

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

[Introdução 1](#_Toc531801351)

[Modelação do *Constraint Satisfaction Problem* 1](#_Toc531801352)

[Estrutura do CSP 1](#_Toc531801353)

[Geração de Sucessores 1](#_Toc531801354)

[Verificação das restrições 2](#_Toc531801355)

[Heurísticas 2](#_Toc531801356)

[Procuras 2](#_Toc531801357)

[Sondagem iterativa 2](#_Toc531801358)

[ILDS 2](#_Toc531801359)

[Procura alternativa – DDS 2](#_Toc531801360)

[Melhor abordagem 3](#_Toc531801361)

[Comparação de resultados 3](#_Toc531801362)

[Melhorias a implementar 3](#_Toc531801363)

[Conclusão 3](#_Toc531801364)

[Bibliografia 3](#_Toc531801365)

# 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.

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. 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. Todos estes estados-filhos são devolvidos apenas após a validação das restrições, e todos eles respeitam todas as restrições.

## Verificação das restrições

A verificação de restrições é feita durante a fase de geração de sucessores, de forma a imediatamente descartar a geração de estados inválidos. Esta validação é efetuada para cada turno do estado proposto, e é durante esta fase que se calculam e definem 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.

**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

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

# Procuras implementadas

## Sondagem iterativa

|  |  |
| --- | --- |
| **1.**  **2.**  **3.**  **4.**  **5.**  **6.** | **ISP**(node):  **if** **isGoal**(node) **return** node  **if** **fail**(node) **return** nil  S = **generate-successors**(node)  index = **generate-random-number**(0, S\_size - 1)  **ISP**(S[index]) |
| **Figura 1. Algoritmo para a Sondagem Iterativa** | |

Foi implementado a versão *1-samp* que não possui retrocesso. Escolheu-se esta abordagem porque os sucessores gerados são sempre válidos.

Na figura 1, o parâmetro *node* é o estado que está a ser de momento explorado. Na linha 1, se o estado actual 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.

## ILDS

|  |  |
| --- | --- |
| **1.**  **2.**  **3.**  **4.**  **5.** | **LDS**(node, n):  **for k = 0 to n:**  **result = ILDProb(node, k, rDepth)**  **if result != nil:**  **return result** |
| **Figura 2. 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 3. Algoritmo para a Improved-Limited Discrepancy Search** | |

A iteração para k discrepâncias gera todos os caminhos com k ou menos ramos.

Para cada nó folha na profundidade máxima é gerada exactamente uma vez. Nós folha abaixo da profundidade máxima, contudo, assim como nós interiores, são gerados mais do que 1 vez.

O parâmetro n é o número de discrepâncias que se pode fazer. Corresponde ao número de variáveis. 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 ao número de tarefas do problema inicial.

O parâmetro *rDepth* corresponde à profundidade ainda por explorar abaixo do nó actual, 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 r-Depth-1. Adia-se portanto a decisão de tomar a discrepância.

Se k for maior que 0, não se segue a heurística. Atualiza-se k para k-1. E a profundidade ainda por explorar para rDepth-1. Para transformar o problema numa árvore binária, escolhe-se o primeiro sucessor da lista de sucessores que não foram escolhidos pela heurística.

## Procura alternativa – DDS

|  |  |
| --- | --- |
| **1.**  **2.**  **3.**  **4.**  **5.**  **6.** | **DDS**(node, n):  **for k = 0 to n:**  **result = ILDProb(node, k, rDepth)**  **if result != nil:**  **return result** |
| **Figura 4. Algoritmo DDS** | |
|  | |  |  | | --- | --- | | 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15. | **DDSProb**(node, k):  **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)**  **if isGoal(result) return result**  **else**  **DDSProb(worst\_S, k-1)** | | **Figura 5. Função auxiliar para o algoritmo DDS** | | |

Se k for igual a 0 segue-se a heurística. Se k igual a 1 não se segue a heurística. Se k for maior que 1 segue-se a heurística.  Atualiza-se o k para k-1. Se para este ramo, não se encontrar uma solução não se segue a heurística.

Ao contrário do ILDS, a profundidade actual é calculada durante a procura.

Começa por fazer discrepâncias perto da raíz. O número de discrepâncias vão aumentando à medida que me afasto da raiz. A heurística tem maior probabilidade de falhar no topo.

Iterativamente vai aumentando o limite da profundidade. Na iteração 0, vai a favor da heurística. Na iteração i explora os ramos onde as disciprancias ocorrem na profundidade i ou menos. Tem-se o cuidado de não revisitar nõs folha vistados em iteração anteriores. Isto é feito fazendo: na profudndiade 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. A procura termina quando o limite é maior que a profundidade da folha mais profunda.

## Melhor abordagem

# Comparação de resultados

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

# 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.