UNIVERSIDADE FEDERAL DE OURO PRETO

INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS

DEPARTAMENTO DE COMPUTAÇÃO

Gabriel Caetano Araújo

OTIMIZAÇÃO DO PLANEJAMENTO DE TRANSPORTE E SEQUENCIAMENTO DA PRODUÇÃO E ESTOCAGEM EM MINAS A CÉU ABERTO

Ouro Preto, MG
2022

Gabriel Caetano Araújo

OTIMIZAÇÃO DO PLANEJAMENTO DE TRANSPORTE E SEQUENCIAMENTO DA PRODUÇÃO E ESTOCAGEM EM MINAS A CÉU ABERTO

Monografia II apresentada ao Curso de Ciência da Computação da Universidade Federal de Ouro Preto como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Puca Huachi Vaz Penna
Coorientador: Túlio Ângelo Machado Toffolo

Ouro Preto, MG
2022

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
INSTITUTO DE CIENCIAS EXATAS E BIOLOGICAS
DEPARTAMENTO DE COMPUTACAO

FOLHA DE APROVAÇÃO

Gabriel Caetano Araújo

Otimização do Planejamento de Transporte e Sequenciamento da Produção e Estocagem em Minas a Céu Aberto

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal
de Ouro Preto como requisito parcial para obtenção do título de Bacharel em Ciência da Computação

Aprovada em 27 de Outubro de 2022.

Membros da banca

Puca Huachi Vaz Penna (Orientador) - Doutor - Universidade Federal de Ouro Preto
Túlio Ângelo Machado Toffolo (Coorientador) - Doutor - Universidade Federal de Ouro Preto
Marco Antonio Moreira de Carvalho (Examinador) - Doutor - Universidade Federal de Ouro Preto
André Luyde da Silva Souza (Examinador) - Escola Politécnica de Minas Gerais

Puca Huachi Vaz Penna, Orientador do trabalho, aprovou a versão final e autorizou seu depósito na Biblioteca Digital de Trabalhos de Conclusão de Curso da UFOP em 27/10/2022.

Documento assinado eletronicamente por **Puca Huachi Vaz Penna**, **PROFESSOR DE MAGISTERIO SUPERIOR**, em 03/11/2022, às 10:21, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site http://sei.ufop.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0 , informando o código verificador **0416023** e o código CRC **DBC32DD1**.

R. Diogo de Vasconcelos, 122, - Bairro Pilar Ouro Preto/MG, CEP 35400-000
Telefone: 3135591692 - www.ufop.br

Resumo

O trabalho tem como objetivo propor soluções heurísticas e modelagem em programação linear para os Problemas de Mistura de Minérios e Alocação de Máquinas. O modelo de programação linear consiste em determinar os locais de retomada de minério, de tal forma que a mistura atenda aos limites de especificações de qualidade e quantidade pré-estabelecidos pelo cliente, visando reduzir o tempo de montagem de cada pedido. Para satisfazer as restrições operacionais do pátio, propõe-se um método heurístico composto por um algoritmo construtivo guloso e pelas meta-heurísticas *Simulated Annealing* e *Late Acceptance Hill-Climbing*, nas quais a exploração do espaço de soluções é feita por meio de várias estruturas de vizinhança. Para validar os métodos desenvolvidos foram propostas novas instâncias para o problema baseadas em dados reais de um pátio de uma indústria mineradora.

Palavras-chave: Heurística. Programação linear. Blendagem de minérios. Sequenciamento de máquinas.

Abstract

The research aims to propose heuristic solutions and modeling in linear programming for the Ore Mixing and Unrelated Parallel Machine Schedule Problem. The linear programming model consists of determining the locations of ore recovery, in such a way that the mixture meets the limits of quality and quantity specifications pre-established by the customer. To satisfy the operational restrictions of the yard, a heuristic method is proposed composed of a greedy constructive algorithm and meta-heuristics Simulated Annealing and Late Acceptance Hill-Climbing, in which the exploration of the space of solutions is done through several neighborhood structures. To validate the developed methods, new instances for the problem were proposed based on real data from a mining industry yard.

Keywords: Heuristics. Linear programming. Ore blending. Machine scheduling.

Lista de Tabelas

Lista de Algoritmos

Lista de Abreviaturas e Siglas

CVRD	Companhia Vale do Rio Doce
DECOM	Departamento de Computação
GRASP	*Greedy Randomized Adaptive Search Procedure*
IBRAM	Instituto Brasileiro de Mineração
JSON	*JavaScript Object Notation*
ILS	*Iterated Local Search*
LAHC	*Late Acceptance Hill-Climbing*
NSGA-II	*Non-dominated Sorting Genetic Algorithm II*
OMP	*Ore Mixing Problem*
PPPVM	Problema de Planejamento de Produção e Vendas de uma Mineradora
RAM	*Random Access Memory*
SA	*Simulated Annealing*
UFOP	Universidade Federal de Ouro Preto
UPMSP	*Unrelated Parallel Machine Schedule Problem*

Sumário

1 Introdução

O faturamento do setor de mineração do Brasil atingiu 50,7 bilhões de reais no terceiro trimestre de 2020, totalizando um faturamento acumulado de 125,95 bilhões de reais ao longo do ano. As produtoras de minério de ferro do Brasil, representaram 62,8% da receita total do setor de mineração do país no segundo trimestre, com 31,86 bilhões de reais (IBRAM, 2020).

De acordo com o Instituto Brasileiro de Mineração (IBRAM), o minério de ferro está entre os recursos minerais mais valiosos do mundo, suas reservas são estimadas em US$ 6,8 trilhões, sendo a matéria prima para a produção do aço, metal mais consumido no mundo, com 1,6 bilhão de toneladas anuais. O Brasil é o terceiro maior produtor e exportador mundial de minério de ferro, possui o segundo maior estoque de reservas e também as reservas provadas e prováveis de Carajás, no sul do Pará, são consideradas as de melhor qualidade. O minério de ferro é um dos principais produtos de exportação do Brasil, com receita anual média de US$ 25 bilhões nesta década. Atualmente, o Estado de Minas Gerais é responsável por 60% da produção brasileira (IBRAM, 2018).

A indústria mineral brasileira é composta por mais de 10 mil empresas, em sua maioria micro e pequenas empresas, empregando mais de 2 milhões de trabalhadores. Além disso, seus resultados operacionais são essenciais para balança comercial brasileira, entre outros benefícios diretos e indiretos para a economia e a sociedade (IBRAM, 2018).

Em uma breve análise do cenário internacional de exportação de minério de ferro, de acordo com Everett (2007), esse setor é um dos principais contribuintes para o economia australiana e 97% dos US$ 8 bilhões da receita anual é gerada pelos 250 milhões de toneladas de minério exportados. Recentemente, o aumento da demanda da China criou um grande aumento na produção de minério de ferro exigida da Austrália Ocidental, o qual levou à expansão das minas existentes e ao planejamento de novas minas.

O minério de ferro é extraído de minas a céu aberto, que estão distantes por vários quilômetros do porto, esse minério é britado, empilhado em estoques, recuperado e carregado em transportadores de minério para envio aos clientes, que utilizam o minério com propósito de alimentar os altos-fornos para siderurgia. Os alto-fornos são ajustados, de modo que o minério entregue deve estar consistentemente próximo da meta de qualidade, não apenas em ferro, mas também em contaminantes como sílica, alumina e fósforo. Normalmente, essa composição é da ordem de 60% de ferro, 4% de sílica, 2% de alumina e 0,1% de fósforo. A consistência do grau de minério entregue é o principal critério de qualidade, monitorados pelos clientes e um fator importante na negociações de preço e quantidade (EVERETT, 2007).

Neste contexto, é importante que tecnologias que permitam aperfeiçoar a esquematização do sistema de transporte, armazenamento e venda de minérios sejam desenvolvidas. As grandes

empresas mineradoras possuem um complexo sistema de transporte e estocagem de minérios com diferentes características, envolvendo terminais ferroviários, rodoviários e portuários. Como consequência, se faz necessário tomar decisões relacionadas à produção e ao transporte dos minérios com bastante frequência.

Periodicamente, as mineradoras devem tomar decisões relacionadas à produção e ao transporte de minérios, tomando como base suas capacidades produtivas e demandas dos mercados interno e externo. Como as capacidades produtivas e as demandas do mercado variam dependendo do cenário da mineradora, é preciso decidir periodicamente quais minérios precisaram ser empilhados e retomados, bem como a quantidade e a forma como se dará o transporte. Tendo em vista que cada minério possui características físicas e químicas diferentes, o planejamento envolve também o atendimento às demandas de massa e qualidade sujeitas às restrições impostas pelo cliente.

Por estes motivos, modelos de otimização capazes de encontrar soluções de boa qualidade para os Problemas de Mistura de Minérios e Planejamento de Transporte e Sequenciamento da Produção são muito desejados pelas empresas mineradoras. O método mais utilizado para determinar a proporção de minério proveniente de cada frente é a Programação Linear, através da resolução do problema da mistura, ou blendagem. Outra abordagem eficiente é a Programação Linear por Metas para resolver o problema da mistura com o atendimento de metas de qualidade e produção. Em casos reais da indústria mineral existe uma série de outras restrições que devem ser consideradas e que normalmente não são abordadas em um mesmo modelo na literatura (COSTA, 2005).

Otimizar o problema de planejamento operacional de uma mina a céu aberto em partes independentes pode gerar conflitos que inviabilizam a implementação das soluções obtidas em cada parte, posto que é necessário resolver o problema da mistura de minério levando em consideração as restrições relacionadas à realidade operacional da mina (PINTO; BIAJOLI; MINE, 2003). Dentre essas restrições, destacamos a relação das características do minério a serem obedecidas e a alocação e disponibilidade de equipamentos de carga e transporte de material.

1.1 Justificativa

O armazenamento de minério em pátios de estocagem de uma indústria mineradora é feito na forma de pilhas compostas por produtos de diferente especificações, tanto químicas, quanto físicas. Para atender as exigências dos clientes, é necessário observar vários aspectos do ponto de vista da qualidade do minério, bem como do ponto de vista operacional do pátio.

Em relação à qualidade do minério é preciso que as metas e os limites mínimos e máximos aceitáveis para cada parâmetro de qualidade para formação do produto final sejam respeitados. Já do ponto de vista operacional, as máquinas retomadoras e empilhadeiras, que são responsáveis por realizar o transporte de minérios, possuem certas limitações, como a possibilidade de realizar

apenas uma atividade (retomar ou empilhar) ou não conseguir acessar determinado conjunto de pilhas. Existe também limitação de tempo para a montagem de um pedido em razão do tempo de preparação e deslocamento das máquinas.

Desta forma, o presente trabalho se justifica, uma vez que permitirá obter uma solução de baixo custo operacional e melhor aproveitamento de recursos que atenda as exigências do produto para uma dada configuração de pátio, além de acelerar o processo de decisão. É importante observar que muitas mineradoras possuem um pátio de estocagem com problemas bastante semelhantes. Assim, a solução adotada pode ser utilizada e adaptada para resolver problemas similares em outras empresas mineradoras.

1.2 Objetivos

O objetivo geral deste projeto é a aplicação de métodos de otimização para abordar os Problemas de Mistura de Minérios e Alocação de Máquinas em uma indústria mineradora. Para atingir este objetivo, os seguintes objetivos específicos foram elencados:

- Estudar o problema de Mistura de Minérios e Alocação de Máquinas em uma empresa mineradora;
- Avaliar diversos modelos de otimização propostos na literatura, reunindo-os em modelos mais amplos, de forma a contemplar os requisitos típicos de uma empresa mineradora;
- Modificar, adaptar e aperfeiçoar algoritmos para os problemas abordados;
- Elaborar um modelo matemático de otimização para o Problema de Mistura de Minérios de uma empresa mineradora, buscando metas de qualidade e quantidade;
- Elaborar uma heurística construtiva para o Problema de Alocação de Máquinas;
- Aplicar as meta-heurísticas *Simulated Annealing* e *Late Acceptance Hill-Climbing* na resolução do problema;
- Desenvolver uma metodologia heurística híbrida para o problema abordado;
- Avaliar e testar os algoritmos desenvolvidos;
- Contribuir com a divulgação de técnicas de otimização mono- e multi-objetivo aplicadas à resolução do problema;
- Contribuir para a consolidação do grupo de pesquisa GOAL (Grupo de Otimização e Algoritmos) da Universidade Federal de Ouro Preto.

2 Revisão Bibliográfica

O Problema de Mistura ou Blendagem de Minérios é um problema de otimização abordado por diversos autores e normalmente está atrelado a outros fatores como Planejamento de Transporte e Sequenciamento da Produção. Todos esses elementos devem ser considerados no modelo de otimização, pois estão intimamente conectados e ignorar parte desse problema pode inviabilizar o modelo, quando aplicado à realidade de uma empresa mineradora.

Toffolo (2009) descreveu o tema como sendo um problema de otimização que consiste em determinar a quantidade de cada minério, proveniente de um conjunto de pilhas, que deve ser misturada para formar um produto final com características que atendam às exigências de um determinado cliente. Como os minérios possuem características diferentes, seja o teor de certo elemento químico ou percentual de um minério em certa faixa granulométrica, é necessário combinar os minérios em determinadas proporções para que a mistura cumpra as metas de qualidade e quantidade.

A composição desse minério geralmente é especificada, não apenas para o ferro, mas também para alumínio, sílica, fósforo e cálcio. Cada decisão tomada no processo produtivo não pode ser feita sem considerar seus efeitos nas decisões futuras, isso ocorre parcialmente porque os objetivos otimização de curto prazo podem se esbarrar ou bloquear os objetivos da otimização de longo prazo (EVERETT, 2001).

Como mencionado anteriormente, o Problema de Mistura de Minério está atrelado ao Problema de Transporte e Sequenciamento da Produção e para armazenar e retirar o material que chega aos pátios é utilizado um maquinário pesado e com alto custo operacional. Esse equipamento é chamado de Empilhadeira/Retomadora, por ser capaz de realizar ambas tarefas, e tratando-se de um mecanismo que se move por trilhos, existe pouca maleabilidade em sua trajetória.

Algumas dessas retomadoras, são controladas de forma remota ou automática para seguir um simples padrão de trajetória, sem qualquer flexibilidade em tempo real caso haja necessidade de se alterar esse trajeto. Nas operações automatizadas, o algoritmo utilizado pelas retomadoras, opera em modo reativo, ou seja, ele é baseado em uma trajetória totalmente fixa sem se importar se o material está de fato sendo retirado da pilha ou não. Por conta disso, algumas das retomadoras recolhem nada além de ar por longos períodos de tempo. Sem o conhecimento adequado de cada material das pilhas antes de seu recolhimento e sem o planejamento ótimo dos caminhos do retomador, uma solução de grau muito pobre pode acabar resultando em altos custos operacionais (LU; MYO, 2011).

Observa-se que o problema de se determinar o ritmo do transporte em um pátio de estocagem de mineração se reduz ao Problema da Mochila Inteira Múltipla com restrições

adicionais. De fato, cada equipamento de carga pode ser considerado uma mochila i de capacidade máxima b_i. Cada frente de lavra pode ser considerada como um objeto j para o qual estão disponíveis u_j unidades. O problema consiste em determinar quantas unidades x_{ij} de cada objeto j alocar à mochila i de forma a maximizar o benefício pelo uso das pilhas de minério (dado pelo atendimento às metas de produção e qualidade), satisfazendo à condição de que cada mochila tenha sua capacidade respeitada e que cada objeto esteja em uma única mochila (COSTA, 2005).

2.1 Trabalhos Relacionados

Nesta seção são apresentadas as principais técnicas e trabalhos sobre a temática desenvolvida, apresentando os conceitos mais importantes, justificativas e características sobre o tema, do ponto de vista da análise feita pelos autores.

2.1.1 Mistura de Minérios

A abordagem mais comum na literatura para o problema em questão é a combinação entre programação linear e métodos heurísticos, pois apesar dos modelos matemáticos retornarem bons resultados, eles se tornam inviáveis conforme o horizonte de planejamento expande.

Moraes et al. (2006) apresentaram um modelo de programação linear por metas para um problema de mistura de minérios no planejamento de curto prazo em uma mina de carvão. Também ressaltaram que a programação linear clássica está sendo vastamente utilizada na modelagem de problemas de produção/blendagem na indústria mineral, porém, a sua formulação é limitada pelo fato de que somente uma função objetivo pode ser utilizada por modelo, quando na realidade o problema de mistura de minérios é multiobjetivo. Dessa forma, esses problemas podem ser mais apropriadamente modelados utilizando-se múltiplos objetivos em vez de restrições rígidas. A solução desses modelos não se resume a maximizar ou minimizar uma função objetivo dentro de um conjunto de restrições, mas envolve a satisfação de uma condição mínima aceitável. Esses são os denominados problemas de programação por metas, ou da literatura inglesa, *Goal Programming*.

Em seu texto, Moraes et al. (2006) focou seu trabalho no Complexo de Itabira, mais especificamente na Mina de Cauê. Após apresentar os detalhes do problema abordado, foi elaborado um modelo de programação matemática para o mesmo. Por fim foram apresentados, discutidos e analisados os resultados obtidos pela aplicação do modelo proposto a um conjunto de diferentes cenários no pátio de estocagem da Mina de Cauê da CVRD, em Itabira (MG). Os resultados computacionais obtidos mostraram que o sistema desenvolvido foi capaz de gerar soluções de qualidade superiores às produzidas pelo método manual. Esses resultados validam, portanto, a utilização dessa ferramenta para a resolução do problema de blendagem dos produtos dos pátios da CVRD, demonstrando que é possível desenvolver metodologias que substituem, com ganho de produtividade, os tradicionais métodos de tentativa e erro.

Alves (2007) abordou o problema utilizando um exemplo muito conhecido de modelo de planejamento da produção, que é o modelo de dimensionamento de lotes ou *lot sizing*. Tal modelo normalmente apresenta algumas características básicas, como o horizonte de planejamento finito e dividido em períodos e a demanda dinâmica de cada item, isto é, variando ao longo do horizonte de planejamento. Em conclusão o autor apresentou a interface do sistema computacional desenvolvido para o Problema de Planejamento de Produção e Vendas de uma Mineradora (PPPVM), que nada mais é que o problema de mistura adicionado das restrições relativas ao planejamento da produção e vendas de minérios. Em seguida são exibidos os resultados obtidos pela aplicação do sistema desenvolvido. Em relação aos resultados, o sistema desenvolvido pelo autor mostrou-se eficiente na resolução do PPPVM. No estudo de caso analisado, o sistema encontrou a solução ótima para o problema rapidamente, atendendo a todas as demandas dos produtos finais e apresentando baixos desvios em relação aos parâmetros de controle.

Para o problema em questão Toffolo (2009) apresentou um modelo de programação inteira multiobjetivo, bem como metodologias heurísticas *relax-and-fix*, GRASP e ILS. Para cada um dos métodos apresentados foi feita uma discussão sobre sua aplicação ao problema. Por se tratar de um problema combinatório, o uso de métodos exatos se torna um pouco restrito, já que não conseguem encontrar uma solução boa em tempo hábil. Devido a esse limitador foi utilizada uma abordagem heurística com intuito de encontrar soluções viáveis de forma mais ágil.

A heurística *relax-and-fix* é uma abordagem de solução baseada em métodos exatos, que tem sido utilizada na solução de diversos tipos de problemas de forma pura ou híbrida. O GRASP é um método iterativo e guloso, que consiste de duas fases: uma fase de construção, na qual uma solução inicial é gerada e uma fase de busca local, na qual um ótimo local na vizinhança da solução construída é pesquisado, por fim a melhor solução pesquisa é retornada. O método ILS é baseado na ideia de que um procedimento de busca local pode ser melhorado gerando-se novas soluções de partida, as quais são obtidas por meio de perturbações na solução ótima local.

Após a descrição dos métodos utilizados, Toffolo (2009) apresentou os cenários de teste aos quais os algoritmos de otimização foram aplicados, exibindo e analisando os resultados obtidos. Para testar as metodologias desenvolvidas, foram gerados cenários de teste baseados na realidade de uma grande empresa situada no Quadrilátero Ferrífero, em Minas Gerais. Esses cenários deram origem a diversas instâncias, que consideram os horizontes de planejamento anual, trimestral, mensal e diário. O modelo matemático multiobjetivo se mostrou eficiente apenas para as instâncias menores, sendo necessário a utilização dos algoritmos heurísticos, que em relação aos tempos computacionais, demandaram pouco tempo para gerar boas soluções, mostrando-se adequados para serem utilizados no desenvolvimento de ferramentas de apoio à decisão.

2.1.2 Transporte e Sequenciamento da Produção

De acordo com Lu e Myo (2011), uma pilha de estocagem é retomada com aproximadamente cerca de 50% do seu potencial produtivo, de acordo com uma pesquisa realizada, em sua

maioria, com minas de minérios de ferro na Austrália Ocidental. Parte significante da receita é resultado da pequena taxa produtiva dos retomadores, pois em sua grande maioria são operadas manualmente pelos pátios de estocagem em todo mundo.

Júnior (2011) propôs a elaboração de métodos para a otimização do problema de planejamento operacional de lavra em minas, considerando a alocação dinâmica de caminhões. Este trabalho consiste em determinar a melhor maneira para se organizar cada frente de lavra, ou seja, o número de viagens para cada tipo de caminhão, a quantidade de caminhões utilizados, porcentagem e granulometria de cada tipo de minério e a quantidade de minério. Como essas especificações variam de acordo com cada cliente, tem-se como objetivo determinar o ritmo de produção na mina de forma a minimizar os desvios da produção e os desvios das metas de qualidade do minério, assim como minimizar o número de caminhões envolvidos no processo. O autor fez uma comparação entre a otimização mono-objetivo e a multiobjetivo para a resolução do problema em questão, na qual a otimização multiobjetivo se diverge da mono-objetivo pelo fato de admitir um conjunto de soluções chamadas de pareto-ótimas, que são consideradas equivalentes.

Por definição, um conjunto de soluções é pareto-ótimo se não existe um outro conjunto viável que possa melhorar algum objetivo, sem causar piora em pelo menos um outro objetivo. Já na otimização mono-objetivo uma única solução é claramente identificada.

No trabalho de Júnior (2011) elaborou-se uma formulação matemática para o problema, um algoritmo genético multiobjetivo e um modelo de simulação computacional capaz de validar os resultados e determinar o sequenciamento de caminhões. Por último o autor apresentou os resultados computacionais obtidos com a aplicação da formulação de programação matemática proposta, do método heurístico e do modelo de simulação computacional. Por se tratar de um problema NP-difícil, para problemas de dimensões maiores o método exato apresentou soluções de baixa qualidade, e com intuito de resolver essa situação foi elaborado um algoritmo genético, baseado no NSGA-II, que foi capaz de apresentar soluções de qualidade em tempo hábil. Após a análise dos resultados foram apresentadas as conclusões sobre o trabalho desenvolvido e recomendações para trabalhos futuros.

Como abordado por Everett (2001), a produção pode ser organizada começando pelo número de minas em operação, já que pode haver mais de uma mina em funcionamento, que nesse caso uma sequência de trens carregados de diferentes minas pode ser esquematizada para despachar os minérios até o porto. Quando os trens chegam ao porto, eles são descarregados e o material é armazenado no estoque. É possível que haja mais de um pátio de estocagem, mas cada um é abastecido por completo antes de se tornarem aptos para carregar um navio. Finalmente, cada navio é carregado com o material recuperado de cada uma das pilhas de estocagem. Para cada um dos três estágios do problema em questão foram desenvolvidas soluções heurísticas usando modelos de simulação, baseado em dados de vários anos de produção de duas minas diferentes. A construção dos modelos foi realizada utilizando a linguagem de simulação Extend.

Vianen, Ottjes e Lodewijks (2015) formularam um modelo de programação inteira mista

para o problema de agendamento com o objetivo de minimizar o *makespan*, tempo total entre o começo da primeira operação e o final da última operação, para um determinado conjunto de operações de manipulação. A abordagem desenvolvida foi baseada em algoritmos genéticos usando dois tipos de representação de cromossomos. No procedimento de atribuição guloso, as operações são designadas para máquinas baseadas em sua disponibilidade, com o tempo de conclusão e de configuração minimizados.

O problema de agendamento empilhador-retomador tem semelhanças com o problema de agendamento da oficina. Uma operação de empilhamento ou recuperação pode então ser definida como um trabalho. Vianen, Ottjes e Lodewijks (2015) observaram também que o problema padrão de agendamento de oficina é estendido, levando em consideração as quebras da máquina e a chegada de novos trabalhos prioritários. O modelo de simulação apresentado pode ser usado para apoiar as decisões sobre a reprogramação da operação de empilhadeiras e recuperadoras para diminuir o tempo médio de porto do trem e ainda garantir o desempenho à beira-mar. Os autores concluíram que com o reagendamento da operação do empilhador-retomador o tempo médio de porto do trem será reduzido.

2.2 Fundamentação Teórica

Nesta seção é apresentada a fundamentação teórica na qual este trabalho se apoia, descrevendo brevemente o funcionamento e a aplicação da Programação Linear em problemas de otimização; também é feita uma breve introdução sobre meta-heurísticas, em mais detalhes os métodos *Late Acceptance Hill-Climbing* e *Simulated Annealing* e o porquê de serem boas opções para problemas de otimização, nas seções 2.2.6 e 2.2.7, respectivamente.

2.2.1 Programação Linear

A programação linear tem como objetivo encontrar a melhor solução para problemas que possam ser representados por expressões lineares, buscando a maximização ou a minimização de uma Função Objetivo, respeitando-se um sistema de restrições. As restrições representam as limitações de recursos disponíveis ou exigências e condições que devem ser cumpridas no problema. São essas restrições que determinam quais soluções são viáveis para o problema. O objetivo da programação linear se resume na determinação de uma solução ótima, a melhor solução dentre as soluções viáveis para um determinado problema (ALVES, 2007).

Entretanto, existem diversas situações no mundo real que exigem que as decisões sejam mais flexíveis. Dessa forma, procura-se satisfazer ou aproximar-se dos objetivos estabelecidos, ao invés de considerá-los como metas rígidas. Esses problemas podem ser modelados utilizando múltiplos objetivos e sua solução não se trata de apenas maximizar ou minimizar uma função objetivo, mas envolve a satisfação de uma condição mínima aceitável (BUENO; OLIVEIRA, 2004).

A programação por metas é uma técnica que permite a modelagem e a busca de soluções para os problemas multiobjetivo. De acordo com Moraes et al. (2006), a principal diferença entre a programação por metas e a programação linear clássica é que a segunda requer a busca pela maximização ou minimização de uma função objetivo, enquanto a primeira, por sua vez, busca-se a minimização dos desvios no alcance das metas.

2.2.2 Programação por Metas

A programação por metas, também conhecida por *Goal Programming*, é uma técnica de pesquisa operacional que permite a modelagem e a busca de soluções para problemas com múltiplos objetivos ou metas a serem atingidas, situação nas quais é necessário satisfazer ou aproximar-se das metas estabelecidas. Costa (2005) caracterizou a programação por metas como sendo uma extensão da programação linear, desenvolvida de modo a permitir a solução simultânea de um sistema com múltiplas metas, podendo apresentar unidades de medidas diferentes.

Na programação por metas a função de avaliação é definida como a minimização dos desvios em relação às metas. Tem-se como exemplo a função de avaliação Arquimediana, na qual cada meta possui uma importância diferente na otimização e são hierarquizadas através de pesos, priorizando-se as principais metas com pesos maiores. (ROMERO, 2004)

Dessa forma, as variáveis de decisão podem assumir valores que otimizem, de modo geral, o problema, sendo possível avaliar o quanto uma restrição fica distante da meta estabelecida, considerando que na programação por metas as restrições deixam de ser rígidas através da inclusão de variáveis de desvio, ampliando o espaço de soluções viáveis.

2.2.3 Métodos Construtivos

Um método construtivo, de acordo com Costa (2005), tem por objetivo gerar uma solução, elemento por elemento. A solução gerada pelo procedimento construtivo pode não ser muito boa, fazendo necessário um refinamento da solução após ser construída.

Um método construtivo guloso, é quando a cada iteração um elemento que ainda não foi selecionado é inserido, sendo este o que traz o maior benefício para a solução construída. Este processo, apesar de normalmente gerar soluções com uma boa qualidade, tem o inconveniente de produzir soluções sem diversidade e por isso se faz necessário um maior tempo computacional para o refinamento.

Outra maneira de gerar uma solução inicial é construí-la de maneira aleatória. Portanto, a cada passo um elemento que ainda não foi selecionado é inserido aleatoriamente no conjunto solução. A grande vantagem dessa metodologia está na simplicidade de implementação e na grande diversidade de soluções. Entretanto, a solução produzida é de baixa qualidade e por essa razão requere um esforço maior na fase de refinamento.

2.2.4 Busca Local

Segundo Souza (2008), os métodos de busca local, ou métodos de refinamento, em problemas de otimização são um conjunto técnicas baseadas na noção de vizinhança. A busca local consiste em mover de uma solução inicial para outra dentro de uma estrutura de vizinhança, de acordo com algumas regras bem definidas.

Especificamente, seja S o espaço de pesquisa de um problema de otimização e f a função objetivo a minimizar. A função N, a qual depende da estrutura do problema tratado, associa a cada solução viável $s \in S$, sua vizinhança $N(s) \subseteq S$. Cada solução $s' \in N(s)$ é chamada de vizinho de s. Denomina-se movimento a modificação m que transforma uma solução s em outra, s', que esteja em sua vizinhança. A eficiência da busca local depende da qualidade da solução construída, bem como o método de exploração da vizinhança. Logo, o procedimento de construção tem então um papel importante na busca local, uma vez que as soluções construídas constituem bons pontos de partida, permitindo assim reduzir o esforço computacional (COSTA, 2005).

2.2.5 Meta-Heurísticas

Em alguns problemas combinatórios, o uso de métodos exatos pode se tornar bastante restrito, isso acontece normalmente quando o problema possui um grande volume de dados para serem processados, fazendo com que os métodos exatos não consigam encontrar uma solução boa em tempo aceitável. Por esse motivo, é muito comum encontrar abordagens heurísticas e meta-heurísticas para solucionar problemas deste nível de complexidade. As meta-heurísticas são métodos para encontrar boas soluções para um determinado problema combinatório e consistem na aplicação de uma heurística subordinada (busca local) em cada uma de suas iterações (RIBEIRO, 1996).

2.2.6 Late Acceptance Hill-Climbing

O *Late Acceptance Hill-Climbing* (LAHC) é uma meta-heurística introduzida por Burke e Bykov (2008), sendo uma adaptação da heurística clássica *Hill-Climbing*, que consiste em analisar um vizinho aleatório de uma vizinhança aleatória e aceitar somente aquele que representar uma melhora no valor atual da função objetivo. A diferença para a meta-heurística é que uma solução candidata pode ser aceita mesmo que seja pior do que a solução atual, uma vez que ela é comparada com a solução obtida há l iterações atrás, em que l é o número de soluções mais recentes consideradas.

De acordo com Santos et al. (2019), este método foi criado para ser um procedimento de pesquisa que dispensa o uso de um fator de resfriamento artificial como o *Simulated Annealing* e que usa um mecanismo de aceitação quase tão simples como o do *Hill-Climbing* (BURKE; BYKOV, 2008).

O Algoritmo 1 mostra o pseudocódigo do *Late Acceptance Hill-Climbing*, onde a lista $\{0, \ldots, l-1\}$ armazena custo das soluções e é inicializada com o custo da solução inicial S. A cada iteração um novo vizinho é gerada e é aceita somente se melhora a solução atual ou se seu custo for menor ou igual ao custo armazenado na posição v da lista. Caso a solução encontrada melhorar em relação à melhor solução encontrada, então ela é atualizada. Em seguida o custo na posição v é atualizado e o procedimento se repete até atingir o critério de parada.

Os valores escolhidos para os parâmetros do *Late Acceptance Hill-Climbing* foram determinados com base nas observações realizadas durante a experimentação prática, sendo $l = 1000$ e o critério de parada como 10^3 iterações sem melhora. A solução vizinha é gerada por meio dos movimentos de troca de uma ou duas tarefas entre duas máquinas diferentes e alteração da ordem de uma ou duas tarefa em uma ou duas máquinas diferentes, todos os movimentos possuem suas versões aleatórias e gulosas e são escolhidos aleatoriamente pela meta-heurística.

Algoritmo 1: Late Acceptance Hill-Climbing

Entrada: solução inicial S e lista de tamanho l

Late Acceptance Hill-Climbing
- $f_v \leftarrow f(S)\ \forall v \in \{0, \ldots, l-1\}$
- $S^* \leftarrow S$
- $v \leftarrow 0$
- **enquanto** *critério de parada não for atingido* **faça**
  - $S' \leftarrow$ vizinho aleatório de uma vizinhança aleatória k, $S' \in N_k(S)$
  - **se** $f(S') \leq f(S^*)$ *ou* $f(S') \leq f_v$ **então**
    - $S \leftarrow S'$
    - **se** $f(S') < f(S^*)$ **então** $S^* \leftarrow S'$
  - $f_v \leftarrow f(S)$
  - $v \leftarrow (v+1) \mod l$
- **retorna** S^*

2.2.7 Simulated Annealing

Proposta por Kirkpatrick, Gelatt e Vecchi (1983), a meta-heurística *Simulated Annealing* (SA) é uma técnica de busca local, que faz uma analogia à termodinâmica ao simular o resfriamento de um conjunto de átomos aquecidos (REEVES, 1993). Este método tem como procedimento principal realizar uma busca partir de uma solução inicial, gerando vizinhos aleatórios a cada iteração do algoritmo. Resumidamente, a cada vizinho gerado o método avalia se houve melhora em relação à solução inicial, caso o vizinho gerado seja melhor a solução é aceita, caso contrário essa solução pode ser aceita com uma probabilidade $e^{-\Delta/T}$, onde T é um parâmetro que indica a probabilidade de se aceitar soluções piores.

Em analogia à termodinâmica, o parâmetro T é chamado de temperatura e possui inicialmente um valor elevado que vai diminuindo, ou resfriando, a cada iteração do método. Após

um número predefinido de iterações, a temperatura é gradativamente diminuída por uma razão de resfriamento α, sendo $0 < \alpha < 1$. Dessa maneira, ao iniciar o algoritmo tem-se uma maior chance de evitar ótimos locais, e conforme ocorre o resfriamento o método se comporta como um método de descida, já que diminui a chance de se aceitar soluções piores (SOUZA, 2008).

O Algoritmo 2 mostra o pseudocódigo do *Simulated Annealing*, onde $f(\cdot)$ é a função objetivo e $N_k(\cdot)$ é a função que retorna uma solução vizinha aleatória da vizinhança k. O algoritmo requer os seguintes argumentos: solução inicial S; temperatura inicial t_0; razão de resfriamento α; número de iterações processadas em cada temperatura sa_{max}.

Os valores escolhidos para os parâmetros do *Simulated Annealing* foram determinados com base nas observações realizadas durante a experimentação prática, sendo $\alpha = 0.9$, $t_0 = 1.0$, $sa_{max} = 1000$ e o critério de parada como número de iterações sem melhora. A solução vizinha é gerada da mesma maneira que em 2.2.6, por meio dos movimentos de troca de uma ou duas tarefas entre duas máquinas diferentes e alteração da ordem de uma ou duas tarefa em uma ou duas máquinas diferentes, todos os movimentos possuem suas versões aleatórias e gulosas e são escolhidos aleatoriamente pela meta-heurística.

Algoritmo 2: Simulated Annealing

Entrada: S, t_0, α, sa_{max}

Simulated Annealing
- $S^* \leftarrow S$
- $t \leftarrow t_0$
- **enquanto** *critério de parada não for atingido* **faça**
  - **para cada** $i \in \{1, \ldots, sa_{max}\}$ **faça**
    - $S' \leftarrow$ vizinho aleatório de uma vizinhança aleatória k, $S' \in N_k(S)$
    - $\Delta \leftarrow f(S') - f(S)$
    - **se** $\Delta \leq 0$ **então**
      - $S \leftarrow S'$
      - **se** $f(S') < f(S^*)$ **então** $S^* \leftarrow S'$
    - **senão**
      - selecione um valor aleatório de $x \in [0, 1]$
      - **se** $x < e^{-\Delta/t}$ **então** $S \leftarrow S'$
  - $t \leftarrow \alpha \times t$
  - **se** $t < \epsilon$ **então** $t \leftarrow t_0$
- **retorna** S^*

3 Desenvolvimento

O objetivo do presente trabalho é propor um modelo de programação linear combinado a métodos heurísticos visando melhorar o aproveitamento de recursos e consequente melhoria na produtividade do processo de mistura e blendagem de minérios. Para tratar o problema por completo, a formulação matemática é focada em solucionar apenas o problema de mistura dos minérios, ignorando os custos operacionais. Já os algoritmos heurísticos definem as tarefas e a forma como cada equipamento irá operar para atender a demanda baseado nos resultados obtidos pelo modelo matemático.

3.1 Formulação Matemática

O modelo de programação linear proposto tem como objetivo solucionar o problema de mistura e blendagem de minérios. A formulação leva em consideração a qualidade e a quantidade de cada minério, combinando-os de tal forma para suprir a demanda. O modelo indica quanto deve ser retirado ou empilhado em cada área de estocagem de minério para cada pedido, de forma que a massa resultante respeite os limites máximos e mínimos de cada parâmetro de qualidade, focando também em manter esse valor o mais próximo possível da meta estipulada para cada um desses parâmetros.

3.1.1 Dados de Entrada

P : Conjunto de Áreas de Estocagem de Minério;

T : Conjunto de Parâmetros de Qualidade;

R : Conjunto de Pedidos;

E : Conjunto Entradas.

q_{ij}^{ini} : parâmetro de qualidade $j \in T$ inicialmente presente na área $i \in P$;

q_{jk}^{goal} : meta para o parâmetro de qualidade $j \in T$ no pedido $k \in R$;

q_{jk}^{min} : limite mínimo para o parâmetro de qualidade $j \in T$ no pedido $k \in R$;

q_{jk}^{max} : limite máximo para o parâmetro de qualidade $j \in T$ no pedido $k \in R$;

q_{jk}^{imp} : importância do parâmetro de qualidade $j \in T$ no pedido $k \in R$;

w_{i}^{ini} : massa inicial de minério da área $i \in P$;

d_k : demanda de minério do pedido $k \in R$.

C_i : capacidade máxima de minério da área $i \in P$;

W_h : quantidade máxima de minério que pode ser retirada da entrada $h \in E$.

3.1.2 Variáveis

x_{ik} : quantidade de minério retirada da área $i \in P$ para o pedido $k \in R$;

y_{hi} : quantidade de minério retirada da entrada $h \in E$ direcionada para a área $i \in P$;

α^+_{jk} : desvio positivo para o limite máximo do parâmetro de qualidade $j \in T$ do pedido $k \in R$;

α^-_{jk} : desvio negativo para o limite mínimo do parâmetro de qualidade $j \in T$ do pedido $k \in R$;

β^+_{jk} : desvio positivo para a meta do parâmetro de qualidade $j \in T$ do pedido $k \in R$;

β^-_{jk} : desvio negativo para a meta do parâmetro de qualidade $j \in T$ do pedido $k \in R$.

3.1.3 Função Objetivo

A função objetivo visa minimizar os desvios de qualidade em relação às metas de qualidade e aos limites máximos e mínimos de cada parâmetro. Os pesos ω_1 e ω_2 penalizam os desvios calculados em (3.1) com o intuito de manter o desvio em relação aos limites o menor possível, portanto $\omega_1 \gg \omega_2$. Já os pesos ω_{ik} e ω_{hi} penalizam a retomada de minério na pilha i para o pedido k e o empilhamento de minério da entrada h na pilha i e funcionam como *feedback* da abordagem heurística elaborada para o problema de sequenciamento.

$$
\begin{aligned}
&Minimizar \\
&\omega_1 \sum_{k \in R} \sum_{j \in T} \left(\frac{q^{imp}_{jk} \times \alpha^-_{jk}}{q^{goal}_{jk} - q^{min}_{jk}} + \frac{q^{imp}_{jk} \times \alpha^+_{jk}}{q^{max}_{jk} - q^{goal}_{jk}} \right) + \\
&\omega_2 \sum_{k \in R} \sum_{j \in T} \frac{\beta^-_{jk} + \beta^+_{jk}}{\min\left(q^{goal}_{jk} - q^{min}_{jk}, q^{max}_{jk} - q^{goal}_{jk}\right)} \\
&+ \sum_{k \in R} \sum_{i \in P} \omega_{ik} \times x_{ik} + \sum_{i \in P} \sum_{h \in E} \omega_{hi} \times y_{hi}
\end{aligned}
\tag{3.1}
$$

3.1.4 Restrições

As restrições em um modelo de programação linear expressam as condições limitantes que surgem de condições tecnológicas ou de recursos limitados, sendo possível definir um conjunto dos pontos viáveis no qual a solução ótima se encontra. Para o modelo matemático em questão foram definidas restrições de capacidade, de estoque, de demanda e de qualidade.

Restrições de Capacidade. As restrições de capacidade impedem que os limites de massa de minérios estocados em cada área ou retirados das entradas sejam desrespeitados. Em (3.2) a massa retirada de cada entrada deve ser menor ou igual à massa da entrada. Em (3.3) a massa

inicial da área de estocagem somada à massa da entrada adicionada à ela deve ser menor ou igual a capacidade limite da área.

$$\sum_{i \in P} y_{hi} \leq W_h \qquad \forall h \in E \tag{3.2}$$

$$\sum_{h \in E} y_{hi} + w_i^{ini} \leq C_i \qquad \forall i \in P \tag{3.3}$$

Restrição de Estoque. A restrição de estoque impede que a massa retirada da área seja maior que a massa armazenada. Em (3.4) a massa retirada da área de estocagem para atender a demanda deve ser menor que a massa inicial da área somada à massa da entrada que foi adiciona à ela.

$$\sum_{k \in R} x_{ik} \leq w_i^{ini} + y_{hi} \qquad \forall h \in E, \forall i \in P \tag{3.4}$$

Restrição de Demanda. A restrição de demanda garante que a massa de minério solicitada em cada pedido seja atendida. Em (3.5) o somatório das massas de minérios retiradas de cada área de estocagem deve ser igual à demanda.

$$\sum_{i \in P} x_{ik} = d_k \qquad \forall k \in R \tag{3.5}$$

Restrições de Qualidade. As restrições de qualidade visam penalizar na função objetivo os desvios de qualidade em relação aos limites máximos e mínimos de cada parâmetro e às metas de qualidade. Em (3.6) e em (3.7) é calculado o desvio caso a qualidade do minério retirado para atender a demanda seja inferior a qualidade mínima de cada pedido ou superior a qualidade máxima de cada pedido, respectivamente. Em (3.8) são calculados os desvios positivos e negativos caso a qualidade do minério retirado para atender a demanda seja diferente da meta de qualidade.

$$\sum_{i \in P} x_{ik} \left(q_{ij}^{ini} - q_{jk}^{min} \right) + \alpha_{jk}^{-} d_k \geq 0 \qquad \forall j \in T, \forall k \in R \tag{3.6}$$

$$\sum_{i \in P} x_{ik} \left(q_{ij}^{ini} - q_{jk}^{max} \right) - \alpha_{jk}^{+} d_k \leq 0 \qquad \forall j \in T, \forall k \in R \tag{3.7}$$

$$\sum_{i \in P} x_{ik} \left(q_{ij}^{ini} - q_{jk}^{goal} \right) + \left(\beta_{jk}^{-} - \beta_{jk}^{+} \right) d_k = 0 \qquad \forall j \in T, \forall k \in R \tag{3.8}$$

3.2 Algoritmos Heurísticos

Os métodos heurísticos e meta-heurísticos propostos têm como objetivo solucionar o problema de transporte e sequenciamento da produção. Tais métodos levam em consideração as restrições de cada equipamento disposto no pátio, como a capacidade de empilhar e retomar os minérios, deslocamento pelos trilhos e acesso à determinadas pilhas de estocagem. Dessa maneira, por meio de um heurística construtiva gulosa, uma solução inicial é criada para futuramente ser refinada pelas meta-heurísticas. Tal solução indica quais tarefas devem ser realizadas por cada unidade operacional e em qual ordem, tendo como objetivo a minimização do tempo gasto no atendimento das demandas.

3.2.1 Heurística Construtiva

O Algoritmo 3 consiste em um método construtivo para gerar uma lista de tarefas para cada máquina de acordo com a demanda de cada pedido e com a funcionalidade de cada equipamento. Inicialmente cria-se uma lista com o horário relativo t_i em que cada máquina consegue acessar determinada área de estocagem de minério, tendo em vista o tempo de deslocamento da máquina até a área, o tempo para configurar o equipamento e também o tempo total que foi gasto para realizar as tarefas anteriores. Sendo assim, em cada pedido é calculado uma lista de tarefas para cada equipamento, visando atender a demanda no menor tempo possível.

Algoritmo 3: Método Construtivo

Entrada: lista de minérios retirados $X = \{x_i^k : \dots\}$ e empilhados $Y = \{y_i : \dots\}$
Saída: lista de tarefas $J = \{j_n^k : \dots\}$

Construtivo
  $M \leftarrow \{1, ..., m\}$ `/* Conjunto de Máquinas */`
  $t_n \leftarrow 0, \forall n \in M$
  para cada $n \in M, k \in R$ **faça**
    $a_n^k \leftarrow \emptyset$ `/* Lista de Áreas */`
    enquanto *todas as áreas do pátio não forem visitadas pela máquina* n **faça**
      $f_i \leftarrow$ área i com menor horário de acesso pela máquina n
      se *máquina* n *é capaz de o trabalho* x_i^k *ou* y_i *da área* f_i **então**
        $t_n \leftarrow$ duração do trabalho + horário de acesso da área f_i
        $a_n^k \leftarrow a_n^k \cup f_i$
    $A \leftarrow a_n^k, \forall n \in M$
    enquanto $A \neq \emptyset$ **faça**
      $f_i \leftarrow$ área i com o menor horário de acesso do conjunto A
      $n \leftarrow$ máquina associada à área f_i
      $A \leftarrow A \setminus \{f_i\}$
      se *existe trabalho* x_i^k *ou* y_i *em* f_i **então**
        remove o trabalho realizado da respectiva lista
        $j_n^k \leftarrow j_n^k \cup f_i$
    $J \leftarrow j_n^k, \forall n \in M$
  retorna J

Portanto, para cada equipamento encontra-se a área de estocagem com o menor horário de acesso, ou seja, aquela na qual a máquina conseguiria iniciar sua atividade mais cedo. Ao encontrar essa área é calculado o tempo gasto para o equipamento realizar sua tarefa, então esse tempo é somado ao horário em que a máquina acessou a área e em seguida adicionado à lista. Adiciona-se a área à rota da máquina e o ciclo continua a partir da última área visitada pelo equipamento até que todas as áreas tenham sido acessadas pela máquina em questão.

Após todas as máquinas possuírem suas rotas, em que cada máquina realiza todas as tarefas possíveis para sua configuração e em seu pátio, as atividades são filtradas para que nenhum trabalho seja feito mais de uma vez ou por mais de um equipamento. Esse procedimento consiste

em escolher para cada tarefa a máquina que vai executá-la mais rapidamente, até que todas as tarefas tenham sido realizadas.

Finalmente, após todos os equipamentos estarem associados às suas atividades, não havendo conflito ou repetição, a lista de rotas de cada máquina em cada pedido é retornada para ser refinada, futuramente, pelas meta-heurísticas *Simulated Annealing* e *Late Acceptance Hill-Climbing*.

3.3 Política de Comunicação entre Sub-Problemas

A solução obtida após a execução das meta-heurísticas é utilizada para definir os pesos ω_{ik} e ω_{hi}, utilizado pela Função Objetivo (3.1). Esse pesos funcionam como *feedback* da abordagem heurística para o modelo linear, penalizando a retomada de minério na pilha i para o pedido k e o empilhamento de minério da entrada h na pilha i. Dessa forma, o modelo é executado novamente, com intuito de obter novos resultados, que são mais uma vez processados pelo método construtivo e pelas meta-heurísticas, este ciclo permanece até atingir o tempo limite.

3.4 Abordagem Inversa

Outra abordagem para o tema deste trabalho é atuar primeiramente com os algoritmos heurísticos no problema de transporte e sequenciamento da produção. Para isso basta fazer pequenas alterações no Algoritmo 3, onde a lista de tarefas de cada máquina é definida independente do modelo matemático. Dessa forma, essa lista de tarefas é criada priorizando o menor custo operacional das máquinas entre as pilhas de minério, de maneira que o pedido seja atendido em relação a massa, mas ignorando os parâmetros de qualidade.

As pilhas de minérios que não forem selecionadas dessa forma serão penalizadas no modelo matemático ao atribuir para ω_{ik} e ω_{hi} um valor elevado, indicando que as pilhas que foram escolhidas pelo método construtivo devem ser priorizadas na formação de um produto final que respeite os limites máximos e mínimos de cada parâmetro de qualidade.

Após a resolução do modelo, o algoritmo se comporta da mesma forma que a abordagem inicial. Porém, esta técnica possibilita que resultados diferentes sejam gerados, devido às novas restrições impostas ao modelo matemático.

4 Experimentos Computacionais

Os experimentos computacionais foram realizados para um conjunto de 3 blocos com 10 instâncias cada, geradas de maneira pseudo-aleatória, considerando diferentes cenários para disposição do maquinário, composição das pilhas de minério e dos pedido realizados. Os dados foram construídos espelhando o funcionamento real de um pátio de minério, variando os valores de mistura, quantidade e disposição das máquinas por um fator previamente determinado de 20 à 40%, selecionado aleatoriamente, de maneira que o conjunto gerado mantenha a similaridade dos dados entre si, porém prevenindo instâncias repetidas. Não houve possibilidade de comparação com resultados reais, uma vez que essas esses dados não são disponibilizados pelas empresas mineradoras.

Em todas as instâncias utilizadas foram considerados 4 parâmetros químicos de controle – Fe, SiO_2, Al_2O_3, P – e 2 físicos (granulometria) – $+31.5$, -6.3 –, cada parâmetro de qualidade é descrito com suas quantidades máximas, mínimas e desejadas. São considerados ainda um conjunto de 2, 4 e 8 unidades operacionais e um conjunto 8, 16 e 32 pilhas de estocagem, cada conjunto dividido em seu respectivo bloco *small*, *medium* e *big*. Ao todo, foram geradas 10 situações de teste para cada bloco. As instâncias estão estruturadas no formato JSON (*JavaScript Object Notation*), que é uma formatação leve de troca de dados, fácil de ler e escrever.

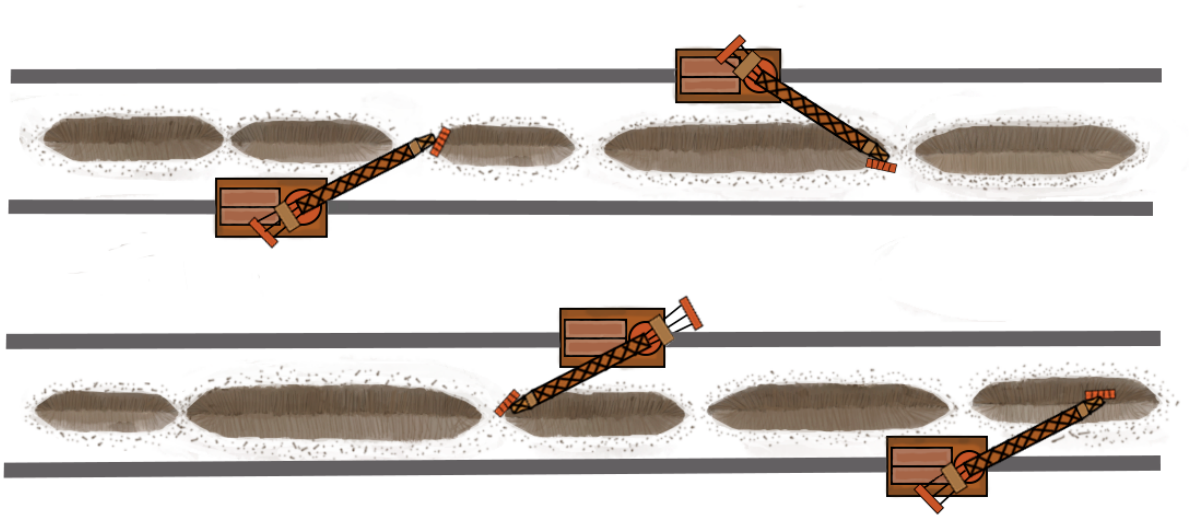

Figura 4.1 – Esquematização do funcionamento de um pátio de minério.

A Figura 4.1 ilustra de maneira simplificada o comportamento das instâncias, onde para cada trilho existe um maquinário associado, capaz de acessar um determinado conjunto de pilhas. Cada pilha de minério possui uma certa quantidade de minério armazenada e também

uma capacidade máxima de minério permitida, o minério armazenado em cada pilha possui composições diferentes que precisam ser combinadas para montar o produto final. Além disso, os maquinários possuem restrições operacionais, alguns são capazes são capazes de realizar as atividades de retomar ou de empilhar (ou ambas) minérios. Além disso, não é possível se movimentar por uma pilha na qual outro maquinário já esteja operando por risco de colisão. Sendo assim, as instâncias geradas simulam todas as restrições associadas ao problema em questão, contendo as composições e quantidades armazenadas em cada pilha, o trilho em que cada maquinário está acoplado e a quais pilhas possuem acesso, a velocidade deslocamento e capacidade operacional de cada equipamento, a quantidade e qualidade do produto final e também a quantidade e qualidade dos minérios que chegam para abastecer o pátio.

Em outras palavras nestas instâncias estão indicadas as metas e limites para cada parâmetro de qualidade na composição do produto final, a posição de cada pilha de minério e suas características (massa inicial, capacidade, qualidade do minérios estocados e trilho de acesso), as características principais de cada unidade operacional (capacidade de empilhamento e retomada, trilho de locomoção e posição inicial), tempo de deslocamento e distância entre cada pilha de minério.

Todos os testes foram executados em um computador Intel Core i7-8750H 2.2GHz com 16GB de memória RAM e sistema operacional Microsoft Windows 10 Home Single Language. Os algoritmos foram desenvolvidos utilizando a linguagem de programação Python 3.8.5 e as bibliotecas NumPy, UltraJSON e PythonMIP. O pacote otimizador Gurobi v9.0 foi utilizado para resolver os modelos de programação linear.

O algoritmo foi executado 10 vezes para cada instância, as instâncias do mesmo bloco se diferem apenas pelo posicionamento inicial de cada equipamento e também como quantidade total de minério é distribuída entre as pilha. Sendo assim os resultados da Tabela 4.1 foram agrupados por blocos, já que por se tratar de uma abordagem exata os valores obtidos para a mistura se mantém constantes em cada instância do bloco. A Tabela 4.1 está disposta da seguinte maneira: a primeira coluna é referente ao parâmetro de qualidade avaliado em cada bloco de instância (*small*, *medium* e *big*) e as demais colunas contém os indicadores de qualidade; z indica a média do teor de minério obtido pela função objetivo analisada para conjunto de instâncias; $goal$ indica o teor de minério da meta de qualidade; min e max os valores mínimos e máximos aceitáveis para cada parâmetro. Pode-se observar que os resultados estão bem próximos da meta estipulada. Vale lembrar que os resultados dispostos na Tabela 4.1 são ótimos, já que o problema da mistura foi resolvido por programação linear, ou seja, o modelo matemático garante a otimalidade quando finalizada a execução.

Parâmetros	small				medium				big			
	min	z	max	$goal$	min	z	max	$goal$	min	z	max	$goal$
Fe	57	**57.0**	100	57	57	65.0	100	57	57	57.3	100	57
SiO_2	0	5.34	5.8	5.8	0	4.5	5.8	5.8	0	4.77	5.8	5.8
Al_2O_3	0	4.61	4.9	4.9	0	4.75	4.9	4.9	0	4.67	4.9	4.9
P	0	0.05	0.07	0.07	0	**0.07**	0.07	0.07	0	0.05	0.07	0.07
$+31.5$	0	7.8	10	10	0	5.0	10	10	0	6.0	10	10
-6.3	0	19.83	25	25	0	17.5	25	25	0	18.33	25	25

Tabela 4.1 – Resultados para mistura de minérios utilizando programação linear.

Seguindo para o problema de sequenciamento, foram utilizadas cinco abordagens heurísticas para tratar o problema: uma em que o problema de mistura é resolvido inicialmente e em seguida o problema de sequenciamento da produção (OMP-UPMSP) utilizando apenas o método construtivo descrito em 3.2.1; outra em que o problema de sequenciamento é resolvido primeiro e então o problema de mistura de minérios (INV) como descrito em 3.4, também utilizando apenas o método construtivo para o sequenciamento; as demais abordagens foram aplicadas na tentativa de refinar a solução inicial de OMP-UPMSP, por meio das meta-heurísticas LAHC (2.2.6) e SA (2.2.7). Além disso, tem-se uma última abordagem na qual foi aplicada a política de *feedback* (FB), descrita em 3.3. Para a abordagem INV, a sequência em que as máquinas vão operar é definida inicialmente ignorando os parâmetros de qualidade da demanda e priorizando o menor custo para retomar a massa de minério solicitada, só então o modelo define quanto deve ser retomado de cada pilha de minério, a fim de respeitar os limites impostos. Para as demais abordagens, a ordem de operação das máquinas é definida com base nos resultados retornados pelo modelo matemático para o problema de mistura, que indica quanto deve ser retomado e/ou empilhado em cada pilha de minério.

Vale lembrar que na abordagem OMP-UPMSP e as demais que refinam seus resultados, como mencionado acima, gera-se inicialmente uma solução do problema da mistura para só então resolver os problema de sequenciamento. Sendo assim, o problema de sequenciamento é resolvido a fim de organizar e programar o maquinário de tal forma que seja viável montar os pedidos com as composições especificadas na Tabela 4.1 com o menor custo possível.

Os valores dispostos na Tabela 4.2, indicam quantas unidades de tempo, *makespan* (ms), foram necessárias para atender a demanda em cada uma das abordagens, também mostra o gap que é a razão entre a solução encontrada e o limite inferior, e por fim o tempo de execução $time$ em minutos de cada abordagem. O limite inferior é encontrado algebricamente considerando um pátio de minério com as configurações ideais, no qual a demanda pode ser atendida apenas realizando a atividade de retomada das unidades operacionais, sem necessidade de locomoção e na qual todas as unidades retomam a mesma quantidade de minério. Sendo assim, é impossível encontrar uma solução cujo valor seja menor que o limite inferior. A primeira coluna é referente ao identificador da instância utilizada.

#	OMP-UPMSP			INV			LAHC			SA			FB		
	ms	*gap*	*time*	*ms*	*gap*	*time*	*ms*	*gap*	*time*	*ms*	*gap*	*time*	*ms*	*gap*	*time*
s_001	183.98	0.37	0.01	223.19	0.48	0.01	183.98	0.37	0.93	183.98	0.37	1.02	183.98	0.37	3.81
s_002	176.23	0.35	0.01	230.88	0.50	0.01	176.23	0.35	1.82	176.23	0.35	1.76	176.23	0.35	7.43
s_003	194.29	0.41	0.01	194.54	0.41	0.01	194.29	0.41	1.05	194.29	0.41	1.44	194.29	0.41	5.70
s_004	194.29	0.41	0.01	194.54	0.41	0.01	194.29	0.41	0.98	194.29	0.41	1.56	194.29	0.41	5.34
s_005	176.23	0.35	0.01	230.88	0.50	0.01	176.23	0.35	2.11	176.23	0.35	1.75	176.23	0.35	8.91
s_006	176.23	0.35	0.01	230.88	0.50	0.01	176.23	0.35	1.68	176.23	0.35	1.81	176.23	0.35	10.37
s_007	129.28	0.11	0.01	253.20	0.54	0.01	129.28	0.11	1.59	129.28	0.11	1.72	129.28	0.11	9.96
s_008	153.90	0.25	0.01	253.15	0.54	0.01	153.90	0.25	0.81	153.90	0.25	1.07	153.90	0.25	4.95
s_009	153.90	0.25	0.01	253.15	0.54	0.01	153.90	0.25	0.83	153.90	0.25	1.15	153.90	0.25	5.04
s_010	129.28	0.11	0.01	253.20	0.54	0.01	129.28	0.11	1.71	129.28	0.11	1.75	129.28	0.11	10.12
m_001	257.57	0.41	0.01	309.08	0.49	0.01	231.81	0.37	1.47	231.81	0.37	2.26	257.57	0.41	5.61
m_002	229.09	0.38	0.01	297.81	0.49	0.01	229.09	0.38	1.87	229.09	0.38	1.86	229.09	0.38	8.04
m_003	252.57	0.41	0.01	303.08	0.49	0.01	227.31	0.36	1.17	227.31	0.36	1.89	252.57	0.41	8.17
m_004	233.48	0.38	0.01	326.87	0.53	0.01	233.48	0.38	1.44	233.48	0.38	2.05	233.48	0.38	9.21
m_005	211.47	0.35	0.01	274.91	0.45	0.01	211.47	0.35	3.29	211.47	0.35	1.97	211.47	0.35	9.04
m_006	220.28	0.37	0.01	286.36	0.48	0.01	220.28	0.37	1.96	220.28	0.37	1.86	220.28	0.37	10.43
m_007	219.71	0.36	0.01	285.62	0.46	0.01	219.71	0.36	1.67	219.71	0.36	1.74	219.71	0.36	10.09
m_008	184.68	0.25	0.01	258.55	0.35	0.01	184.68	0.25	0.85	184.68	0.25	1.17	184.68	0.25	5.69
m_009	200.07	0.29	0.01	280.09	0.41	0.01	200.07	0.29	1.05	200.07	0.29	1.38	200.07	0.29	5.78
m_010	206.84	0.31	0.01	289.57	0.43	0.01	206.84	0.31	1.82	206.84	0.31	1.79	206.84	0.31	10.35
b_001	283.32	0.45	0.01	311.65	0.49	0.01	254.98	0.40	1.81	254.98	0.40	2.49	283.32	0.45	7.04
b_002	193.85	0.31	0.01	252.01	0.41	0.01	193.85	0.31	1.88	193.85	0.31	1.92	193.85	0.31	10.61
b_003	213.71	0.33	0.01	277.82	0.43	0.01	213.71	0.33	1.75	213.71	0.33	2.83	213.71	0.33	13.72
b_004	217.60	0.34	0.01	261.12	0.41	0.01	217.60	0.34	2.03	217.60	0.34	2.81	217.60	0.34	10.37
b_005	232.61	0.38	0.01	279.13	0.46	0.01	232.61	0.38	4.09	232.61	0.38	1.92	232.61	0.38	10.95
b_006	242.31	0.40	0.01	290.77	0.48	0.01	230.19	0.38	2.29	230.19	0.38	1.90	242.31	0.40	10.49
b_007	197.73	0.32	0.01	237.27	0.38	0.01	197.73	0.32	1.78	197.73	0.32	1.94	197.73	0.32	10.24
b_008	203.14	0.32	0.01	243.76	0.38	0.01	203.14	0.32	0.93	203.14	0.32	1.76	203.14	0.32	8.60
b_009	220.07	0.35	0.01	286.09	0.45	0.01	220.07	0.35	1.02	220.07	0.35	1.83	220.07	0.35	8.71
b_010	227.52	0.36	0.01	295.78	0.46	0.01	227.52	0.34	1.97	227.52	0.34	1.81	227.52	0.36	10.67

Tabela 4.2 – Resultados para o problema de sequenciamento com abordagens heurísticas.

Pela Tabela 4.2 e Figura 4.2 é possível notar que OMP-UPMSP performou melhor que a abordagem INV para todos os blocos de instâncias que a e também que a variação nos resultados de uma mesma abordagem é mínima. Isso ocorre pois as instâncias analisadas em cada bloco têm pouca variação entre si, logo, o algoritmo tem pouca margem para fazer alterações na solução sem torná-la inviável. Em função disso, como a segunda abordagem define primeiramente quais pilhas serão utilizadas, ignorando os parâmetros de qualidade, o modelo precisa definir a massa que deve ser retomada priorizando um sub-conjunto pré-definido de pilhas de estocagem, que muitas vezes não possui as melhores condições para realizar a blendagem, se comparado ao conjunto completo de pilhas de minério.

Pela Figura 4.2 também nota-se que para as instâncias *medium* as meta-heurísticas LAHC e SA conseguiram refinar os resultados obtidos para alguns casos, apesar do tempo computacional ser ligeiramente maior. Já para a abordagem FB é possível perceber que não houveram melhorias no *makespan*, mas um considerável aumento no tempo de execução, já que o modelo de programação linear para a mistura de minérios é executado diversas vezes seguido do método construtivo para o problema de sequenciamento.

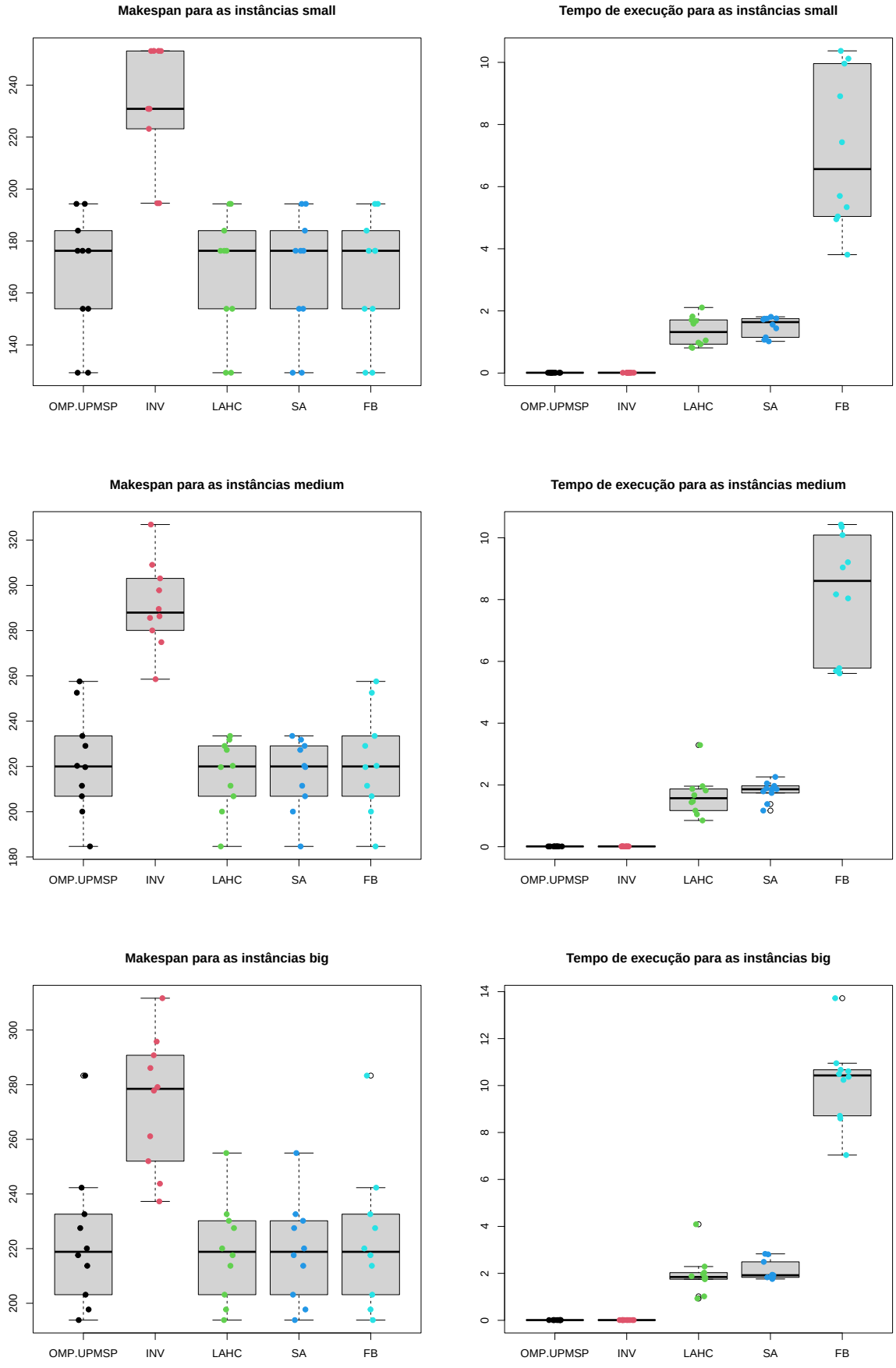

Figura 4.2 – Resultados e tempo de execução para o problema de sequenciamento.

Foi realizado também uma análise estatística para cada um dos blocos de instâncias individualmente. As observações realizadas nas instâncias fornecem informações para concluir que os dados analisados são estatisticamente dependentes, pois as amostras contêm o mesmo conjunto de observações ou contêm observações diferentes mas que se emparelharam de forma significativa. Sendo assim, seguiu-se o fluxo de testes paramétricos da Figura 4.3 para amostras dependentes e normalmente distribuídas com mais de três algoritmos e o fluxo de testes não paramétricos da Figura 4.4 para amostras dependentes mas não normalmente distribuídas.

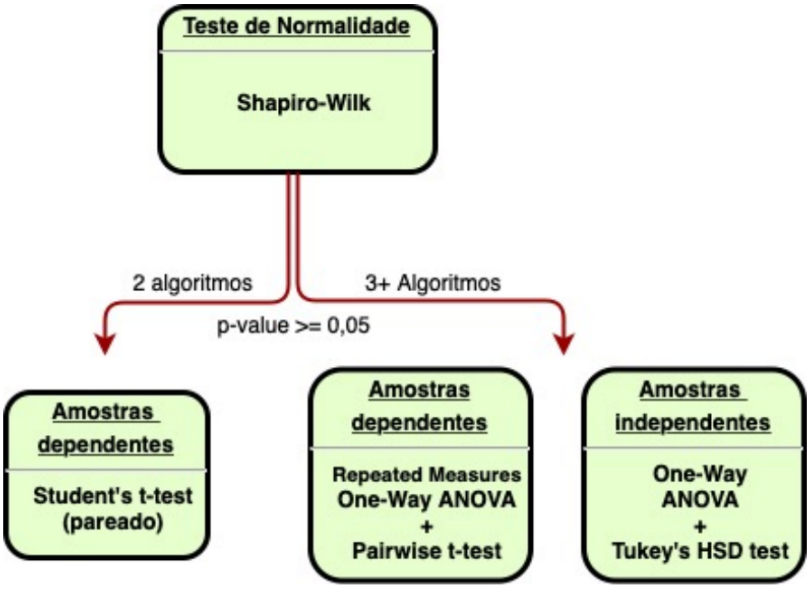

Figura 4.3 – Fluxograma de escolha de testes paramétricos.

Dessa forma, para cada bloco de instâncias, foram realizados os testes de normalidade Shapiro-Wilk (SHAPIRO; WILK, 1965), no qual a hipótese nula é de que a distribuição amostral é normal. Se $p - value$ é menor do que o valor crítico dado pelo nível de significância α (normalmente 0,05), então o pressuposto de normalidade é rejeitado. Ou seja, existem evidência de que os dados testados não pertencem a uma população normalmente distribuída.

Após a aplicação do teste de normalidade e caso existam evidências de que os dados são normalmente distribuídos, segue-se para o *Repeated measures One-Way ANOVA* (BOBBITT, 2020), que é um teste de comparações múltiplas de populações consideradas normalmente distribuídas no qual a hipótese nula é de que as populações possuem distribuições idênticas. Em outras palavras, se $p - value$ for menor do que nível de significância α é possível concluir que há diferença significativa entre as populações. Entre os dados retornados tem-se a coluna $Pr(> F)$, que corresponde ao $p - value$.

O *pairwise t-test* realiza comparações par a par entre as médias das populações com correções para comparações múltiplas, este teste é utilizado comumente em conjunto com o *Repeated measures One-Way ANOVA*, caso haja diferença significativa. O intuito desse teste é

identificar onde reside essa diferença. É utilizada a correção de *Bonferroni* no $p - value$, para contornar o problema de múltiplas comparações e evitar resultados falsos-positivos. O resultado é reportado por uma matriz triangular dos $p - values$ entre todos os grupos comparados e um valor menor do que o nível de significância α nos permite concluir que há diferença significativa entre as populações.

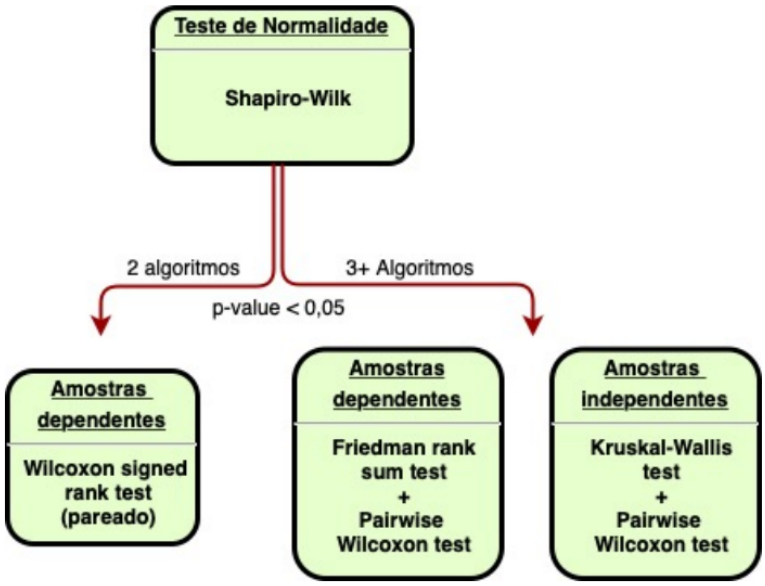

Figura 4.4 – Fluxograma de escolha de testes não paramétricos.

Caso existam evidências de que os dados não são normalmente distribuídos após a aplicação do teste de normalidade, segue-se para o *Friedman rank sum test*, que é um teste de comparações múltiplas de populações, no qual a hipótese nula é de que não há diferença entre as populações. Sendo assim, se $p - value$ for menor do que nível de significância α é possível concluir que há diferença significativa entre as populações. Caso essa diferença exista é possível usar o *pairwise Wilcoxon test* para calcular comparações de pares entre níveis de grupo com correções para vários testes e identificar onde essa diferença reside.

A Figura 4.5 mostra o relatório dos testes realizados em R, que é uma linguagem e ambiente para computação estatística, para o grupo de instâncias *small*. A partir da saída do *Friedman rank sum test*, tem-se que existe uma diferença significativa entre os grupos, mas não sabemos quais pares de grupos são diferentes. Sendo assim, pelo resultado do *pairwise Wilcoxon test*, podemos observar pela a matriz triangular dos $p - values$ entre todos os grupos comparados um valor menor do que o nível de significância α, logo não há uma diferença estatisticamente significativa nos resultados entre as abordagens OMP-UPMSP, LAHC e SA, porém existe diferença significativa entre a abordagem INV e as demais.

```
Shapiro-Wilk normality test

data: OMP-UPMSP |   W = 0.88224, p-value = 0.1385
data: INV       |   W = 0.82408, p-value = 0.02839
data: LAHC      |   W = 0.88224, p-value = 0.1385
data: SA        |   W = 0.88224, p-value = 0.1385
data: FB        |   W = 0.88224, p-value = 0.1385
------------------------------------------------------------
Friedman rank sum test

data:  values, method and instance
Friedman chi-squared = 40, df = 4, p-value = 4.328e-08
------------------------------------------------------------
Pairwise comparisons using Wilcoxon rank sum test

data:  values and method

     OMP.UPMSP INV    LAHC   SA
INV  0.0017    -      -      -
LAHC 1.0000    0.0017 -      -
SA   1.0000    0.0017 1.0000 -
FB   1.0000    0.0017 1.0000 1.0000

P value adjustment method: bonferroni
```

Figura 4.5 – Resultados dos testes estatísticos para as instâncias *small*.

A Figura 4.6 mostra o relatório dos testes realizados para o grupo de instâncias *medium*. A partir dos resultados do *Repeated measures One-Way ANOVA*, tem-se que existe uma diferença significativa entre os grupos, mas não sabemos quais pares de grupos são diferentes. Sendo assim, pelo resultado do *pairwise t-test*, podemos observar pela a matriz triangular dos $p - values$ entre todos os grupos comparados um valor menor do que o nível de significância α, logo não há uma diferença estatisticamente significativa nos resultados entre as abordagens OMP-UPMSP, LAHC, SA e FB, mas existe diferença significativa entre a abordagem INV e as demais.

```
Shapiro-Wilk normality test

data: OMP-UPMSP |   W = 0.9749, p-value = 0.9322
data: INV       |   W = 0.98754, p-value = 0.9928
data: LAHC      |   W = 0.91776, p-value = 0.3387
data: SA        |   W = 0.91776, p-value = 0.3387
data: FB        |   W = 0.9749, p-value = 0.9322
------------------------------------------------------------
Repeated measures one-way ANOVA

Error: factor(instance)
          Df Sum Sq Mean Sq F value Pr(>F)
Residuals  9  15154    1684

Error: Within
               Df Sum Sq Mean Sq F value Pr(>F)
factor(method)  4  41927   10482   211.1 <2e-16 ***
Residuals      36   1788      50
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
------------------------------------------------------------
Pairwise comparisons using t tests with pooled SD

data:  values and method

     OMP.UPMSP INV     LAHC SA
INV  3.2e-09   -       -    -
LAHC 1         4.5e-10 -    -
SA   1         4.5e-10 1    -
FB   1         3.2e-09 1    1

P value adjustment method: bonferroni
```

Figura 4.6 – Resultados dos testes estatísticos para as instâncias *medium*.

A Figura 4.7 mostra o relatório dos testes realizados para o grupo de instâncias *big*. Como explicado para o grupo de instâncias *medium*, podemos observar que de acordo com o $p - value$ do *pairwise t-test*, também não há uma diferença estatisticamente significativa nos resultados entre as abordagens OMP-UPMSP, LAHC, SA e FB, porém existe diferença significativa entre a abordagem INV e as demais.

```
Shapiro-Wilk normality test

data: OMP-UPMSP |   W = 0.89582, p-value = 0.197
data: INV       |   W = 0.96209, p-value = 0.8094
data: LAHC      |   W = 0.96257, p-value = 0.8147
data: SA        |   W = 0.96257, p-value = 0.8147
data: FB        |   W = 0.89582, p-value = 0.197
------------------------------------------------------------
Repeated measures one-way ANOVA

Error: factor(instance)
          Df Sum Sq Mean Sq F value Pr(>F)
Residuals  9  21934    2437

Error: Within
               Df Sum Sq Mean Sq F value Pr(>F)
factor(method)  4  22111    5528   117.9 <2e-16 ***
Residuals      36   1688      47
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
------------------------------------------------------------
Pairwise comparisons using t tests with pooled SD

data:  values and method

     OMP.UPMSP INV     LAHC    SA
INV  0.00012   -       -       -
LAHC 1.00000   3.3e-05 -       -
SA   1.00000   3.3e-05 1.00000 -
FB   1.00000   0.00012 1.00000 1.00000

P value adjustment method: bonferroni
```

Figura 4.7 – Resultados dos testes estatísticos para as instâncias *big*.

Pelos resultados e análise estatística realizada é possível concluir que a abordagem OMP-UPMSP performou melhor que as demais para todos os blocos de instâncias, tanto para os valores médios do gap em relação ao limite inferior quanto para o tempo de execução. Avaliando o bloco de instâncias *medium*, é possível perceber que as meta-heurísticas LAHC e SA obtiveram um valor médio de $makespan$ ligeiramente menor, entretanto pelas análises estatísticas tem-se que essa diferença não significativa em nenhum bloco. Além disso, a abordagem INV foi a única que apresentou uma diferença estatisticamente diferente das demais, entretanto seus valores médios para o $makespan$ foram muito piores. Para a abordagem FB houve um considerável aumento no tempo de execução, devido as diversas iterações entre modelo de programação linear e construção da solução para o problema de sequenciamento.

Podemos concluir também que devido as configurações do pátio de minério, existe uma grande quantidade de restrições, como acesso do maquinário a uma quantidade limitada de pilhas, tipo de operação de cada equipamento, colisão entre as máquinas ao tentar acessar certas áreas de estocagem, que impossibilitam a geração de soluções vizinhas viáveis. Sendo assim, nota-se que em poucos casos as meta-heurísticas conseguiram refinar a solução gulosa gerada pelo construtivo. Logo, a abordagem OMP-UPMSP se destaca pela semelhança entre os resultados médios para o $makespan$, quando comparado aos demais métodos, e principalmente pelo tempo computacional extremamente baixo.

5 Considerações Finais

Neste trabalho foram propostas soluções para tratar o Problema de Mistura e Blendagem de Minérios e o Problema de Sequenciamento e Planejamento da Produção da indústria mineradora, visando minimizar o tempo gasto para atender as demandas, o custo operacional e o não atendimento às metas de qualidade.

Foi desenvolvido um modelo de programação linear considerando um cenário real, no qual existe um grande número de variáveis e restrições. Também foram implementados um algoritmo construtivo guloso, as meta-heurísticas *Simulated Annealing* e *Late Acceptance Hill-Climbing* e mais duas abordagens heurísticas, a fim de gerar soluções viáveis para o problema. Entre as duas abordagens heurísticas desenvolvidas, temos a primeira delas gerando soluções por meio do modelo matemático para o problema de mistura e em seguida para o problema de sequenciamento que fornece um *feedback* para o modelo de programação linear gerar novas soluções. A segunda abordagem atua primeiramente no problema de sequenciamento e em seguida indica para o modelo o conjunto de pilhas que devem ser priorizadas no problema de mistura.

Os resultados obtidos pelos algoritmos implementados são promissores. No entanto, ainda é necessário uma análise comparativa com dados reais como forma de avaliar a qualidade das soluções obtidas. Nota-se que o projeto tem grande espaço na Indústria 4.0, contribuindo para o aprimoramento dos processos de uma empresa mineradora e a otimização dos recursos utilizados. Além disso, o algoritmo desenvolvido tem grande possibilidade de evolução, avaliando outras abordagens heurísticas e refinando os métodos já implementados.

5.1 Trabalhos Futuros

Como trabalho futuro aponta-se a necessidade de melhoria dos métodos heurísticos e do modelo de programação linear, realizar experimentos em instâncias mais variadas, preferencialmente com dados reais de uma empresa mineradora. Para atingir este objetivo, foram elencados alguns objetivos específicos, tais como avaliar outros modelos de otimização propostos na literatura, de forma a contemplar os requisitos típicos de uma empresa mineradora; aplicar e desenvolver outras meta-heurísticas para o problema, avaliando sua performance no refinamento da solução; avaliar e comparar os resultados entre as mais diversas abordagens heurísticas, a fim de encontrar o método mais adequado para o problema; expandir a resolução do problema para cenários de médio e longo prazo, realizando o planejamento para um conjunto maior de pedidos. Sendo essa última uma abordagem mais interessante para a realidade de uma empresa mineradora.

Referências

ALVES, J. M. d. C. B. Um sistema para o planejamento de produção e vendas de uma empresa mineradora. Programa de Pós-Graduação em Engenharia Mineral. Departamento de Engenharia, 2007.

BOBBITT, Z. *How to Perform a Repeated Measures ANOVA in R*. 2020. Disponível em: <https://www.statology.org/repeated-measures-anova-in-r/>.

BUENO, A. F.; OLIVEIRA, M. C. Goal programming (programação multiobjetivo). *Pesquisa Operacional para decisão em Contabilidade*, 2004.

BURKE, E. K.; BYKOV, Y. A late acceptance strategy in hill-climbing for exam timetabling problems. In: *PATAT 2008 Conference, Montreal, Canada*. [S.l.: s.n.], 2008. p. 1–7.

COSTA, F. Aplicações de técnicas de otimização a problemas de planejamento operacional de lavra em minas a céu aberto. *Programa de Pós-Graduação em Engenharia Mineral Ouro Preto*, 2005.

EVERETT, J. Iron ore production scheduling to improve product quality. *European journal of operational research*, Elsevier, v. 129, n. 2, p. 355–361, 2001.

EVERETT, J. Computer aids for production systems management in iron ore mining. *International Journal of Production Economics*, Elsevier, v. 110, n. 1-2, p. 213–223, 2007.

IBRAM. *Eleições 2018: políticas públicas para a indústria mineral*. [S.l.]: Instituto Brasileiro de Mineração - IBRAM, 2018.

IBRAM. *Apresentação Coletiva Setor Mineral 3º Trimestre 2020*. [S.l.]: Instituto Brasileiro de Mineração - IBRAM, 2020.

JÚNIOR, G. P. Métodos de otimização multiobjetivo e de simulação aplicados ao problema de planejamento operacional de lavra em minas a céu aberto. Programa de Pós-Graduação em Engenharia Mineral. Departamento de Engenharia, 2011.

KIRKPATRICK, S.; GELATT, C. D.; VECCHI, M. P. Optimization by simulated annealing. *science*, American association for the advancement of science, v. 220, n. 4598, p. 671–680, 1983.

LU, T.-F.; MYO, M. T. R. Optimal stockpile voxel identification based on reclaimer minimum movement for target grade. *International Journal of Mineral Processing*, Elsevier, v. 98, n. 1-2, p. 74–81, 2011.

MORAES, E. F.; ALVES, J. M. d. C. B.; SOUZA, M. J. F.; CABRAL, I. E.; MARTINS, A. X. Um modelo de programação matemática para otimizar a composição de lotes de minério de ferro da mina cauê da cvrd. *Rem: Revista Escola de Minas*, SciELO Brasil, v. 59, n. 3, p. 299–306, 2006.

PINTO, L. R.; BIAJOLI, F. L.; MINE, O. M. Uso de otimizador em planilhas eletrônicas para auxílio ao planejamento de lavra. *Relatório técnico, Programa de Pós-graduação em Engenharia Mineral, Universidade Federal de Ouro Preto, Ouro Preto, Minas Gerais*, 2003.

REEVES, C. R. *Modern heuristic techniques for combinatorial problems*. [S.l.]: John Wiley & Sons, Inc., 1993.

RIBEIRO, C. C. Metaheuristics and applications. *Advanced School on Artificial Intelligence, Estoril, Portugal*, 1996.

ROMERO, C. A general structure of achievement function for a goal programming model. *European Journal of Operational Research*, Elsevier, v. 153, n. 3, p. 675–686, 2004.

SANTOS, H. G.; TOFFOLO, T. A.; SILVA, C. L.; BERGHE, G. V. Analysis of stochastic local search methods for the unrelated parallel machine scheduling problem. *International Transactions in Operational Research*, Wiley Online Library, v. 26, n. 2, p. 707–724, 2019.

SHAPIRO, S. S.; WILK, M. B. An analysis of variance test for normality (complete samples). *Biometrika*, JSTOR, v. 52, n. 3/4, p. 591–611, 1965.

SOUZA, M. J. F. Inteligência computacional para otimização. *Notas de aula, Departamento de Computação*, Universidade Federal de Ouro Preto, 2008. Disponível em: <http://www.decom.ufop.br/prof/marcone/InteligenciaComputacional/InteligenciaComputacional.pdf>.

TOFFOLO, T. A. M. Otimizaçao do fluxo de produtos de uma empresa mineradora. Universidade Federal de Minas Gerais, 2009.

VIANEN, T. V.; OTTJES, J.; LODEWIJKS, G. Simulation-based rescheduling of the stacker–reclaimer operation. *Journal of Computational Science*, Elsevier, v. 10, p. 149–154, 2015.

Apêndices

APÊNDICE A – Documentação das Classes

As classes implementadas para o algoritmo construído para solucionar o problema tema deste trabalho estão documentadas neste apêndice. A documentação do código está inteiramente em inglês, bem como os nomes das classes, métodos e variáveis, com intuito de torná-lo mais genérico e acessível. Estão presentes neste documento apenas os métodos e atributos públicos de cada classe. A documentação que se segue foi produzida usando a ferramenta Sphinx, que é um gerador de documentação escrito e usado pela comunidade Python.

Código-fonte disponível em: <https://github.com/gabriaraujo/omp-upmsp>

Constructive Module

Constructive

class Constructive(*problem*, *solution*)

This class contains simple constructive procedures for the Machine Scheduling Problem. Generates and returns a greedy solution. The jobs are added in a specific order to the machine to which they incur the smallest increase in the makespan.

build()

This method defines the operations performed on each stockpile and their durations.

To use this method outside the run() method, you must manually assign the value of the output_id attribute of the Constructive class and the routes attribute of the Solution class before using it.

Return type
None

property inputs: List[float]

List with the stacked weights, in which the columns are the stockpiles.

Type
List[float]

Return type
List[float]

property output_id: Optional[int]

The output request identifier.

Type
Optional[int]

Return type
Optional[int]

property problem: *Problem*

The problem considered.

Type
Problem

Return type
Problem

reset_inputs()

This method is called whenever the input list should be reset (mainly to avoid the need of creating another object).

Return type
None

run(*has_routes=False*)

Executes the Constructive for all output requests.

Parameters
has_routes (*bool*) – Flag to indicate if the routes are already defined. True if the routes have already been established, False otherwise. Note that the False option will define the routes automatically and greedily. Defaults to False.

Return type
None

`set_jobs(routes)`

This method defines greedily where each machine will act based on its routes and the start time of each job.

Return type
`None`

`set_route(start_time, engine)`

This method greedily defines the operating order of each individual machine. It assigns all possible jobs to each machine, which must be further refined by set_jobs().

Parameters

- `start_time` (*List[float]*) – List with the time when each engine can start a new task.
- `engine` (`Engine`) – The engine reference.

Returns

List of tuples whose first
element is the access time of the machine to the stockpile, the second element is the engine ID, the third element is its position and the last element is its configuration.

Return type
List[Tuple[float, int, int, str]]

`set_routes()`

This method defines the order of operation of all machines and save the result in the routes attribute of the Solution class.

Return type
`None`

`property solution: Solution`

The solution reference.

Type
Solution

Return type
Solution

`property weights: List[List[float]]`

List of lists with the weights retrieved from each output request, in which the lines are the requests and the columns, the stockpiles.

Type
List[List[float]]

Return type
`List[List[float]]`

Linear Model

`class LinModel(problem)`

This class represents a Linear Model that is built using the Python-MIP package.

To instantiate this class, you must install and import the Python-MIP package first. Python-MIP requires Python 3.5 or newer. Since Python-MIP is included in the Python Package Index, once you have a Python installation, installing it is as easy as entering in the command prompt:

$ pip install mip

Python-MIP is a collection of Python tools for the modeling and solution of Mixed-Integer Linear programs (MIPs). The package also provides access to advanced solver features like cut generation, lazy constraints, MIP starts and solution pools. For more information, access https://www.python-mip.com.

`add_weights(`*variable*, *weights*`)`

This method assigns weights to the variables. It must be called whenever it is necessary to send to send feedback to this model.

Parameters

- `variable` (*str*) – Indicator of which variable weights are defined. It must be 'x' or 'y'.
- `weights` (*Dict[Tuple[int, int], int]*) – Dict of weights reclaimed or stacked resulting from a previous execution of this model.

Return type
`None`

`resolve()`

This method resolves the linear model and write the its details in a .lp file (in CPLEX or Gurobi lp format).

Returns
A tuple whose first element is the objective value of the model, the second element is a dictionary of entries whose keys are the stockpiles IDs and values are lists with the stacked weights, the last element is a dictionary of recoveries whose keys are the IDs of each request and the values are lists with the reclaimed weights.

Return type
Tuple[Optional[float], Dict[str, List[float]], Dict[str, List[float]]]

Post-Model

`class PostModel(`*problem*, *solution*`)`

This class contains simple constructive procedures for the Machine Scheduling Problem. Generates and returns a greedy solution. The jobs are added in a specific order to the machine to which they incur the smallest increase in the makespan.

`build()`

This method defines the operations performed on each stockpile and their durations.

To use this method outside the run() method, you must manually assign the value of the output_id attribute of the Constructive class and the routes attribute of the Solution class before using it.

Return type
`None`

`reset_inputs()`

This method is called whenever the input list should be reset (mainly to avoid the need of creating another object).

Return type
`None`

`run(`*has_routes=False*`)`

Executes the Constructive for all output requests.

Parameters
`has_routes` (*bool*) – Flag to indicate if the routes are already defined. True if the routes have already been established, False otherwise. Note that the False option will define the routes automatically and greedily. Defaults to False.

Return type
None

`set_jobs(`*routes*`)`

This method defines greedily where each machine will act based on its routes and the start time of each job.

Return type
None

`set_route(`*start_time*, *engine*`)`

This method greedily defines the operating order of each individual machine. It assigns all possible jobs to each machine, which must be further refined by set_jobs().

Parameters

- `start_time` (*List[float]*) – List with the time when each engine can start a new task.
- `engine` (`Engine`) – The engine reference.

Returns

List of tuples whose first
element is the access time of the machine to the stockpile, the second element is the engine ID, the third element is its position and the last element is its configuration.

Return type
List[Tuple[float, int, int, str]]

`set_routes()`

This method defines the order of operation of all machines and save the result in the routes attribute of the Solution class.

Return type
None

Pre-Model

`class PreModel(`*problem*, *solution*`)`

This class contains simple constructive procedures for the Machine Scheduling Problem. Generates and returns a greedy solution. The jobs are added in a specific order to the machine to which they incur the smallest increase in the makespan.

`build()`

This method defines the operations performed on each stockpile and their durations.

To use this method outside the run() method, you must manually assign the value of the output_id attribute of the Constructive class and the routes attribute of the Solution class before using it.

Return type
None

`reset_inputs()`

This method is called whenever the input list should be reset (mainly to avoid the need of creating another object).

Return type
None

`run(`*has_routes=False*`)`

Executes the Constructive for all output requests.

Parameters
`has_routes` (*bool*) – Flag to indicate if the routes are already defined. True if the routes have already been established, False otherwise. Note that the False option will define the routes automatically and greedily. Defaults to False.

Return type
None

`set_jobs`(*routes*)

This method defines greedily where each machine will act based on its routes and the start time of each job.

Return type
None

`set_route`(*start_ time*, *engine*)

This method greedily defines the operating order of each individual machine. It assigns all possible jobs to each machine, which must be further refined by set_jobs().

Parameters

- `start_time` (*List [float]*) – List with the time when each engine can start a new task.
- `engine` (Engine) – The engine reference.

Returns

List of tuples whose first
element is the access time of the machine to the stockpile, the second element is the engine ID, the third element is its position and the last element is its configuration.

Return type
List[Tuple[float, int, int, str]]

`set_routes`()

This method defines the order of operation of all machines and save the result in the routes attribute of the Solution class.

Return type
None

Heuristic Module

Heuristic

`class Heuristic`(*problem*, *name*)

This class represents a Heuristic (or Local Search method). The basic methods and neighborhood selection are included.

`accept_move`(*move*)

This method accepts a move.

Parameters
`move` (Move) – The move to be accepted.

Return type
None

`add_move`(*move*)

This method adds a move to the heuristic.

Parameters
`move` (*int*) – The move to be added.

Return type
None

property best_solution: Optional[*Solution*]

Best solution found by the heuristic.

Type
Optional[*Solution*]

Return type
Optional[*Solution*]

property iters: int

Iteration counter.

Type
int

Return type
int

property moves: List[*Move*]

List with all the movements present in this heuristic.

Type
List[*Move*]

Return type
List[*Move*]

property name: str

The name of the heuristic.

Type
str

Return type
str

property problem: *Problem*

The problem reference.

Type
Problem

Return type
Problem

reject_move(*move*)

This method rejects a move.

Parameters
move (Move) – The move to be rejected.

Return type
None

select_move(*solution*)

This method selects a move.

Parameters
solution (Solution) – The solution.

Returns
a randomly selected move (neighborhood).

Return type
Move

Late Acceptance Hill-Climbing

`class LAHC(`*problem*, *size*`)`

This class is a Late Acceptance Hill-Climbing implementation.

`run(`*initial_solution*, *max_iters*, *best_known=False*`)`

Executes the Late Acceptance Hill-Climbing and updates the best solution.

Parameters

- `initial_solution` (`Solution`) – The initial (input) solution.
- `max_iters` (*`int`*) – The maximum number of iterations to execute.
- `best_known` (*`bool`*) – True if the initial best_solution have already been established, False otherwise. Note that the False option will define the initial best_solution as the initial_solution. Defaults to False.

Return type
`None`

`property size: int`

The number os most recent solutions.

Type
int

Return type
`int`

Simulated Annealing

`class SA(`*problem*, *alpha*, *t0*, *sa_max=1000*`)`

This class is a Simulated Annealing implementation.

`property alpha: float`

Cooling rate for the simulated annealing.

Type
float

Return type
`float`

`property eps: float`

Conceptual zero (mainly to avoid zero division error).

Type
float

Return type
`float`

`run(`*initial_solution*, *max_iters*, *best_known=False*`)`

Executes the Simulated Annealing and updates the best solution.

Parameters

- `initial_solution` (`Solution`) – The initial (input) solution.
- `max_iters` (*`int`*) – The maximum number of iterations to execute.
- `best_known` (*`bool`*) – True if the initial best_solution have already been established, False otherwise. Note that the False option will define the initial best_solution as the initial_solution. Defaults to False.

Return type
None

property sa_max: int

The maximum number of iterations without improvements to execute.

Type
int

Return type
int

property t0: float

Initial temperature.

Type
float

Return type
float

Neighborhood Module

Move

class Move(*problem*, *constructive*, *name*)

This class represents a Move (or Neighborhood). The basic methods as well as several counters (for future analysis) are included.

accept()

This method must be called whenever the modification made by this move is accepted. It ensures that the solution as well as other structures are updated accordingly.

Return type
None

property constructive: *Constructive*

The move constructive procedure.

Type
Constructive

Return type
Constructive

property current_solution: *Solution*

The current solution reference.

Type
Solution

Return type
Solution

property delta_cost: float

The move delta cost.

Type
float

Return type
float

`do_move(`*solution*`)`

This method returns does the move and returns the impact (delta cost) in the solution.

Parameters
`solution` (`Solution`) – The solution to be modified.

Returns
The impact (delta cost) of this move in the solution.

Return type
float

`gen_move(`*solution*`)`

This method generates a random candidate for the movement that must be subsequently validated by has_move ().

Parameters
`solution` (`Solution`) – The solution to be modified.

Return type
`None`

`has_move(`*solution*`)`

This method returns a boolean indicating whether this neighborhood can be applied to the current solution.

Parameters
`solution` (`Solution`) – The solution to be evaluated.

Returns

True if this neighborhood can be applied to the current
solution, False otherwise.

Return type
bool

`property improvements: int`

The move improvements counter.

Type
int

Return type
`int`

`property initial_cost: float`

The move initial cost.

Type
float

Return type
`float`

`property intermediate_state: bool`

flag to indicate whether the movement is in the intermediate state.

Type
bool

Return type
`bool`

`property iters: int`

The move interaction counter.

Type
int

Return type
`int`

`property name: str`

The move name.

Type
str

Return type
`str`

`property problem: Problem`

The problem reference.

Type
Problem

Return type
Problem

`reject()`

This method must be called whenever the modification made by this move are rejected. It ensures that the solution as well as other structures are updated accordingly.

Return type
`None`

`property rejects: int`

the move rejects counter.

Type
int

Return type
`int`

`reset()`

This method is called whenever the neighborhood should be reset (mainly to avoid the need of creating another object).

Return type
`None`

`property sideways: int`

the move sideways counter.

Type
float

Return type
`int`

`property worsens: int`

The move worsens counter.

Type
int

Return type
`int`

Swap

`class Swap(`*problem*, *constructive*`)`

This class represents a Swap Move. A neighbor in the Swap Neighborhood is generated by swapping two jobs between two machines. Note that the swapped jobs may be placed in any position on the other machine, i.e. the original positions are not taken into account. This Move first removes the two jobs and then reinserts them in random positions.

`accept()`

This method must be called whenever the modification made by this move is accepted. It ensures that the solution as well as other structures are updated accordingly.

Return type
`None`

`do_move(`*solution*`)`

This method returns does the move and returns the impact (delta cost) in the solution.

Parameters
`solution` (`Solution`) – The solution to be modified.

Returns
The impact (delta cost) of this move in the solution.

Return type
float

`property engine_1:` ***Engine***

The first engine reference.

Type
Engine

Return type
Engine

`property engine_2:` ***Engine***

The second engine reference.

Type
Engine

Return type
Engine

`gen_move(`*solution*`)`

This method generates a random candidate for the movement that must be subsequently validated by has_move().

Parameters
`solution` (`Solution`) – The solution to be modified.

Return type
`None`

`has_move(`*solution*`)`

This method returns a boolean indicating whether this neighborhood can be applied to the current solution.

Parameters
`solution` (`Solution`) – The solution to be evaluated.

Returns

True if this neighborhood can be applied to the current
solution, False otherwise.

Return type
bool

property job_1: Tuple[int, str]

The selected job from the first engine route.

Type
Tuple[int, str]

Return type
Tuple[int, str]

property job_2: Tuple[int, str]

The selected job from the second engine route.

Type
Tuple[int, str]

Return type
Tuple[int, str]

property pos_1: Optional[int]

The index of the first job.

Type
Optional[int]

Return type
Optional[int]

property pos_2: Optional[int]

The index of the second job.

Type
Optional[int]

Return type
Optional[int]

reject()

This method must be called whenever the modification made by this move are rejected. It ensures that the solution as well as other structures are updated accordingly.

Return type
None

reset()

This method is called whenever the neighborhood should be reset (mainly to avoid the need of creating another object).

Return type
None

property route_1: List[Tuple[int, str]]

The route of the first engine.

Type
List[Tuple[int, str]]

Return type
List[Tuple[int, str]]

property route_2: List[Tuple[int, str]]

The route of the second engine.

Type
List[Tuple[int, str]]

Return type
List[Tuple[int, str]]

Switch

class Switch(*problem*, *constructive*)

This class represents a Switch Move. A neighbor in the Switch Move structure is generated by switching the order of two jobs of a machine.

accept()

This method must be called whenever the modification made by this move is accepted. It ensures that the solution as well as other structures are updated accordingly.

Return type
None

do_move(*solution*)

This method returns does the move and returns the impact (delta cost) in the solution.

Parameters
solution (Solution) – The solution to be modified.

Returns
The impact (delta cost) of this move in the solution.

Return type
float

property engine: *Engine*

The engine reference.

Type
Engine

Return type
Engine

gen_move(*solution*)

This method generates a random candidate for the movement that must be subsequently validated by has_move ().

Parameters
solution (Solution) – The solution to be modified.

Return type
None

has_move(*solution*)

This method returns a boolean indicating whether this neighborhood can be applied to the current solution.

Returns

True if this neighborhood can be applied to the current
solution, False otherwise.

Return type
bool

property job_1: Optional[int]

The index of the first job.

Type
Optional[int]

Return type
`Optional[int]`

`property job_2: Optional[int]`

The index of the second job.

Type
Optional[int]

Return type
`Optional[int]`

`reject()`

This method must be called whenever the modification made by this move are rejected. It ensures that the solution as well as other structures are updated accordingly.

Return type
`None`

`reset()`

This method is called whenever the neighborhood should be reset (mainly to avoid the need of creating another object).

Return type
`None`

SimpleSwap

`class SimpleSwap(`*problem*, *constructive*`)`

This class represents a Simple Swap Move. A neighbor in the Simple Swap Neighborhood is generated by swapping two jobs between two machines. Note that the swapped jobs will be placed in the same position on the other machine, i.e. the original positions are taken into account. This Move first removes the two jobs and then reinserts them in the equivalent positions.

`accept()`

This method must be called whenever the modification made by this move is accepted. It ensures that the solution as well as other structures are updated accordingly.

Return type
`None`

`do_move(`*solution*`)`

This method returns does the move and returns the impact (delta cost) in the solution.

Parameters
`solution` (`Solution`) – The solution to be modified.

Returns
The impact (delta cost) of this move in the solution.

Return type
float

`property engine_1: Engine`

The first engine reference.

Type
Engine

Return type
Engine

property engine_2: *Engine*

The second reference.

Type
Engine

Return type
Engine

gen_move(*solution*)

This method generates a random candidate for the movement that must be subsequently validated by has_move().

Parameters
`solution` (`Solution`) – The solution to be modified.

Return type
`None`

has_move(*solution*)

This method returns a boolean indicating whether this neighborhood can be applied to the current solution.

Parameters
`solution` (`Solution`) – The solution to be evaluated.

Returns

True if this neighborhood can be applied to the current
solution, False otherwise.

Return type
bool

property job_1: `Tuple[int, str]`

The selected job from the first engine route.

Type
Tuple[int, str]

Return type
`Tuple`[`int`, `str`]

property job_2: `Tuple[int, str]`

The selected job from the second engine route.

Type
Tuple[int, str]

Return type
`Tuple`[`int`, `str`]

property pos_1: `Optional[int]`

The index of the first job.

Type
Optional[int]

Return type
`Optional`[`int`]

property pos_2: `Optional[int]`

The index of the second job.

Type
Optional[int]

Return type
`Optional[int]`

`reject()`

This method must be called whenever the modification made by this move are rejected. It ensures that the solution as well as other structures are updated accordingly.

Return type
`None`

`reset()`

This method is called whenever the neighborhood should be reset (mainly to avoid the need of creating another object).

Return type
`None`

`property route_1: List[Tuple[int, str]]`

The route of the first engine.

Type
List[Tuple[int, str]]

Return type
`List[Tuple[int, str]]`

`property route_2: List[Tuple[int, str]]`

The route of the second engine.

Type
List[Tuple[int, str]]

Return type
`List[Tuple[int, str]]`

SmartShift

`class SmartShift(`*problem*, *constructive*`)`

This class represents a Smart Shift Move. A neighbor in the Smart Shift Move is generated by re-scheduling one job from a machine with the largest total execution time to another position in the machine.

`accept()`

This method must be called whenever the modification made by this move is accepted. It ensures that the solution as well as other structures are updated accordingly.

Return type
`None`

`do_move(`*solution*`)`

This method returns does the move and returns the impact (delta cost) in the solution.

Parameters
`solution` (`Solution`) – The solution to be modified.

Returns
The impact (delta cost) of this move in the solution.

Return type
float

`property engine_id: Optional[int]`

The engine id.

Type
Optional[int]

Return type
`Optional[int]`

`gen_move(`*solution*`)`

This method generates a random candidate for the movement that must be subsequently validated by has_move().

Parameters
`solution` (`Solution`) – The solution to be modified.

Return type
`None`

`has_move(`*solution*`)`

This method returns a boolean indicating whether this neighborhood can be applied to the current solution.

Parameters
`solution` (`Solution`) – The solution to be evaluated.

Returns

True if this neighborhood can be applied to the current solution, False otherwise.

Return type
bool

`property job: Tuple[int, str]`

The selected job from the engine route.

Type
Tuple[int, str]

Return type
`Tuple[int, str]`

`property make_span: List[Tuple[float, int]]`

List containing a tuple with the makespan and the id of each engine.

Type
List[Tuple[float, int]]

Return type
`List[Tuple[float, int]]`

`property pos: Optional[int]`

The index of the job.

Type
Optional[int]

Return type
`Optional[int]`

`reject()`

This method must be called whenever the modification made by this move are rejected. It ensures that the solution as well as other structures are updated accordingly.

Return type
`None`

`reset()`

This method is called whenever the neighborhood should be reset (mainly to avoid the need of creating another object).

Return type
`None`

`property route: List[Tuple[int, str]]`

The route of the engine.

Type
List[Tuple[int, str]]

Return type
`List[Tuple[int, str]]`

SmartSimpleSwap

`class SmartSimpleSwap(`*problem*, *constructive*`)`

This class represents a Smart Simple Swap Move. A neighbor in the Smart Simple Swap Neighborhood is generated by swapping two jobs between two machines, with at least one of these machines has the largest total execution time. Note that the swapped jobs will be placed in the same position on the other machine, i.e. the original positions are taken into account. This Move first removes the two jobs and then reinserts them in the equivalent positions.

`accept()`

This method must be called whenever the modification made by this move is accepted. It ensures that the solution as well as other structures are updated accordingly.

Return type
`None`

`do_move(`*solution*`)`

This method returns does the move and returns the impact (delta cost) in the solution.

Parameters
`solution` (`Solution`) – The solution to be modified.

Returns
The impact (delta cost) of this move in the solution.

Return type
float

`property engine_1_id: Optional[int]`

The first engine id.

Type
Optional[int]

Return type
`Optional[int]`

`property engine_2_id: Optional[int]`

The second engine id.

Type
Optional[int]

Return type
`Optional[int]`

`gen_move(`*solution*`)`

This method generates a random candidate for the movement that must be subsequently validated by has_move().

Parameters
`solution` (`Solution`) – The solution to be modified.

Return type
`None`

`has_move(`*solution*`)`

This method returns a boolean indicating whether this neighborhood can be applied to the current solution.

Parameters
`solution` (`Solution`) – The solution to be evaluated.

Returns

True if this neighborhood can be applied to the current solution, False otherwise.

Return type
bool

`property job_1: Tuple[int, str]`

The selected job from the first engine route.

Type
Tuple[int, str]

Return type
`Tuple[int, str]`

`property job_2: Tuple[int, str]`

The selected job from the second engine route.

Type
Tuple[int, str]

Return type
`Tuple[int, str]`

`property make_span: List[Tuple[float, int]]`

List containing a tuple with the makespan and the id of each engine.

Type
List[Tuple[float, int]]

Return type
`List[Tuple[float, int]]`

`property pos_1: Optional[int]`

The index of the first job.

Type
Optional[int]

Return type
`Optional[int]`

`property pos_2: Optional[int]`

The index of the second job.

Type
Optional[int]

Return type
`Optional[int]`

`reject()`

This method must be called whenever the modification made by this move are rejected. It ensures that the solution as well as other structures are updated accordingly.

Return type
`None`

`reset()`

This method is called whenever the neighborhood should be reset (mainly to avoid the need of creating another object).

Return type
`None`

`property route_1: List[Tuple[int, str]]`

The route of the first engine.

Type
List[Tuple[int, str]]

Return type
`List[Tuple[int, str]]`

`property route_2: List[Tuple[int, str]]`

The route of the second engine.

Type
List[Tuple[int, str]]

Return type
`List[Tuple[int, str]]`

SmartSwap

`class SmartSwap(`*problem*, *constructive*`)`

This class represents a SmartSwap Move. A neighbor in the SmartSwap Neighborhood is generated by swapping two jobs between two machines, with at least one of these machines has the largest total execution time. Note that the swapped jobs may be placed in any position on the other machine, i.e. the original positions are not taken into account. This Move first removes the two jobs and then reinserts them in random positions.

`accept()`

This method must be called whenever the modification made by this move is accepted. It ensures that the solution as well as other structures are updated accordingly.

Return type
`None`

`do_move(`*solution*`)`

This method returns does the move and returns the impact (delta cost) in the solution.

Parameters
`solution` (`Solution`) – The solution to be modified.

Returns
The impact (delta cost) of this move in the solution.

Return type
float

`property engine_1_id: Optional[int]`

The first engine id.

Type
Optional[int]

Return type
`Optional[int]`

`property engine_2_id: Optional[int]`

The second engine id.

Type
Optional[int]

Return type
`Optional[int]`

`gen_move(`*solution*`)`

This method generates a random candidate for the movement that must be subsequently validated by has_move().

Parameters
`solution` (`Solution`) – The solution to be modified.

Return type
`None`

`has_move(`*solution*`)`

This method returns a boolean indicating whether this neighborhood can be applied to the current solution.

Parameters
`solution` (`Solution`) – The solution to be evaluated.

Returns

True if this neighborhood can be applied to the current solution, False otherwise.

Return type
bool

`property job_1: Tuple[int, str]`

The selected job from the first engine route.

Type
Tuple[int, str]

Return type
`Tuple[int, str]`

`property job_2: Tuple[int, str]`

The selected job from the second engine route.

Type
Tuple[int, str]

Return type
`Tuple[int, str]`

`property make_span: List[Tuple[float, int]]`

List containing a tuple with the makespan and the id of each engine.

Type
List[Tuple[float, int]]

Return type
List[Tuple[float, int]]

property pos_1: Optional[int]

The index of the first job.

Type
Optional[int]

Return type
Optional[int]

property pos_2: Optional[int]

The index of the second job.

Type
Optional[int]

Return type
Optional[int]

reject()

This method must be called whenever the modification made by this move are rejected. It ensures that the solution as well as other structures are updated accordingly.

Return type
None

reset()

This method is called whenever the neighborhood should be reset (mainly to avoid the need of creating another object).

Return type
None

property route_1: List[Tuple[int, str]]

The route of the first engine.

Type
List[Tuple[int, str]]

Return type
List[Tuple[int, str]]

property route_2: List[Tuple[int, str]]

The route of the second engine.

Type
List[Tuple[int, str]]

Return type
List[Tuple[int, str]]

SmartSwitch

class SmartSwitch(*problem*, *constructive*)

This class represents a SmartSwitch Move. A neighbor in the SmartSwitch Move structure is generated by switching the order of two jobs of a machine with the largest total execution time.

accept()

This method must be called whenever the modification made by this move is accepted. It ensures that the solution as well as other structures are updated accordingly.

Return type
None

`do_move(`*solution*`)`

This method returns does the move and returns the impact (delta cost) in the solution.

Parameters
`solution` (`Solution`) – The solution to be modified.

Returns
The impact (delta cost) of this move in the solution.

Return type
float

`property engine_id: Optional[int]`

The engine id.

Type
Optional[int]

Return type
`Optional[int]`

`gen_move(`*solution*`)`

This method generates a random candidate for the movement that must be subsequently validated by has_move ().

Parameters
`solution` (`Solution`) – The solution to be modified.

Return type
`None`

`has_move(`*solution*`)`

This method returns a boolean indicating whether this neighborhood can be applied to the current solution.

Returns

True if this neighborhood can be applied to the current
solution, False otherwise.

Return type
bool

`property job_1: Optional[int]`

The index of the first job.

Type
Optional[int]

Return type
`Optional[int]`

`property job_2: Optional[int]`

The index of the second job.

Type
Optional[int]

Return type
`Optional[int]`

`property make_span: List[Tuple[float, int]]`

List containing a tuple with the makespan and the id of each engine.

Type
List[Tuple[float, int]]

Return type
List[Tuple[float, int]]

`reject()`

This method must be called whenever the modification made by this move are rejected. It ensures that the solution as well as other structures are updated accordingly.

Return type
None

`reset()`

This method is called whenever the neighborhood should be reset (mainly to avoid the need of creating another object).

Return type
None

Classes

Engine

`class Engine(`*id*, *speed_stack*, *speed_reclaim*, *pos_ini*, *rail*, *yards*`)`

This class represents an Engine or a Machine. The attributes indicate which rail the equipament is on, the yards to which it has access and its working configuration, whether as a stacker, reclaimer or both.

`property id: int`

The Engine identifier.

Type
int

Return type
int

`property pos_ini: int`

The starting position of the engine.

Type
int

Return type
int

`property rail: int`

The rail to which the engine is attached.

Type
int

Return type
int

`property speed_reclaim: float`

The reclaiming speed of the engine. If this attribute is different from zero, then the equipment can perform the reclaiming function.

Type
float

Return type
float

property speed_stack: float

The stacking speed of the engine. If this attribute is different from zero, then the equipment can perform the stacking function.

Type
float

Return type
float

property yards: List[int]

List with the ore yards that the engine has access to.

Type
List[int]

Return type
List[int]

Input

class Input(*id*, *weight*, *quality*, *time*)

This class represents an ore Input. The attributes indicate the weight of ore available in the input, the quality parameters of these ores and when the input is available.

property id: int

The input identifier.

Type
int

Return type
int

property quality: List[*Quality*]

List with the quality parameters.

Type
List[*Quality*]

Return type
List[*Quality*]

property time: float

Time when input is available.

Type
float

Return type
float

property weight: float

The input ore weight.

Type
float

Return type
float

Output

`class Output(`*id*, *destination*, *weight*, *quality*, *time*`)`

This class represents an ore Output. The attributes indicate the weight of ore requested in the output, the quality parameters of these ores and when the output request can be started.

`property destination: int`

The destination identifier of the output.

Type
int

Return type
`int`

`property id: int`

The output identifier.

Type
int

Return type
`int`

`property quality: List[`*`Request`*`]`

List with the requested quality parameters.

Type
List[*Request*]

Return type
`List[`*`Request`*`]`

`property time: float`

Time when the output request can be started.

Type
float

Return type
`float`

`property weight: float`

The weight of ore requested in the output.

Type
float

Return type
`float`

Quality

`class Quality(`*parameter*, *value*`)`

This class represents a Quality parameter. The attributes indicate the parameter name and its percentage.

`property parameter: str`

The quality parameter name.

Type
str

Return type
`str`

`property value: float`

The percentage of the quality parameter.

Type
float

Return type
`float`

Request

`class Request(`*parameter*, *minimum*, *maximum*, *goal*, *importance*`)`

This class represents a Quality Request parameter. The attributes indicate the parameter name, its maximun and minium percentage, the goal percentage and its importance on the request.

`property goal: float`

The goal percentage of the quality parameter.

Type
float

Return type
`float`

`property importance: int`

The importance of the quality parameter.

Type
int

Return type
`int`

`property maximum: float`

The maximum percentage of the quality parameter.

Type
float

Return type
`float`

`property minimum: float`

The minimum percentage of the quality parameter.

Type
float

Return type
`float`

Stockpile

`class Stockpile(`*id*, *position*, *yard*, *rails*, *capacity*, *weight_ini*, *quality_ini*`)`

This class represents an ore Stockpile. The attributes indicate the stockpile position, the yard where its located, the rails that have access to it, its ore capacity, initial weight and quality parameters.

`property capacity: float`

The stockpile ore capacity.

Type
float

Return type
`float`

property id: int

The stockpile identifier.

Type
int

Return type
int

property position: int

The stockpile position.

Type
int

Return type
int

property quality_ini: List[*Quality*]

List of quality parameters presents in the stockpile.

Type
List[*Quality*]

Return type
List[*Quality*]

property rails: List[int]

List of rails that have access to the stockpile.

Type
List[int]

Return type
List[int]

property weight_ini: float

The stockpile initial weight.

Type
float

Return type
float

property yard: int

The yard where the stockpile is located.

Type
int

Return type
int

Problem

class Problem(*instance_path*)

This class represents a Problem that will be used to build the Ore Mixing Problem and the Machine Scheduling Problem from a .json file and, for that, the UltraJSON packege is necessary.

UltraJSON is an ultra fast JSON encoder and decoder written in pure C with bindings for Python 3.5+. To install it just run pip as usual on the command prompt:

$ pip install ujson

For more information, access https://pypi.org/project/ujson/.

property distances_travel: List[List[float]]

Matrix with the distances between each stockpile.

Type
List[List[float]]

Return type
List[List[float]]

property engines: List[*Engine*]

List with engine data.

Type
List[*Engine*]

Return type
List[*Engine*]

property info: List[Union[str, int]]

List with the instance name and the omega values for the linear model.

Type
List[Union[str, int]]

Return type
List[Union[str, int]]

property inputs: List[*Input*]

List with ore input data.

Type
List[*Input*]

Return type
List[*Input*]

property outputs: List[*Output*]

list with ore output data.

Type
List[*Output*]

Return type
List[*Output*]

property stockpiles: List[*Stockpile*]

List with stockpile data.

Type
List[*Stockpile*]

Return type
List[*Stockpile*]

property time_travel: List[List[float]]

Matrix with the time needed to travel from one stockpile to another.

Type
List[List[float]]

Return type
List[List[float]]

Solution

`class Solution(`*problem*`)`

This class represents a Solution of the Machine Scheduling Problem and the Ore Mixing Problem.

`property cost: float`

The solution cost.

Type
float

Return type
`float`

`property deliveries: List[Dict[str, Union[float, List[Dict[str, Union[str, int, float]]]]]]`

List[Dict[str, Union[float, List[Dict[str, Union[str, int, float]]]]]]: List of dictionaries with the final results, such as delivery time, ore weight, quality parameters and other information for each request.

Return type
`List[Dict[str, Union[float, List[Dict[str, Union[str, int, float]]]]]]`

`property gap: List[float]`

List with the gap between the optimal and current delivery duration. The indexes are associated with the IDs of each output.

Type
List[float]

Return type
`List[float]`

`property has_deliveries: bool`

Flag that indicates whether deliveries are defined before writing them to a file.

Type
bool

Return type
`bool`

`property inputs: Dict[str, List[float]]`

dictionary of entries whose keys are the stockpiles IDs and values are lists with the stacked weights.

Type
Dict[str, List[float]]

Return type
`Dict[str, List[float]]`

`property objective: Optional[float]`

The solution objective value.

Type
Optional[float]

Return type
`Optional[float]`

`property problem: `*`Problem`*

The problem considered.

Type
Problem

Return type
Problem

property reclaims: List[Dict[str, Union[int, float]]]

Dictionary with the reclaiming data to be recorded in a .json file, in which the keys are the names of the attributes and the values are their information.

Type
List[Dict[str, Union[int, float]]]

Return type
List[Dict[str, Union[int, float]]]

reset()

This method is called whenever the solution should be reset (mainly to avoid the need of creating another object).

Return type
None

property routes: List[List[Tuple[int, str]]]

Matrix of tuples whose the first element is the stockpile position and the last is the engine configuration in that stockpile. Each line indicates an engine, based on its IDs.

Type
List[List[Tuple[int, str]]]

Return type
List[List[Tuple[int, str]]]

set_deliveries()

This method saves the output data for each order, which consists of the total mass ordered, at the time the order was initiated and its duration, as well as the quality of each ore delivered.

Return type
None

set_objective(*objective*)

This method sets the objectives for the Machine Scheduling Problem from the results of Ore Mixing Problem.

Parameters
objective (*Tuple[Optional[float], Dict[str, List[float]], Dict[str, List[float]]]*) – A tuple whose first element is the objective value of the linear model, the second element is a dictionary of entries whose keys are the stockpiles IDs and values are lists with the stacked weights, the last element is a dictionary of recoveries whose keys are the IDs of each request and the values are lists with the reclaimed weights.

Return type
None

property stacks: List[Dict[str, Union[int, float]]]

Dictionary with the stacking data to be recorded in a .json file, in which the keys are the names of the attributes and the values are their information.

Type
List[Dict[str, Union[int, float]]]

Return type
List[Dict[str, Union[int, float]]]

property start_time: List[float]

List with the time when each engine can start a new task. The indexes are associated with the IDs of each engine.

Type
List[float]

Return type
List[float]

update_cost(*id*)

This method calculates and updates the solution cost.

Parameters
id (*int*) – The request identifier.

Return type
None

property weights: Dict[str, List[float]]

dictionary of recoveries whose keys are the IDs of each request and the values are lists with the reclaimed weights.

Type
Dict[str, List[float]]

Return type
Dict[str, List[float]]

work_time(*id*)

This method calculates and returns the time the request was initiated and completed.

Parameters
id (*int*) – The request identifier.

Returns

A tuple which the first value is the start
time and the last value is the end time.

Return type
Tuple[float, float]

write(*file_path*)

This method writes the solution in a .json file and, for that, the UltraJSON packege is necessary.

UltraJSON is an ultra fast JSON encoder and decoder written in pure C with bindings for Python 3.5+. To install it just run pip as usual on the command prompt:

$ pip install ujson

For more information, access https://pypi.org/project/ujson/.

Parameters
file_path (*str*) – The output file path.

Return type
None