colaterais.

Rectangle_<Nome_do_Algoritmo>:

- Propriedades:
  - aVertices - Array que contém os índices dos vértices desse retângulo
  - iNumVertices - Número de vértices do retângulo usado como propriedade para evitar a chamada do aVertices.length e assim ser menos linhas de código
  - iGuarded - Número de guardas que vê o retângulo
  - iSub - Inteiro que faz de booleano. Indica se pertence ao subconjunto que o utilizador quer que seja vigiado.
- Construtores:
  - Rectangle(int) - Novo Retângulo com x número de Vértices
  - Rectangle(Rectangle) - Usado para fazer uma cópia exata de um Retângulo

- -Métodos:
  - AddVertice(int) - Acrescenta o id de um Vertice ao retângulo
  - AddGuard() - Incrementa o número de guardas que vigia o retângulo
  - DecreaseGuard() - Decrementa o número de guardas que vigia o retângulo
  - GetGuarded() - Retorna o número de guardas que vigia o retângulo
  - GetVertices() - Retorna o array de índices de Vértices que estão contidos no retângulo
  - GetSub() - Retorna 1 se faz parte do subconjunto a ser vigiado e 0 caso contrário
  - SetSub(int) - Setter da propriedade iSub

Vertice_<Nome_do_Algoritmo>:

- Propriedades:
  - iNumVertices - Número de vértices
  - id - Id do vértice
  - bHasGuard - Booleano que indica se existe um guarda naquela posição
  - lRectangles - Lista de retângulo que esse Vértice vê
  - iNumRectangles - Número de retângulos visto pelo vértice. Para evitar chamar o lRectangles.size()
  - x - posição x
  - y - posição y
- Construtores:
  - Vertice(int) - Novo vértice com id = int
  - Vertice(Vertice) - Usado para fazer uma cópia exata de um Vertice

- Métodos:
  - AddRectangle(int) - Adiciona o id do retângulo
  - CompareToXY(int,int) - Compara o x e o y
  - GetX() - Retorna o X
  - GetY() - Retorna o Y
  - GetRectangles() - Retorna os retângulos vigiados pelo vértice
  - GetId() - Retorna o id do vértice
  - GetGuarded() - Retorna se o vértice tem um guarda ou não
  - SetGuarded(Rectangle[]) - O guarded passa a true e propaga a vigia para os retângulos
  - UnsetGuarded(Rectangle[]) - O guarded passa a false e propaga a vigia para os retângulos

State_<Nome_do_Algoritmo>:

- Propriedades:
  - lGuardedVertices - Lista de vértices
  - aRectangles - Array de retângulos
  - numOfGuards - Número de guardas no estado
  - maxVerticeIndex - Onde foi colocado o último vértice
  - fitness - Por razões de performance, o fitness é guardado em variável
- Construtores:
  - State(LinkedList<Vertice_BranchBound>, Rectangle_BranchBound[]) - Novo estado com os vértices e retângulos
- Métodos:
  - GetNextState() - Retorna o próximo melhor estado
  - GetNextStates() - Retorna todos os estados possíveis a seguir

- GetFitness() - Retorna o Fitness do estado
- GetGuardedVertices() - Getter da propriedade lGuardedVertices
- GetRectangles() - Getter da propriedade aRectangles
- GetNumOfGuards() - Getter do numOfGuards
- PrintRectangles() - Escreve o número de guardas que vê cada retângulo
- PrintVertices() - Escreve a abcissa de cada vértice (deprecated, não serve para nada)
- compareTo(State)- Método da interface comparable usado principalmente para a heap do algoritmo A*

Field_<Nome_do_Algoritmo>:

- Serve somente para representar graficamente o campo de retângulos.
- Propriedades:
  - field - matriz com inteiros onde 1 significa que é m vértice, 2 significa que tem guarda e 0 significa que não tem nada
  - cField - matriz equivalente à matriz field onde o 1 é representado por 'O', e o 2 por 'X'
  - x - o campo varia de 0 a 2 * x
  - y - o campo varia de 0 a 2 * y
- Construtores:
  - Field (LinkedList<Vertice_BranchBound>, Rectangle_BranchBound []) - Novo campo
- Métodos:
  - PrintField - Escreve o campo na consola.

- Heurística:
  - A heurística é dada por um fitness que será:
    - Para cada retângulo r, será dado um fitness de:
      - ✓ f(r) = sub * 3 - (guards - 1)

*onde sub é o valor de 1 ou 0 a indicar se faz parte do subconjunto de retângulos que o utilizador quer ver

*onde guards é o número de guardas que vigia o retângulo

- O valor de fitness é o somatório de todos os f(r)

- Explicação dos métodos:
  - Definição de solução ótima consoante a heurística definida:
    - faz parte do subconjunto a vigiar === sub = 1
    - cada retângulo é vigiado apenas por um guarda === guards = 1
    - Ou seja:
      - ✓ Fitness = Somatório(1 * 3 - (1 - 1)) == Fitness = nºRetângulos * 3
  - Definição de solução sub-ótima consoante a heurística definida:
    - faz parte do subconjunto a vigiar === sub = 1
    - todos os retângulos são vigiados, mas pelo menos 1 ou 2 ou 3 deles é vigiado 2 vezes === guards = 1 em quase todos os retângulos e guards = 2 | 3 em um retângulo
    - Ou seja:
      - ✓ Fitness = Somatório(1 * 3 - (1 - 1)) + 1 * 3 - (2 - 1) == Fitness = (nºRetângulos-1) * 3 + 2
      - ✓ Ou Fitness = Somatório(1 * 3 - (1 - 1)) + 1 * 3 - (3 - 1) == Fitness = (nºRetângulos-1) * 3 + 1

### Testes A* search:

- 10 retângulos – ASO *1
- 20 retângulos – ASO *1
- 30 retângulos – ASO *1
- 40 retângulos – Não ASO *2
- 50 retângulos – Não ASO *2

*1 - Maioritariamente encontra solução ótima.

*2 - Não consegue encontrar solução por questões de eficiência.

### Testes Hill Climbing:

- 10 retângulos – ASO *1
- 20 retângulos – ASO *1
- 30 retângulos – Não ASO *2
- 40 retângulos – Não ASO *2
- 50 retângulos – Não ASO *2

*1 - Maioritariamente encontra solução ótima.

*2 - A partir daqui, é muito rápido, mas começa a ter problemas em encontrar um solução sub-ótima.

### Testes Branch and Bound:

- 10 retângulos – ASO *1
- 20 retângulos – Não ASO *2
- 30 retângulos – Não ASO *2
- 40 retângulos – Não ASO *2
- 50 retângulos – Não ASO *2

*1 - Maioritariamente encontra solução ótima.

*2 - Demora demasiado tempo a executar. Problemas de eficiência de código.

# 6. SA - SIMULATED ANNEALING

No Simulated Annealing foram implementadas algumas funções da implementação greedy, mais concretamente as funções utilizadas para inicializar a estrutura de dados e para introduzir na lista guards (Linha 57) os vértices ordenados por ocorrência. Isto foi feito, de modo a poder ter um ponto de partida para o Simulated Annealing.

De forma a representar os vértices que são colocadas como guardas, é utilizado uma Hashtable estática arr_X, que para cada vértice indica se é colocado como guarda (1) ou não (0). Existe ainda outras variáveis estáticas como a temperatura, que indica a temperatura inicial , o cooling factor , que ao longo da implementação mantem se constante (0.95) e o limit que indica a que instâncias vamos reduzir a temperatura.

Inicialmente, começamos por inicializar o arr_X a partir da função str_State() , altera -se as variáveis estáticas caso seja necessário e corremos o algoritmo.

Existe algumas funções que fazem de suporte para outras funções criadas, como por exemplo , para a função one_flip() , dado um vértice que vai ser retirado aleatoriamente a partir da função get_RandomCoor(), vai ser dado "flip" que consiste em remover ou colocar um vértice como guarda, que faz com que seja removido/adicionado incidências em certos retângulos , logo de forma a manter os dados consistentes existe duas funções, updateGuards() que é chamado quando colocamos um guarda num vértice arbitrário e updateBackGuards() quando removemos um guarda. Estas funções não só alteram os dados, mas também incrementam/decrementam o F_counter que representa o número de retângulos que não são vigiados pelo arr_X. É importante observar que no início do programa o F_counter é igual ao número de retângulos do problema e o G_counter é inicializado a zero. Estas duas variáveis são utilizadas para calcular o custo de um certo estado e são incrementadas/decrementadas devidamente após um "flip" de um vértice.

**Testes:**

Este algoritmo, devido ao facto de não retornar soluções corretas em maioria dos casos, apenas encontra soluções para problemas com uma instância, para dez retângulos em certas situações (Ver linha 316). Para problemas com mais retângulos, devido ao comportamento indefinido do programa , quando a temperatura se encontra a um baixo nível nem sempre a solução encontrada é uma solução para o problema em causa.

Nota: Na linha 330 existe uma opção para ver o algoritmo a correr (print dos vértices escolhidos, da temperatura e dos retângulos que estão ou não vigiados).

# 7. CONSTRAINT PROPAGATION – MAC COM AC3

**Modelo Matemático:**

Para o modelo matemático, estabelecemos as restrições para os vértices (X_i) e para os retângulos (R_i). Sabemos que para cada vértice do problema, X_i :: 0..2 , ou seja , para cada vértice o domino possível é entre 0 e 2, em que inicialmente X_i = 2 (o vértice ainda não foi visitado) e ao decorrer do tempo se for decidido que um vértice V , não é escolhido como guarda , então X_v = 0 , caso contrario, X_v = 1. Para os retângulos , R_i :: 0..1 definimos o domínio entre 0 e 1, em que se R_i = 0 o retângulo ainda não esta coberto caso contrário R_i = 1.

Sabendo isto, conseguimos concluir que para obter uma solução para todos os retângulos do problema R_i >= 1 e sabemos ainda que para cada vértice incidente a R_i a sua soma tem de ser maior do que zero, $\sum$ X_i_R >= 1.

Na implementação do MAC com AC3, ao começar o problema inicializamos para cada vértice o X_i a 2 e R_i a 0, que representa que os vértices e os retângulos ainda não foram visitados. Apos isto, corremos o DFS e ao gerir os vizinhos (função gerirViz()) , apenas vamos escolher o vértice que ainda não foi visitado com maior ocorrência que pertence ao retângulo que também ainda não visitado que apresenta menor número de vértices. Depois, se esse vértice for o último vértice do retângulo assumimos como guarda (X_i = 1) , caso contrário, continuamos a pesquisa sem considerar o vértice como guarda (X_i = 0).

**Testes:**

- 10 retângulos – ASO *1
- 20 retângulos – Não ASO (encontra soluções com 11-13 guardas)
- 30 retângulos – Não ASO (encontra soluções com 17-19 guardas)
- 40 retângulos – Não ASO (encontra soluções com 16-24 guardas)
- 50 retângulos – Não ASO (encontra soluções com 25-30 guardas)

*1 - Por vezes, encontra soluções com +1/+2 guardas)

**4 c) e 5:**

Para este exercício, apesar de não conseguirmos implementar por completo o exercício, iniciamos o exercício e tentamos completar o máximo possível.

Nota: O exercício 4 c) encontra-se no ficheiro recP1_GP.ecl e os dados no ficheiro dados_rec.

## CONCLUSÃO

Em suma , a partir dos testes realizados de cada algoritmo para um número variado de retângulos, os algoritmos que apresentaram melhor desempenho relativamente a atingir uma solução ótima, ou uma solução perto da solução ótima , são o A*, DFS e IDS. Enquanto que os algoritmos que apresentam pior desempenho são  o BFS, Branch and Bound e o MAC com AC3. No caso do BFS, devido a sua pesquisa aprofundada e devido a sua ineficiência, nem sempre encontra as melhores soluções mas apresenta maior confiabilidade, relativamente ao Branch and Bound devido a sua ineficiência apresenta dificuldades em encontrar soluções para um número de retângulos superior a dez. Todos os algoritmos foram testados da mesma maneira (com 12 instâncias para um número variado de retângulos) expeto o A*, Branch and Bound e o Hill Climbing que apenas foi testado para 1 instância mas com números variados de retângulos. Surgiram ainda outros problemas no Simulated Annealing, devido a sua implementação e no 4 c), pois falta lhe restrições.

# BIBLIOGRAFIA

Russell, S., & Norvig, P. (2010). Artificial Intelligence A Modern Approach Third Edition. In *Pearson*. https://doi.org/10.1017/S0269888900007724