

Procura e Planeamento

Campus da Alameda

IST @ 2018/2019

**Relatório – Projeto**

*Grupo 06*

*- Ana Margarida Pataca da Costa – 83425,*

*- Mariana Francisca Carrilho Loureiro – 83520.*

Índice

[1. Introdução 3](#_Toc531898314)

[2. Modelação do *Constraint Satisfaction Problem* 3](#_Toc531898315)

[2.1. Estrutura do CSP 3](#_Toc531898316)

[2.2. Geração de Sucessores 3](#_Toc531898317)

[2.3. Verificação das restrições 3](#_Toc531898318)

[2.4. Heurísticas 4](#_Toc531898319)

[3. Estratégias de procura implementadas 4](#_Toc531898320)

[3.1. Sondagem Iterativa 4](#_Toc531898321)

[3.2. *Iterative Limited Depth Search* (ILDS) 5](#_Toc531898322)

[3.3. Procura alternativa – *Depth-Bounded Discrepancy Search* (DDS) 6](#_Toc531898323)

[3.4. Melhor abordagem 7](#_Toc531898324)

[4. Comparação de resultados 7](#_Toc531898325)

[5. Melhorias a implementar 8](#_Toc531898326)

[6. Conclusão 8](#_Toc531898327)

[7. Bibliografia 8](#_Toc531898328)

# Introdução

O problema a que este projeto pretende dar resposta consiste em obter uma afetação completa das tarefas de condução de veículos de transporte, a realizar pelos trabalhadores durante um dia de trabalho, a turnos de serviço, respeitando todas as restrições do problema.

Para esse propósito, foi modelado um problema de satisfação de restrições, sobre o qual foram posteriormente executadas várias estratégias de procura. Este relatório analisa os resultados desses algoritmos. Os valores calculados pela execução do código do projeto, e apresentados abaixo, foram registados por um *core Intel i7* a 4.00GHz.

# Modelação do *Constraint Satisfaction Problem*

## Estrutura do CSP

Foi criada uma estrutura da linguagem *Lisp* para representar o conceito de *estado* na procura, com uma abordagem construtiva em mente. É composto pelos atributos *variáveis por atribuir* e *turnos com as tarefas já atribuídas*, que se representam em lista.

Adicionalmente, a estrutura mantém (para cada estado) os valores de *custo* (como definido no enunciado – somatório das durações temporais dos turnos definidos nesse estado, com um mínimo de 6 horas para cada turno), *número de turnos definidos*, *número de viagens sem serviço na atribuição dos turnos*, *número total de tarefas no problema* e uma variável para indicar se nesse estado foi acrescentado um *novo turno* em relação ao estado-pai, e a duração das tarefas que ainda não foram atribuídas.

A função de *custo* devolve o valor de custo descrito acima, e foram também definidas as funções *objetivo* e *igualdade* *de* *estados*, para além da função de *geração de sucessores* descrita em baixo. Desta forma, é possível testar todas as técnicas de procura fornecidas no ficheiro *procura.lisp*.

## Geração de Sucessores

O CSP foi modelado de forma a permitir ser explorado como um problema de otimização. Nesta perspetiva, a geração de sucessores é feita *incluindo* a verificação de restrições, para a maioria das procuras. Para a procura com as estratégias de *ILDS* e *DDS*, são gerados todos os estados-filhos potenciais, mesmo que violem restrições.

As tarefas são ordenadas temporalmente no início da execução do projeto, pelo que a solução modelada nunca assumiu um sucessor que atribuísse uma tarefa numa posição anterior a outra tarefa previamente atribuída – a nova tarefa é sempre colocada no final do novo turno.

Cada estado gera filhos atribuindo a nova tarefa a cada um dos turnos já existentes (um novo filho por turno), mais um estado-filho adicional em que a tarefa é a primeira num novo turno gerado.

## Verificação das restrições

Para todas as procuras exceto *ILDS* e *DDS*, a verificação de restrições é feita durante a fase de geração de sucessores, de forma a imediatamente descartar a geração de estados inválidos. Todos estes estados-filhos são devolvidos apenas após a validação das restrições, e todos eles respeitam todas as restrições. Para o *ILDS* e o *DDS*, a validação é feita no final de cada iteração da estratégia.

O teste das restrições é efetuado para cada turno do estado proposto, e é durante esta fase que se calculam e definem, para cada estado, o *número de viagens sem serviço na atribuição dos turnos* e o *custo*. O turno é iterado uma só vez para todo este processo. As consistências espacial e temporal são também validadas durante a iteração. Desta forma, a complexidade desta fase é linear.

**Restrições**:

1. Todos os turnos devem começar no local L1;
2. A duração máxima de um turno de serviço é de 8:00;
3. A duração de um turno de serviço é calculada do início do serviço até ao fim do serviço. Se a duração for menos de 6 horas, então conta como se tivesse 6 horas;
4. Os turnos de serviço devem ter no máximo uma pausa para refeição;
5. Uma pausa de refeição deve ter a duração de 40 minutos;
6. O trabalhador não pode tomar refeição enquanto está a ser transportado;
7. O tempo de condução antes de ser necessário uma pausa de refeição não pode exceder as 4:00;
8. O tempo de condução inclui potenciais espaços entre tarefas que não sejam tomadas de refeição, isto é conta-se todo o tempo desde o inicio da primeira tarefa do bloco de tarefas até ao fim da última tarefa do bloco de tarefas. Nesta contabilização, no caso de a primeira tarefa do bloco ser a primeira tarefa do turno e o seu local de início não ser L1, deve-se incluir o tempo de deslocar o trabalhador de L1 até esse local (é necessário usar o mesmo raciocínio para a última tarefa de um turno e o seu local de fim).

## Heurísticas

A heurística 1 é admissível. A variável *numT* corresponde ao número de turnos já atribuídos no estado. A variável newT, corresponde aos novos turnos criados no estado.

Na heurística 2 o tempo começa a 0. O tempo do turno corresponde à soma da duração de cada tarefa. Os tempos de todos os turnos são somados.

As heurísticas encontradas e definidas acima foram testadas para 5 problemas fornecidos para teste na página da cadeira. Os resultados foram registados nas seguintes tabelas:

|  |  |  |
| --- | --- | --- |
| *Problema\Heurística* | **Heurística 1 (H1)** | **Heurística 2 (H2)** |
| **1** |  |  |
| **2** |  |  |
| **3** |  |  |
| **4** |  |  |
| **5** |  |  |

Tabela 1 – Número de turnos na solução

|  |  |  |
| --- | --- | --- |
| *Problema\Heurística* | **Heurística 1 (H1)** | **Heurística 2 (H2)** |
| **1** |  |  |
| **2** |  |  |
| **3** |  |  |
| **4** |  |  |
| **5** |  |  |

Tabela 2 – Tempo (s) de execução

|  |  |  |
| --- | --- | --- |
| *Problema\Heurística* | **Heurística 1 (H1)** | **Heurística 2 (H2)** |
| **1** |  |  |
| **2** |  |  |
| **3** |  |  |
| **4** |  |  |
| **5** |  |  |

Tabela 3 – Nós gerados na execução

Tabela 4 – Nós expandidos na execução

|  |  |  |
| --- | --- | --- |
| *Problema\Heurística* | **Heurística 1 (H1)** | **Heurística 2 (H2)** |
| **1** |  |  |
| **2** |  |  |
| **3** |  |  |
| **4** |  |  |
|  |  |  |

# Estratégias de procura implementadas

Os algoritmos foram implementados de forma a enviar várias sondas ao longo de um tempo útil para explorar o máximo de caminhos possíveis, tendo como objetivo encontrar o caminho com menor custo.

## Sondagem Iterativa

|  |  |
| --- | --- |
| **1.**  **2.**  **3.**  **4.**  **5.**  **6.**  **7.**  **8.**  **9.** | **ISP**(node):  solucao\_optima = NULL  **while** t\_util**:**  solucao = ISPProb(node)  **if** solucao\_optima == NULL:  **solucao\_optima = solucao**  **else if custo(solucao\_optima) > custo(solucao):**  solucao\_optima = solucao  **return** solucao |
| **Figura 1. Algoritmo para a Sondagem Iterativa** | |

|  |  |
| --- | --- |
| **1.**  **2.**  **3.**  **4.**  **5.**  **6.**  **7.** | **ISPProb**(node):  **if** **isGoal**(node) **return** node  **if** **fail**(node) **return** nil  **while t\_util:**  S = **generate-successors**(node)  index = **generate-random-number**(0, S\_size - 1)  **ISPProb**(S[index]) |
| **Figura 2. Função auxiliar para o algoritmo de Sondagem Iterativa** | |

Foi implementado a versãode procura em árvore. Se o problema for pequeno suficiente, é possível explorar todo o espaço de estados, encontrando a solução óptima.

Na figura 1, o parâmetro *node* é o nó inicial do problema. A função ISP chama a função ISPNode para explorar caminhos a partir do nó raíz até ao nó objectivo. Num tempo definido, procura o máximo de caminhos que conseguir, comparando os seus custos e guardando aquele com custo menor. Ao encontrar um nó objetivo, retrocede para o nó raiz.

Na figura 2, o parâmetro *node* é o estado que está a ser de momento explorado. Na linha 1, se o estado atual é objetivo este é retornado. Na linha 2 se não se consegue gerar sucessores, retorna-se NIL. A função ***generate-successors*** expande o nó, gerando os seus sucessores. A função ***generate-random-number*** devolve um índice aleatório para aceder a S, lista de sucessores. Desta forma, escolhe-se aleatoriamente um sucessor para ser o novo estado a explorar.

Foram experimentados vários valores para o número de sondas usadas no algoritmo, de forma a encontrar o número mais eficiente para encontrar a solução ótima. Em baixo, os gráficos demonstram o comportamento obtido para diferentes valores, para um único teste:

O número de sondas não informa significativamente o desempenho do algoritmo. Há uma diminuição ligeira inicial para *n*=15, pelo que esse foi o valor selecionado.

## *Iterative Limited Depth Search* (ILDS)

|  |  |
| --- | --- |
| **1.**  **2.**  **3.**  **4.**  **5.**  **6.**  **7.**  **8.**  **9.** | **LDS**(node, n):  solucao\_optima = NULL  **for k = 0 to n:**  solucao = **ILDSProb(node, k, rDepth)**  **if** solucao\_optima == NULL:  **solucao\_optima = solucao**  **else if custo(solucao\_optima) > custo(solucao):**  solucao\_optima = solucao  **return** solucao\_optima |
| **Figura 3. Algoritmo para a Improved-Limited Discrepancy Search** | |
| |  | | --- | | 1 | | 2 | | 3 | | 4 | | 5 | | 6 | | 7 | | 8 | | 9 | | **ILDSProb**(node, k, rDepth):  **if** **isGoal**(node) **return** node  **if** **fail**(node) **return** nil  **S = generate-successors(node)**  best\_suc = choose-best-successor(S)  **worst\_suc = disjunction(best\_suc)**  **if rDepth > k:**  **ILDSProb(best\_suc, k, rDepth – 1)**  **if rDepth <= k:**  ILDSProb (first(worst\_suc) ,k-1,rDepth-1) | | |
| **Figura 4. Algoritmo para a Improved-Limited Discrepancy Search** | | | |

A iteração para k discrepâncias gera todos os caminhos com k ou menos ramos (linha 2, figura 3). Para cada iteração, compara-se qual o caminho encontrado com menor custo.

O parâmetro n do algoritmo ILDS, é o número de discrepâncias que se pode fazer. A função LDS chama a função ILDSProbe, aumentando o número de discrepâncias k até uma solução ser encontrada ou o máximo de discrepâncias ser feita. O máximo de discrepâncias corresponde à profundidade máxima do problema.

O parâmetro *rDepth* corresponde à profundidade ainda por explorar abaixo do nó atual, onde as discrepâncias que ainda podem ser tomadas.

Se a profundidade ainda por explorar for maior que k, segue-se a heurística. Atualiza-se a profundidade ainda por explorar para menos 1 unidade. Adia-se, portanto, a decisão de tomar a discrepância. Se k for maior que 0, não se segue a heurística. Diminui-se o k e a profundidade ainda por explorar em 1 unidade. Estas verificações impedem que o mesmo nó seja re-expandido entre iterações.

Para transformar o problema numa árvore binária, escolhe-se o primeiro sucessor da lista de sucessores que não foram escolhidos pela heurística.

!!!!!!!!!!!!!!!!!!!!!Heurística 12, problema 1 (DIZER ISTO PARA TODAS AS PROCURAS COM GRAFICOS (SONDAGEM ITERATIVA NÃO USA HEURISTICA))

## Procura alternativa – *Depth-Bounded Discrepancy Search* (DDS)

|  |  |
| --- | --- |
| **1.**  **2.**  **3.**  **4.**  **5.**  **6.** | **DDS**(node, n):  solucao\_optima = NULL  **for k = 0 to n:**  solucao = D**DSProb(node, k,0)**  **if** solucao\_optima == NULL:  **solucao\_optima = solucao**  **else if custo(solucao\_optima) > custo(solucao):**  solucao\_optima = solucao  **return** solucao\_optima |
| **Figura 4. Algoritmo DDS** | |
|  | |  |  | | --- | --- | | 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15. | **DDSProb**(node, k, prof):  **if** **isGoal**(node) **return** node  **if** **fail**(node) **return** nil  **S = generate-successors(node)**  best\_suc = choose-best-successor(S)  **worst\_S = disjunction(best\_suc)**  **if k = 0:**  **DDSProb(best\_suc, k)**  **if k = 1:**  DDSProb (worst\_suc,k-1)  **if k > 1:**  **result = DDSProb(best\_suc, k-1, prof+1)**  **if isGoal(result) return result**  **else**  **DDSProb(worst\_S, k-1, prof + 1)** | | **Figura 5. Função auxiliar para o algoritmo DDS** | | |

O algoritmo DDS, começa por fazer discrepâncias perto da raiz. Os números de discrepâncias vão aumentando à medida que me afasto da raiz. A heurística tem maior probabilidade de falhar no topo. Em semelhança ao ILDS, compara-se os custos das soluções encontradas entre iterações de forma a encontrar a solução com menor custo. A profundidade actual vai sendo calculada ao longo do algoritmo.

Iterativamente vai aumentando o limite da profundidade. Na iteração 0, vai a favor da heurística. Na iteração i explora os ramos onde as discrepâncias ocorrem na profundidade i ou menos. Tem-se o cuidado de não revisitar nós folha visitados em iteração anteriores. Isto é feito fazendo: na profundidade i da iteração i, vai-se contra a heurística já que ramos que vão a favor da heurística iria nos levar a nós folha visitados anteriormente em iterações anteriores. Em profundidades menores, podemos tomar tanto os ramos a favor como contra heurística. Em profundidades maiores, devemos sempre seguir a heurística. Isto é garantindo pelas verificações feitas da linha 7 à 15 da figura 5.

## Melhor abordagem

# Comparação de resultados

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *Problema\*  *Procura* | **Largura** | **Profundidade** | **Profundidade Iterativa** | **A\* (H1)** | **A\* (H2)** | **IDA\*** | **Sondagem Iterativa** | **ILDS** | **DDS** |
| **1** |  |  |  |  |  |  |  |  |  |
| **2** |  |  |  |  |  |  |  |  |  |
| **3** |  |  |  |  |  |  |  |  |  |
| **4** |  |  |  |  |  |  |  |  |  |
| **5** |  |  |  |  |  |  |  |  |  |

Tabela 5 – Número de turnos na solução

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *Problema\*  *Procura* | **Largura** | **Profundidade** | **Profundidade Iterativa** | **A\* (H1)** | **A\* (H2)** | **IDA\*** | **Sondagem Iterativa** | **ILDS** | **DDS** |
| **1** |  |  |  |  |  |  |  |  |  |
| **2** |  |  |  |  |  |  |  |  |  |
| **3** |  |  |  |  |  |  |  |  |  |
| **4** |  |  |  |  |  |  |  |  |  |
| **5** |  |  |  |  |  |  |  |  |  |

Tabela 6 – Tempo (s) de execução

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *Problema\*  *Procura* | **Largura** | **Profundidade** | **Profundidade Iterativa** | **A\* (H1)** | **A\* (H2)** | **IDA\*** | **Sondagem Iterativa** | **ILDS** | **DDS** |
| **1** |  |  |  |  |  |  |  |  |  |
| **2** |  |  |  |  |  |  |  |  |  |
| **3** |  |  |  |  |  |  |  |  |  |
| **4** |  |  |  |  |  |  |  |  |  |
| **5** |  |  |  |  |  |  |  |  |  |

Tabela 7 – Nós gerados na execução

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *Problema\*  *Procura* | **Largura** | **Profundidade** | **Profundidade Iterativa** | **A\* (H1)** | **A\* (H2)** | **IDA\*** | **Sondagem Iterativa** | **ILDS** | **DDS** |
| **1** |  |  |  |  |  |  |  |  |  |
| **2** |  |  |  |  |  |  |  |  |  |
| **3** |  |  |  |  |  |  |  |  |  |
| **4** |  |  |  |  |  |  |  |  |  |
| **5** |  |  |  |  |  |  |  |  |  |

Tabela 8 – Nós expandidos na execução

# Melhorias a implementar

# Conclusão

# Bibliografia

1. Angelo Oddi et al., Iterative-Sampling Search for Job Shop Scheduling with Setup Times, 2009;
2. Richard E. Korf, Improved Limited Discrepancy Search, 1995;
3. Toby Walsh, Depth-Bounded Discrepancy Search, 1996;
4. Patrick Prosser, Chris Unsworth, Limited Discrepancy Search Revisited, 2011.