FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Sequenciamento sustentável – abordagens heurísticas para operações deterioráveis ou perecíveis

Daniel Santos do Carmo Roseta

DISSERTAÇÃO

Mestrado em Engenharia Eletrotécnica e de Computadores

Orientador: Dalila Benedita Machado Martins Fontes
Coorientador: Fernando Arménio da Costa Castro e Fontes

21 de Agosto de 2022

Resumo

Com a competitividade empresarial a aumentar cada vez mais é necessário encontrar maneiras de ganhar vantagem e ter uma boa resolução do problema do job shop e suas variantes. Incorporar a deterioração de tarefas é uma variante relevante e pouco estudada. Esta variante consiste no sequenciamento de operações de um conjunto de tarefas num conjunto de máquinas (JSP) incorporando a deterioração dessas mesmas tarefas (JSPDJ), o que pode ter implicações na conclusão da produção das tarefas e consequentemente na entrega atempada ao cliente. Neste relatório vai ser possível explicar o contexto em que o problema é abordado bem como reportar a parca literatura existente para o mesmo. É abordado o algoritmo (BRKGA), que foi escolhido para solucionar o problema, com a sua explicação, funcionamento e testagem. O impacto da deterioração é demonstrado através de diversos testes variando diferentes parâmetros.

Palavras-chave: problema job shop; deterioração; sequenciamento; meta-heurística; BRKGA; JSPDJ

Abstract

With business competitiveness increasing more and more, it is necessary to find ways to gain advantage and have a good resolution of the job shop problem and its variants. Incorporating task deterioration is a relevant and little studied variant. This variant consists of sequencing the operations of a set of tasks on a set of machines (JSP) incorporating the deterioration of these same tasks (JSPDJ), which can have implications for the completion of the production of tasks and consequently the timely delivery to the customer. In this report it will be possible to explain the context in which the problem is addressed as well as to report on the scarce existing literature for the same. The algorithm (BRKGA), which was chosen to solve the problem, is discussed, with its explanation, operation and testing. The impact of deterioration is demonstrated through several tests varying different parameters.

Keywords: job shop problem; deterioration; sequencing; meta-heuristic; BRKGA; JSPDJ

Agradecimentos

Aos meus pais por nunca terem me deixado faltar nada.
Aos meus amigos e colegas por me sempre apoiarem.
À professora Dalila Fontes por me ter orientado e apoiado durante este trabalho.

Daniel Santos do Carmo Roseta

Conteúdo

Lista de Figuras

Lista de Tabelas

Abreviaturas e Símbolos

PME	Pequenas e Médias Empresas
JSP	Problema de Sequenciamento de tarefas em ambientes de *job shop* (*job shop scheduling problem*)
JSPDJ	Problemas de Sequenciamento em ambiente *job shop* com tarefas Deterioráveis (*job shop scheduling problem with deteriorating jobs*)
BRKGA	Algoritmo genético de chaves aleatórias enviesadas (*bieased random key genetic algorithm*)
Pij	tempo de processamento da operação j da tarefa i
(alpha)ij	coeficiente de deteriorização da operação j da tarefa i
t	tempo em que a operação começou
n	tamanho de cada cromossoma
p	tamanho da população
pe	fração da população que é elite
pm	fração da população substituída por mutantes
rhoe	probabilidade de receber o valor do parente elite

Capítulo 1

Introdução

1.1 Contexto e Motivação

Muitas pequenas e médias empresas (PMEs) enfrentam dificuldades decorrentes de concorrência agressiva, legislação restritiva, produtividade insuficiente, elevados custos energéticos e das matérias-primas, entre outros. O atual contexto pandémico veio intensificar os problemas sentidos, pelo menos alguns.

Assim, para assegurar a sobrevivência, é necessário decidir melhor e mais rápido, utilizando os recursos cada vez mais limitados e/ou mais caros da forma mais eficiente, garantindo a satisfação do cliente. Para isto e necessário um planeamento rigoroso de todas as etapas da produção.

Os problemas de *job shop* estão associados a processos de produção onde e produzido um elevado número de produtos diferentes e em pequenas quantidades; frequentemente customizados, i.e., de acordo com as especificações do cliente.

Como a maior parte das PMEs, que constituem uma parte significativa do tecido empresarial, têm processos de produção do tipo *job shop* escolheu-se estudar o problema de sequenciamento de tarefas em ambientes de *job shop*.

Em problemas de *job shop*, o tempo de processamento é constante e conhecido; no entanto, recentemente começaram a ser estudados problemas onde o tempo de processamento aumenta com a data de início do processamento, designados na literatura por problemas de sequenciamento em ambiente *job shop* com tarefas deterioráveis - *job shop scheduling problem with deteriorating jobs* (JSPDJ). Este tipo de problemas de sequenciamento pode ser encontrado em diversas aplicações tais como, indústria (onde o aço arrefece enquanto espera, requerendo subsequente aquecimento), operações de socorro (deterioração das condições atmosférica ou luminosidade), assistência médica (deterioração da saúde do paciente), combate a incêndios (progressão do fogo), entre outras. Em todas estas situações a conclusão da tarefa em mãos requer mais tempo à medida que o tempo passa sem que a tarefa seja iniciada.

1.2 Objetivos

Com o aumento da competitividade entre as empresas a dissertação tem como objetivos:

- Melhorar a competitividade e eficiência das organizações
- Fazer face à legislação e requisitos sociais e económicos, cada vez mais exigentes
- Abordar um problema quase nada explorado
- Através de uma meta-heurística estudar o impacto da deterioração

1.3 Estrutura do Relatório

Para além da introdução, este relatório contém mais 5 capítulos. No capítulo 2 é apresentada o estado da arte com análise de alguns artigos de algumas vertentes do problema. No capítulo 3 são abordadas a definição do problema e do algoritmo como o seu desenvolvimento e testagem de verificação de qualidade. No capítulo 4 é estudado o impacto da deterioração através da variação tanto do coeficiente de deteriorização, como do número de operações e o número de máquinas. Assim vão ser mencionadas formas de tentar mitigar o seu efeito. No capítulo 5 consiste num sumário das ideias abordadas nos outros capítulos. Por fim no capítulo 6 tem a parte do código onde estão as variáveis para serem alteradas.

Capítulo 2

Estado da arte

2.1 JSPDJ

O problema de sequenciamento de tarefas em ambientes de *job shop* (JSP - *job shop scheduling problem*) envolve a determinação da sequência de processamento das operações de um conjunto de tarefas (ou ordens de produção) num conjunto de máquinas (ou por um conjunto de recursos). Cada tarefa consiste num conjunto ordenado de operações. Cada operação tem de ser executada numa máquina pré-definida (ou por um recurso pré-definido) e o tempo de execução é constante e conhecido. Cada máquina ou recurso executa uma operação de cada vez sem interrupções e as operações de cada tarefa têm de ser executadas uma de cada vez.

Ao longo dos anos várias medidas de performance têm sido propostas, sendo a mais comum a minimização do *makespan*, isto é, o tempo necessário para executar todas as operações de todas as tarefas. O JSP é um problema NP-*hard*, ou seja, é um problema para o qual não existem algoritmos que o resolva em tempo polinomial. Logo há medida que a dimensão do problema vai aumentando, o número de operações que um algoritmo tem de realizar para o resolver aumenta muito rapidamente.

A incorporação da deterioração das tarefas consiste numa extensão ao problema do JSP original e por isso conduz a um problema também NP-*hard*. A literatura em problemas de sequenciamento com deterioração das tarefas remonta ao início dos anos 90. Desde então vários problemas têm vindo a ser abordados tais como *single-machine scheduling problem*, *flow shop* e *open shop*. No entanto o JSP não tem sido muito investigado, apesar da sua enorme importância e relevância.

A Figura 2.1 exemplifica uma situação onde a deterioração da tarefa pode conduzir à não conformidade em termos de qualidade. Caso tal se verifique é necessário reprocessamento, o que, naturalmente, implica tempo e custos de produção adicionais.

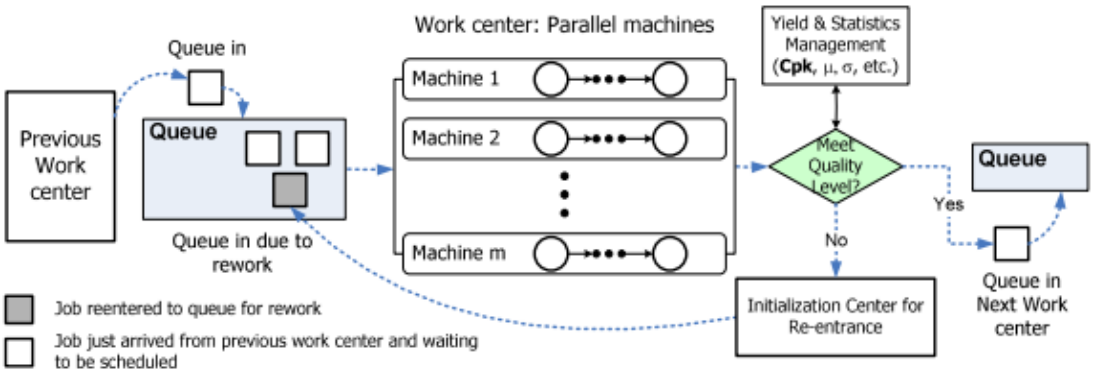

Figura 2.1: Exemplo de um problema de *Job Shop* com deterioração, adaptado de [1]

A grande diferença entre o JSP clássico e o problema a abordar é o facto de o tempo de processamento deixar de ser constante. Há várias questões que podem conduzir a essa variação, nomeadamente os recursos alocados, o número de tarefas já executadas e o início da execução. Nesta dissertação vamos nos concentrar nestes últimos. Na literatura podem ser encontrados três modelos de tempo de processamento variável em função do início do processamento (*time-dependent processing times*), de acordo com as implicações no tempo de processamento [2], sendo sobre este artigo até ao fim da secção:

- No primeiro grupo quanto mais tarde for realizada a tarefa maior é o seu tempo de processamento. São chamados de *deteriorating jobs* [3] [4] [5];
- No segundo grupo quanto mais tarde a tarefa for realizada menor é o seu tempo de processamento. São chamados de *shortening jobs* [6] [7];
- No terceiro grupo é englobado tudo o que não é considerado dos outros dois grupos, por exemplo tarefas cujo tempo de processamento aumento em alguns períodos e diminui noutros. O tempo de processamento varia de forma arbitrária. São chamados de *alterable jobs* [8].

Nesta dissertação apenas os problemas envolvendo tempos de processamento crescentes (ou não decrescentes) serão abordados.

O problema de sequenciamento numa máquina (*single-machine scheduling*) é o mais estudado na presença de tempos de processamento variáveis. A maior parte dos trabalhos envolve *deteriorating jobs*, mas também há algum trabalho publicado envolvendo *shortening jobs*. O tempo de processamento pode variar de forma linear ou não linear, isto é, ou aumenta ou diminui em função do início do processamento, mas sempre ao mesmo ritmo ou então altera com ritmos diferentes ao longo do tempo, por exemplo, aumenta mais num certo período do que noutro.

Segue-se o problema de sequenciamento em máquinas paralelas (*parallel-machine scheduling*). Para este problema é explicado o mesmo que no anterior.

Por fim com o problema de sequenciamento numa máquina dedicada (*dedicated-machine*) (não sendo exatamente sobre o JSP é o problema que mais se aproxima do tema pretendido a ser

tratado neste relatório intermédio) é abordado as mesmas variantes explicadas para o sequenciamento numa máquina.

Sendo o artigo estudado em relação às últimas quatro décadas é possível concluir que relativamente ao JSP com a deterioração de tarefas praticamente é inexistente o estudo deste problema, ao contrário do que acontece com as suas outras variantes como o *flexible job shop*, *open shop* e *flow shop*. Através destas variantes somos capazes de usar os métodos utilizados alterando só o número de máquinas e a ordem das operações.

Neste artigo analisado, o autor direciona-se mais para o *open shop* e *flow shop*. Com os diversos teoremas abordados várias conclusões são tiradas como:

- O *flow shop* é um problema difícil de fazer aproximações, mesmo com alguns graus de deterioração iguais;
- O *flow shop* tem propriedades semelhantes em relação quando os tempos de processamento são fixos;
- O problema das três máquinas *open shop* dependendo do tempo de processamento é um problema intratável, mesmo com restritos graus de deterioração.

Estas conclusões apesar de não serem do problema pretendido é possível aplicá-las na mesma visto que as similaridades entre o JSP com as variantes são muitas.

Um outro exemplo de *flow shop* é segundo [9], em que as tarefas são processadas em m máquinas e seguindo sempre o mesmo caminho, o tempo de processamento normal é fixo para cada tarefa enquanto o tempo de processamento atual varia de acordo com o tempo inicial. Assumem que cada máquina só consegue processar uma tarefa de cada vez, que cada tarefa só pode ser processada numa máquina, que o processamento de uma tarefa não pode ser interrompido, que todas as tarefas estão prontas para serem processadas no início e sem nenhuma restrição acerca de antecedentes necessários e que todas as máquinas estão disponíveis.

De forma a minimizarem o *tardy time* criam uma função objetivo com várias restrições. A metodologia que foi usada foi um algoritmo meta heurístico denominado MVO (*multiverse optimizer*) que utiliza seleção elitista e pesquisa local.

Esta metodologia não vai ser aplicada pois:

- Algoritmo demasiado complexo com diversas operações complicadas e de difícil interpretação
- Com a pesquisa local existe a possíbilidade de não ser ótimo uma vez que o número de iterações pode não ser suficiente para encontrar a solução ótima

Sendo o problema de sequenciamento numa máquina com deterioração um dos problemas mais estudados também é possível encontrar vários exemplos acerca disso sendo o mais relevante para o desenvolvimento desta dissertação a ser apresentado de seguida.

A diferença entre o modelo de uma máquina com o JSP é o número de possibilidades de sequenciamento existentes, isto é, enquanto que no problema a ser trabalhado nesta dissertação as tarefas podem ser feitas em diversas máquinas, quando só há uma máquina as possibilidades de combinação decresce muito porque só existe essa hipótese onde podem ser realizadas as diferentes operações tendo só que seguir a ordem definida das tarefas. Quanto à deterioração não existe diferenças entre os dois modelos.

Segundo [10], as tarefas podem ser agrupadas em grupos dependendo da similaridade das suas operações em que cada um executa um determinado número de forma consecutiva. Designa (alfa)ij como um coeficiente de deterioração da operação Jij (j é a posição da operação no grupo e i é o grupo i) e utiliza tempos de processamento independentes de grupo para grupo. A fórmula que foi usada para o tempo de processamento foi:

$$P_{ij} = \alpha_{ij}.(a+bt) \tag{2.1}$$

Na equação 2.1 a letra a e a letra b são maiores do que zero e a letra t corresponde ao tempo em que a operação Jij começou. Esta fórmula vai ser usada na dissertação para calcular o tempo de processamento.

Com esta expressão a deterioração é proporcionalmente linear pois o tempo de processamento aumenta há medida que t cresce.

Os métodos que usaram foram diversos teoremas consoante se o número de grupos no início era conhecido ou não porém esta metodologia não vai ser seguida pois:

- Não dá a certeza de a solução encontrada ser a ótima
- É direcionado para uma só máquina quando o problema desta dissertação é com várias máquinas o que tornaria os teoremas demasiado complexos

2.1.1 Localização no estado da arte

Apesar de não se ter encontrado artigos diretamente relacionados com o problema, através de variantes foi possível detetar informação útil a ser usada tal como a expressão 2.1 e talvez alguns dados para serem usados no problema, algo ainda a ser estudado mais posteriormente com o desenvolvimento da dissertação.

Pode fornecer um contributo grande às empresas quando revolvido eficazmente solucionando várias contrariadades e permitindo baixar os custos de produção, o que consequentemente fará aumentar os lucros, sendo este o objetivo de qualquer empresa.

Assim é possível dizer que este trabalho sendo realizado de forma eficiente tem um lugar privilegiado no estado da arte por ser um assunto quase nada abordado, no sentido em que as variantes sendo muito mais exploradas existe muito mais informação acerca delas o que seria muito mais complexo para um artigo se poder posicionar dentro dessa bibliografia.

2.2 BRKGA

O BRKGA começa a aparecer na literatura de forma mais estudada a partir dos anos 2000, porém só são mesmo aplicados em JSP em dois artigos e para o JSPDJ não se encontra nenhum.

Em [11] e em [12] utilizam o BRKGA num JSP com a função objetivo de minimizar o tempo total em que são realizadas todas as operações. Com o algoritmo a ser comparado diversos outros algoritmos e a apresentar excelentes resultados. No primeiro artigo o algoritmo consegue encontrar os melhores tempos para as ordens de sequenciamento tendo conseguido para 72% dos testes.

O BRKGA é apresentado em diversos outros artigos para outras funcionalidades:

- Problema de redes [13]
- Empacotamento bidimensional [14]
- Minimização do trânsito nas redes rodoviárias [15]
- Empacotamento tridimensional [16]

Capítulo 3

JSP e BRKGA

3.1 JSP

O problema de sequenciamento de tarefas em ambientes de *job shop* (JSP - *job shop scheduling problem*) envolve a determinação da sequência de processamento das operações de um conjunto de tarefas (ou ordens de produção) num conjunto de máquinas (ou por um conjunto de recursos). As características a ter em atenção são as seguintes:

- Cada tarefa consiste num conjunto ordenado de operações;
- Cada operação tem de ser executada numa máquina pré-definida (ou por um recurso pré-definido);
- Cada máquina ou recurso executa uma operação de cada vez sem interrupções;
- As operações de cada tarefa têm de ser executadas uma de cada vez.

Quanto à deterioração das tarefas vai ser aplicado uma fórmula em que quanto mais tarde for o processamento maior será o tempo necessário para ser concluída essa mesma tarefa, ou seja, o tempo de processamento aumenta. A expressão matemática é a seguinte:

$$P_{ij} \quad = \quad P_{ij} + \alpha_{ij}.t \tag{3.1}$$

Nesta equação os parâmetros têm o seguinte significado:

- Pij - tempo de processamento da operação j da tarefa i
- (alpha)ij - coeficiente de deteriorização da operação j da tarefa i
- t - tempo em que a operação começou

Anteriormente tinha sido estudado a expressão 2.1, porém não se seguiu por esta porque se a operação começa-se no tempo 0 não ficaria com o seu tempo de processamento inicial sem deterioração.

A função objetivo vai ter como objetivo minimizar o tempo em que todas as operações.

A formulação matemática pode ser vista em [17].

Ficou decidido o uso de um algoritmo genético de chaves aleatórias enviesadas.

Os algoritmos genéticos lidam com populações de soluções em que cada solução é um código. Começa por fazer uma seleção dos melhores indivíduos, os mais adaptáveis são aqueles que têm maior probabilidade de serem escolhidos, visto através do valor do fitness. Posteriormente os códigos sofrem um crossover, mantendo uma parte do seu código original e substituindo a outra pelo código de outra solução. Arbitrariamente no final ainda pode haver mutação, por exemplo, a alteração de um número do código.

O algoritmo de chaves aleatórias enviesadas é uma variante do algoritmo genético.

3.2 BRKGA

Num BRKGA começa-se com uma população de cromossomas, em que cada cromossoma é um vetor de chaves aleatórias, e essa população dura até um certo número de iterações que neste caso são as gerações. Esse vetor é gerado de forma aleatória com números entre 0 e 1, sendo que 1 está fora deste intervalo.

Com a função objetivo calcula-se o valor de cada cromossoma, vai ser o seu fitness, e com isso faz-se a escolha entre os indivíduos elite e os não elite. Os elite correspondem à parte da geração com melhor valor.

De forma a evoluir a população é criada uma nova geração em que mutantes são inseridos. O grupo elite e copiado e além disso através de *crossovers* é produzido o número de cromossomas restantes para a nova geração. Num BRKGA o *crossover* é feito através de um elite com um não elite. É adicionado uma probabilidade em que o cromossoma do *crossover* pode receber o valor do pai de elite e se não se confirmar essa probabilidade recebe o valor do pai não elite.

Com a população concluída é calculado o valor de cada cromossoma através do descodificador e volta a haver a divisão entre elite e não elite.

Para o BRKGA foram utilizados os seguintes valores nos parâmetros:

- n - tamanho de cada cromossoma - Vai variar consoante o teste
- p - tamanho da população - Começou-se por utilipar p = 100, porém num estado avançado de testagem verificou-se que com 200 dava melhores resultados e então repetiu-se tudo
- pe - fração da população que é elite - 0.2
- pm - fração da população substituída por mutantes - 0.1
- rhoe - probabilidade de receber o valor do parente elite - 0.7

Os valores de pe, pm, rhoe foram aconselhados pelo autor do código do BRKGA.

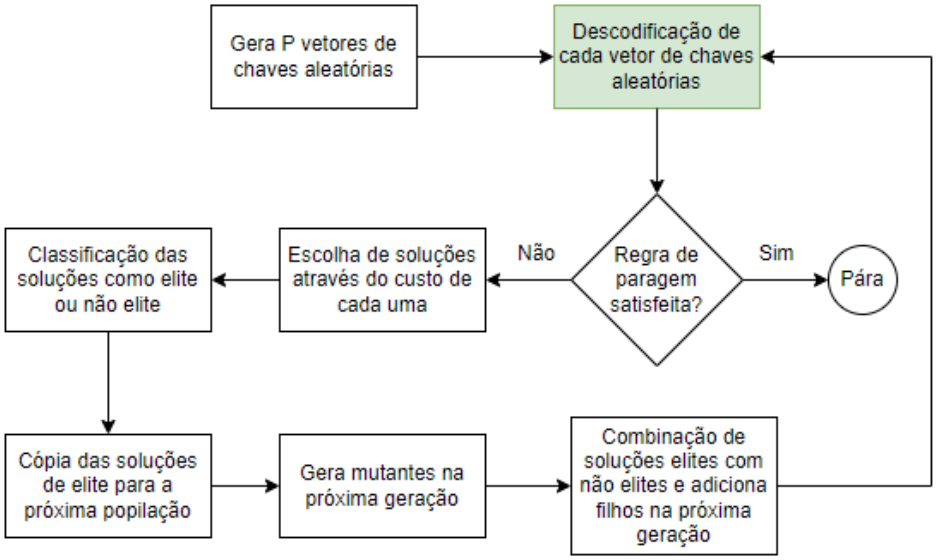

Figura 3.1: Estrutura do código do algoritmo

O descodificador é o que está a verde na figura 3.1 e o BRKGA é obtido em [18]

3.3 Descodificador

Para descodificar o vetor de chaves aleatórias usou-se o seguinte algoritmo:

1	Tarefa 1 2	3	Tarefa 2 4	5	Tarefa 3 6	7	Tarefa 4 8
0.5	0.9	0.4	0.7	0.2	0.8	0.05	0.6
7	5	3	1	8	4	6	2
4	3	2	1	4	2	3	1
O41	O31	O21	O11	O42	O22	O32	O12

O41 = primeira operação da tarefa 4

Figura 3.2: Algoritmo a ser implementado

- Ordenar o vetor recebido

Depois de ser recebido o cromossoma foi necessário ordenar o vetor por ordem crescente. À medida que a ordenação é feita, o número aleatório é substituído pelo seu índice, ou seja, coloca-se a posição na qual se encontrava no vetor original.

Para este passo foi preciso fazer uma cópia do vetor fornecido de maneira a se proceder a quaisquer alterações no vetor cópia e não no original. Foi criado também um vetor inicialmente vazio que, com a ajuda dos vetores original e cópia, foi posteriormente preenchido com os índices ordenados, como foi explicado no parágrafo anterior.

O código em baixo demonstra como foi feita a ordenação dos índices.

Listing 3.1: Código para o primeiro passo

```
// copiar o vetor para nao estar a mexer no fornecido
vector<double> copia = chromosome;
vector<int> indices_ordenados;

// colocar indices do vetor original ordenados no vetor indices_ordenados
for (unsigned i = 0; i < copia.size(); i++) {
    int indice = min_element(copia.begin(), copia.end()) - copia.begin();
    copia[indice] = 1;
    indices_ordenados.push_back(indice);
}
```

- Tarefas correspondentes

O passo seguinte passou por substituir cada número pelo número da respetiva tarefa a que corresponde.

Através do vetor com a quantidade de operações que cada tarefa usa, que é algo fornecido pelos dados do problema, procedeu-se à substituição dos valores no vetor cópia.

Listing 3.2: Código para o segundo passo

```
// criar vetor tarefas a partir do vetor DIVISAO_TAREFAS fornecido,
//que vai conter a tarefa correspondente para cada indice.
// substituir no vetor copia ao produto a que correspondem,
//por exemplo, se o trabalho 1 tiver duas operacoes os numeros na copia
//que correspondem as posicoes 0 e 1 no vetor
// aleatorio v o ter o numero 1
vector<int> tarefas;

for (unsigned i = 0; i < DIVISAO_TAREFAS.size(); i++) {
```

```
        for (int j = 0; j < DIVISAO_TAREFAS[i]; j++) {
            tarefas.push_back(i+1);
        }
    }

    // criar vetor tarefas_ordenas com as tarefas correspondentes
    //a cada indice por ordem dos indices ordenados
    vector<int> tarefas_ordenadas;

    for (unsigned i = 0; i < indices_ordenados.size(); i++) {
        tarefas_ordenadas.push_back(tarefas[indices_ordenados[i]]);
    }
```

- Expressão Ox-x

Por último substitui-se cada número pela expressão Ox-x em que o primeiro valor corresponde à tarefa e o segundo à operação dessa mesma tarefa. Por exemplo, no primeiro 1 que encontrar será O1-1 (primeira operação da tarefa 1) e no segundo 1 coloca-se O1-2 (segunda operação da tarefa 1).

Quanto ao hífen na expressão, inicialmente não era para ser utilizado, porém tornou-se necessário para poder distinguir o número da tarefa do número da ordem da operação quando um destes, ou ambos, apresentam mais que um algarismo. Por exemplo, no caso da operação "O8-11", sem o hífen ("O811") seria impossível de determinar se esta seria a décima primeira operação da tarefa 8 ou a primeira operação da tarefa 81. O uso do hífen permite distinguir de melhor forma os números correspondentes à tarefa e à ordem da operação.

A partir do vetor de tarefas ordenadas, criou-se o vetor "operacoes"que contém as operações ordenadas no formato "Ox-x"em que o primeiro "x"corresponde à tarefa e o segundo "x"à ordem. Para determinar a ordem das operações foi criado um vetor que mantém o registo e controla o número de vezes que cada tarefa foi utilizada.

Listing 3.3: Código para o terceiro passo

```
// criar vetor operacoes com as operacoes ordenadas no formato Ox-x,
//em que o primeiro x corresponde a tarefa e o segundo x a ordem
vector<string> operacoes;
vector<int> tarefas_utilizadas(DIVISAO_TAREFAS.size(), 1);

for (unsigned i = 0; i < tarefas_ordenadas.size(); i++) {
    string operacao = "O" + to_string(tarefas_ordenadas[i]) + "-" +
    to_string(tarefas_utilizadas[tarefas_ordenadas[i]-1]);
    tarefas_utilizadas[tarefas_ordenadas[i]-1]++;
```

```
        operacoes.push_back(operacao);
    }
```

- Cálculo da função objetivo

No código, de maneira a se conseguir obter o resultado da função objetivo correspondente ao cromossoma é necessário calcular os tempos iniciais e finais de cada operação e máquina, tendo em conta o tempo de processamento e coeficiente de deterioração de cada operação, bem como a distribuição das operações pelas máquinas, dados fornecidos do problema. Com estes cálculos é possível determinar o tempo a que termina a última operação de todas, que é o resultado da função objetivo.

O tempo inicial de cada operação na máquina é sempre calculado através do máximo entre o tempo final atual da respetiva máquina e o tempo final da operação com a ordem anterior (se existir).

Listing 3.4: Código para o quarto passo

```
// calcular tempos das maquinas e operacoes
vector<double> maquinas_tempos(NUMERO_MAQUINAS, 0);
unordered_map<string, double> operacoes_tempos;

for (unsigned i = 0; i < operacoes.size(); i++) {
    string operacao = operacoes[i];

    double duracao = OPERACOES_DURACOES.at(operacao);
    int maquina = OPERACOES_MAQUINAS.at(operacao);
    double inicio = maquinas_tempos[maquina-1];
    int ordem = stoi(operacao.substr(operacao.find("-")+1));

    // se ordem for maior que 1 e preciso verificar se o final da operacao
    //com a ordem anterior foi depois do inicio calculado.
    // se sim, o inicio e aquando o fim da operacao com a ordem anterior.
    // se nao, o inicio mantem-se o mesmo calculado anteriormente.
    if (ordem > 1) {
        string operacao_anterior = operacao.substr(0, operacao.find("-")+1)
                                   + to_string(ordem-1);
        double fim_operacao_anterior =
        operacoes_tempos.at(operacao_anterior);

        if (fim_operacao_anterior > inicio)
```

```
                inicio = fim_operacao_anterior;
        }

        // calcular tempo final tendo em conta coeficiente e a duracao
        double coeficiente = OPERACOES_COEFICIENTES.at(operacao);
        double fim = inicio + inicio*coeficiente + duracao;

        maquinas_tempos[maquina-1] = fim;
        operacoes_tempos[operacao] = fim;
    }

    double tempo_maximo = *max_element(maquinas_tempos.begin(),
                                       maquinas_tempos.end());

    return tempo_maximo;
```

3.3.1 Verificação do descodificador

Antes de conectar o descodificador ao código principal formando assim o BRKGA procedeu-se à sua verificação de maneira a observar o seu funcionamento. Através de dados criados por mim sempre com vetores de baixa dimensão para se saber o resultado que deve dar, os resultados foram os seguintes:

Com o cromossoma [0.3 0.1 0.2 0.05 0.6] e os dados da tabela 3.1

tarefa	operação	máquina	tempo processamento
1	1	2	3
1	2	1	2
2	1	1	2
3	1	1	5
3	2	2	5

Tabela 3.1: Dados para verificação do descodificador

Com o cromossoma [0.2 0.03 0.7 0.5 0.3] e os dados da tabela 3.2

tarefa	operação	máquina	tempo processamento
1	1	2	3
1	2	1	2
2	1	2	2
3	1	1	5
3	2	2	5

Tabela 3.2: Dados para verificação do descodificador

Com o cromossoma [0.4 0.6 0.2 0.9 0.03 0.1] e os dados da tabela 3.3

tarefa	operação	máquina	tempo processamento
1	1	3	3
1	2	1	2
2	1	3	2
2	2	2	2
3	1	1	5
3	2	2	5

Tabela 3.3: Dados para verificação do descodificador

Com o cromossoma [0.5 0.1 0.01 0.7 0.03 0.2] e os dados da tabela 3.4

tarefa	operação	máquina	tempo processamento
1	1	3	3
1	2	1	2
2	1	3	2
2	2	2	2
3	1	1	5
3	2	2	5

Tabela 3.4: Dados para verificação do descodificador

Com o cromossoma [0.3 0.1 0.2 0.05 0.6 0.4] e os dados da tabela 3.5

tarefa	operação	máquina	tempo processamento
1	1	2	3
1	2	1	2
2	1	1	2
3	1	1	5
3	2	2	5
4	1	3	15

Tabela 3.5: Dados para verificação do descodificador

Obtiveram-se os seguintes resultados:

tabela dos dados	ordem de processamento	valor esperado	valor obtido
tabela 3.1	O3-1, O1-1, O2-1, O1-2, O3-2	10	10
tabela 3.2	O1-1, O1-2, O3-1, O3-2, O2-1	17	17
tabela 3.3	O3-1, O3-2, O2-1, O1-1, O1-2, O2-2	12	12
tabela 3.4	O2-1, O3-1, O1-1, O3-2, O1-2, O2-2	12	12
tabela 3.5	O3-1, O1-1, O2-1, O1-2, O4-1, O3-2	15	15

Tabela 3.6: Verificação do descodificador

Estes resultados permitiram concluir com segurança que o descodificador estava bem elaborado.

Para o resto dos testes juntou-se o descodificador e o código em 6 ao código principal. Os dados dos problemas foram sempre introduzidos manualmente.

3.4 Qualidade do algoritmo

Com o descodificador já conectado ao BRKGA, recebendo o cromossoma deste e procedendo ao algoritmo criado já descrito anteriormente foi necessário perceber a qualidade do método utilizado sem contar com a deterioração de forma a se poder mais à frente criar dados para se comparar permitindo observar o impacto da deterioração.

Este passo foi algo complicado de se tentar mostrar porque não foi encontrado um artigo que fornecesse os dados todos necessários para a realização deste problema.

O artigo [19] foi o que mais próximo teve disso e então foram comparados alguns resultados de forma a se testar a qualidade do algoritmo.

Os dados utilizados foram 10 máquinas, 10 tarefas com 10 operações cada. Cada tarefa tem que passar pelas máquinas todas e o tempo de processamente usado foi de 75 para cada operação. A sequência de máquinas que cada tarefa tem de cumprir foi obtida de forma aleatória.

Os coeficientes da deterioração foram todos postos a zero de forma a se proceder à comparação.

número de máquinas	**número de tarefas**	**número de operações**	**valor obtido**
10	10	100	1125
10	10	100	1250
10	10	100	1125
10	10	100	1050
10	10	100	1200

Tabela 3.7: Qualidade do algoritmo

Os valores obtidos são próximos dos do artigo mencionado permitindo dizer que a qualidade do algoritmo é boa.

Capítulo 4

Deterioração

A deterioração é algo que muitas empresas sofrem com as suas matérias-primas pois quanto mais tempo demorarem a começar a trabalhar nelas maior será o tempo de processamento porque vai exigir um esforço extra para tratar os problemas causados pela degradação do material. Se não for tratado com o devido cuidado correm o risco de não conseguir cumprir os prazos estipulados de entrega que consequentemente provoca um descontentamento nos clientes e pode levar a que estes passem para a concorrência fortalecendo-os e enfraquecendo as empresas onde se encontravam originalmente.

Neste capítulo vão ser realizados vários testes de maneira a se poder obervar o impacto que a deterioração tem num problema de sequenciamento em ambiente *job shop* e vão ser abordadas possíveis maneiras de mitigar o máximo possível os efeitos desta.

4.1 Variação do coeficiente de deteriorização

Os dados que foram criados e usados para este ponto serviram de base para os restantes testes e são apresentados na seguinte tabela:

tarefa	operação	máquina	tempo processamento
1	1	1	10
1	2	3	16
1	3	2	7
2	1	1	13
2	2	3	4
3	1	2	20
3	2	3	15
3	3	1	11
3	4	3	9
3	5	1	17
4	1	2	3
4	2	1	8
4	3	3	4
4	4	2	10
5	1	3	15

Tabela 4.1: Dados do problema

O n para o BRKGA passou a ser 15.

Variando o coeficiente de deteriorização de 0 a 1 em intervalos de 0.1 obteve-se:

coeficiente de deteriorização	resultado obtido
0	77
0.1	100.339
0.2	125.697
0.3	156.956
0.4	198.16
0.5	251.281
0.6	317.301
0.7	398.622
0.8	497.946
0.9	618.288
1	763

Tabela 4.2: Resultados da variação do coeficiente de deteriorização

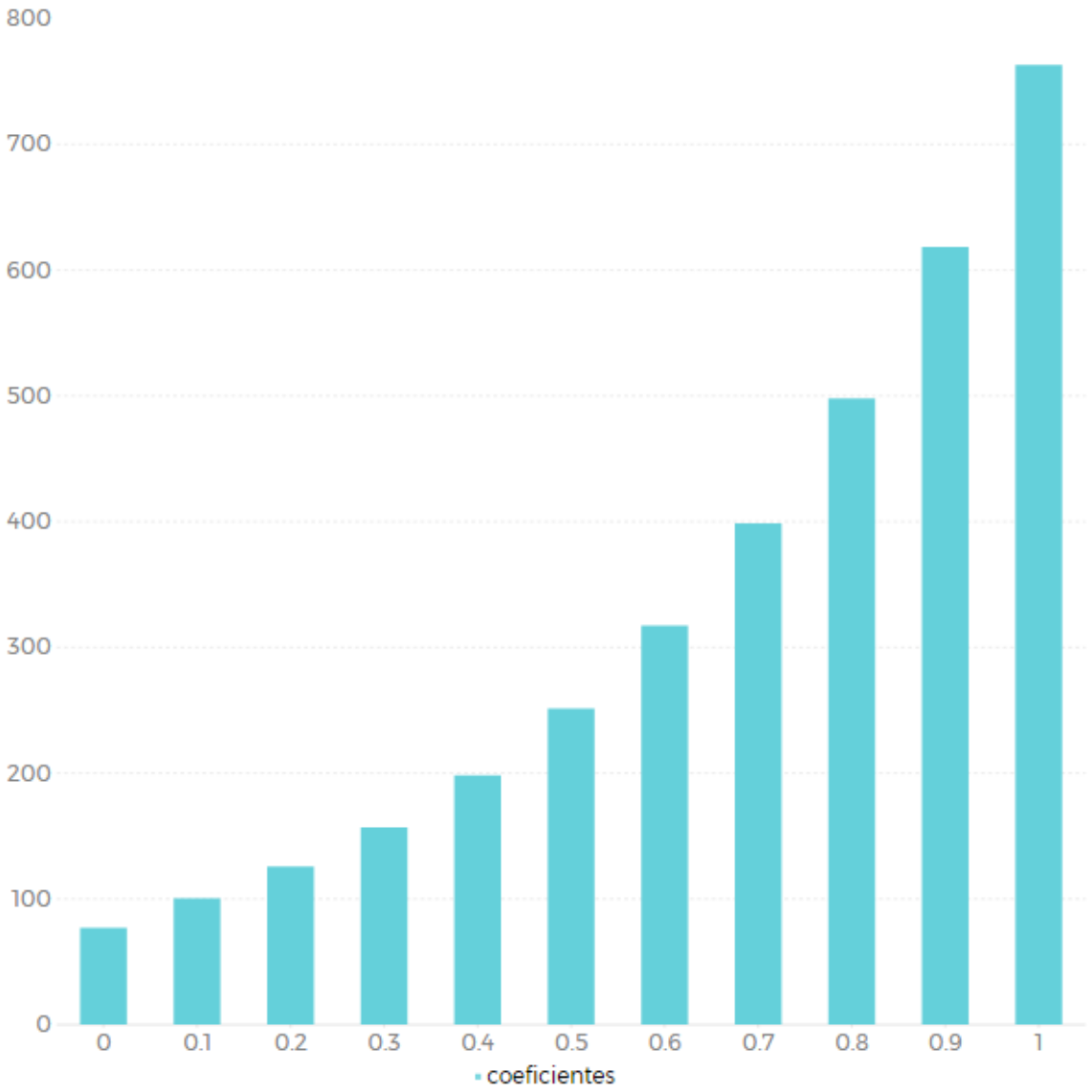

Figura 4.1: Variação do coeficiente de deteriorização

O eixo do x corresponde ao coeficiente de deteriorização e o eixo do y aos resultados obtidos.

É possível concluir que quanto maior o coeficiente de deteriorização o intervalo de tempo entre o início da primeira operação e o fim da última operação aumenta de forma considerável.

Um coeficiente maior significa que a matéria-prima se degrada mais depressa fazendo com que o tempo de processamento necessário aumente também notavelmente.

Observando os valores obtidos sempre que se aumenta 0.1 o resultado cada vez sofre uma variação maior. Para coeficientes baixos como 0.1, 0.2 e 0.3 atingiu-se 100.339, 125.697 e 156.956 respetivamente, e quando comparado com o JSP normal, houve um amento de 30%, 63% e 104%, sendo que já é um aumento considerável. Então quando se compara com o coeficiente 1 o aumento é brutal passando a ser de 891%.

4.2 Variação do número de operações

Aos dados da tabela 4.1 foram adicionados os seguintes:

tarefa	operação	máquina	tempo processamento
1	4	1	10
2	3	1	10
3	6	2	10
4	5	2	10
5	2	3	10

Tabela 4.3: Acrescento para o número de operações igual a 20

O n para o BRKGA passou a ser 20 e conseguiu-se:

deterioração	resultado
0.1	122.76
0.2	179.499
0.3	247.818
0.5	463.422
0.8	1107.13
1	1884

Tabela 4.4: Resultados para o número de operações igual a 20

Acrescentando outras 5 operações:

tarefa	operação	máquina	tempo processamento
1	5	3	10
2	4	3	10
3	7	3	10
4	6	1	10
5	3	1	10

Tabela 4.5: Acrescento para o número de operações igual a 25

O n para o BRKGA passou a ser 25 e obteve-se:

deterioração	resultado
0.1	167.548
0.2	278.376
0.3	453.718
0.5	1197.94
0.8	4732.55
1	10982

Tabela 4.6: Resultados para o número de operações igual a 25

Outras 5 operações:

tarefa	**operação**	**máquina**	**tempo processamento**
1	6	2	10
2	5	2	10
3	8	1	10
4	7	3	10
5	4	2	10

Tabela 4.7: Acrescento para o número de operações igual a 30

O n para o BRKGA passou a ser 30 e os resultados foram:

deterioração	**resultado**
0.1	198.273
0.2	344.051
0.3	599.833
0.5	1806.9
0.8	8528.6
1	21974

Tabela 4.8: Resultados para o número de operações igual a 30

Mais 5 operações:

tarefa	**operação**	**máquina**	**tempo processamento**
1	7	1	10
2	6	1	10
3	9	2	10
4	8	2	10
5	5	3	10

Tabela 4.9: Acrescento para o número de operações igual a 35

O n para o BRKGA passou a ser 35 e conseguiu-se:

deterioração	**resultado**
0.1	261.134
0.2	509.993
0.3	1035.12
0.5	4072.81
0.8	27888.9
1	90454

Tabela 4.10: Resultados para o número de operações igual a 35

Por fim, outras 5 operações:

tarefa	**operação**	**máquina**	**tempo processamento**
1	8	3	10
2	7	3	10
3	10	3	10
4	9	1	10
5	6	1	10

Tabela 4.11: Acrescento para o número de operações igual a 40

O n para o BRKGA passou a ser 40 e por fim obteve-se:

deterioração	**resultado**
0.1	324.571
0.2	742.207
0.3	1714.55
0.5	8941.2
0.8	87324.5
1	343798

Tabela 4.12: Resultados para o número de operações igual a 40

De forma a se poder comparar os resultados mais facilmente criou-se o seguinte gráfico:

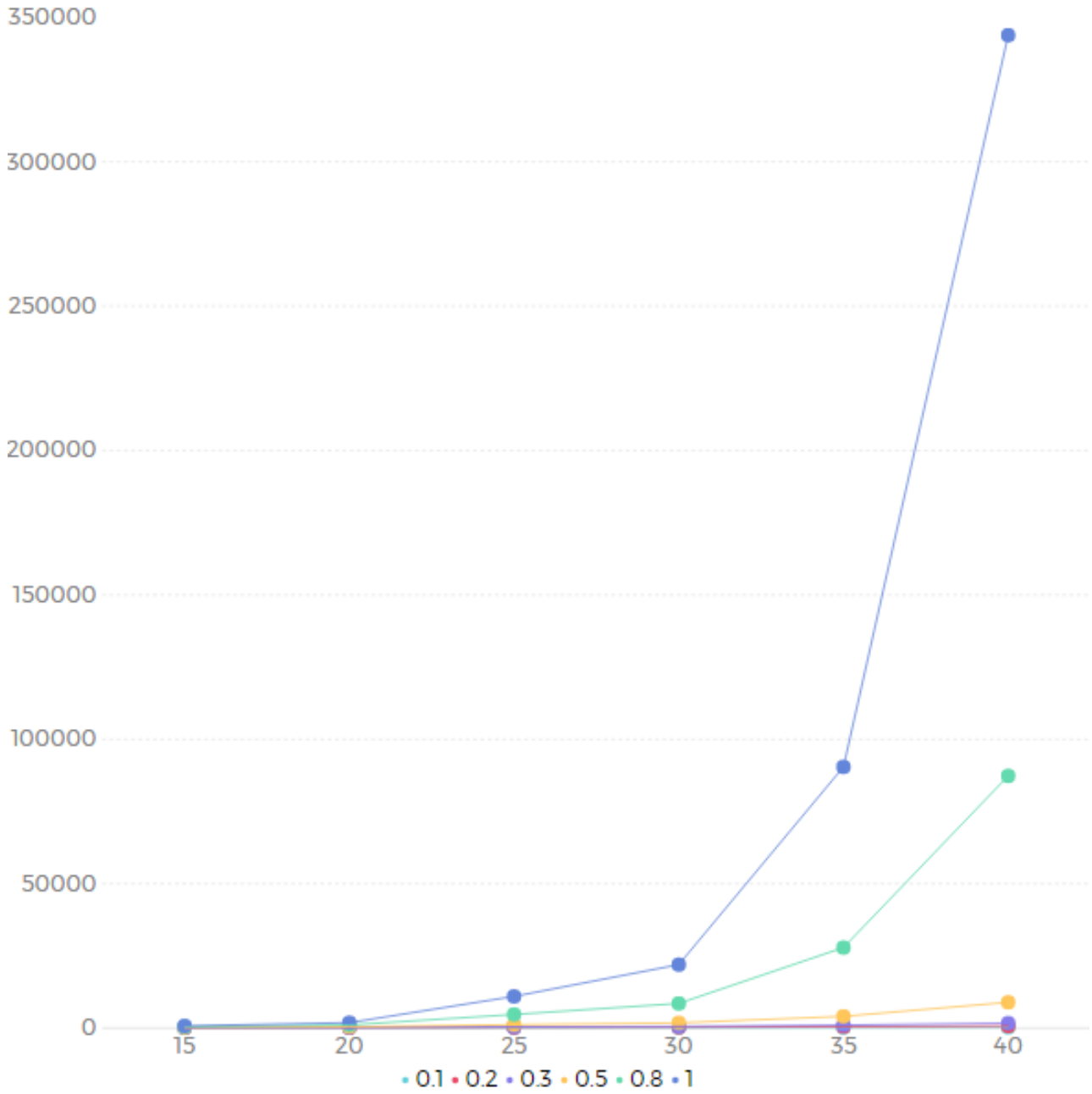

Figura 4.2: Variação do número de operações

O eixo do x corresponde ao número total de operações e o eixo do y ao resultado obtido.

É possível concluir que quanto mais operações o problema tiver maior é o tempo que demora as realizar todas.

Para os coeficientes mais elevados o aumento é brusco, as variações mais elevadas são com o coeficiente 1 em que quando passa de 35 para 40 operações o tempo aumenta substancialmente, cerca de 280%, e é consideravelmente superior quando comparado com o coeficiente 0.8.

Mais uma vez observa-se que para os baixos coeficientes a variação de operações provoca um incremento ligeiro se bem que mesmo assim é grandinho e para os coeficientes altos o aumento

é gigante, sendo cada vez maior o valor que é acrescentado à medida que o número de operações vai aumentando.

De notar que o gráfico é para ser observado juntamente com as tabelas desta secção porque os valores exatos estão nas tabelas e para os coeficientes mais baixos não se nota as diferenças no gráfico parecendo que têm valores 0 mas não.

4.3 Variação do número de máquinas

Aos dados usados anteriormente para a variação do número de operações modificou-se:

tarefa	operação	máquina	tempo processamento
1	2	4	16
2	4	4	10
3	4	4	9
3	10	4	10
4	7	4	10
5	5	4	10

Tabela 4.13: Modificação para o número de máquinas igual a 4

O número de máquinas passou a ser 4 e conseguiu-se:

deterioração	resultado
0.1	295.255
0.2	617.419
0.3	1318.56
0.5	5888.83
0.8	46647.1
1	162134

Tabela 4.14: Resultados para o número de máquinas igual a 4

Alterando outros dados:

tarefa	operação	máquina	tempo processamento
1	4	5	10
2	7	5	10
3	8	5	10
4	4	5	10
5	2	5	10

Tabela 4.15: Modificação para o número de máquinas igual a 5

O número de máquinas passou a ser 5 e obteve-se:

deterioração	**resultado**
0.1	228.951
0.2	419.902
0.3	779.777
0.5	2645.17
0.8	14522.4
1	40790

Tabela 4.16: Resultados para o número de máquinas igual a 5

Po último, acrescentando ainda outra máquina subituiu-se:

tarefa	**operação**	**máquina**	**tempo processamento**
1	1	6	10
2	3	6	10
3	9	6	10
4	5	6	10
5	3	6	10

Tabela 4.17: Modificação para o número de máquinas igual a 6

O número de máquinas passou a ser 6 e os resultados foram:

deterioração	**resultado**
0.1	205.122
0.2	356.179
0.3	638.804
0.5	1877.73
0.8	8215.79
1	19990

Tabela 4.18: Resultados para o número de máquinas igual a 6

Fazendo o mesmo que na secção anterior criou-se um gráfico juntando todos os resultados:

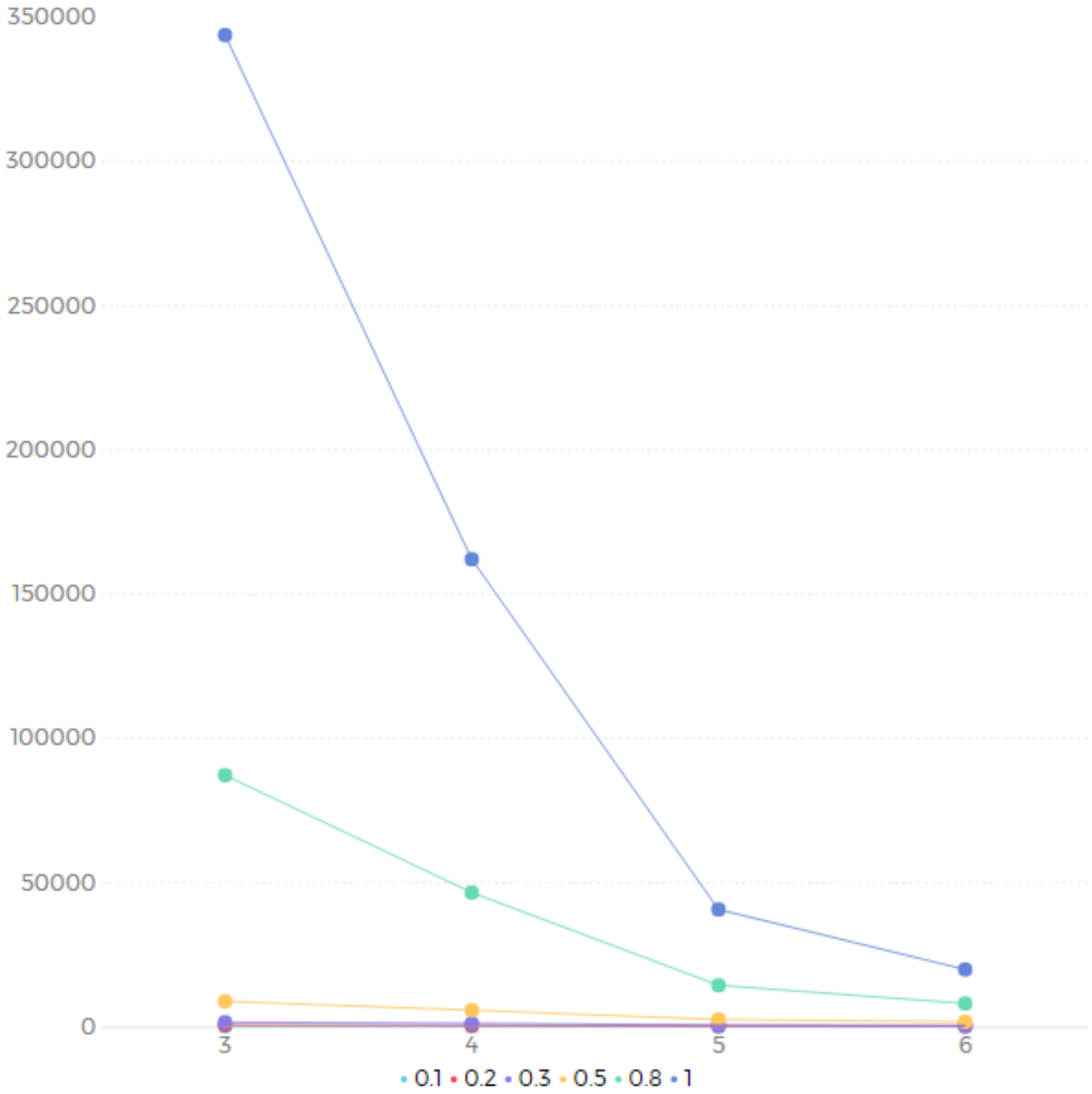

Figura 4.3: Variação do número de máquinas

O eixo do x corresponde ao número de máquinas e o eixo do y ao resultado obtido.

Com este gráfico conclui-se que à medida que o número de máquinas aumenta a ordem de processamento é realizada em menos tempo.

À semelhança dos outros testes para um coeficiente alto a variação é maior quando comparado com um baixo coeficiente, porém de uma forma inversa, ou seja, a variação é maior mas em decrescendo. Por exemplo, com o número de máquinas de 5 para 6, para o coeficiente 1 diminui cerca de 51% enquanto que para o coeficiente 0.1 a descida é mais ligeira sendo de aproximadamente 10%.

De notar que o gráfico é para ser observado juntamente com as tabelas desta secção porque os valores exatos estão nas tabelas e para os coeficientes mais baixos não se nota as diferenças no gráfico parecendo que têm valores 0 mas não.

4.4 Formas de mitigar

As empresas para conseguirem fazer frente à concorrência precisam de alguma forma tentar diminuir os efeitos que a deteriorização causa, contudo para se conseguir tal feito é preciso ou algum poder financeiro ou alguma paciência não querendo ser muito ambiciosos e controlando as coisas.

Duas possíveis maneiras de atenuar as consequências da deteriorização são:

- Aumento do número de máquinas

  Conforme o concluído com os testes realizados no ponto anterior com os respetivos resultados no gráfico da figura 4.3 e nas tabelas da secção 4.3, se conseguirmos disponibilizar um maior número de máquinas de forma a poder aliviar a pressão noutras e a distribuir melhor as operações o tempo com que é executado a ordem de processamento é menor.

  Para este caso é o necessário o já referido poder financeiro. Um maior número de máquinas exige um maior investimento para a empresa, contudo os ganhos podem ser grandes pois entrega os produtos de forma mais rápida, consegues com que os clientes fiquem mais felizes e podem conseguir adquirir novos clientes enfraquecendo a concorrência.

- Ordens de processamento pequenas

  Na impossibilidade de investir grandes quantias em máquinas de forma a concorrer com outras empresas, colocar um limite de operações na ordem de processamento pode ser uma opção válida para se conseguir realizá-la num bom tempo.

  Aqui entra a já mencionada paciência, não querendo ser muito ambiciosos e controlando as coisas. É verdade que com ordens de processamento pequenas o dinheiro que se recebe é menor mas isto permite que os clientes não esperem muito tempo e de forma a contrariar esses ganhos menores pela ordem, os clientes podem sempre vir mais frequentemente.

  Com isto sempre se pode fazer um pouco de frente à concorrência que tenha um maior número de máquinas, permitindo assim que o cliente venha frequentemente para receber produtos mais rapidamente e assim conseguir lucrar um pouco mais.

Capítulo 5

Conclusões

Apesar de ser um problema muito importante e que pode dar vantagem às pequenas e médias empresas aquando da sua resolução, o JSP com a deterioração de tarefas quase nada foi explorado, sendo dada mais relevância pelos diversos autores às suas variantes.

Mesmo sem muita investigação feita ao longo dos anos sobre o JSP com estas características é possível concretizá-lo através de heurísticas e algoritmos, sendo o escolhido o algoritmo de chaves aleatórias enviesadas.

O maior desafio foi implementar o código do algoritmo. Ocurreram diversos problemas durante a sua elaboração que demoraram tempo a serem solucionados para depois. Com dados criados de raíz verificou-se que funciona de forma eficiente e não apresenta erros de execução quando o número de tarefas começa a ser elevado.

Quanto ao estudo do impacto da deterioração foram testados em diversos cenários aumentando o coeficiente, aumentando o número de operações e aumentando o número de máquinas. Concluiu-se que com o aumento das máquinas é possível atenuar os efeitos da deteriorização sendo esta diminuição mais sentida para coeficientes mais elevados, ainda que para os mais baixos também atenua mas não de forma tão acentuada. Com o número de operações a subir notou-se um acréscimo no tempo que demora a realizar todas as operações da ordem de processamento (*makespan*), sendo este, mais uma vez, notado de forma mais expressiva nos coeficientes altos mas desta vez de forma negativa, pois o que se pretende é um tempo baixo e não um muito alto. Já quanto ao coeficiente de deteriorização quanto mais elevado for maior será o *makespan*, notando-se mais sgnificativamente esse aumento nos coeficientes mais altos.

Ainda há um longo caminho a percorrer quanto a problemas com deteriorização de maneira a não se desperdiçar matérias-primas e permitindo um tempo total de processamento de todas as operações considerável baixo para atrair clientes e consequentemente ganhar mais dinheiro, que é para isso que as empresas todas trabalham tentando superar a concorrência.

Capítulo 6

Apêndice

6.1 Código com as variáveis a introduzir

Listing 6.1: Código com as variáveis a introduzir

```
#ifndef DECODER_H
#define DECODER_H

#include <stdio.h>
#include <stdlib.h>
#include <vector>
#include <iostream>
#include <algorithm>
#include <string>
#include <unordered_map>

using namespace std;

class Decoder {
public:
        const vector<int> DIVISAO_TAREFAS{8, 7, 10, 9, 6};
        const int NUMERO_MAQUINAS = 6;
        const unordered_map<string, double> OPERACOES_DURACOES =
                                                {{"O1-1", 10},
                                                {"O1-2", 16},
                                                {"O1-3", 7},
                                                {"O1-4", 10},
                                                {"O1-5", 10},
                                                {"O1-6", 10},
                                                {"O1-7", 10},
```

```
                                                {"O1-8", 10},
                                                {"O2-1", 13},
                                                {"O2-2", 4},
                                                {"O2-3", 10},
                                                {"O2-4", 10},
                                                {"O2-5", 10},
                                                {"O2-6", 10},
                                                {"O2-7", 10},
                                                {"O3-1", 20},
                                                {"O3-2", 15},
                                                {"O3-3", 11},
                                                {"O3-4", 9},
                                                {"O3-5", 17},
                                                {"O3-6", 10},
                                                {"O3-7", 10},
                                                {"O3-8", 10},
                                                {"O3-9", 10},
                                                {"O3-10", 10},
                                                {"O4-1", 3},
                                                {"O4-2", 8},
                                                {"O4-3", 4},
                                                {"O4-4", 10},
                                                {"O4-5", 10},
                                                {"O4-6", 10},
                                                {"O4-7", 10},
                                                {"O4-8", 10},
                                                {"O4-9", 10},
                                                {"O5-1", 15},
                                                {"O5-2", 10},
                                                {"O5-3", 10},
                                                {"O5-4", 10},
                                                {"O5-5", 10},
                                                {"O5-6", 10}};

const unordered_map<string, int> OPERACOES_MAQUINAS =
                                                {{"O1-1", 6},
                                                {"O1-2", 4},
                                                {"O1-3", 2},
                                                {"O1-4", 5},
                                                {"O1-5", 3},
```

```
                                                {"O1-6", 2},
                                                {"O1-7", 1},
                                                {"O1-8", 3},
                                                {"O2-1", 1},
                                                {"O2-2", 3},
                                                {"O2-3", 6},
                                                {"O2-4", 4},
                                                {"O2-5", 2},
                                                {"O2-6", 1},
                                                {"O2-7", 5},
                                                {"O3-1", 2},
                                                {"O3-2", 3},
                                                {"O3-3", 1},
                                                {"O3-4", 4},
                                                {"O3-5", 1},
                                                {"O3-6", 2},
                                                {"O3-7", 3},
                                                {"O3-8", 5},
                                                {"O3-9", 6},
                                                {"O3-10", 4},
                                                {"O4-1", 2},
                                                {"O4-2", 1},
                                                {"O4-3", 3},
                                                {"O4-4", 5},
                                                {"O4-5", 6},
                                                {"O4-6", 1},
                                                {"O4-7", 4},
                                                {"O4-8", 2},
                                                {"O4-9", 1},
                                                {"O5-1", 3},
                                                {"O5-2", 5},
                                                {"O5-3", 6},
                                                {"O5-4", 2},
                                                {"O5-5", 4},
                                                {"O5-6", 1}};

const unordered_map<string, double> OPERACOES_COEFICIENTES =
                                                {{"O1-1", 1},
                                                {"O1-2", 1},
                                                {"O1-3", 1},
```

```
                                        {"O1-4",  1},
                                        {"O1-5",  1},
                                        {"O1-6",  1},
                                        {"O1-7",  1},
                                        {"O1-8",  1},
                                        {"O2-1",  1},
                                        {"O2-2",  1},
                                        {"O2-3",  1},
                                        {"O2-4",  1},
                                        {"O2-5",  1},
                                        {"O2-6",  1},
                                        {"O2-7",  1},
                                        {"O3-1",  1},
                                        {"O3-2",  1},
                                        {"O3-3",  1},
                                        {"O3-4",  1},
                                        {"O3-5",  1},
                                        {"O3-6",  1},
                                        {"O3-7",  1},
                                        {"O3-8",  1},
                                        {"O3-9",  1},
                                        {"O3-10",  1},
                                        {"O4-1",  1},
                                        {"O4-2",  1},
                                        {"O4-3",  1},
                                        {"O4-4",  1},
                                        {"O4-5",  1},
                                        {"O4-6",  1},
                                        {"O4-7",  1},
                                        {"O4-8",  1},
                                        {"O4-9",  1},
                                        {"O5-1",  1},
                                        {"O5-2",  1},
                                        {"O5-3",  1},
                                        {"O5-4",  1},
                                        {"O5-5",  1},
                                        {"O5-6",  1}};

    Decoder();
    ~Decoder();
```

```
        double decode(const std::vector< double >& chromosome) const;

private:
};

#endif
```

Referências

[1] H. J. Shin. A dispatching algorithm considering process quality and due dates: na application for re-entrant production lines. *International Journal of Advanced Manufacturing Technology*, vol. 77:pp. 249–259, Mar. 2015. doi:10.1007/s00170-014-6436-9.

[2] S. Gawiejnowicz. A review of four decades of time-dependent scheduling: main results, new topics, and open problems. *Journal of Scheduling*, vol. 23:pp. 3–47, Feb. 2020. doi:10.1007/s10951-019-00630-w.

[3] A. Arigliano; G. Ghiani; A. Grieco. Single-machine time-dependent scheduling problems with fixed rate-modifying activities and resumable jobs. *4OR*, vol. 15:pp. 201–215, June 2017. doi:10.1007/s10288-016-0333-z.

[4] Q. Chen; L. Lin; Z. Tan; Y. Yan. Coordination mechanisms for scheduling games with proportional deterioration. *European Journal of Operational Research*, vol. 263:pp. 380–389, Dec. 2017. doi:10.1016/j.ejor.2017.05.021.

[5] L. He M. Cheng, S. Sun. Flow shop scheduling problems with deteriorating jobs on no-idle dominant machines. *European Journal of Operational Research*, vol. 183:pp. 115–124, Nov. 2007. doi:10.1016/j.ejor.2006.10.043.

[6] V. Strusevich; K. Rustogi. Scheduling with Time-Changing Effects and RateModifying Activities. *International Series in Operations Research Management Science*, vol. 243, Jan. 2017. doi:10.1007/978-3-319-39574-6.

[7] L.Y. Kang; T.C.E. Cheng; C.T. Ng; M. Zhao. Scheduling to Minimize Makespan with Time-Dependent Processing Times. *Lecture Notes in Computer Science*, vol. 3827:pp. 925–933, Dec. 2005. doi:10.1007/11602613$_9$2.

[8] F. Jaehn; H.A. Sedding. Scheduling with time-dependent discrepancy times. *Journal of Scheduling*, vol. 19:pp. 737–757, Dec. 2016. doi:10.1007/s10951-016-0472-2.

[9] Hongfeng Wang; Min Huang; Junwei Wang. An effective metaheuristic algorithm for flowshop scheduling with deteriorating jobs. *Journal of Intelligent Manufacturing*, vol. 30:pp. 2733–2742, May 2018. doi:10.1007/s10845-018-1425-8.

[10] Feng Liu; Jing Yang; Yuan-Yuan Lu. Solution algorithms for single-machine group scheduling with ready times and deteriorating jobs. *Engineering Optimization*, vol. 51:pp. 862–874, Feb. 2018. doi:10.1080/0305215X.2018.1500562.

[11] J.F. Gonçalves; J.J.M. Mendes; M.G.C. Resende. A hybrid genetic algorithm for the job shop scheduling problem. *European Journal of Operational Research*, vol. 167:pp. 77–95, Nov. 2005. doi:10.1016/j.ejor.2004.03.012.

[12] J.F. Gonçalves; M.G.C. Resende. A Biased Random-Key Genetic Algorithm For Job-Shop Scheduling. *ATT Labs Research Technical Report*, vol. 46, Jan. 2011 [Online] Available: https://www.researchgate.net/publication/265628674$_{Ab}iased_{r}andom-key_{g}enetic_{a}lgorithm_{f}or_{j}ob-shop_{s}cheduling$.

[13] M. Ericsson; M.G.C. Resende; P. Pardalos. A genetic algorithm for the weight setting problem in OSPF routing. *Journal of Combinatorial Optimization*, vol. 6:pp. 299–333, Jan. 2002. doi:10.1023/A:1014852026591.

[14] J.F. Gonçalves. A hybrid genetic algorithm-heuristic for a two-dimensional orthogonal packing problem. *European Journal of Operational Research*, vol. 183:pp. 1212–1129, Feb. 2007. doi:10.1016/j.ejor.2005.11.062.

[15] L.S. Buriol M. Ritt; M.J. Hirsch; P.M. Pardalos; T. Querido; M.G.C. Resende. A biased random-key genetic algorithm for road congestion minimization. *Optimization Letters*, vol. 4:pp. 619–633, Nov. 2010. doi:10.1007/s11590-010-0226-6.

[16] J.F. Gonçalves; M.G.C. Resende. A parallel multi-population biased random-key genetic algorithm for a container loading problem. *Computers Operations Research*, vol. 39:pp. 179–190, Feb. 2012. doi:10.1016/j.cor.2011.03.009.

[17] R. Tavakkoli-Moghaddam; M. Daneshmand-Mehr. A computer simulation model for job shop scheduling problems minimizing makespan. *Computers Industrial Engineering*, vol. 48:pp. 811–823, June 2005. doi:10.1016/j.cie.2004.12.010.

[18] R.F.Toso; M.G.C.Resende. brkgaAPI. *[Online]*, Available: http://mauricio.resende.info/src/brkgaAPI/.

[19] Joseph Adams; Egon Balas; Daniel Zawack. The Shifting Bottleneck Procedure for Job Shop Scheduling. *Management Science*, vol. 34, no. 3:pp. 249–259, Mar. 1988.