Classificação:

Nº Total de Horas Gastas: ≈ 100 h

Procura e Planeamento

Relatório – Job Shop Schedulling Problem

Grupo 001 – Alameda

António Machado – 68122
Diogo Costa – 72770
Rodrigo Monteiro – 73701

ÍNDICE

Índice 2
Índice de Imagens 2
1 Introdução 3
2 Abordagem Incremental ao Problema 4
2.1 O Problema 4
2.2 A Solução 6
3 Estruturas de Dados 7
4 Modelo do Problema e Funções Principais 9
5 Heurísticas e Estratégia de Cortes 11
6 Estratégias de Procura 13
7 Análise dos Resultados 14
7.1 Definição das Amostras 14
7.2 Avaliação da *Performance* das Estratégias de Procura 14
7.2.1 Procuras Cegas 15
7.2.2 Procuras Informadas 16
7.2.3 Procuras Não-Sistemáticas 17
8 Conclusão 19
Anexo I – Tabelas de Execução das Procuras Cegas e Informadas 20
Anexo II – Tabelas de Execução das Procuras Não-Sistemáticas 21

ÍNDICE DE IMAGENS

FIGURA 1 - ESQUEMA DA RESOLUÇÃO DE UM *JSSP* 6
FIGURA 2 - CORTE DE ESTADOS GERADOS 2ª VEZ 11

GRÁFICO 1 - *PERFORMANCE* DAS PROCURAS CEGAS 15
GRÁFICO 2 - *PERFORMANCE* DAS PROCURAS INFORMADAS 16
GRÁFICO 3 - PERFORMANCE DAS PROCURAS NÃO-SISTEMÁTICAS 17

TABELA 1 - EXECUÇÃO DAS PROCURAS CEGAS SOBRE *JSSP* 20
TABELA 2 - EXECUÇÃO DAS PROCURAS INFORMADAS SOBRE *JSSP* 20
TABELA 3 - EXECUÇÃO DAS PROCURAS NÃO-SISTEMÁTICAS SOBRE *JSSP* 21

1 INTRODUÇÃO

Um problema de calendarização da produção industrial (Job Shop Schedulling Problem, referido como *JSSP* no decurso deste relatório) é constituído por uma **lista de encomendas**. As encomendas, por sua vez, são constituídas por uma sequência de **tarefas ordenadas** em que cada uma é realizada única e exclusivamente por **uma máquina**. A solução consiste num estado cujas tarefas têm atribuído um **tempo de início**, tendo em conta as restrições do problema, com o objetivo de **minimizar o tempo despendido** para completar todas as encomendas da lista.

Existem dois tipos de restrições neste problema, e são elas:

a. **Restrições de capacidade** – cada máquina tem um limite de *uma tarefa* num dado *intervalo de tempo*, ou seja, nenhuma máquina pode estar a realizar mais do que uma tarefa nesse intervalo.

b. **Restrições de precedência** – as tarefas no início da lista precedem as seguintes e, como tal, não é possível começar a realizar uma tarefa de índice N se a tarefa que a precede (N-1) ainda não tiver sido concluída.

Encontrar uma solução válida, que respeite as restrições anteriores, não é muito complicado, no entanto, otimizar o programa de forma a encontrar a solução mais eficiente possível depende não só da implementação mas também do algoritmo de procura utilizado para a encontrar. Nos capítulos que se seguem descrevemos a nossa visão da resolução do problema e efetuamos uma análise à eficiência das várias procuras aplicadas sobre o mesmo.

Os testes de eficiência que fizemos foram realizados no *LispWorks 6.1 Personal Edition* a correr numa plataforma Windows 8.1 numa máquina com Intel Core i7 4510U (Dual-Core) @ 2GHz, com 12GB de memória RAM.

2 ABORDAGEM INCREMENTAL AO PROBLEMA

2.1 O Problema

Tendo por base as restrições impostas pelo *JSSP*, a nossa abordagem consiste na subdivisão do problema introduzido em dois *sub-problemas*: ***a)*** *problema de precedências* e ***b)*** *problema de capacidades*. No primeiro, a cada tarefa $\boldsymbol{T_n}$ de uma encomenda ***E***, é atribuído um tempo de início de forma a que só se inicie a sua execução quando $\boldsymbol{T_{n-1}}$ tiver terminado:

$$Start.Time(\boldsymbol{T_n}) \geq Start.Time(\boldsymbol{T_{n-1}}) + Duration(\boldsymbol{T_{n-1}})$$

Restrição 1 - Restrição de Precedência

No segundo, a cada tarefa $\boldsymbol{T_n}$ sob encargo de uma máquina $\boldsymbol{M_i}$, é atribuído um tempo de início de forma a que só se inicie a sua execução quando $\boldsymbol{T_\beta}$ tiver terminado **ou** $\boldsymbol{T_\beta}$ ainda não tiver começado. Por outras palavras, a execução de $\boldsymbol{T_\alpha}$ **não pode intercetar** a execução de $\boldsymbol{T_\beta}$:

$$Start.Time(\boldsymbol{T_\alpha}) \geq Start.Time(\boldsymbol{T_\beta}) + Duration(\boldsymbol{T_\beta})$$
$$\vee$$
$$Start.Time(\boldsymbol{T_\alpha}) + Duration(\boldsymbol{T_\alpha}) \leq Start.Time(\boldsymbol{T_\beta})$$

Restrição 2 - Restrição de Capacidade

A entidade que concerne à resolução de um problema de precedência é a ***Encomenda***. A resolução deste problema passa por pegar em cada tarefa de cada encomenda E_x e validar a restrição de precedência com a tarefa anterior pertencente **à mesma encomenda**.

Por outro lado, a entidade ***Máquina*** é a esfera de preocupação para a resolução de um problema de capacidade. A resolução deste problema passa por pegar em cada tarefa associada à máquina $\boldsymbol{M_i}$ e verificar que nenhuma se interceta com as restantes tarefas **da mesma máquina**. Esta abordagem tem a vantagem de limitar o número de estados gerados a partir de um estado-pai ao número de máquinas de um problema. Assim, se um problema possui ***m*** máquinas, então a expansão de um estado gerará ***m*** estados novos. Nas secções seguintes deste relatório mostraremos como é que este número ainda consegue ser reduzido drasticamente.

A maior dificuldade que se coloca logo à partida é o facto de o *JSSP* ser, na verdade, a soma de dois problemas em que, numa abordagem incremental, cada um se **resolve em deterioração do outro**. Ou seja, por forma a resolver um problema de capacidade entre tarefas é possível que, simultaneamente, se esteja a criar um novo problema de precedência, e vice-versa. Serve de exemplo o seguinte cenário:

E_a	E_a
Task0, Machine0, ***Duration 5, Start.time 0***	*Task0, Machine0, Duration 5, Start.time 0*
Task1, Machine1, Duration 10, Start.time 5	*Task1, Machine1, Duration 10, Start.time 5*
Task2, Machine2, Duration 10, Start.time 15	*Task2, Machine2, Duration 10, Start.time 15*
Resolve Capacidade →	
E_b	E_b
Task0, Machine0, ***Duration 15, Start.time 0***	*Task0, Machine0,* ***Duration 15, Start.time 5***
Task1, Machine1, Duration 10, Start.time 15	*Task1, Machine1,* ***Duration 10, Start.time 15***
Task2, Machine2, Duration 5, Start.time 25	*Task2, Machine2, Duration 5, Start.time 25*

Cenário 1 - Resolução da Restrição de Capacidade

O *Cenário 1* começa com um estado que **viola a restrição de capacidade** entre as tarefas *Task0* das encomendas E_a *e* E_b. Após resolução deste problema, verifica-se que o estado obtido **passa a violar a restrição de precedência**. Deste modo, gera-se temporariamente um ciclo de *proximidade – afastamento* à solução, até que se atinge um **equilíbrio** em que a resolução de um problema de capacidade deixa de afetar a precedência, e/ou vice-versa.

E_a	E_a
Task0, Machine0, Duration 5, Start.time 0	*Task0, Machine0, Duration 5, Start.time 0*
Task1, Machine1, Duration 10, Start.time 5	*Task1, Machine1, Duration 10, Start.time 5*
Task2, Machine2, Duration 10, Start.time 15	*Task2, Machine2, Duration 10, Start.time 15*
Resolve Precedência →	
E_b	E_b
Task0, Machine0, ***Duration 15, Start.time 5***	*Task0, Machine0, Duration 15, Start.time 5*
Task1, Machine1, ***Duration 10, Start.time 15***	*Task1, Machine1, Duration 10, Start.time 20*
Task2, Machine2, Duration 5, Start.time 25	*Task2, Machine2, Duration 5, Start.time 30*

Cenário 2 - Resolução da Restrição de Precedência

Como se pode ver através do *Cenário 2*, depois de resolvido o problema de precedência não se voltou a gerar um problema de capacidade.

2.2 A Solução

Com base nesta análise, e tendo em conta que por definição um *JSSP* começa com todas as tarefas sem tempo de início atribuído, para poupar o trabalho da geração de estados inúteis, este é ***inicializado*** como sendo *estritamente* um ***problema de capacidade***, tal como no início do *Cenário 1*. Ou seja, à partida apenas garantimos que não existem violações da restrição de precedência.

Adicionalmente, com o fim de *acelerar* a chegada ao ***ponto de equilíbrio*** entre a resolução de restrições referido anteriormente, nunca são gerados estados-sucessores com problemas de precedência. Ou seja, um estado com um problema de precedência é considerado apenas um *potencial* estado-sucessor. Chamemos a este tipo de estados ***meta-estado / meta-sucessor.*** É fácil perceber que, gerar um *meta-sucessor* é pôr mais palha no palheiro, enquanto se os estados-sucessores apenas se traduzirem num problema de capacidade, temos sempre a certeza de estar mais rapidamente a desbravar caminho até à agulha. Deste modo, antes de ser inserido na lista de sucessores do estado-pai, o problema de precedência de um *meta-estado* é imediatamente resolvido, dando origem a um **novo problema de capacidade** ou a **uma solução**. Resumindo, em qualquer momento do percurso da procura por uma solução válida ao *JSSP*, **um estado ou é um problema de capacidade ou é um estado objetivo**[1].

O esquema abaixo resume a lógica e a mecânica por detrás do método de resolução implementado:

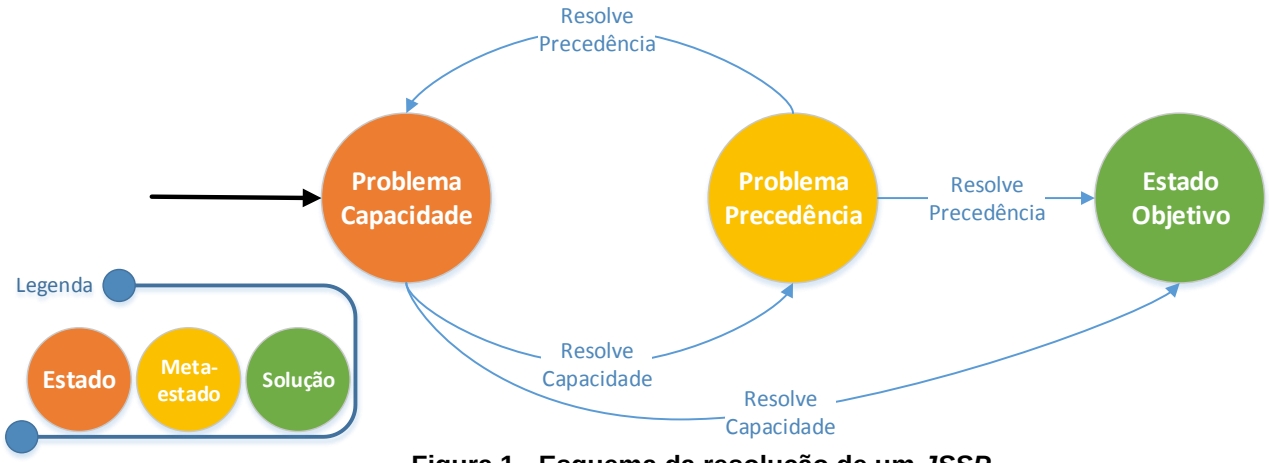

Figura 1 - Esquema da resolução de um *JSSP*

[1] Não há nada que impeça a implementação contrária, ou seja, iniciar o *JSSP* sem problemas de capacidade, e tratá-lo exclusivamente como um problema de precedências.

3 ESTRUTURAS DE DADOS

Algumas estruturas de dados já estavam implementadas quando nos foi facultado o enunciado do problema e código de apoio, nomeadamente:

a. Estrutura **Problema (Problem)** – constituída por uma lista de encomendas, um nome e dois inteiros que representam o número de encomendas e o número de máquinas desse problema.

```
(defstruct job-shop-problem
        name
        n.jobs
        n.machines
        jobs)
```

b. Estrutura **Encomenda (Job)** – constituída por uma lista de tarefas e um inteiro que representa o número de tarefas dessa encomenda.

```
(defstruct job-shop-job
        job.nr
        tasks)
```

c. Estrutura **Tarefa (Task)** – constituída por cinco inteiros que representam, respetivamente, o número da encomenda, o número da tarefa, o número da máquina em que vai ser executada, a duração e tempo de início da mesma.

```
(defstruct job-shop-task
        job.nr
        task.nr
        machine.nr
        duration
        start.time)
```

Para além destas foi necessária a criação de uma nova estrutura – ***state*** – que representasse o estado do problema sem o alterar diretamente, e permitisse a adição de outras variáveis que nos facilitassem a sua resolução.

```
(defstruct state
                jobs-lst
                n.jobs
                n.machines
                discrepancy
                heuristic-eval
                precedence-solved)
```

A estrutura ***state*** é constituída por um(a):

- ***Lista de encomendas*** do problema – sobre a qual vão ser efetuadas operações por forma a alcançar um estado objetivo;
- ***Número de encomendas*** – com o fim de evitar a recorrente contagem de encomendas da lista de encomendas;
- ***Número de máquinas*** – com o fim de evitar a recorrente contagem do número de máquinas da lista de encomendas;
- ***Número de discrepâncias*** – com o fim de complementar a estratégia de procura *ILDS*, discutida na *secção 6 – Estratégias de Procura;*
- ***Avaliação da heurística*** – é inicializada com o valor da avaliação de heurística do estado-pai e é posteriormente atualizada com o valor a atribuir ao estado corrente;
- ***Número de precedências resolvidas*** – guarda o número de precedências entre tarefas que, após geração de um estado que não viola a restrição de capacidade, tiveram de ser corrigidas.

As duas últimas variáveis de controlo permitir-nos-ão construir heurísticas baseadas na proximidade de um estado à solução.

4 MODELO DO PROBLEMA E FUNÇÕES PRINCIPAIS

Tendo definido a estrutura de dados que representa o problema na secção anterior – ***state*** – o próximo passo será munirmo-nos de um conjunto de operações que nos permita manipular essa representação de forma a chegar a um estado que é solução do problema.
Para este efeito foram definidas algumas funções de importância capital para a resolução do problema, são elas:

a) Função que verifica se um estado é objectivo - *is-goal? (state)*

Uma das coisas mais importantes a definir é o conceito de estado objetivo, já que sem ele nunca se chega a uma solução. Um estado objetivo é simplesmente aquele que respeita as restrições do problema (o que não implica ser ótimo), nomeadamente, as restrições de precedência e de capacidade. Devido ao nosso desenho de implementação descrito nas secções anteriores, a única restrição que efetivamente teremos de validar é a restrição de capacidade. Isto porque o problema é sempre tratado de forma a que *a expansão de um estado nunca finde num problema de precedência*. Assim, esta função apenas valida se nenhuma máquina tem agendada a execução de duas tarefas em simultâneo, e em caso afirmativo, então estamos perante um estado objetivo.

b) Função que compara dois estados, verificando se são iguais - *eq-states? (state1 state2)*

Esta função recebe dois estados e devolve ***T*** caso sejam iguais e ***nil*** caso contrário. Dois estados são iguais se todas as encomendas e tarefas que as constituem são iguais. Por forma a refatorizar e tornar o código mais legível, esta função usa a função ***eq-jobs? (job1 job2)*** para comparar encomendas, que por sua vez usa a função ***eq-tasks? (task1 task2)*** para comparar os atributos de cada tarefa de cada encomenda, nomeadamente o *job.nr, task.nr, machine.nr, duration* e *start.time.*

c) Função que converte um problema num estado inicial – *make-state-from-problem (problem)*

Esta função recebe como *input* um problema e instancia uma estrutura – ***state*** – acedendo ao número de encomendas e de máquinas desse problema e à sua lista de encomendas. O *número de conflitos* é inicializado a ***nil***, uma vez que só é calculado no momento em que a heurística atribui um valor ao estado; e o número de *precedências resolvidas* é inicializado a zero.

d) Função auxiliar que "prepara" um problema para ser convertido em estado – *setup-init-state (state)*

Esta função recebe um problema e, substituindo o *start-time* de todas as tarefas de modo a que se verifique a restrição de precedência, devolve o estado resultante dessas substituições.

e) Função que gera sucessores (expande o estado atual) – *expand (state)*

Esta função recebe um estado e expande-o, retornando uma lista com todos os seus sucessores. Para isso faz uso de duas funções auxiliares. A primeira, ***gen-capacity-complient-state (state machine)***, gera um estado ***new-state*** cuja máquina ***machine*** respeita a restrição de capacidade. Posteriormente, a função ***maintain-precedence-restriction (state)*** trata de resolver imediatamente qualquer problema de precedência levantado pela resolução do problema de capacidade por ***gen-capacity-complient-state***, e incrementa a variável ***precedence-solved*** do estado por cada precedência entre tarefas uniformizada.

Adicionalmente, de modo a reduzir a lista de sucessores gerados pela expansão de um estado, são feitas duas verificações:

1. Se um estado sucessor ***new-state*** for igual ao estado-pai, então ***new-state*** não é acrescentado à lista de sucessores;
2. Se um estado sucessor ***new-state*** já se encontrar na lista de sucessores[2], então ***new-state*** não é acrescentado à lista de sucessores;

Ainda, foi definida uma variável global ****ALL-GEN-STATES**** responsável por guardar numa lista todos os estados gerados pela função ***expand***, pelo que, no fim da expansão de um estado, a sua lista de sucessores é adicionada à lista de ****ALL-GEN-STATES****. Esta variável tem como fim a posterior prática de cortes a aplicar a determinados ramos durante a procura de uma solução.

[2] A função ***member? (state state-list)*** verifica se um dado estado pertence a uma determinada lista de estados.

5 HEURÍSTICAS E ESTRATÉGIA DE CORTES

A cada heurística foram introduzidas duas situações de corte que permitem descartar a tentativa de expansão de estados inúteis. Estes cortes baseiam-se nas seguintes premissas:

- Se, na lista de todos os estados gerados ***ALL-GEN-STATES***, existe mais do que um estado igual ao estado que a heurística se encontra a avaliar, então o valor deste estado passa a ser ∞.

- Se o valor atribuído ao estado a ser avaliado pela heurística for maior do que a avaliação do estado-pai, então o valor desse estado passa a ser ∞.

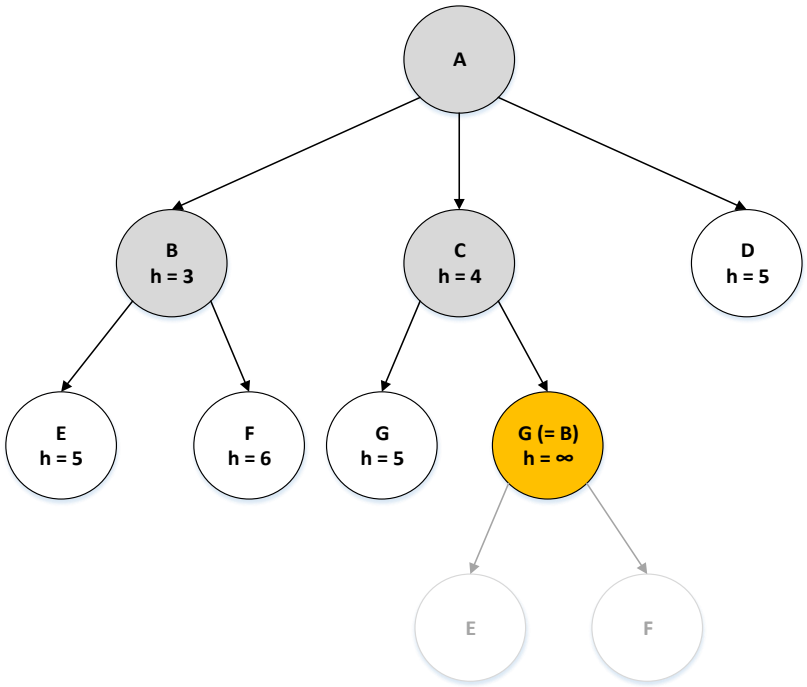

Figura 2 - Corte de estados gerados 2ª vez

Tendo em conta a *Figura 2*, a primeira decisão justifica-se com base no facto de que se se expandisse o estado G iriam resultar os mesmos sucessores de B, o que inutilmente obrigaria à resolução de um problema de capacidade já resolvido em B e aumentaria a lista de todos os estados gerados. Deste modo evitam-se perdas de tempo a resolver problemas replicados e uma explosão exponencial de expansões desnecessárias.

Por sua vez, a segunda premissa justifica-se com o facto de que se um estado-pai gera um estado com pior avaliação então estamo-nos a afastar do estado objetivo, pelo que, à partida, não é de grande interesse que este estado seja candidato a ser expandido.

As duas heurísticas desenvolvidas foram baseadas em conceitos completamente distintos. São elas:

A. Num-conflicts-to-goal *(heuristic-conflicts)*

Esta função recebe um estado e devolve um inteiro que corresponde ao número tarefas que violam a restrição de capacidade, ou seja, o número de conflitos entre os intervalos de execução de cada tarefa de cada máquina. Esta é uma heurística admissível uma vez que quanto maior o número de conflitos mais longe nos encontramos da solução, e vice-versa. Para tornar este valor ainda mais preciso, somamos ao número de conflitos o número de precedências que tiveram de ser resolvidas após geração do estado recorrendo à variável do estado *precedence-solved*. Isto porque, por senso-comum, quantos mais problemas de precedência a resolução de um problema de capacidade gerar, significa que essa expansão não foi provavelmente a mais inteligente.

B. Longest-end-time *(heuristic-max)*

Esta função recebe um estado e devolve um inteiro que corresponde ao tempo de fim mais alto de entre todas as tarefas de um *JSSP*, ou seja, o tempo que demora a executar todas as tarefas de um problema. Através da aplicação desta heurística, claramente um problema que demore menos tempo a acabar será escolhido para expansão em prol de um que demore mais, pelo que esta heurística tende a preferir estados ótimos a estados válidos.

Pela descrição de ambas as heurísticas, facilmente se consegue antever que a heurística **A** se irá focar em encontrar uma solução válida, enquanto a heurística **B** se irá preocupar em encontrar uma solução ótima. Deste modo, e como demonstrado em detalhe na *secção 7 – Análise de Resultados*, a heurística **A** encontrará uma solução bastante mais rápido do que a **B**, embora a **B** certamente irá encontrar melhores soluções que **A.**

6 ESTRATÉGIAS DE PROCURA

Para este problema, desenvolveram-se quatro estratégias de procura não-sistemáticas de forma a servirem de termo de comparação com as procuras sistemáticas de que dispúnhamos à partida. São elas:

i. ***Sondagem iterativa*** – esta estratégia lança uma *sonda* que percorre um caminho aleatório da árvore de um problema e termina quando encontra uma solução (não necessariamente ótima) ou atinge o tempo limite definido para a procura.

ii. ***Discrepância melhorada*** (ILDS) – sempre que, na expansão de um estado, é gerada a lista de novos sucessores, esta é ordenada de acordo com uma heurística, de modo a termos no início da lista os estados com melhor *avaliação*. De seguida atribui um valor de discrepância a cada estado da lista de sucessores, sendo que um sucessor no início da lista possui 0 discrepâncias e o que se encontrar no fim possui ***r*** discrepâncias (tantas quanto o número de ***r***amos de um determinado nível).
Inicialmente percorremos o caminho de estados que possuem 0 discrepâncias. Caso não se encontre uma solução, em cada iteração é incrementado o valor máximo de discrepâncias possíveis num dado estado permitindo a exploração de caminhos onde é mais difícil obter uma perceção de proximidade, ou não, a uma solução.
Simbolicamente, este valor representa o número de *erros* que o algoritmo permite em cada iteração, de forma a poder explorar todos os caminhos possíveis até encontrar uma solução válida.
Assim, o número de discrepâncias aceites não pode ser maior que o máximo aceite para a respetiva iteração. Este algoritmo tem a vantagem de que apenas explora sucessores de um estado apenas uma vez, algo que não acontece na sua versão inicial – *LDS* – que em cada iteração explora sempre todos os nós explorados em iterações anteriores. Isto é possível guardando na primeira iteração a profundidade máxima que foi atingida e verificando nas iterações seguintes se já se chegou a essa profundidade. Em caso afirmativo, apenas são explorados os nós que não ultrapassam o valor de discrepância máximo para aquela iteração.

iii. ***Têmpora-simulada*** – esta estratégia torna possível explorar, temporariamente, estados que são piores que o estado atual, de modo a mais tarde poder melhorá-lo. Em vez de escolher o melhor sucessor, o algoritmo escolhe um aleatoriamente. Se for melhor que o estado atual, o sucessor é sempre aceite, caso contrário é aceite com uma probabilidade $p < 1$. O parâmetro temperatura é utilizado para determinar a probabilidade da escolha de um estado pior, e à medida que este se aproxima de 1, estados considerados piores tendem a não ser aceites.

iv. ***Discrepância limitada*** (LDS) – esta estratégia é idêntica à *ILDS* com a exceção de que, em cada iteração, continua a explorar estados que já foram explorados em iterações anteriores. Foi implementada com o objetivo de, na *secção 7 – Análise de Resultados* se poder avaliar qual a intensidade do impacto na resolução de um JSSP em comparação com a procura *ILDS*.

7 ANÁLISE DOS RESULTADOS

Para estudarmos toda a implementação descrita nas secções anteriores deste relatório, foi desenhado um ambiente de testes que se focou em:

1. **Definir um modelo de amostras de *runs*** para aplicar cada estratégia de procura sobre vários *JSSP*;
2. **Avaliar a *performance* das estratégias de procura;**
3. **Avaliar a *performance* das heurísticas;**

7.1 Definição das Amostras

Primeiramente, definimos que iriam existir duas amostras, **A** e **B**, a primeira sob a influência da heurística ***num-conflicts-to-goal*** e segunda sob influência da heurística ***longest-end-time***:

- Foram definidos 6 *JSSP* com **E** encomendas, **T** tarefas/encomenda e **M** máquinas – $JSSP_x < \mathrm{E}, \mathrm{T}, \mathrm{M} >$ – sobre os quais as procuras irão ser aplicadas:
  - $JSSP_1 < 4, 3, 3 >$
  - $JSSP_2 < 6, 6, 6 >$
  - $JSSP_3 < 10, 5, 5 >$
  - $JSSP_4 < 10, 10, 10 >$
  - $JSSP_5 < 20, 5, 5 >$
  - $JSSP_6 < 50, 10, 10 >$
- Para **cada uma** das *9 estratégias de procura* analisadas serão executadas **10 *runs*** sobre **cada** *JSSP*[3];

7.2 Avaliação da *Performance* das Estratégias de Procura

A **medida de *performance*** baseia-se na razão entre o número de estados gerados e o número de estados expandidos e o tempo que se demorou até encontrar uma solução válida:

$$\boldsymbol{Performance} = \frac{\#Estados\ Gerados / \#Estados\ Expandidos}{Tempo\ de\ Execução + 1}$$

Esta é uma medida bastante razoável, uma vez que não podemos concentrar-nos apenas se uma procura precisou de expandir poucos estados para encontrar uma solução em função do número de estados que foram gerados; o tempo que demorou a encontrá-la é também um fator extremamente importante. No caso em que o tempo de execução é menor que um obteríamos *performances* astronómicas, pelo que somamos 1 para manter a proporção.

[3] *JSSP's* em que a procura demora **mais de 5 minutos** a terminar, apenas foram efetuadas **3 *runs*,** uma vez que ficarão excluídas da amostra.

Ao analisarmos esta medida, facilmente se percebe que estamos a preferir procuras que expandem poucos estados para encontrar uma solução e no menor tempo possível:

$$\boldsymbol{Performance\ Ótima} = \frac{\lim {}^{\#Estados\ Gerados}/_{\#Estados\ Expandidos} \to \infty}{\lim Tempo\ de\ Execução \to 1} \to \infty$$

7.2.1 Procuras Cegas

Os resultados de *performance* obtidos para as procuras cegas são os expressos no *Gráfico 1* abaixo:

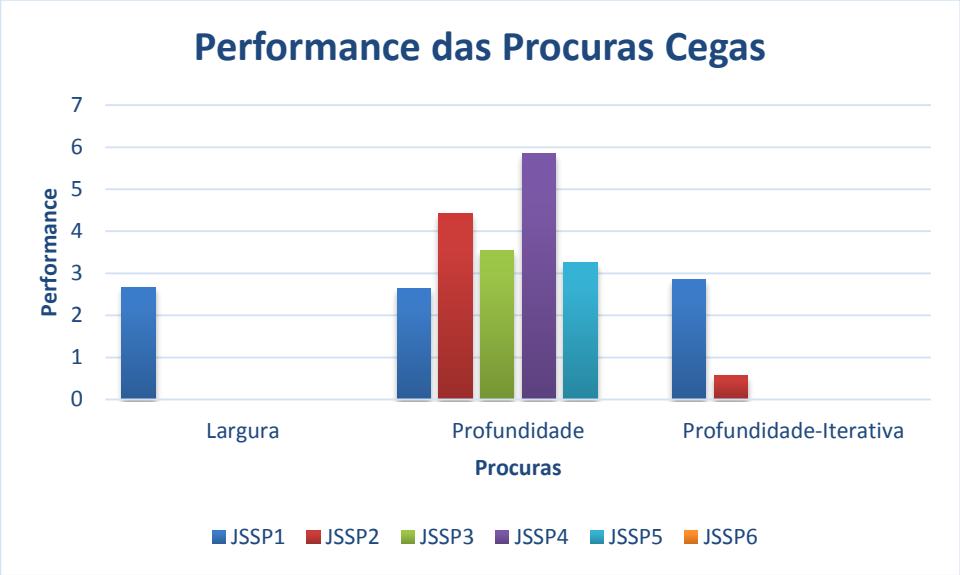

Gráfico 1 - *Performance* das Procuras Cegas

- **Largura-Primeiro**

Sabemos de antemão que esta estratégia de procura se foca numa expansão horizontal da árvore de estados de um problema, dando, por norma, origem a mais estados gerados que qualquer outra procura. No entanto permite logo de início diferenciar um estado em todos os seus possíveis sucessores (ao contrário da profundidade-primeiro que se foca num único sucessor e explora-o até não poder mais), dotando a procura de grande dinamismo mas tornando-a inviável em problemas cuja solução se forma em níveis mais fundos da árvore. Deste modo, apenas foi possível correr com sucesso esta procura no $JSSP_1$, sendo que no $JSSP_2$ demora mais de 5 minutos a concluir e nos problemas mais complexos torna-se inviável a espera por uma solução.

- **Profundidade-Primeiro**

Como se pode observar no *Gráfico 1* esta foi a procura com melhor desempenho, tendo conseguido arranjar solução para todos os *JSSP* exceto o mais complexo de todos. Isto porque, e após toda a explicação da implementação do problema segundo a perspetiva abordada, a solução de um *JSSP* raramente se encontra em níveis baixos da árvore do problema. A cíclica deterioração/resolução das restrições do problema obrigam a que o ponto de equilíbrio mencionado em secções anteriores só se atinja a uma determinada profundidade, pelo que qualquer que seja o ramo que se escolha

aprofundar, certamente se encontrará uma solução assim que esse equilíbrio é alcançado. Expandir todos os estados em largura, como na estratégia de *largura-primeiro*, em nada se está a contribuir para que esse equilíbrio aconteça, daí que esta seja a melhor estratégia de procuras cegas a aplicar a este problema.

- **Profundidade-Iterativa**

Sendo esta uma estratégia que procura combinar as duas anteriores, ainda conseguiu arranjar solução para os dois primeiros problemas – um a mais que a estratégia de *largura-primeiro*. No problema mais simples conseguiu inclusive superar a *performance* da estratégia de procura em profundidade, no entanto, por todos os motivos já explanados, excedeu o tempo e o espaço de expansão nos restantes. De relevar que, embora tenha um menor desempenho que a procura em profundidade, esta encontrou uma solução melhor, visto combinar o dinamismo já referido da *largura-primeiro*.

A *Tabela 1* do ***Anexo I*** deste relatório detalha os resultados obtidos de aplicar cada procura aos problemas estudados.

7.2.2 Procuras Informadas

Os resultados de *performance* obtidos para as procuras informadas são os expressos no *Gráfico 2* abaixo:

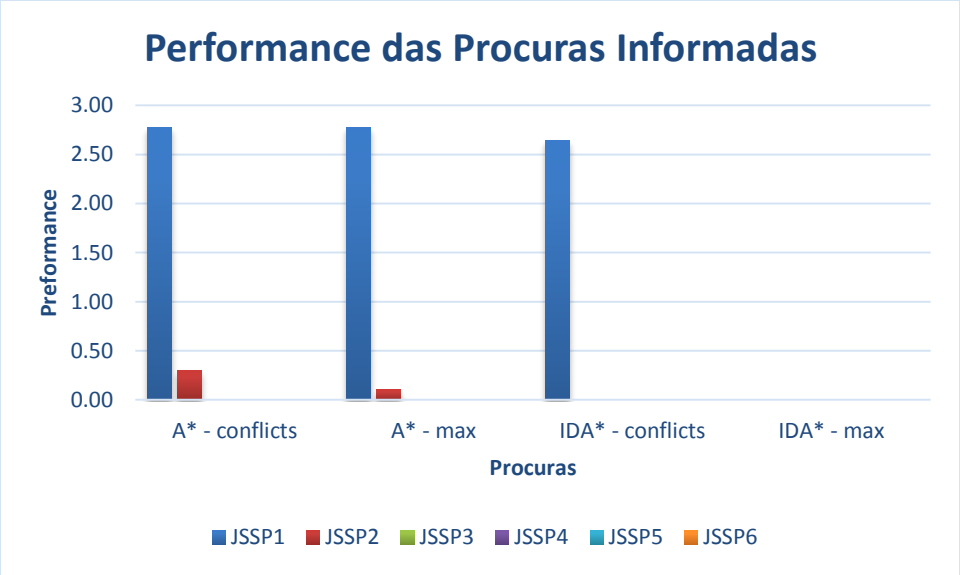

Gráfico 2 - *Performance* das Procuras Informadas

- **A* e IDA***

Embora pareça uma surpresa, é expectável que, nesta implementação, as procuras informadas **sistemáticas** não apresentem tão bons resultados quanto uma estratégia de profundidade. Pelo mesmo motivo de uma *largura-primeiro*, as procuras informadas não *insistem* na exploração de um estado até à chegada do **ponto de equilíbrio** entre restrições. Ao invés, estas escolhem sempre o estado que aparentemente se encontra mais próximo da solução, tal como ditado pela avaliação da heurística. O problema coloca-se quando um estado que parece estar próximo da solução acaba por gerar sucessores que afinal tornam o caminho até à solução impossível. Estas situações são devidas ao **fenómeno de deterioração** que a resolução da violação de uma restrição provoca na outra, como mencionado no início deste relatório. Assim, mesmo como uma boa heurística, estas situações acabam por tornar inviável a prática destas procuras em problemas mais complexos no espaço de tempo útil definido para a amostra.

- ***Heuristic-conflicts* e *Heuristic-max***

Tal como referido na *secção 5 – **Heurísticas e Estratégias de Corte***, conseguimos ver através do *Gráfico 2* que as estratégias apresentam uma melhor *performance* com a *heuristic-conflicts* uma vez que o foco desta heurística é simplesmente encontrar um estado válido; por outro lado, com a *heuristic-max* a performance diminui drasticamente, já que o seu foco é uma solução ótima. No entanto, há que ter em conta que, ao contrário da *heuristic-conflicts*, a *heuristic-max* encontra ***sempre*** uma solução ótima. Posto isto, uma vez que o diferencial de *performance* entre as procuras com as diferentes heurísticas é, através da observação do *Gráfico 2*, claramente visível, podemos destacar como **melhor heurística** a *heuristic-conflits*.

A *Tabela 2* do ***Anexo I*** deste relatório detalha os resultados obtidos de aplicar cada procura aos problemas estudados.

7.2.3 Procuras Não-Sistemáticas

Os resultados de *performance* obtidos para as procuras não-sistemáticas são os expressos no gráfico abaixo:

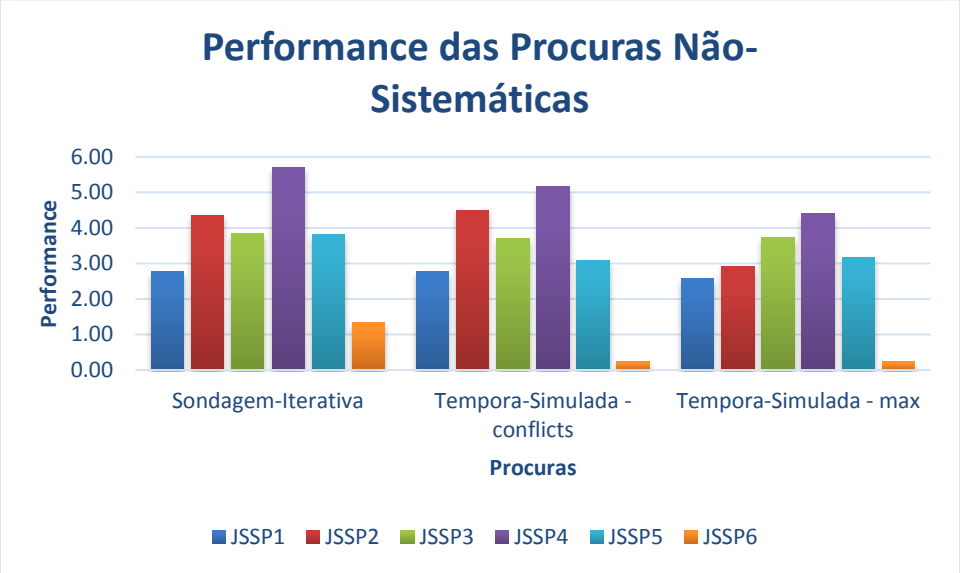

Gráfico 3 - Performance das Procuras Não-Sistemáticas

- **Sondagem-Iterativa**
  Como observável através do *Gráfico 3*, a sondagem-iterativa apresenta a melhor *performance* de todas as procuras analisadas até agora. Isto porque a sua característica de aleatoriedade acelera bastante o processo de encontrar um estado objetivo. No entanto, e visto esta estratégia não utilizar quaisquer heurísticas, as soluções que devolve estão longe de serem ótimas. Deste modo, apenas podemos contar com a expectativa de uma solução válida para o problema.

- **Têmpora-Simulada**
  Os resultados obtidos para esta estratégia em pouco diferem da anterior, no entanto a sua *performance* revela ser menor uma vez que o processo iterativo da escolha de um novo sucessor é consideravelmente mais lento. Isto porque esta estratégia faz uso das heurísticas definidas, pelo que, apesar de **mais lenta**, encontra em **90% dos casos** melhores soluções do que a *sondagem-iterativa*.

- ***Heuristic-conflicts* e *Heuristic-max***
  Mais uma vez pudemos verificar que a *Heuristic-max* contribuiu distintivamente para que se alcançasse uma solução ótima; ao invés da *Heuristic-conflicts* que apenas permitiu encontrar melhores soluções do que a sua suplente em **20% dos casos.** Uma vez que, e como se pode inferir a partir do *Gráfico 3*, a performance das procuras usando cada uma das heurísticas é praticamente idêntica, podemos dizer que a **melhor heurística** para as procuras não-sistemáticas é a *Heuristic-max*.

8 CONCLUSÃO

Este projeto tinha como objetivo a resolução de problemas do tipo *Job Shop Schedulling Problem* utilizando diferentes técnicas de procura, bem como efetuar uma análise comparativa entre as mesmas. O 1º passo para atingir esse objetivo foi modelar o problema de forma a que conseguíssemos ter informação suficiente no estado que o representa (mas não excedentária) para aplicar as estratégias que viemos a desenvolver.

A nível dos algoritmos de procura, começámos pelas procuras cegas que, surpreendentemente, conseguiram atingir resultados bastante interessantes, nomeadamente no caso da procura em **profundidade-primeiro**. De facto, analisando mais a fundo o problema conseguimos compreender que, como referido logo na *secção 1* deste relatório, devido à necessidade de **maturação do equilíbrio entre restrições**, uma solução raramente se encontra em níveis iniciais da árvore do problema. Assim, é compreensível que esta procura tenha revelado bons resultados ao *insistir* na exploração de um caminho até que esse equilíbrio tivesse sido atingido.

No que diz respeito às procuras informadas, desenvolvemos duas heurísticas com resultados bastante díspares, o que seria de esperar dada a diferença conceptual por detrás da implementação das mesmas. Enquanto a heurística *conflicts* se revelou ser a mais adequada a situações cujo objetivo é encontrar uma solução **o mais rápido possível**, a heurística *max* é a melhor escolha quando o objetivo se preocupa mais com a **procura de uma solução ótima**. Deste modo, a estratégia de procura informada que obteve melhores resultados foi a procura A* com a heurística ***conflicts***, isto porque esta se foca apenas em encontrar uma solução válida, logo atinge um estado-objetivo muito mais rapidamente que uma heurística que se preocupe em encontrar uma solução ótima em prol de uma que seja válida – heurística ***max***. No entanto é de assinalar que nos problemas de maior dimensão/complexidade, nenhuma estratégia deste tipo obteve resultados em tempo útil.

Quanto às procuras não-sistemáticas, como já foi mencionado nos pontos anteriores, as que implementámos foram a ***iterative sampling, ILDS*** e, como extras, implementámos ainda ***LDS e simulated annealing.*** Convém ressalvar que tanto a estratégia de procura ***ILDS*** como a ***LDS,*** apesar de estarem completamente implementadas, apresentam um *bug* que não conseguimos resolver e que impossibilita a descoberta de uma solução em tempo útil. No que às restantes diz respeito, é notório o aumento da velocidade na descoberta de solução, o que se traduz em resultados de *performance* bastante melhores, comparativamente com as estratégias até então analisadas. Este aumento de velocidade é justificado pela aleatoriedade inerente a estas estratégias, as quais aliadas à heurística ***max*** (que apesar de mais lenta encontra melhores soluções) produzem muito melhores resultados. A estratégia ***iterative sampling*** é, em média, 500% mais rápida a encontrar uma solução, no entanto essa solução é tipicamente preterível à devolvida pela ***simulated-annealing.*** Deste modo, embora a estratégia de ***sondagem-iterativa*** tenha a melhor *performance* segundo a métrica que desenvolvemos para o seu cálculo, a ***simulated-annealing*** é aquela que, sem dúvida, encontra as melhores soluções, e num espaço de tempo que compensa a espera.

Posto isto, resta concluir que apesar de o desafio de encontrar uma solução ótima não ser de todo trivial, estamos bastante satisfeitos com o resultado do nosso trabalho. Pensamos ter desenvolvido uma solução elegante e facilmente expansível para o problema de Job Shop Schedulling.

ANEXO I – TABELAS DE EXECUÇÃO DAS PROCURAS CEGAS E INFORMADAS

	JSSP1				***JSSP2***				***JSSP4***			
Procuras\Problemas	Gerados	Expandidos	Tempo	***Perf.***	Gerados	Expandidos	Tempo	***Perf.***	Gerados	Expandidos	Tempo	***Perf.***
Largura	27	10	0,01	**2.67**	-	-	-	**0**	-	-	-	**0**
Profundidade	8	3	0,01	**2.64**	82	18	0,03	**4.42**	130	35	0,05	**3.54**
Profundidade-Iter.	23	8	0,01	**2.85**	16100	3047	8,20	**0.57**	-	-	-	**0**
Melhor Solução	IGUAL				Profundidade-Iterativa				N/A			
	JSSP3				***JSSP5***				***JSSP6***			
	Gerados	Expandidos	Tempo	***Perf.***	Gerados	Expandidos	Tempo	***Perf.***	Gerados	Expandidos	Tempo	***Perf.***
	-	-	-	**0**	-	-	-	**0**	-	-	-	**0**
	547	72	0,3	**5.84**	145	37	0,2	**3.27**	-	-	-	**0**
	-	-	-	**0**	-	-	-	**0**	-	-	-	**0**
	N/A				N/A				N/A			

Tabela 1 - Execução das Procuras Cegas sobre *JSSP*

	JSSP1				***JSSP2***				***JSSP3***			
Procuras\Problemas	Gerados	Expandidos	Tempo	***Perf.***	Gerados	Expandidos	Tempo	***Perf.***	Gerados	Expandidos	Tempo	***Perf.***
A* - *conflicts*	14	5	0.01	**2.77**	2145	520	13	**0.29**	-	-	-	**0**
A* - *max*	14	5	0.01	**2.55**	3910	728	50	**0.11**	-	-	-	**0**
IDA* - *conflicts*	8	3	0.01	**2.64**	-	-	-	**0**	-	-	-	**0**
IDA* - *max*	-	-	-	**0**	-	-	-	**0**	-	-	-	**0**
Melhor Solução	IGUAL				A* - *max*				N/A			

Tabela 2 - Execução das Procuras Informadas sobre *JSSP*

- $JSSP_1 < 4, 3, 3 >$
- $JSSP_2 < 6, 6, 6 >$
- $JSSP_4 < 10, 5, 5 >$
- $JSSP_3 < 10, 10, 10 >$
- $JSSP_5 < 20, 5, 5 >$
- $JSSP_6 < 50, 10, 10 >$

ANEXO II – TABELAS DE EXECUÇÃO DAS PROCURAS NÃO-SISTEMÁTICAS

	JSSP1				***JSSP2***				***JSSP4***			
Procuras\Problemas	Gerados	Expandidos	Tempo	***Perf.***	Gerados	Expandidos	Tempo	***Perf.***	Gerados	Expandidos	Tempo	***Perf.***
Sondagem-Iterativa	14	5	0,01	**2,77**	89	20	0,02	**4,36**	95	24	0,03	**3,84**
Tempora-Simulada - *conflicts*	14	5	0,01	**2,77**	74	16	0,03	**4,49**	103	26	0,07	**3,70**
Tempora-Simulada - *max*	13	5	0,01	**2,57**	63	21	0,03	**2,91**	104	26	0,07	**3,74**
Melhor Solução	IGUAL				Tempora-Simulada - *max*				Tempora-Simulada - *max*			
	JSSP3				***JSSP5***				***JSSP6***			
	Gerados	Expandidos	Tempo	***Perf.***	Gerados	Expandidos	Tempo	***Perf.***	Gerados	Expandidos	Tempo	***Perf.***
	434	61	0,25	**5,69**	215	50	0,13	**3,81**	2193	245	5,66	**1,34**
	441	61	0,40	**5,16**	210	50	0,36	**3,09**	2172	243	36	**0,24**
	426	59	0,64	**4,40**	189	45	0,32	**3,18**	2169	241	38	**0,23**
	Tempora-Simulada - *max*				Tempora-Simulada - *max*				Tempora-Simulada - *max*			

Tabela 3 - Execução das Procuras Não-Sistemáticas sobre *JSSP*[4]

- $JSSP_1 < 4, 3, 3 >$
- $JSSP_2 < 6, 6, 6 >$
- $JSSP_4 < 10, 5, 5 >$
- $JSSP_3 < 10, 10, 10 >$
- $JSSP_5 < 20, 5, 5 >$
- $JSSP_6 < 50, 10, 10 >$

[4] Os valores obtidos resultam da média das **10 *runs*** definidas em amostra para cada procura