UniFil

CIÊNCIA DA COMPUTAÇÃO

MATHEUS MURIEL

ANÁLISE DA EFICIÊNCIA DA IMPLEMENTAÇÃO DE COMPONENTE DINÂMICO EM ALGORITMOS DE ENXAME DE PARTÍCULAS APLICADOS AO PROBLEMA DE JOB SHOP FLEXÍVEL

Londrina

2021

MATHEUS MURIEL

ANÁLISE DA EFICIÊNCIA DA IMPLEMENTAÇÃO DE COMPONENTE DINÂMICO EM ALGORITMOS DE ENXAME DE PARTÍCULAS APLICADOS AO PROBLEMA DE JOB SHOP FLEXÍVEL

Trabalho de conclusão de curso apresentado ao Centro Universitário Filadélfia como parte dos requisitos para obtenção de graduação em Ciência da Computação. Orientador: Ricardo Inacio Alvares e Silva.

Londrina

2021

MATHEUS MURIEL

ANÁLISE DA EFICIÊNCIA DA IMPLEMENTAÇÃO DE COMPONENTE DINÂMICO EM ALGORITMOS DE ENXAME DE PARTÍCULAS APLICADOS AO PROBLEMA DE JOB SHOP FLEXÍVEL

Trabalho de conclusão de curso apresentado à banca examinadora do curso de Ciência da Computação do Centro Universitário Filadélfia de Londrina em cumprimento a requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

APROVADO PELA **COMISSÃO EXAMINADORA**
EM LONDRINA, 2021.

_____
Ricardo Inacio Alvares e Silva - Orientador

_____
Professor 1 - Marc Antonio Queiroz

_____
Professor 2 - Kleber Marcio de Souza

AGRADECIMENTOS

Gostaria de agradecer primeiramente ao meu orientador Prof. Ricardo por ter me guiado nessa pesquisa sendo sempre muito esclarecedor e puxando meu pé quando necessário, além de durante a graduação ter sido o professor que mais me ensinou e que me fez sempre ter uma fome de aprender cada vez mais, por não ter sido somente um professor, mas sim um verdadeiro mestre. Também gostaria de agradecer aos professores Marc, Mario, Kaneshima, Tânia, Simone e Sérgio por durante esses 5 anos de graduação terem sidos excelentes mentores e feito eu me apaixonar pela computação. Ao grupo de estudos em Inteligencia Artificial guiado pelos professores Sérgio Tanaka e Luppércio Luppi por me dar ajuda e embasamento teórico, pelo networking e pelas oportunidades. Agradecer a minha namorada Isabeli por ter me ajudado a me manter motivado durante este longo projeto e também ao longo do curso. A minha prima Stephanie por insistir que eu tentasse uma bolsa e entrasse na faculdade, ao ProUni por tornar possível que eu tenha estudado em uma excelente universidade. E por fim agradecer a minha mãe Cristina Muriel por todo o seu esforço para garantir que eu tenha tido boa educação e por estar sempre ao meu lado nos momentos mais difíceis.

"Sem requisitos ou design, a programação é a arte de adicionar erros a um arquivo de texto vazio."
(Louis Srygley)

MURIEL; MATHEUS, F. **Análise da eficiência da implementação de componente dinâmico em algoritmos de enxame de partículas aplicados ao problema de Job Shop Flexível**. Trabalho de Conclusão de Curso (Graduação) - Centro Universitário Filadélfia. Londrina, 2021.

RESUMO

O *Flexible Job Shop Problem* (FJSP) é um complexo problema de otimização de análise combinatória pertencente à classe de problemas NP-Hard. Esse problema consiste em um cenário onde existem diversos conjuntos de tarefas, chamadas *jobs*, e diversas máquinas diferentes que podem processar essa tarefa, e tem como objetivo encontrar um agendamento que execute todas as operações na menor quantidade de tempo possível. Por se tratar de um problema NP-Hard não é possível encontrar a melhor solução, então ao longo do tempo foram propostos diversos tipos de algoritmos para encontrar uma solução boa o suficiente para o problema, uma dessas abordagens são os algoritmos bio inspirados nos quais se utilizam padrões observados na natureza para resolver problemas. Um desses padrões observados na natureza é o comportamento de enxames de abelhas e o movimento migratório de bandos de aves, o comportamento desses bandos é usado como heurística no algoritmo *Particle Swarm Optimization* (PSO) onde cada individuo de uma população de partículas utiliza a média entre sua melhor posição já visitada (*pBest*) e a melhor posição já visitada entre todos da população (*gBest*) para obter uma nova posição, assim a cada rodada a população converge para um resultado ótimo. Porém, em alguns casos o algoritmo PSO pode convergir prematuramente para um mínimo local e chegar a uma solução não ótima. Já foram propostas modificações dinâmicas no PSO para resolver o problema de FJSP, mas apenas em cenários multiobjetivo, ou seja, com mais de um critério de avaliação de qualidade. Esse trabalho foca no cenário mono objetivo e descreve a implementação, testes e resultados de uma nova abordagem, introduzindo componentes de inércia dinâmicos nas partículas e obteve resultados que demonstram uma redução de $3,90\%$ no tempo final de *makespan*, de $16,63\%$ no desvio padrão e de $33,76\%$ na variância. Uma significativa melhora na confiabilidade e na chance de se obter melhores resultados na execução do algoritmo em comparação com PSO base.

Palavras-chave: PSO; Otimização por enxame de partículas; Particle Swarm Optimization; FJSP; Flexible Job Shop Problem; Inteligência Populacional; Componentes dinâmicos.

MURIEL; MATHEUS, F. **Análise da eficiência da implementação de componente dinâmico em algoritmos de enxame de partículas aplicados ao problema de Job Shop Flexível**. Trabalho de conclusão de curso (Graduação) - Centro Universitário Filadélfia. Londrina, 2021.

ABSTRACT

The Flexible Job Shop Problem (FJSP) is a complex combinatorial analysis optimization problem belonging to the class of NP-Hard problems. This problem consists of a scenario where there are several sets of tasks, called jobs, and several different machines that can process this task, and its objective is to find a schedule that performs all operations in the shortest amount of time possible. As it is an NP-Hard problem, it is not possible to find the best solution, so over time, several types of algorithms have been proposed to find a good enough solution to the problem, one of these approaches is the bio-inspired algorithms in which they use patterns observed in nature to solve problems. One of these patterns observed in nature is the behavior of bee swarms and the migratory movement of flocks of birds, the behavior of these flocks is used as a heuristic in the Particle Swarm Optimization (PSO) algorithm where each individual of a population of particles uses the average between its best position ever visited (pBest) and the best position already visited among all of the population (gBest) to obtain a new position so that each round the population converges to an optimal result. However, in some cases, the PSO algorithm may prematurely converge to a local minimum and arrive at a non-optimal solution. Dynamic modifications in the PSO have already been proposed to solve the FJSP problem, but only in multi-objective scenarios, that is, with more than one quality assessment criterion. This work focuses on the single objective scenario and describes the implementation, tests, and results of a new approach, introducing dynamic inertial components into particles and obtaining results that demonstrate a reduction of 3.90% in the final time of makespan, 16.63% for the standard deviation and 33.76% for the variance. A significant improvement in reliability and the chance of obtaining better results in the execution of the algorithm in comparation with the standard PSO.

***Keywords**: PSO; Particle Swarm Optimization; FJSP; Flexible Job Shop Problem; Population Intelligence; Dynamic components.*

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE SIGLAS

FJSP	Flexible Job Shop Problem.
JSP	Job Shop Problem.
PFJSP	Parcial Flexible Job Shop Problem.
PSO	Particle Swarm Optimization.
TFJSP	Total Flexible Job Shop Problem.

LISTA DE ALGORITMOS

SUMÁRIO

1 INTRODUÇÃO

Na sociedade contemporânea, a crescente demanda pelas mais variadas categorias de produtos tem tornado necessário melhorias na produtividade das indústrias, de modo que acompanhe o ritmo de crescimento da demanda.

Essa necessidade acontece devido a um ambiente altamente dinâmico onde a demanda por um produto pode aumentar ou diminuir em um curto espaço de tempo, e a um cenário de grande concorrência entre as indústrias, e que para atender a essas necessidades as indústrias devem buscar meios de otimizar seus mecanismos de produção, e que essa melhoria é importante para o sucesso dessas empresas (WARI; ZHU, 2016).

Segundo Xhafa e Abraham (2008) uma dessas áreas importantes para a produtividade da linha de produção é a programação de produção. A programação de produção é a parte da administração de produção responsável por decidir o local, o momento e a ordem nas quais serão realizadas as operações. Por ser uma areá importante e ter um escopo com variáveis e objetivos bem definidos a programação de produção se mostra como um bom ponto para se otimizar por meio da computação.

De acordo com a Bagchi (1999), o problema de programação de produção se enquadra como um problema de escalonamento, que o autor define como um tipo de problema onde existe um conjunto de demandas e um conjunto de recursos, e o objetivo é encontrar um agendamento que distribua as demandas entre os recursos de forma que o tempo de corrido entre o início e o fim da execução de todas as tarefas (chamado *makespan*) seja o menor possível.

Bagchi (1999) também define algumas sub divisões do problema de escalonamento, dentre elas o *JSP* e uma variação sua o *FJSP*. Que é um problema onde existem m máquinas diferentes entre si e n *jobs* formados por diversas operações O em uma ordem específica.
Cada operação O_{ij} tem um tempo de execução diferente em cada máquina, ou seja, uma operação O_{ij} demora um tempo x se for executada na máquina M_a e um tempo y se for executa na maquina M_b.
Porém, existem algumas restrições no problema de *FJSP* como:

- Não é possível executar simultaneamente duas operações de um mesmo *job*.
- As maquinas são heterogêneas, ou sejam, não são iguais entre si.
- A execução de uma operação é atômica e não podem ser interrompidas.
- As máquinas são podem executar uma operação de cada vez.
- Um *job* não pode ser processado duas vezes.

O número de possibilidades de arranjos pode ser representado por: $(n!)^m$, ou seja, o número de soluções possíveis cresce exponencialmente conforme o número de máquinas e de *jobs*. Esse característica de crescimento exponencial faz com que um problema se torne inviável de ser resolvido de formas tradicionais após um certo tamanho, pois é necessário computar todas as suas possibilidades de soluções. Isso faz com que o *FJSP* seja definido como um problema de otimização com crescimento exponencial, o que de acordo com ESWARAMURTHY; TAMILARASI o define como um problema pertencente a classe de problemas *NP-Hard*.

A classe de problemas *NP-Hard* é composta por problemas cujo a resposta não pode ser encontrada computacionalmente em um tempo polinomial, ou seja, em um tempo aceitável, porém uma solução pode ser verificada em um tempo polinomial (ESWARAMURTHY; TAMILARASI, 2009). Para solucionar problemas da classe *NP-Hard* são geralmente utilizados métodos de aproximação, que visam ao invés de tentar encontrar a melhor solução possível, tentar encontrar uma solução boa o suficiente para o problema em questão. Um dos algoritmos que fazem essa busca por uma solução via aproximação são os algoritmos bio inspirados, que se baseiam técnicas observadas na natureza.

A ideia de se basear em um comportamento biológico visa usar técnicas que passaram pela seleção natural ao longo de milhares de anos, e que como resultado dessa seleção sobraram somente as técnicas mais eficientes. Uma subcategoria dos algoritmos bioinspirados são os algoritmos populacionais, baseados no comportamento de populações de animais como aves, abelhas e insetos. Esses algoritmos visam utilizar o conceito de inteligência de bando, simulando indivíduos interagindo entre si e com o ambiente para chegar a um objetivo. Existem diferentes algoritmos populacionais como o *Ant Colony Optimization*, o *Stochastic Diffusion Search* e o *Particle Swarm Optimization*.

1.1 PARTICLE SWARM OPTIMIZATION (PSO)

O algoritmo PSO, ou Optimização por Enxame de Partículas em tradução para o português, se baseia na convergência de um enxame de partículas em um objetivo. A estrutura básica de um algoritmo *PSO* é uma população de partículas em que cada partícula tem uma velocidade e uma direção, além das informações da sua posição, o quão boa sua posição atual é, qual foi a melhor posição na, qual ela já esteve ($pBest$), e qual foi a melhor posição em que qualquer um dos indivíduos da população já esteve ($gBest$). A cada rodada as partículas se movimentam para uma direção intermediaria entre $pBest$ e $gBest$, por uma distância que é calculada pelo valor de velocidade menos o valor de inércia da partícula multiplicado por um valor aleatório entre 0.1 e 0.9. E assim a cada rodada do algoritmo a população vai chegando cada vez mais perto da melhor solução.

Alguns dos problemas que podem acontecer no *PSO* é uma convergência prematura da população em mínimos locais, que são casos aonde entre as soluções existe uma que é melhor que a média das soluções a sua volta, porém não é o melhor entre todas.
Como a movimentação das partículas se baseia na média entre as melhores posições locais e gerais do grupo, caso a maioria das partículas vá para em direção ao mínimo local o $gBest$ fica no mínimo local, e as partículas tentem a não saírem dessa região.

Algumas medidas podem ser tomadas para diminuir essas possibilidades de convergência Prematura. Como as variáveis de inércia e de velocidade são variáveis individuais que influenciam a movimentação da partícula, é possível utilizar técnicas como valores dinâmicos e pesos relativos para as variáveis, e assim mudar de maneira dinâmica as características de movimentação das partículas.

Nesse trabalho são analisados os fatores de projeto, implementação e a utilização de uma abordagem dinâmicas nas variáveis de partícula e como essas modificações influenciam na convergência do *PSO* aplicado em cenários de aplicação para a solução de problemas do tipo *FJSP*.

Atualmente existem poucos estudos sobre o comportamento do PSO com componentes dinâmicos em cenários mono objetivos, principalmente em problemas com o FJSP. Com as implementações feitas nesse trabalho se espera não uma análise crua de números e métricas de desempenho, mas sim ideias de como certas alterações afetam o desempenho do algoritmo, e quais são as tendências de mudanças.

A nova abordagem e a abordagem padrão do PSO foram testadas em diversos cenários e com tamanhos diferentes de espaços de solução e foram obtidos resultados positivos e promissores, tendo melhores resultados e com uma menor taxa de variância, demostrando que essa abordagem de inércia dinâmica proposta traz uma contribuição significativa para o algoritmo PSO aplicado no problema de FJSP.

2 FUNDAMENTAÇÃO

Nesse capítulo são desenvolvidos os conceitos teóricos tanto dos problemas de agendamento em geral quanto do FJSP, além de descrever o algoritmo PSO.

2.1 PROBLEMAS DE AGENDAMENTO

Problemas de agendamento são muito comuns atualmente, pois estão diretamente presentes em diversos cenários, tais como planejamento de produção, sistemas de manufatura, linhas de montagem, processamento de informação, planejamento e gerenciamento de rotas logísticas e mesmo que de maneira indireta, também estão presentes nos computadores através de escalonadores de processos do sistema operacional. Com tantos possíveis cenários de aplicação é esperado que existam diversas variações deste problema, cada uma com suas peculiaridades.

Porém, a maioria destes problemas tem características em comum. Geralmente eles se baseiam em um conjunto de recursos e um conjunto de demandas, e tem como objetivo organizar as execuções dessas demandas e distribuí-las entre os recursos, de modo a alcançar da melhor maneira possível um ou mais objetivos, que podem ser, por exemplo:

- O tempo total para o algoritmo executar o algoritmo e encontrar uma solução, ou seja, um agendamento.
- O tempo total decorrido entre a primeira e a última tarefa do agendamento, também chamado *fitness* ou *makespan*.
- Ou a soma do tempo de ociosidade das máquinas durante o período do agendamento.

Devido a esta característica de buscar um melhor resultado para atender a um objetivo os problemas de agendamento são caracterizados como problemas de otimização, problemas nos quais o objetivo é encontrar uma solução que melhor atenda os critérios de avaliação do problema em questão.

Os problemas de otimização são caracterizados por terem um número muito grande de soluções possíveis, algumas vezes tendendo ao infinito, o que torna inviável a computação de todas essas possibilidade. Por causa dessa característica um algoritmo de otimização não necessariamente precisa encontrar a melhor solução possível entre todas as possibilidades, mas sim uma solução boa o suficiente, também chamado solução ótima, conforme os critérios de avaliação do problema.

Os problemas de otimização podem ser classificados em dois tipos, sendo eles problemas "mono-objetivos", e problemas "multi-objetivos".

Os problemas do tipo multi-objetivo consideram simultaneamente, mais de um critério de avaliação, podendo ter ou não o mesmo peso (nível de importância), e espera se que sejam atingidos de forma satisfatória todos os critérios de avaliação. Por causa dessa diversidade de objetivos que podem muitas vezes ser conflitantes entre si, por exemplo, ser mais rápido e gastar a menos energia, problemas do tipo multi-objetivos são os mais comuns.

Já os problemas do tipo mono-objetivo lidam somente com um critério de avaliação, o que diminui significativamente a dificuldade para encontrar uma solução, porém, em contrapartida, espera se um resultado melhor para o objetivo em questão, sendo necessário um maior nível de refinamento da solução.

Como foi definido por BAGCHI um problema de agendamento (também chamado problema de escalonamento) é um problema de otimização aonde diversas tarefas (chamadas *jobs*) são alocados em uma ordem sequencial entre os recursos (máquinas), com diferentes tarefas podendo ser executadas simultaneamente em máquinas diferentes, porém cada atividade devendo ser executada somente em uma máquina por vez.

Assim como a grande maioria dos problemas de origem combinatória, o problema de escalonamento pertence à classe de problemas NP-Hard, onde não é possível se encontrar a melhor solução em um tempo polinomial, ou seja, um tempo computacionalmente aceitável.

Isso acontece devido à quantidade de operações necessárias para verificar todas as possibilidades de soluções para o problema em questão, crescerem de forma exponencial com base no tamanho do problema, o que torna os problemas dessa classe inviáveis de serem resolvidos através de métodos comuns de cálculos.

Por causa disso, nos casos de problemas NP-Hard são normalmente utilizados algoritmos de aproximação, que tenham um tempo de execução razoável e consigam encontrar uma solução boa o suficiente, a chamada "Solução Ótima", com base nos critérios de avaliação do problema, como visto por (LAWLER et al., 1993).

As características de uma solução ótima variam conforme o problema, em alguns casos é mais importante que a solução seja encontrada em um curto espaço de tempo, do que seja alguns milissegundos mais rápida do que as outras soluções já encontradas pelo algoritmo. Isso se aplica, por exemplo, em ambientes onde o agendamento deve ser feito em tempo real. Já em um cenário onde essa solução precisa ser encontrada apenas uma vez no dia e depois ser aplicada varias vezes, como em uma estrutura de linha de montagem, pode ser vantajoso esperar alguns minutos a mais se essa solução economizar tempo de produção (e consequentemente dinheiro) ao longo de seu uso. Então esse critério de avaliação que define uma solução ótima deve ser muito bem analisado de caso a caso.

Nesses ambientes de soluções aproximadas, alguns algoritmos que se destacam são os algoritmos bio inspirados como algoritmos populacionais e algoritmos

evolutivos. Que consistem em simular uma inteligência biológica ou comportamentos já observados na natureza e selecionados pela evolução, para encontrar uma solução ótima para o problema em questão, de maneira similar a qual um ser vivo faria.

Vários autores já classificaram diversas variações de problemas de escalonamento, dentre eles: (GRAHAM et al., 1979), (LENSTRA; Rinnooy Kan, 1979), e (MACCARTHY; LIU, 2007). E o que diferencia esses problemas é o fluxo dos *jobs* a serem processados e a capacidade dos recursos.
Dentre esses problemas estão:

- ***Open Shop:*** Os *job* não tem uma ordem de execução, porém as operações de cada *job* tem uma ordem específica de execução.
- ***Flow Shop:*** Os *jobs* devem ser executados em um fluxo unidirecional e em somente uma máquina. E não exite uma divisão do *job* em operações.
- ***Flexible Flow Shop:*** Semelhante ao *Flow Shop*, porém os *jobs* podem ser divididos em operações.
- ***Job Shop:*** Diferentemente do *Flow Shop* o *Job Shop* pode ter execuções paralelas, assim como ser dividido em operações.
- ***Flexible Job Shop:*** Uma extensão do *Job Shop* onde as operações de cada *job* podem ser executados em máquinas diferentes. Esse problema tem duas sub divisões:
  - ***Flexible Job Shop Total***: Em que todas as máquinas podem executar todas as operações.
  - ***Flexible Job Shop Parcial***: Em que existem limitações para quais máquinas podem executar quais operações.

Além disso, cada um dos problemas acima podem ser classificado em mono-objetivos ou multi-objetivos. Esse trabalho é focado no problema ***Flexible Job Shop Total***, com um contexto mono-objetivo.

2.1.1 Problema de Job Shop — JSP

Como citado anteriormente o JSP é um problema de escalonamento de tarefas pertencente à classe de problemas NP-Hard.

Como visto por Cheng, Gen e Tsujimura (1996) o problema de *Job Shop* desperta muito interesse de pesquisadores por ser um problema com diversas aplicações no mundo real, e em vários cenários diferentes, como industria, computação e manufatura.

De acordo com (CHENG; GEN; TSUJIMURA, 1996) o problema de *Job Shop*

consiste em m máquinas distintas e n *jobs* diferentes entre si, sendo cada *job* formado por diversas operações O em uma ordem específica, e cada operação O_{ij} tem seu respectivo tempo de execução.

Como visto por (BAGCHI, 1999) existem algumas restrições no problema de *Job Shop* dentre elas estão:

- Não é possível executar simultaneamente duas operações de um mesmo *job*.
- Não existe mais de uma máquina de um mesmo tipo.
- As máquinas podem ficar ociosas durante o período de escalonamento.
- As execuções de um *job* são atômicas e não podem ser interrompidas.
- Uma máquina não pode executar mais de uma operação simultaneamente.
- Um *job* não é processado duas vezes na mesma máquina.
- Não é possível interromper a execução de uma operação.

Para ser possível que um algoritmo encontre uma solução para o agendamento, ele precisar conhecer algumas informações, dentre elas:

- Quantas máquinas existem;
- Quantos *jobs* existem;
- Quantas operações cada *job* possui;
- Em qual máquina cada operação pode ser executada;
- Quanto tempo cada operação leva para ser executada em sua respectiva máquina;

Na Tabela 1 é possível ver um exemplo de uma instância de problema de *Job Shop*. A onde existem duas máquinas $M_1,$ e M_2, dois *jobs* $J_1,$ e J_2 e que cada *job* possui duas operações $O_{j,1}$ e $O_{j,2}$.

Tabela 1 – Exemplo de instância de um problema *Job Shop Problem*

	Operação 1		Operação 2	
Job	Máquina	Tempo	Máquina	Tempo
J_1	M_2	9	M_1	5
J_1	M_1	1	M_2	7

Sendo assim:

- A operação $O_{1,1}$ na máquina M_2 tem o tempo de execução 9

- A operação $O_{1,2}$ na máquina M_1 tem o tempo de execução 5
- A operação $O_{2,1}$ na máquina M_1 tem o tempo de execução 1
- A operação $O_{2,2}$ na máquina M_2 tem o tempo de execução 7

2.1.2 Problema de Job Shop Flexível — FJSP

O FJSP é uma extensão do problema de JSP onde é permitido que uma operação seja executada em mais de uma máquina. Então no momento de definir a ordem e a máquina de execução de cada *job* as possibilidades de arranjo são muito maiores, o que aumenta a complexidade por se tratar de um problema exponencial. De um lado isso traz uma maior complexidade para o algoritmo, porém possibilita uma maior número de possíveis soluções e deixa o algoritmo mais flexível para encontrar melhores soluções. Mas por esse aumento de fatores a se considerar o *Flexible Job Shop* é definido como uma extensão mais complexa do *Job Shop*.

Como abordado na Seção 2.1 , existem duas sub divisões do problema de *FJSP*, sendo elas a PFJSP se uma operação só pode ser processada por um certo sub conjunto de máquinas, ou TFJSP caso uma operação possa ser processada por qualquer máquina. E para ambas as sub divisões do *Flexible Job Shop* são aplicadas as mesmas restrições do *Job Shop* com exceção da que diz que um *job* não pode ser processado duas vezes em uma mesma máquina.

As representações do *Flexible Job Shop* são praticamente as mesma do *Job Shop* com exceção da forma de representar o ambiente inicial do problema. Devido a sua característica de ter uma maior possibilidade de alocações entre as máquinas e os *jobs*, uma representação do de um problema de *Flexible Job Shop* precisa definir quanto tempo cada máquina utilizaria para processar cada operação.

No caso de um problema parcial (P-FJSP) é necessário definir os tempos somente das operações que cada máquina pode executar, na Tabela 2 é demonstrado um exemplo de uma representação de uma problema P-FJSP.

Já no caso de um problema de *Flexible Job Shop* Total, ou seja, em que qualquer máquina pode processar qualquer operação, a representação precisa definir o tempo de processamento de todas as operações para todas as máquinas como pode ser visto na Tabela 3 . Na tabela é mostrada uma instância de um problema de *Flexible Job Shop* Total. De tamanho 10×10 retirado do benchmark de (KACEM; HAMMADI; BORNE, 2002).

Nessa representação é possível ver dez máquinas $(M_1, M_2, M_3, ..., M_{10})$ e dez *jobs* $(J_1, J_2, J_3, ..., J_{10})$ onde cada *job* possui três operações (O_{j1}, O_{j2}, O_{j3}).
Cada índice ji representa o tempo Tji de execução de O_{ji}, para a máquina M_k, sendo $k = [1, ..., m]$, onde m é a quantidade de máquinas e n a quantidade de jobs.

A Figura 1 apresenta um Diagrama de Gantt que representa uma solução

Tabela 2 – Exemplo de problema 8×8 de *P-FJSP* de Kacem et al. 2002

Job	O_{ji}	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8
J_1	$O_{1,1}$	5	3	5	3	3	–	10	9
	$O_{1,2}$	10	–	5	8	3	9	9	6
	$O_{1,3}$	–	10	–	5	6	2	4	5
J_2	$O_{2,1}$	5	7	3	9	8	–	9	–
	$O_{2,2}$	–	8	5	2	6	7	10	9
	$O_{2,3}$	–	10	–	5	6	4	1	7
	$O_{2,4}$	10	8	9	6	4	7	–	–
J_3	$O_{3,1}$	10	–	–	7	6	5	2	4
	$O_{3,2}$	–	10	6	4	8	9	10	–
	$O_{3,3}$	1	4	5	6	–	10	–	7
J_4	$O_{4,1}$	3	1	6	5	9	7	8	4
	$O_{4,2}$	12	11	7	8	10	5	6	9
	$O_{4,3}$	4	6	2	10	3	9	5	7
J_5	$O_{5,1}$	3	6	7	8	9	–	10	–
	$O_{5,2}$	10	–	7	4	9	8	6	–
	$O_{5,3}$	–	9	8	7	4	2	7	–
	$O_{5,4}$	11	9	–	6	7	5	3	6
J_6	$O_{6,1}$	6	7	1	4	6	9	–	10
	$O_{6,2}$	11	–	9	9	9	7	6	4
	$O_{6,3}$	10	5	9	10	11	–	10	–
J_7	$O_{7,1}$	5	4	2	6	7	–	10	–
	$O_{7,2}$	–	9	–	9	11	9	10	5
	$O_{7,3}$	–	8	9	3	8	6	–	10
J_8	$O_{8,1}$	2	8	5	9	–	4	–	10
	$O_{8,2}$	7	4	7	8	9	–	10	–
	$O_{8,3}$	9	9	–	8	5	6	7	1
	$O_{8,4}$	9	–	3	7	1	5	8	–

para o problema representado na Figura **??**, que se trata de um problema de tamanho 10×10 do benchmark de (KACEM; HAMMADI; BORNE, 2002).

Nessa solução é possível observar que:

- A máquina M_1 executa as operações $[O_{1,1}, O_{7,1}, O_{2,1}]$.
- Cada *job* utiliza no mínimo duas máquinas para ser executado, como o *job* J_1 executado nas máquinas $[M_1, M_3, M_4]$.
- O *fitness* dessa solução é 7.

O exemplo mais claro de aplicação do problema de *Flexible Job Shop* é na produção industrial e em sistemas de manufatura, o que traz muito interesse econômico para encontrar boas soluções para essa categoria de problema.

Nos tempos modernos onde existe uma grande competição e constantes mudanças e melhorias tecnológicas, a organização e otimização desses processos in-

Tabela 3 – Exemplo de problema 10×10 de *T-FJSP* de Kacem et al. 2002

Job	O_{ji}	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8	M_9	M_{10}
	$O_{1,1}$	1	4	6	9	3	5	2	8	9	5
J_1	$O_{1,2}$	4	1	1	3	4	8	10	4	11	4
	$O_{1,3}$	3	2	5	1	5	6	9	5	10	3
	$O_{2,1}$	2	10	4	5	9	8	4	15	8	4
J_2	$O_{2,2}$	4	8	7	1	9	6	1	10	7	1
	$O_{2,3}$	6	11	2	7	5	3	5	14	9	2
	$O_{3,1}$	8	5	8	9	4	3	5	3	8	1
J_3	$O_{3,2}$	9	3	6	1	2	6	4	1	7	2
	$O_{3,3}$	7	1	8	5	4	9	1	2	3	4
	$O_{4,1}$	5	10	6	4	9	5	1	7	1	6
J_4	$O_{4,2}$	4	2	3	8	7	4	6	9	8	4
	$O_{4,3}$	7	3	12	1	6	5	8	3	5	2
	$O_{5,1}$	7	10	4	5	6	3	5	15	2	6
J_5	$O_{5,2}$	5	6	3	9	8	2	8	6	1	7
	$O_{5,3}$	6	1	4	1	10	4	3	11	13	9
	$O_{6,1}$	8	9	10	8	4	2	7	8	3	10
J_6	$O_{6,2}$	7	3	12	5	4	3	6	9	2	15
	$O_{6,3}$	4	7	3	6	3	4	1	5	1	11
	$O_{7,1}$	1	7	8	3	4	9	4	13	10	7
J_7	$O_{7,2}$	3	8	1	2	3	6	11	2	13	3
	$O_{7,3}$	5	4	2	1	2	1	8	14	5	7
	$O_{8,1}$	5	7	11	3	2	9	8	5	12	8
J_8	$O_{8,2}$	8	3	10	7	5	13	4	6	8	4
	$O_{8,3}$	6	2	13	5	4	3	5	7	9	5
	$O_{9,1}$	3	9	1	3	8	1	6	7	5	4
J_9	$O_{9,2}$	4	6	2	5	7	3	1	9	6	7
	$O_{9,3}$	8	5	4	8	6	1	2	3	10	12
	$O_{10,1}$	4	3	1	6	7	1	2	6	20	6
J_{10}	$O_{10,2}$	3	1	8	1	9	4	1	4	17	15
	$O_{10,3}$	9	2	4	2	3	5	2	4	10	23

dustriais se torna um gargalo a ser resolvido e pode ser um fator decisório no sucesso de uma indústria (WARI; ZHU, 2016).

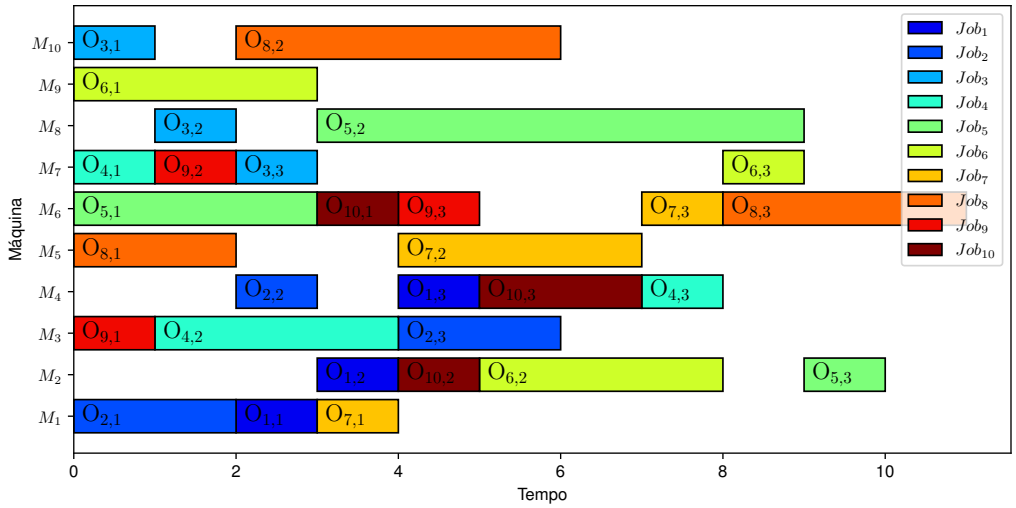
2.2 SOLUÇÕES EXISTENTES

Ao longo do tempo foram propostas, testadas, revisadas e aprimoradas diversas soluções para os problemas de agendamento, mas como cada cenário de aplicação é único não é possível concluir que uma solução é melhor que outra, pois isso depende dos critérios de análise de cada problema. Contudo, durante o tempo algumas soluções chamaram a atenção de pesquisadores, seja por serem mais eficientes ou por serem mais adaptáveis a novas demandas, cenários e objetivos.

Um tipo de solução muito recorrente são as soluções baseadas em comporta-

Figura 1 – Diagrama de Gantt para um agendamento

Problema 3



mento biológico. Ao longo de milhões de anos a vida biológica no planeta Terra evoluiu e se adaptou para encontrar modos mais eficientes de resolver os diversos problemas que a natureza impõe. Então observando esses comportamentos adaptativos, foram propostos diversos algoritmos, como os algoritmos populacionais que simulam como populações de animais, como bandos de aves, enxames de abelhas ou colonias de formigas usam o comportamento de bando para solucionar um problema como o de encontrar alimento, ou fugir de predadores. Ou algoritmos que simulam as próprias regras de seleção natural para selecionar e evoluir indivíduos de uma população para gerar e acumular mutações, de modo a obter uma população de indivíduos mais adaptados e solucionar um problema, esses são os Algoritmos Genéticos e Algoritmos Evolutivos.

Os Algoritmos Evolutivos são baseados no mecanismo de seleção natural observado na natureza e descritos por Charles Darwin. Onde um individuo que tem alguma mutação que gere uma característica positiva que o destaque dos outros indivíduos da população, tem maior chance de se reproduzir e transmitir essa característica para os seus descendentes.

A estrutura básica de um Algoritmo Evolutivo é:

1. Inicialização da população.
2. Analise a qualidade de cada indivíduo.
3. Selecionar os melhores indivíduos.
4. Cruzar esses indivíduos melhores entre si, de modo a gerar uma nova população com melhores indivíduos.

5. Voltar e repetir a partir do passo 2.

Essa estrutura se repete até que seja atingido o critério de parada estabelecido pelo problema em questão.

Além da estrutura básica é possível melhorar um Algoritmo Evolutivo através de um aprendizado continuo, ou seja, em um ambiente em que tenha mudanças constantes um Algoritmo Evolutivo pode se adaptar e assim se auto melhorar sem a necessidade de uma intervenção do programador.

Uma derivação dos Algoritmos Evolutivos são os Algoritmos Genéticos, utilizados para buscas e para otimizações, como nos problemas de escalonamento. Essa abordagem chama atenção por conseguir lidar com uma grande diversidade de soluções, o que a torna interessante especialmente em problemas de multi-objetivo como visto por (BAGCHI, 1999).

Os Algoritmos Genéticos são a abordagem mais utilizada dentre os Algoritmos Evolutivos, o que algumas vezes podem gerar a impressão de que Algoritmos Genéticos e Evolutivos são a mesma coisa, porém como o Algoritmo Genético lida com uma simulação de cromossomos isso o faz com que suas estruturas sejam muito mais dinâmicas, enquanto as técnicas de Algoritmos Evolutivos geralmente tem estruturas mais fixas.

Para ter essa dinamicidade os Algoritmos Genéticos trabalham com o conceito de mutações aleatórias, que podem ou não serem boas para o indivíduo, porém por ser uma derivação dos Algoritmos Evolutivos o Algoritmo Genético conta também com os mecanismos de seleção e cruzamento entre indivíduos da população, o que tende a selecionar apenas os indivíduos com mutações que são positivas do ponto de vista do objetivo estabelecido pelo problema.

Outra abordagem dos algoritmos bio inspirados são os Algoritmos Populacionais, nos quais normalmente não se é utilizado técnicas de evolução de indivíduos específicos, mas o foco é dado no comportamento da população. Seus cenários de aplicação vão além dos algoritmos de otimização e também são utilizados em algoritmos de busca e em efeitos visuais, sendo amplamente utilizada em filmes. Os Algoritmos Populacionais usam o conceito de inteligência de bando, onde indivíduos simples interagem entre si e com o ambiente e juntos convergem para uma solução. A inteligência de bando pode ser vista em diversos lugares da natureza, como em colonias de formigas, enxames de abelhas, na forma de voo migratório de aves, na forma de caça de aves predatórias como águias e na organização de cardumes de peixes. O que torna essa abordagem interessante é a sua característica decentralizada e de auto organização, o que a torna mais extensível e adaptável a cenários distribuídos. Dentre os principais algoritmos populacionais estão:

- Otimização por Colonia de Formigas (*Ant Colony Optimization*)

- Difusão Estocástica de Busca (*Stochastic Diffusion Search*)
- Otimização por Enxame de Partículas (*Particle Swarm Optimization*)

Todos eles se baseiam em como a população de indivíduos encontram juntos um objetivo. Nesse trabalho é utilizado o algoritmo de Otimização por Enxame de Partículas (*Particle Swarm Optimization*).

2.3 PARTICLE SWARM OPTIMIZATION — PSO

O algoritmo de Otimização por Enxame de Partículas ou PSO da sigla em inglês para (*Particle Swarm Optimization*) é um algoritmo baseado nas teorias de inteligência de enxame. E diferentemente de outros algoritmos baseados em populações, como o *Ant Colony Optimization* o *Particle Swarm Optimization* é baseado em uma população genérica de indivíduos, embora sejam normalmente ilustrados como uma população de aves.

O algoritmo *Particle Swarm Optimization* foi proposto em 1995 por (KENNEDY; EBERHART, 1995) e desde então tem se mostrado muito promissor para a solução de diversos problemas de otimização. Por ser um algoritmo bem simples e flexível, ao longo do tempo já foram propostas varias variações para ele.

O *Particle Swarm Optimization* já foi alvo de várias discussões na área, pois alguns autores discordam de ele ser classificada como parte dos Algoritmos Evolutivos (areá da qual os Algoritmos Populacionais fazem parte), pois embora haja variações que sim, o *Particle Swarm Optimization* básico não implementa mecanismos de seleção, cruzamento ou mutação, critérios básicos para a classificação como Algoritmo Evolutivo. Atualmente o *Particle Swarm Optimization* é classificado como parte da família dos algoritmos de *Swarm Intelligence*.

A definição básica do algoritmo *Particle Swarm Optimization* é uma população de partículas (também chamadas indivíduos), em que cada partícula tem uma velocidade e uma direção, além das informações da sua posição, qual é a qualidade de sua posição, qual foi a melhor posição na, qual ela já esteve, e acesso a um conhecimento compartilhado entre todos os indivíduos com a melhor posição em que qualquer um deles já esteve. E assim a cada rodada as partículas se movimentam com base na sua melhor posição e na melhor posição geral, e assim a cada rodada a população converge para uma solução ótima.

As variáveis mais importantes do PSO são justamente as melhores posições, sendo a local normalmente chamada de $pBest$ e a geral normalmente chamada de $gBest$. Elas vão ser utilizadas para obter uma média que será a nova posição da partícula. Porém, existem resistências para que a partícula mude de direção, isso acontece por meio da inércia normalmente representada por w que representa a força que tende a fazer a partícula seguir a direção onde ela já está se movendo. Essa força de inércia

cresce conforme o valor de velocidade da partícula, normalmente representada por v, ou seja, partículas com uma maior velocidade v tendem a tem uma maior inércia w.

No Pseudocódigo 1 é possível uma representação de um algoritmo PSO básico, esse algoritmo tem como base a formulação de (Fernández Martínez; García Gonzalo, 2009).

Algoritmo 1 Pseudocódigo de um PSO básico

```
Inicia o espaço de soluções
Inicia a população em localizações aleatórias
Avalia as partículas e define o gBest
while não atingir o critério de parada do
    Atualiza a posição de todas as partículas
    Atualiza o fitness de todas as partículas
    if algum pBest é melhor que gBest then
        gBest ← pBest
    end if
end while
```

Na definição base do algoritmo não existe uma especificação de uma fórmula para o cálculo da movimentação da partícula. Porém, na Equação 2.1 é possível ver uma representação de uma fórmula base para o cálculo de uma nova posição para a partícula.

$$\vec{x}_i(t+1) = \vec{x}_i + \vec{v}_i(t) + a_1 r_1(pBest - \vec{x}_i(t)) + a_2 r_2(gBest - \vec{x}_i(t)) \tag{2.1}$$

sendo i o número da partícula; t o número da rodada; a_1 e a_2 os valores de aceleração local e global respectivamente; r_1 e r_2 números aleatórios gerados por uma função de probabilidade uniforme em um intervalo de $[0.1, 0.9]$.

Alguns cuidados devem ser tomados para a implementação do *Particle Swarm Optimization* pois o algoritmo se mal configurado através dos parâmetros de velocidade, inercia e tamanho da população, tende a convergir prematuramente para uma solução não ótima, esse problema é normalmente chamado "Convergência Prematura". Algumas medidas podem ser tomadas para diminuir essa possibilidade de convergência Prematura, como valores dinâmicos de inércia e uma ponderação nos valores de importância de $pBest$ e $gBest$.

Algumas técnicas podem ser usadas para melhorar o desempenho do PSO em cada cenário de aplicação. Porém, o efeito dessas abordagens varia de acordo com qual problema esta sendo resolvido e dos recursos e limitações de processamento e memoria do ambiente de processamento do algoritmo.

Uma abordagem com bons resultados é o processamento distribuído do cálculo de movimentação de cada partícula como visto por (THONGKRAIRAT; CHUTCHAVONG, 2019) e (KIM et al., 2011). Essa paralelização é possível graças a natureza distribuída dos algoritmos populacionais. No caso do PSO o único fator a se considerar para uma implementação distribuída é a atualização da variável $gBest$.

Outra melhoria que tem demonstrado bons resultados é a hibridização de PSO com outros algoritmos, como com Algoritmos Genéticos como demostrado por (CARVALHO; FERNANDES, 2014), ou com outros Algoritmos Evolutivos. Essas abordagens de hibridizações com Algoritmos Evolutivos tendem a ser tão boas devido às características básicas do PSO como a de não ter componentes evolutivos em seus indivíduos, então uma adição de componentes evolutivos não atrapalha nenhum ponto do PSO. Esses componentes evolutivos normalmente trabalham nos valores de inércia e de velocidade, criando partículas com uma tendência maior a fugir da convergência do grupo, o que pode ajudar a escapar de mínimos locais, evitando assim uma convergência prematura.

Um cenário que tem um bom proveito da hibridização evolutiva do PSO são os cenários com multi-objetivo, pois os componentes de mutação de Algoritmos Genéticos podem trazer uma maior diversidade para os indivíduos da população, gerando assim uma maior gama de exploração no espaço de soluções.

Outra abordagem promissora é a implementação de componentes dinâmicos na população, ou seja, que possam mudar suas características como velocidade e inercia de maneira dinâmica, por exemplo, se uma partícula perceber que esta longe do $gBest$ e está se movendo pouco ela pode reconhecer que possivelmente esta em um mínimo local e diminuir seu valor de inércia, ou aumentar o fator de importância do $gBest$. Essa abordagem pode ser feita de maneira constante nas partículas, fazendo elas se atualizarem em tempo real.

3 DESENVOLVIMENTO

Nesse capítulo são descritos os métodos e técnicas utilizadas no desenvolvimento tanto do algoritmo PSO com a nova abordagem proposta quanto da abstração para o problema de FJSP. Também são descritos os critérios de parada escolhidos e modo de execução e os resultados dos testes.

3.1 METODOLOGIA DE PESQUISA

Para os testes deste trabalho estão sendo usados os mesmo problemas utilizados por (KACEM; HAMMADI; BORNE, 2002), e já são largamente usados na literatura. Sendo representados por $[j, o, m]$ em que j é a quantidade de *jobs*, o é a quantidade de operações e m a quantidade de máquinas. Para observar o comportamento do algoritmo em diversos cenários foram escolhidos problemas de tamanhos variados, sendo eles $[4, 12, 5]$, $[10, 29, 7]$, $[10, 30, 10]$ e $[15, 56, 10]$.

Os algoritmos foram executados no ambiente de nuvem Google Colab Code sendo executado no motor *Python 3 Google Compute Engine Backend*. Para a execução foi utilizado um *runtime* com processadores comuns (CPU) e 12 Gigabytes de memória RAM.

Para que possibilitar uma análise de dados mais detalhada e a geração de gráficos para representação nesse trabalho, foram implementados no código pontos de salvamento de dados, para salvar em formato de binários do NumPy, utilizando a função `np.savedata`, as informações de execuções e de datasets.

Dentre esses dados estão o espaço de soluções gerado, a quantidade de rodadas realizadas, o histórico da variável $gBest$, o número de mudanças de $pBest$, o histórico de posição das partículas e as soluções finais encontradas.

3.2 ARQUITETURA

A arquitetura de um projeto de é um fator que impacta diretamente no seu desempenho, porém como o objetivo desse trabalho é entender como o algoritmo PSO se comporta diante das alterações propostas, foi necessário o desenvolvimento de uma arquitetura que represente o problema de maneira mais abstrata.

3.2.1 Design do Projeto

Por esse trabalho ser desenvolvimento dentro de um grupo de estudos, foi optado por usar uma estrutura de projeto mais didática e de fácil entendimento, de maneira que possa ser facilmente reutilizada por futuros estudantes. Porém, essa es-

colha acarreta uma certa perda de desempenho, porém, como se trata de um trabalho que visa explorar como cada tipo de alteração impacta o algoritmo, essa perca de desempenho não se torna um defeito.

O projeto foi desenvolvido na linguagem de programação Python versão 3.9.7 e utiliza além das bibliotecas básicas da linguagem, as bibliotecas:

- NumPy (versão 1.21.3)
- MatPlotlib (versão 3.4.3)

O MatPlotlib foi utilizado para a geração de:

- Gráficos de Dispersão (*Scatter Plot*) para representar a posição das partículas da população.
- Gráficos de Superfície (*Surface Plot*) para representar o espaço de soluções.
- Gráficos de Barra Horizontal (*Horizontal Bar Plot*) para representar um diagrama de Gantt com a solução do agendamento.

O NumPy foi utilizado para:

- Grandes arranjos multi dimensionais de dados.
- Funções randômicas de escolha.
- Ordenação de dados.
- Operações de cálculo como raiz quadrada e potenciação.
- Funções de escolha de valores máximos e mínimos.
- Modelagem, união e remodelagem de matrizes.
- Representações e calculo de vetores de movimentação.

Para trazer mais clareza ao código foi optado por criar algumas classes para a representação de entidades e abstração de lógicas. Dentre elas estão:

- Classe de Partícula, que faz a representação de uma entidade de partícula e armazena sua posição, velocidade e direção. Faz a inicialização da partícula com valores aleatórios de posição, velocidade e direção. E estabelece métodos para calcular o movimento da partícula e o valor da qualidade de sua nova posição.
- Classe de *encoding*, responsável por ler e interpretar os arquivos de instância de um problema, criar uma representação para de solução, e então gerar o espaço de soluções com soluções geradas aleatoriamente.

- Classe de *decoding*, responsável por decodificar uma solução no padrão de representação estabelecido pela classe de *encoding* e calcular o *fitness* dessa solução. Essa classe também estabelece o método que gera o gráfico de Gantt para representar visualmente a solução.

Como o PSO é um algoritmo estocástico, ou seja, seu resultado é fruto de eventos não previsíveis foi decidido que para obter uma média real dos desempenhos de cada algoritmo, em cada uma das 30 execuções será gerado um novo espaço de soluções e uma população inicial diferente.

E em cada uma dessas execuções será passado para cada um dos algoritmos uma *deepcopy* do espaço de soluções e da população inicial, para que os diferentes algoritmos trabalhem com base nos mesmos dados iniciais, para assim tem uma análise mais justa.

Para que a análise dos dados possa ser feita mais detalhadamente, foram implementados alguns pontos de salvamento de dados, para salvar em arquivo texto as representações da população e do espaço de soluções gerados aleatoriamente, assim como a quantidade de rodadas realizadas, o histórico da variável $gBest$, o número de mudanças de $pBest$, o histórico de posição das partículas e as soluções finais encontradas.

3.2.2 Espaço de Soluções

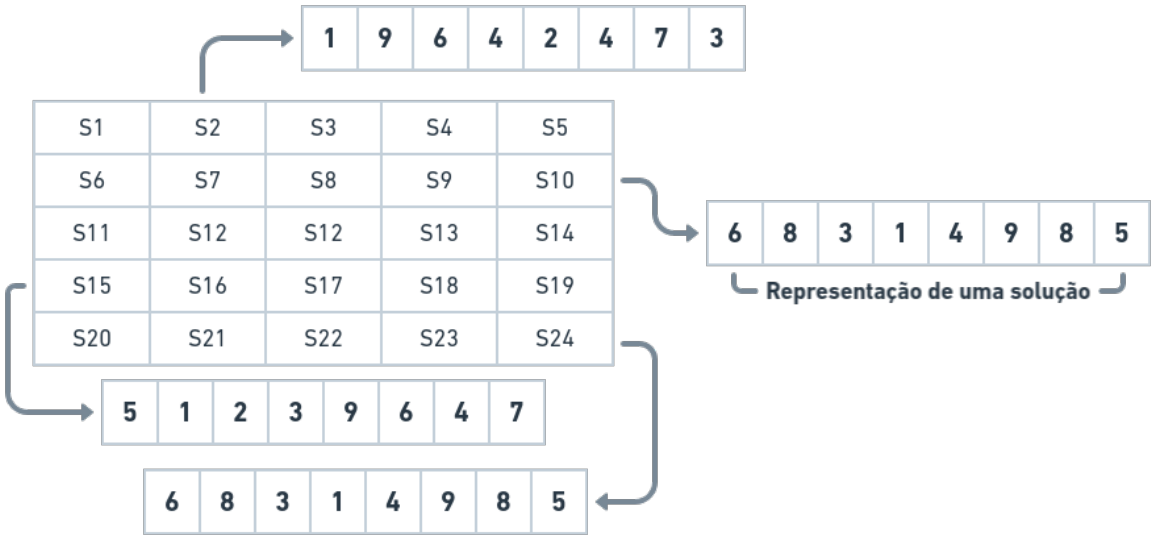
O espaço de soluções (também chamado espaço de busca) é um conceito que define um espaço imaginário aonde cada ponto desse espaço é uma possível solução para o problema em questão. Um espaço de soluções pode ser representado de diversas maneiras, como listas e grafos, mas em problemas de optimização normalmente se usa uma representação na forma de uma matriz.

Nessa representação em forma de matriz cada localização $[i, j]$ tem uma solução valida para o problema, e o algoritmo em questão tem a tarefa de encontrar qual dessas posições tem a melhor solução, ou uma solução boa o suficiente. Na Figura 2 é mostrado uma representação conceitual criada para ilustrar um espaço de soluções em forma de matriz.

Algumas das implementações encontradas em bibliotecas de algoritmos populacionais de código aberto como *PySwarms* utilizam internamente um sistema de representação de espaço de soluções de uma única dimensão, ou seja, uma lista simples, e utiliza uma função para transformar uma posição $[x, y]$ em um índice da lista. Essa implementação provavelmente foi escolhida por ser geralmente, mais rápida em comparação a uma abordagem de duas dimensões.

Como esse trabalho tem como foco uma exploração e implementação mais didática foi optado a utilização de uma representação em forma de uma matriz com duas dimensões, utilizando a biblioteca *NumPy*. Essa abordagem em matriz de duas

Figura 2 – Representação de uma matriz de espaço de soluções



dimensões também facilita a geração de gráficos para representação visual, como, por exemplo, passar a matriz de soluções pela função de avaliação e assim obter um valor de *fitness* para cada ponto do espaço de soluções, como é possível ver na Figura 3 na qual os valores mais acima e mais escuros tem um valor de *fitness* mais alto (ou seja, pior), e os valores mais abaixo e mais claros tem um *fitness* mais baixo (ou seja, melhor).

Esses gráficos se mostraram úteis ao decorrer do desenvolvimento deste trabalho, servindo para validar detectar erros na função de inicialização do espaço de soluções.

A geração do espaço de soluções é muito importante para algoritmos como o PSO que funciona com base na movimentação conforme as médias de melhores posições, devido a isso, o PSO tente a ter problemas com convergência prematura em mínimos locais.

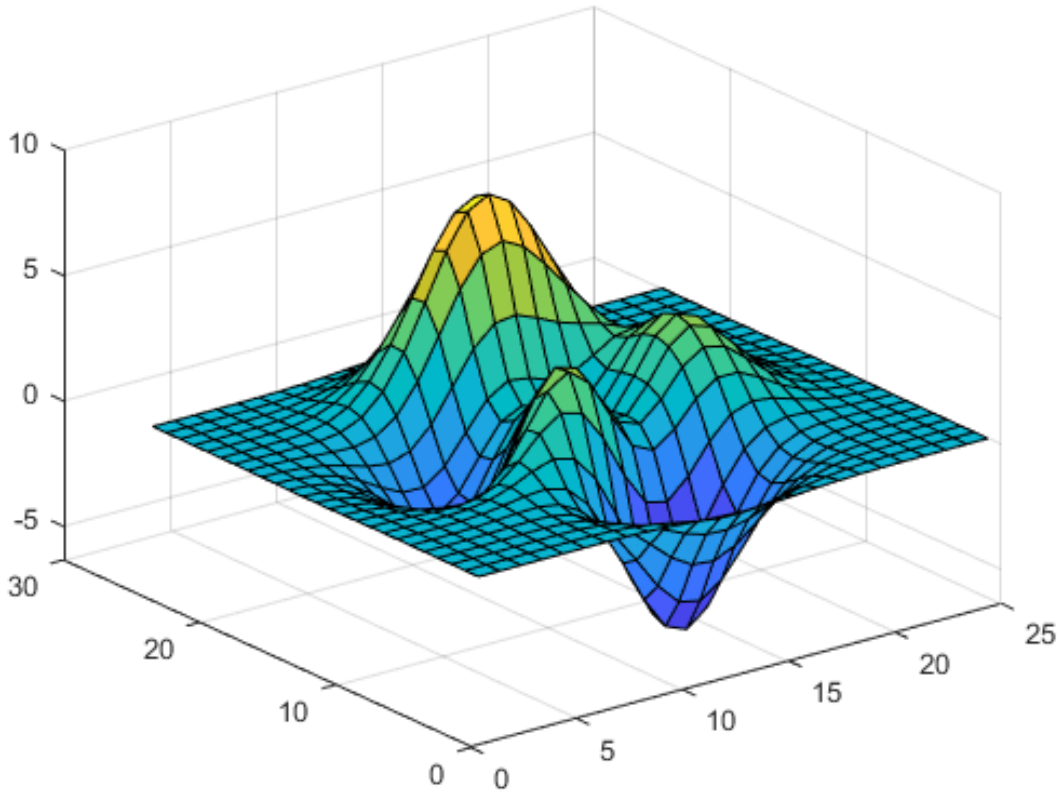
Os mínimos locais são fenômenos onde dentro de uma parte do espaço de soluções existe um ponto que é melhor que a média dos pontos a sua volta, porém não é o melhor entre o todo o espaço de soluções, como é possível ver na Figura 4.

Por esse ponto se destacar entre seus vizinhos o algoritmo pode ficar preso nesta região, pois a média local sempre tende para o ponto onde está o mínimo local.

Como o PSO se baseia na média local e global, se houver mais de um mínimo local e quantidades equivalentes de partículas da população forem distribuídas para esses pontos, as partículas podem ficar presas já que as forças se balanceiam e o $gBest$ não muda mais.

Já a convergência prematura acontece quando a um mínimo local e por coincidência a posição inicial da maioria das partículas da população tende a ser atraído para o mínimo local, logo a maioria da população para nesse ponto e o algoritmo ter-

Figura 3 – Gráfico de Superfície representando um espaço de soluções



mina antes de encontrar uma solução tão boa quando seria possível.

Durante a implementação de um algoritmo para geração do espaço de soluções, foram percebidos diversos detalhes que influenciam fortemente na qualidade final do algoritmo. Nessa seção será analisado alguns desses pontos.

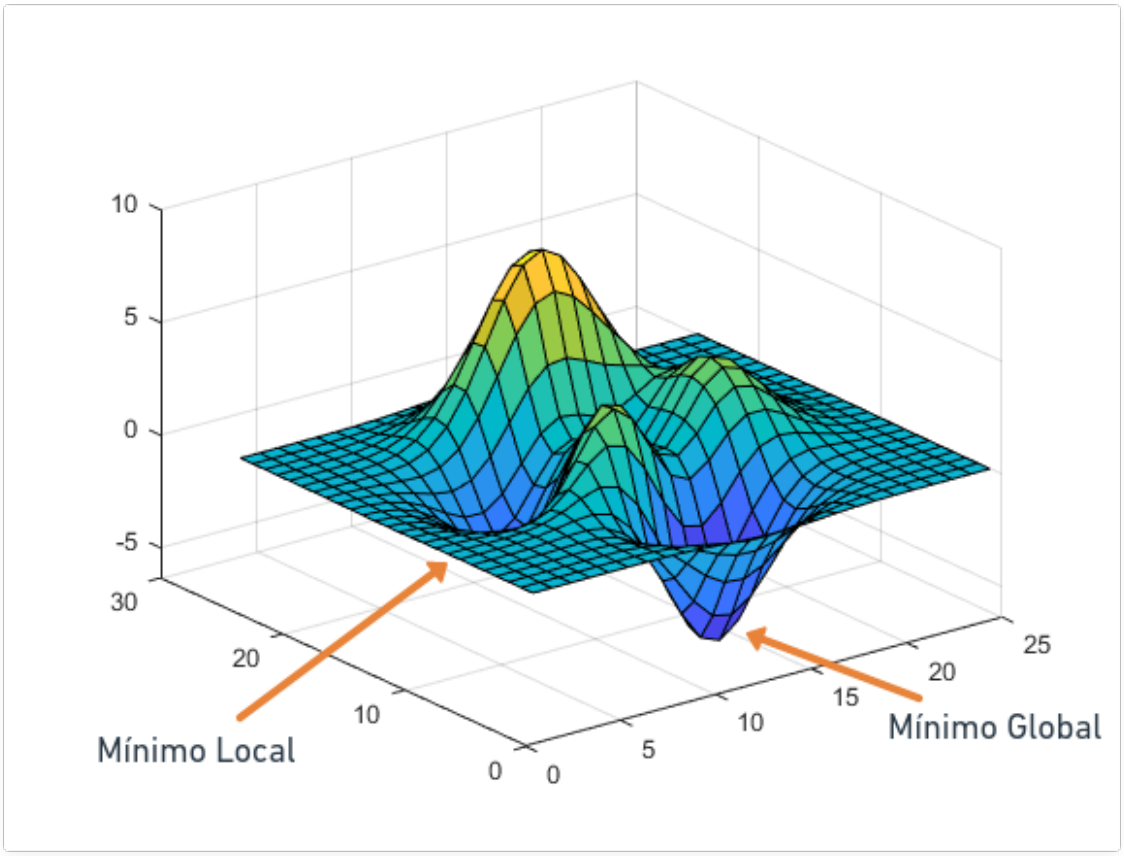
Um ponto importante percebido na implementação é como sera inicializado o *array* onde será armazenado o espaço de soluções.

O NumPy disponibiliza trés métodos de inicialização para um novo *array* sem um valor de preenchimento, dos quais cada um demonstrou um problema diferente para a implementação. Dentre eles:

- *numpy.zeros*: ao inicializar o *array* com todos os valores preenchidos com o valor "0", devido ao processo de embaralhamento das operações, o algoritmo pode acabar gerando resultados errôneos. Por não existir uma máquina M_0 algumas operações não eram consideradas no resultado, e devido à falta de algumas operações, a solução em questão tinha um valor de *fitness* menor, o que tendia a fazer essa solução errônea a melhor.

- *numpy.ones*: ao inicializar o *array* com todos os valores preenchidos com o valor "1", o algoritmo gerava soluções que tendiam a sobre carregar a máquina M_1 e

Figura 4 – Exemplo de um mínimo local



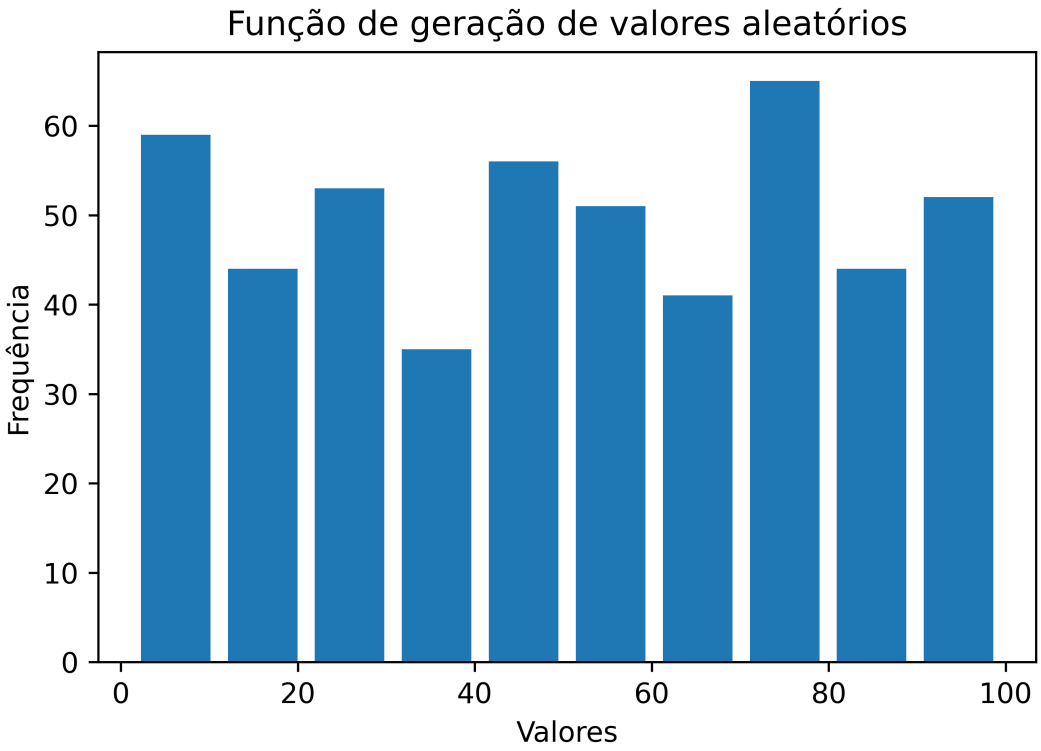
assim os resultados finais tendiam a ter um valor de *fitness* mais alto.

- *numpy.empty*: esse método de inicialização diferentemente dos outros, não defini nenhum valor para os registros da matriz, somente aloca um bloco de memória (semelhante ao *malloc* da linguagem C), o que o torna mais rápido, porém ele traz o revés de possivelmente ter valores errados, pois por simplesmente alocar um bloco de memória, o *array* pode ser inicializado com valores estranhos devido a lixo de memória.

Devido aos problemas citados acima, acaba sendo necessário de qualquer maneira percorrer todo o *array* para setar valores.
Então foi optado por utilizar um método de preenchimento de valores aleatório, gerados pela função *numpy.random.choice* que gera um *array* de valores aleatórios seguindo uma distribuição uniforme, conforme demonstrado no Gráfico da Figura 5.

Após as análises citadas acima foi feito uma modelagem e desenvolvido para esse trabalho o algoritmo de inicialização do espaço de soluções representado pelo Pseudocódigo 2.

Figura 5 – Gráfico de distribuição de valores gerados pela função aleatória



Algoritmo 2 Pseudocódigo de geração do espaço de soluções

```
espaço soluções ← lista vazia
contador ← 0
while contador ≤ tamanho espaço de soluções do
    solução ← lista vazia
    embaralhar(operações)
    embaralhar(jobs)
    embaralhar(máquinas)
    solução[operações] ← remover ultimo elemento(operações)
    solução[jobs] ← remover ultimo elemento(jobs)
    solução[máquinas] ← remover ultimo elemento(máquinas)
    espaço soluções ← adicionar(solução)
end while
```

3.2.3 Partícula

Por se tratar de um algoritmo populacional focado em partículas, por isso o nome *Particle Swarm Optimization* que significa Otimização por Enxame de Partículas, a representação dessa entidade, assim como sua geração, controle e movimentação, se mostra um dos principais pilares do algoritmo. Devido a critérios de legibilidade e facilitação de futuros estudos nesse trabalho foi optado por criar uma estrutura de classe bem definida para essa representação e controle. Nessa classe são guardados os valores de: posição, velocidade e $pBest$, além de outros dados utilizados para fa-

cilitar o gerenciamento como: valor da solução da sua atual posição, assim como seu valor de *fitness* e o tamanho do espaço de soluções.

Para fazer uma melhor movimentação foi optado por não usar uma variável de direção, mas sim uma variável que guarda a última posição onde a partícula esteve. Além da armazenagem dos valores da partícula, a classe da Partícula também implementa algumas funções como:

- Função ***fill_with_random_values*** que gera uma posição inicial e uma velocidade aleatória para a partícula.
- Função ***evaluate_value*** que faz o cálculo do valor de *fitness* da atual posição e o compara com a variável $pBest$ e caso seja melhora, atualiza o $pBest$.
- Função ***update_position*** que faz a movimentação da partícula.

Em implementações de código aberto como a já citada *PySwarms*, também se utiliza uma representação de classe, porém sem nenhuma responsabilidade de cálculo de *fitness*, atualização de posição ou de preenchimento de valores aleatórios na partícula.

A movimentação da partícula no espaço de soluções é o ponto mais importante dessa entidade. Pois, se não for feita da maneira certa pode fazer o algoritmo não desempenhar tão quanto o possível.

Como a implementação do espaço de soluções foi feita utilizando uma representação matricial, o mecanismo de movimentação mais intuitivo seria uma movimentação por soma de inteiros.

Aonde se tem uma direção e com base nessa direção se soma ou subtrai o valor de velocidade do eixo de movimento e assim se obtém uma nova posição, como mostrado no Pseudocódigo 3.

Ou seja, tendo uma posição $[15, 7]$ sendo 15 o valor de x e 7 o valor de y, uma velocidade 2 e a direção do movimento ser *CIMA*, a nova posição é obtida incrementado a velocidade ao valor de y da posição, ou seja $[15, 7 + 2]$, então a nova posição seria $[15, 9]$.

Essa abordagem funciona bem para uma movimentação simples em espaços 2D. Sendo inclusive muito utilizada no desenvolvimento de jogos com mapas baseados em ladrilhos (*Tile-based* / *Grid-based*).

Porém, no caso do algoritmo PSO é necessário considerar a inércia e uma média entre o $pBest$ e o $gBest$ obtendo assim uma nova posição, o que gera muitas vezes uma movimentação em ângulos mais específicos o que torna a abordagem de soma de inteiros muito complexa e limitada para esse cenário. Então nesse trabalho foi desenvolvido um método de movimentação baseado em cálculos vetoriais que elimina esses problemas de movimentação além de facilitar a implementação de novos pontos de comparação (além do $pBest$ e $gBest$) para o cálculo da nova posição.

Algoritmo 3 Pseudocódigo de movimentação por soma de inteiros

```
if direcao = CIMA then
    posicaoY = posicaoY + 1
else if direcao = ESQUERDA then
    posicaoX = posicaoX - 1
else if direcao = DIREITA then
    posicaoX = posicaoX + 1
else if direcao = BAIXO then
    posicaoY = posicaoY - 1
else if direcao = SUPERIOR ESQUERDA then
    posicaoX = posicaoX - 1
    posicaoY = posicaoY + 1
else if direcao = SUPERIOR DIREITA then
    posicaoX = posicaoX + 1
    posicaoY = posicaoY + 1
else if direcao = INFERIOR ESQUERDA then
    posicaoX = posicaoX - 1
    posicaoY = posicaoY - 1
else if direcao = INFERIOR DIREITA then
    posicaoX = posicaoX + 1
    posicaoY = posicaoY - 1
end if
```

Essa abordagem de movimentação é baseado no conceito de *Produto Escalar* definido pela Álgebra Linear, e representa o produto interno padrão do espaço euclidiano sendo definido como uma operação binaria entre dois vetores. Em um cenário aonde a posição da partícula é $position = [12, 8]$ e $pBest = [10, 15]$, $gBest = [16, 20]$ e $velocidade = [2, 4]$. Para encontrar a nova posição de uma partícula são utilizados:

- $\vec{p}$ Que representa o vetor entre a partícula e o $pBest$ (Figura 7).
- $\vec{g}$ Que representa o vetor entre a partícula e o $gBest$ (Figura 8).
- $\vec{r}$ Que representa o vetor final de movimento (Figura 9).

A nova posição da partícula é dada pela Equação 3.1 E a partir dela temos o vetor $\vec{r}$ resultante, então a nova posição da partícula é $[12, 10]$. .

$$\vec{r} = ((\vec{g} + \vec{p})/2) + \vec{v} \tag{3.1}$$

Na implementação em Python para realizar os cálculos vetoriais foi utilizado a biblioteca *Numpy*. Para isso primeiro é calculado um vetor médio entre $pBest$ e $gBest$, e o vetor $\vec{w}$ representando a inércia. Então a posição atual da partícula é deduzido do valor médio entre $pBest$ e $gBest$, isso é feito para que calcular o vetor como se a posição atual da partícula fosse o ponto $[0, 0]$ do mapa.

Figura 6 – Vetor $\vec{v}$ de movimento

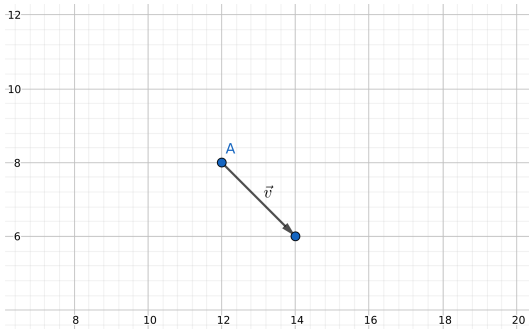


Figura 7 – Vetor $\vec{p}$ de o $pBest$

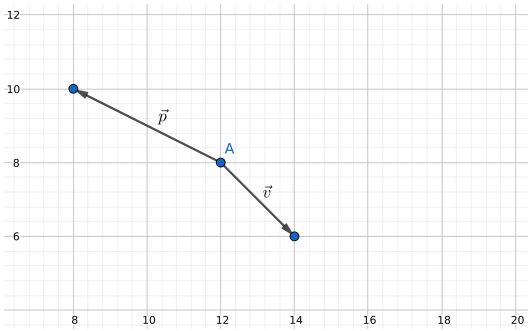


Figura 8 – Vetor $\vec{g}$ de $gBest$

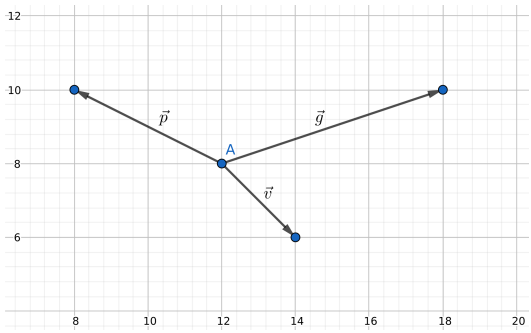
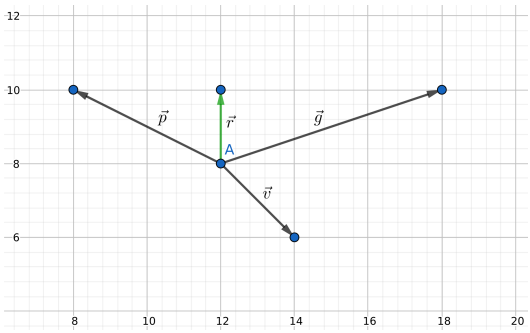


Figura 9 – Vetor $\vec{r}$ de movimento final



Então somado com o vetor de inércia divido por um valor de significância para a inércia (Definido pelos parâmetros de configuração do algoritmo). No Pseudocódigo 4 é possível ver a representação dessa função de cálculo

Algoritmo 4 Pseudocódigo de movimentação de partícula

```
vetor de inércia ← (posição inicial ∗ velocidade)
melhor media ← (pBest + gBest) / 2
vetor final ← (melhor media + posição inicial) / 2
```

3.2.4 População

Pelo algoritmo PSO ser um algoritmo baseado em inteligência populacional, ele é muito influenciado pela geração da população inicial, mesmo nos algoritmos com variações aonde existem mutações e consequentemente evolução dos indivíduos da população, as características como inercia, velocidade, direção e posição desses indivíduos da população inicial são de grande importância.

Os parâmetros como o tamanho da população são definidos nas configurações de hiper parâmetros do algoritmo. Já as funções utilizadas na geração dos indivíduos iniciais utilizam alguns valores aleatórios que seguem um padrão de distribuição uniforme. Os atributos que utilizam valores aleatórios são:

- A velocidade, dada por um valor aleatório entre 0.1 e 0.2.

- A posição inicial da partícula, dada por dois números aleatórios entre 1 e o limite do espaço de soluções -1.
- A última posição da partícula, obtida através de um número inteiro aleatório entre 1 e -1 diferente de zero.

Assim, como valores iniciais seguindo uma distribuição uniforme é observado uma geração de população inicial bem distribuída, dificultando o acontecimento de convergência prematura.
Sendo assim, todas as variações do algoritmo PSO implementadas nesse trabalho utilizam a mesma função geradora de população inicial. Garantindo assim um ponto de comparação mais justo entre os algoritmos.

3.2.5 Algoritmos

A execução dos algoritmos é chamada pela classe principal de modelagem do problema, responsável por realizar a execução e controle dos dados de análise.
Na classe do PSO estão as funções da condição de parada, movimento e *evaluete* do *fitness* da partícula
Como critério de parada para o algoritmo foi utilizado o fator de atualização do $gBest$. Quando a última atualização do $gBest$ for a mais rodadas do que metade da quantidade de partículas, o algoritmo termina e o resultado é o $gBest$ atual.

O que diferencia as abordagens analisadas nesse trabalho são as heurísticas de movimentação das partículas.
A movimentação básica de uma partícula do PSO é definido pela Equação 3.2 .

$$\vec{v_1}(t+1) = \vec{v_i}(t) + a_1 r_1(pBest - \vec{x_i}(t)) + a_2 r_2(gBest - \vec{x_i}(t)) \tag{3.2}$$

Como foi observado nos testes a variável aleatória é importam de para que as partículas não entrem em um estado de equilíbrio e parem de se movimentar. Com a introdução de elementos aleatórios a movimentação se torna mais caótica e evita o fim prematuro do algoritmo.

Esse impacto na movimentação inspirou uma abordagem que considera o nível de atualizações de $gBest$ e assim adiciona mais um fator aleatório no cálculo. como é representado no Pseudocódigo 5.

3.3 EXECUÇÕES

Foram utilizados como parâmetros de configuração:

- `population_size` Quantidade de partículas
- `coeficente_populacional`: coeficiente entre a quantidade de partículas e o tamanho do espaço de soluções

Algoritmo 5 Pseudocódigo de movimentação com componente dinâmico

vetor de inércia ← (posição inicial * velocidade)
melhor media ← (($pBest$ + $gBest$) / 2)
vetor final ← ((melhor media + posição inicial) / 2)
if trocas de $gBest$ é menor que (tamanho da populacao / 10) **then**
    vetor final ← vetor de movimento $+ 0.1 *$ (inercia$*$random(0.1, 0.9))
else
    vetor final ← vetor de movimento $+ 0.1 *$ random(0.1, 0.9)
end if

O tamanho do espaço de soluções é calculado pela raiz quadrada do arrendamento do tamanho da população elevado ao coeficiente populacional, representado pela equação Equação 3.3, sendo p o tamanho da população e c o coeficiente populacional. Essa equação é necessária, pois para gerar um espaço de soluções $n \times n$ pois para isso o número precisa ser uma raiz quadrada perfeita.

$$space\_size = \lfloor \sqrt{\lfloor p^c \rceil} \rceil \tag{3.3}$$

Por exemplo, no caso de uma população de tamanho 30 e um coeficiente populacional 2.3, temos $\lfloor 30^{2.3} \rceil = 2497$ e então $\lfloor \sqrt{2497} \rceil = 50$.
Para essa analise foram usados os valores de `population_size = 30` e o 3 valores diferentes de coeficiente populacional que são:

- `coeficiente_populacional = 2.3`: Gera um mapa de 50×50
- `coeficiente_populacional = 2.5`: Gera um mapa de 70×70
- `coeficiente_populacional = 2.645`: Gera um mapa de 90×90

Foram gerados 20 espaços de solução diferentes para cada um dos 3 coeficientes e cada um dos 4 cenários de problema diferentes, totalizando assim 240 espaços de soluções diferentes.

Pela natureza variável do ambiente em nuvem e por se tratar de um algoritmo que utiliza fatores aleatórios, a execução de cada uma das 2 abordagens foi repetida 20 vezes para assim obter uma média mais precisa das soluções de cada algoritmo.

Para assegurar uma análise posterior mais detalhada, os dados gerados pelos algoritmos foram salvos em aquivo binário. Os dados salvos são a população inicial, o espaço de soluções, o número de rodadas, o histórico da variável $gBest$, o histórico da posição das partículas e as Soluções encontradas.

3.4 RESULTADOS

Para avaliar as execuções dos algoritmos foram estabelecidos os seguintes critérios de análise: A partir das médias estatísticas das execuções, são utilizados como critérios de avaliação do algoritmo:

- Makespan final das soluções.

E são parâmetros de análise:

- Taxa de mudança da variável $gBest$ ao longo da execução.
- Quantidade de rodadas necessárias para atingir o critério de parada.
- Taxa de mudanças de $pBest$ por rodada.
- Histórico de movimentação das partículas.

Nas Figuras 10, 11, 12 e 13 estão representadas as frequências de *Makespan* das execuções, os dados na cor azul representam os resultados obtidos com a implementação padrão do algoritmo PSO, e na cor laranja estão representados os resultados alcançados com a implementação do componente dinâmico de inércia.

Foram utilizados nos testes *datasets* com tamanhos variados, sendo eles, problema A 4×5, problema B 10×7, problema C 10×10, problema D 15×10. Todos esses *datasets* foram retirados dos *benchmarks* de (KACEM; HAMMADI; BORNE, 2002).

3.5 PROBLEMA A

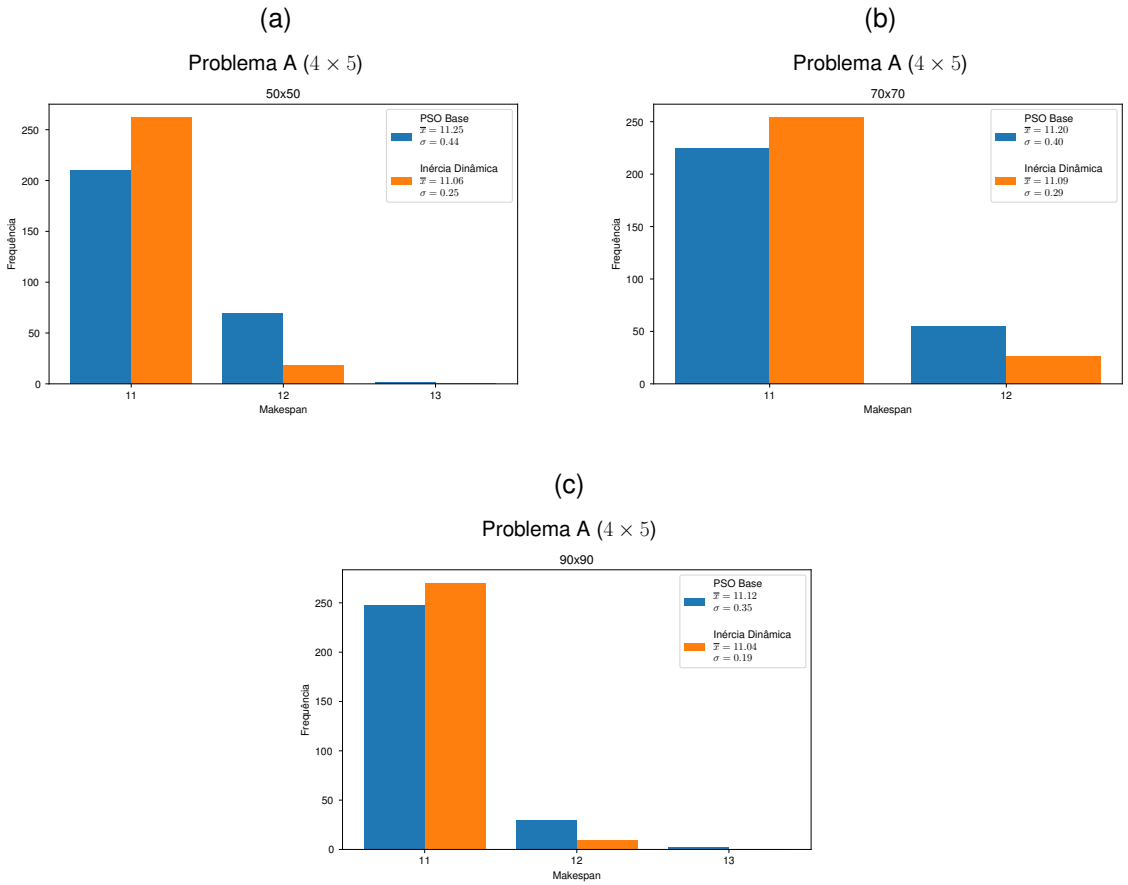
O problema A tem um tamanho 4×5, ou seja, com 4 *jobs* e 5 máquinas, sendo esse o menor dos *datasets* analisados. Na Figura 10 é possível perceber que o componente dinâmico obteve uma pequena tendência a obter um valor menor de *makespan*, ou seja, atingir um melhor resultado, principalmente nas execuções com um espaço de soluções maior, como na Figura 10c. Na Tabela 4 é representado as estatísticas entre os valores de *makespan* obtidos pelas execuções de cada abordagem.

Tabela 4 – Tabela de dados estatísticos da execução do problema A

Problema A 4×5	50×50		70×70		90×90	
	PSO Base	**Dinâmico**	**PSO Base**	**Dinâmico**	**PSO Base**	**Dinâmico**
Média	11.253	11.254	11.196	11.093	11.121	11.036
Mediana	11.0	11.0	11.0	11.0	11.0	11.0
Desvio Padrão	0.443	0.443	0.397	0.29	0.348	0.186
Variância	0.196	0.196	0.158	0.084	0.121	0.034
Mínimo	11	11	11	11	11	11
Máximo	13	13	12	12	13	12

É possível ver que a abordagem dinâmica apresentou em um *makespan* em média $1,16\%$ menor e com um desvio padrão 40% menor e uma variância $62,5\%$ menor em relação à abordagem padrão.

Figura 10 – Problema A — Gráfico de Frequência de Makespan



3.6 PROBLEMA B

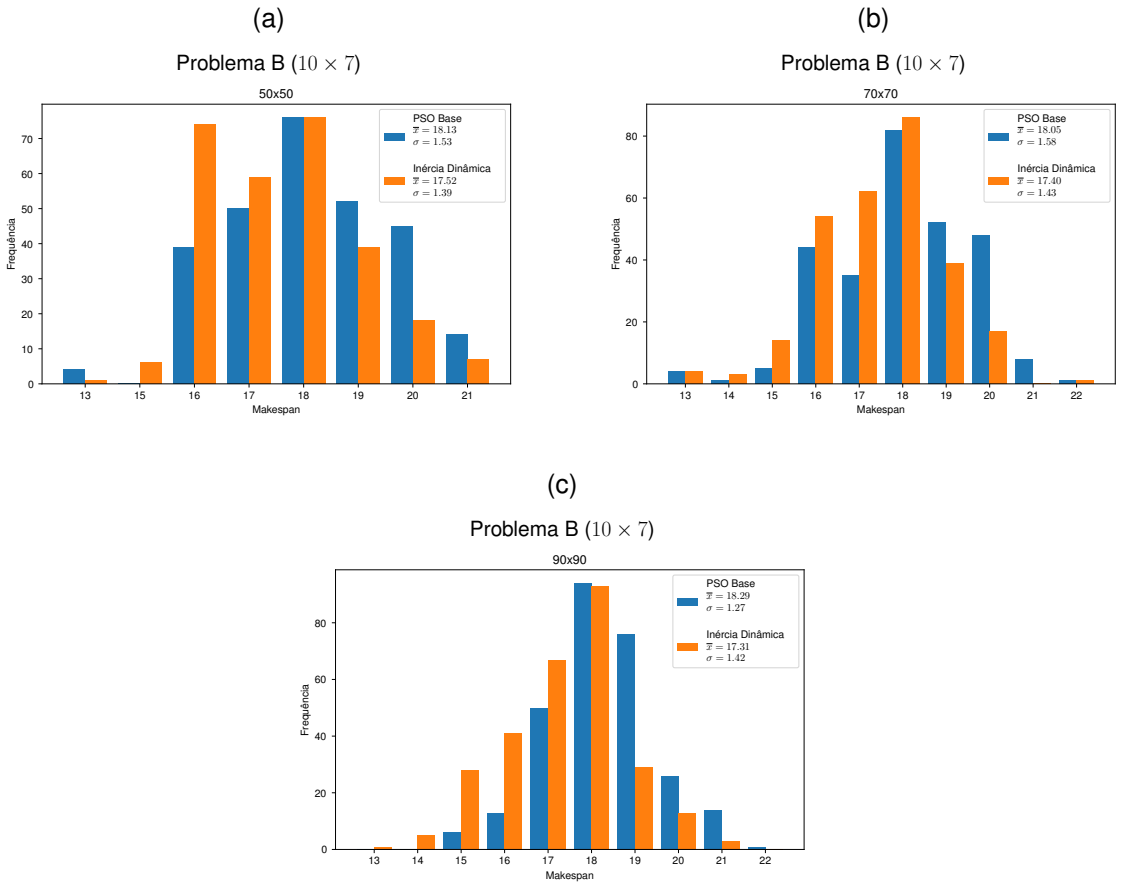
O problema B tem um tamanho 10×7, ou seja, com 10 *jobs* e 7 máquinas, sendo esse o segundo menor dos *datasets* analisados. Na Figura 11 é possível perceber que o componente dinâmico obteve uma tendência de atingir um valor menor de *makespan*, uma tendência maior do que a observada na Figura 10. Na Tabela 4 é representado as estatísticas entre os valores de *makespan* obtidos pelas execuções de cada abordagem. É possível ver que a abordagem dinâmica apresentou em

Tabela 5 – Tabela de dados estatísticos da execução do problema B

Problema B 10×7	50×50		70×70		90×90	
	PSO Base	**Dinâmico**	**PSO Base**	**Dinâmico**	**PSO Base**	**Dinâmico**
Média	18.129	17.521	18.05	17.404	18.286	17.307
Mediana	18.0	17.5	18.0	18.0	18.0	17.0
Desvio Padrão	1.535	1.394	1.585	1.434	1.275	1.416
Variância	2.355	1.942	2.512	2.055	1.626	2.006
Mínimo	13	13	13	13	15	13
Máximo	21	21	22	22	22	21

um *makespan* em média $4,07\%$ menor e com um desvio padrão $4,08\%$ menor e uma variância $7,83\%$ menor em relação à abordagem padrão.

Figura 11 – Problema B — Gráfico de Frequência de Makespan



A porcentagem de melhora em relação ao PSO padrão se mostrou melhor na média de *makespan* obtidos, porém foi menor nos valores de variância e de desvio padrão, isso parece estar relacionado ao tamanho do *dataset* anterior ser pequeno, o que gera menos resultados possíveis e não existe muito espaço de melhora.

3.7 PROBLEMA C

O problema C tem um tamanho 10×10, ou seja, com 10 *jobs* e 10 máquinas, sendo esse o segundo maior dos *datasets* analisados. Na Figura 12 Na Tabela 6 é representado as estatísticas entre os valores de *makespan* obtidos pelas execuções de cada abordagem. É possível ver que a abordagem dinâmica apresentou em um *makespan* em média $6,23\%$ menor e com um desvio padrão $10,38\%$ menor e uma variância $19,94\%$ menor em relação à abordagem padrão. Essa porcentagem monstra que a tendência de melhora é reforçada em *datasets* maiores.

Figura 12 – Problema C — Gráfico de Frequência de Makespan

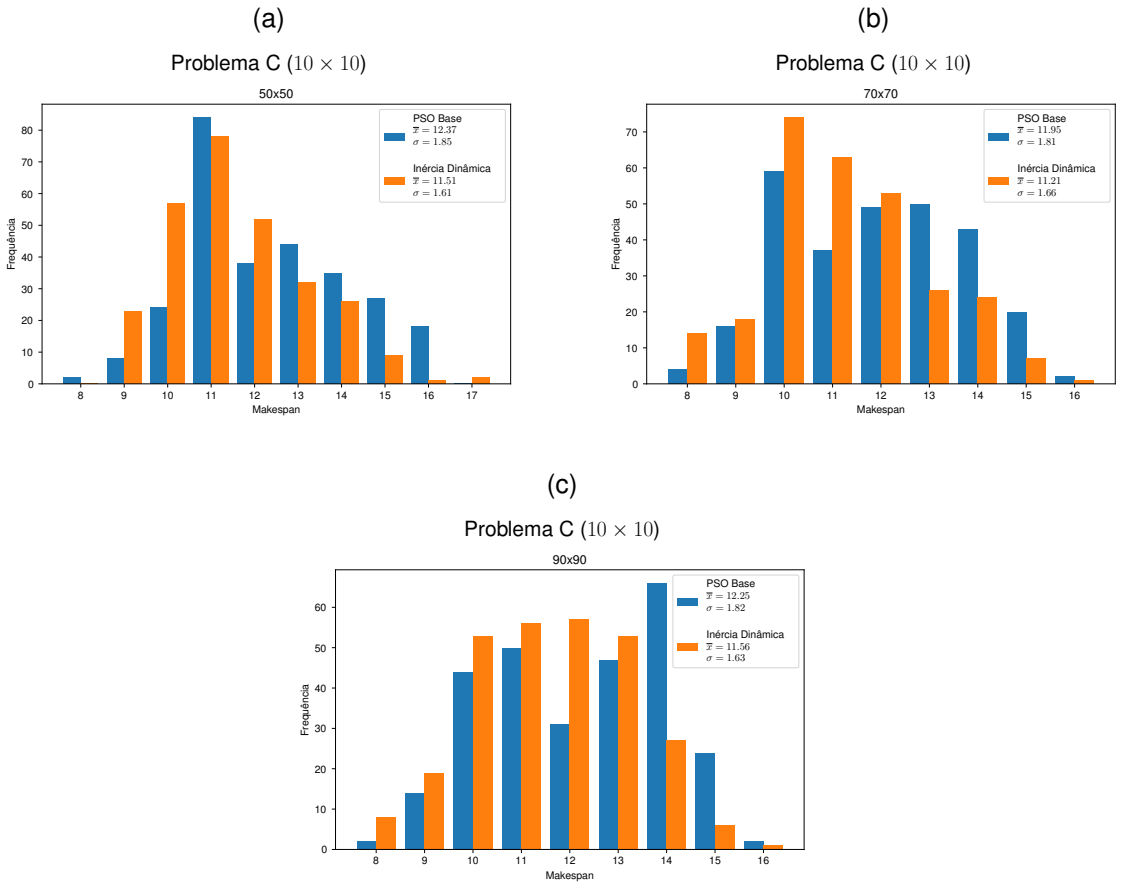


Tabela 6 – Tabela de dados estatísticos da execução do problema C

Problema C 10×10	50×50		70×70		90×90	
	PSO Base	**Dinâmico**	**PSO Base**	**Dinâmico**	**PSO Base**	**Dinâmico**
Média	12.368	11.514	11.946	11.207	12.254	11.564
Mediana	12.0	11.0	12.0	11.0	12.0	12.0
Desvio Padrão	1.853	1.615	1.807	1.658	1.818	1.631
Variância	3.433	2.607	3.265	2.75	3.304	2.66
Mínimo	8	9	8	8	8	8
Máximo	16	17	16	16	16	16

3.8 PROBLEMA D

O problema D tem um tamanho 15×10, ou seja, com 10 *jobs* e 10 máquinas, sendo esse o maior dos *datasets* analisados. Na Figura 13 Na Tabela 7 é representado as estatísticas entre os valores de *makespan* obtidos pelas execuções de cada abordagem. É possível ver que a abordagem dinâmica apresentou em um *makespan* em média $4,14\%$ menor e com um desvio padrão $12,06\%$ menor e uma variância $22,77\%$ menor em relação à abordagem padrão.

Nesse caso vemos uma redução na melhora de comparado aos testes anteriores, porém continua a aumentar a redução do desvio padrão e da variância. Porém, é

Figura 13 – Problema D — Gráfico de Frequência de Makespan

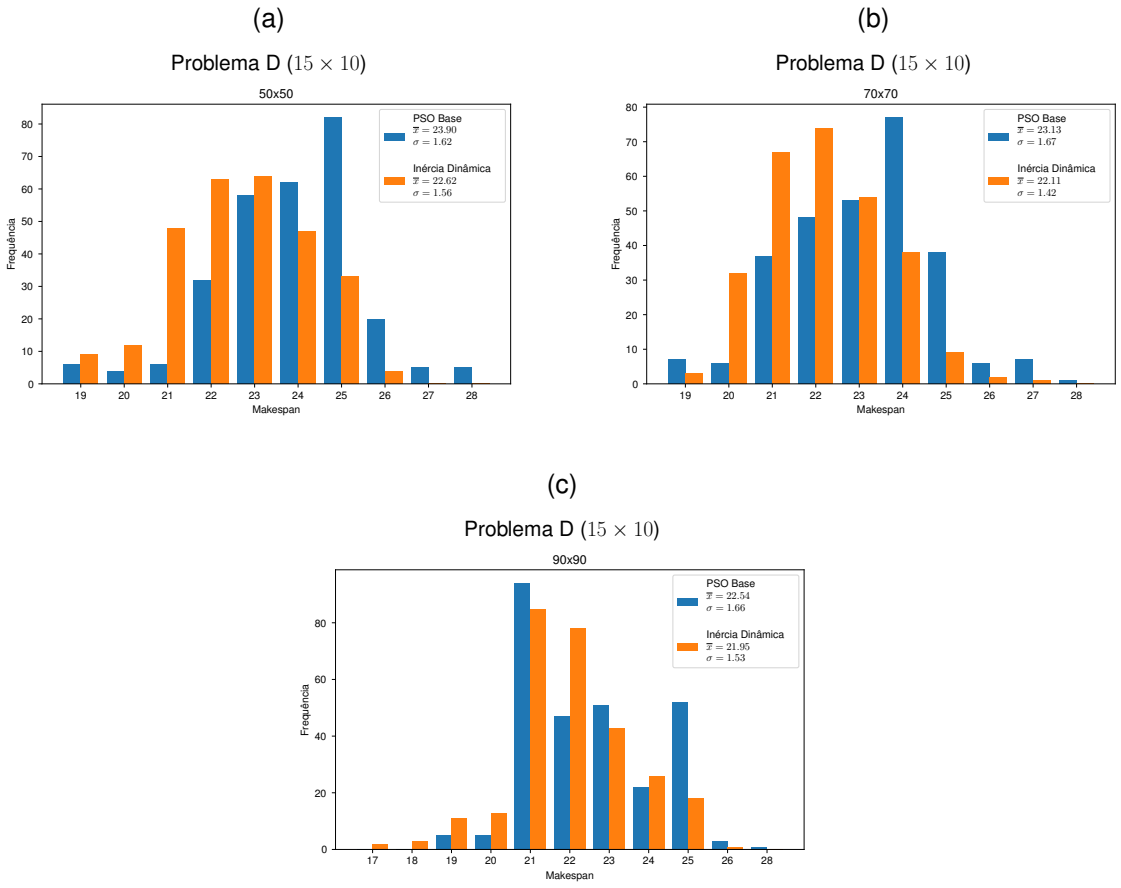


Tabela 7 – Tabela de dados estatísticos da execução do problema D

Problema D 15×10	50×50		70×70		90×90	
	PSO Base	**Dinâmico**	**PSO Base**	**Dinâmico**	**PSO Base**	**Dinâmico**
Média	23.896	22.621	23.129	22.107	22.536	21.954
Mediana	24.0	23.0	23.0	22.0	22.0	22.0
Desvio Padrão	1.623	1.561	1.666	1.418	1.658	1.533
Variância	2.636	2.435	2.776	2.01	2.749	2.351
Mínimo	19	19	19	19	19	17
Máximo	28	26	28	27	28	26

possível ver que na Figura 13c a abordagem dinâmica entrou soluções com um valor de *makespan* de 17 e 18, valores que não foram encontrados pela abordagem padrão.

Por fim, podemos concluir que com as reduções de *makespan* de 1.16%, 4.07%, 6.23%, 4.14%, de desvios padrões de 40%, 4.08%, 10.38%, 12.06%, e de variâncias de 62.5%, 7.83%, 19.94%, 44.77%, vemos que na média houve uma redução de $3,90\%$ no *makespan*, de $16,63\%$ no desvio padrão e de $33,76\%$ na variância. Essa abordagem não traz uma redução significativa no tempo final de *makespan*, porém traz um menor desvio padrão e menor variância, fazendo assim com que o algoritmo PSO tenha uma maior confiabilidade de bons resultados.

4 CONCLUSÃO E TRABALHOS FUTUROS

O FJSP é um problema complexo de análise combinatória e por isso já foram propostos diversas abordagens para conseguir uma solução, dentre essas abordagens está o algoritmo PSO, um algoritmo populacional bio inspirado que simula o comportamento de enxames de abelhas e bandos de aves para encontrar um objetivo em um espaço de soluções. Contudo, esse algoritmo pode acabar convergindo para um mínimo local e assim atingindo uma solução não ótima.

Nesse trabalho foi proposto e implementado um componente dinâmico de cálculo de inércia na movimentação das partículas, com o objetivo de conseguir fazer as partículas escaparem desses mínimos locais. Essa nova abordagem foi testada em conjunto com a abordagem padrão do PSO 20 vezes em 20 espaços de soluções distintos em 5 *datasets* de problemas diferentes, os mesmos utilizados nos *benchmarks* de Kacem, Hammadi e Borne (2002).

Os resultados apontam que houve em média uma redução de $3,90\%$ no tempo total de *makespan* da solução, com um desvio padrão $16,63\%$ menor e com $33,76\%$ menos variância. Indicando que essa abordagem dinâmica trouxe melhores resultados e com uma menor variação, tornando assim esses bons resultados mais frequentes.

A partir desses resultados é possível indicar trabalhos futuros que adicionem novos mecanismos dinâmicos em variáveis de direção das partículas, ou que mostrem a variância desse resultado em espaços de soluções com diferentes padrões.

REFERÊNCIAS

BAGCHI, T. P. Multiobjective scheduling by genetic algorithms. Kluwer Academic Publishers, p. 358, 1999. 12, 16, 18, 23

CARVALHO, L. C. F.; FERNANDES, M. A. Multi-objective Flexible Job-Shop scheduling problem with DIPSO: More diversity, greater efficiency. *Proceedings of the 2014 IEEE Congress on Evolutionary Computation, CEC 2014*, Institute of Electrical and Electronics Engineers Inc., p. 282–289, sep 2014. 26

CHENG, R.; GEN, M.; TSUJIMURA, Y. A tutorial survey of job-shop scheduling problems using genetic algorithms—I. representation. *Computers & Industrial Engineering*, Pergamon, v. 30, n. 4, p. 983–997, sep 1996. ISSN 0360-8352. 17

ESWARAMURTHY, V. P.; TAMILARASI, A. Hybridization of ant colony optimization strategies in tabu search for solving job shop Scheduling problems. *International Journal of Information and Management Sciences*, v. 20, n. 2, p. 173–189, 2009. ISSN 10171819. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.551.1898>. 13

Fernández Martínez, J. L.; García Gonzalo, E. The PSO family: deduction, stochastic analysis and comparison. *Swarm Intelligence 2009 3:4*, Springer, v. 3, n. 4, p. 245–273, oct 2009. ISSN 1935-3820. Disponível em: <https://link.springer.com/article/10.1007/s11721-009-0034-8>. 25

GRAHAM, R. L. et al. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics*, Elsevier, v. 5, n. C, p. 287–326, jan 1979. ISSN 01675060. 17

KACEM, I.; HAMMADI, S.; BORNE, P. Pareto-optimality approach for flexible job-shop scheduling problems: Hybridization of evolutionary algorithms and fuzzy logic. *Mathematics and Computers in Simulation*, v. 60, n. 3-5, p. 245–276, sep 2002. 19, 20, 27, 39, 44

KENNEDY, J.; EBERHART, R. Particle swarm optimization. In: *Proceedings of ICNN'95 - International Conference on Neural Networks*. [S.l.: s.n.], 1995. v. 4, p. 1942–1948 vol.4. 24

KIM, J. Y. et al. Optimal power system operation using parallel processing system and PSO algorithm. *International Journal of Electrical Power & Energy Systems*, Elsevier, v. 33, n. 8, p. 1457–1461, oct 2011. ISSN 0142-0615. 25

LAWLER, E. L. et al. Sequencing and scheduling: Algorithms and complexity. *Handbooks in Operations Research and Management Science*, Elsevier, v. 4, n. C, p. 445–522, jan 1993. ISSN 09270507. 16

LENSTRA, J. K.; Rinnooy Kan, A. H. Computational Complexity of Discrete Optimization Problems. *Annals of Discrete Mathematics*, Elsevier, v. 4, n. C, p. 121–140, jan 1979. ISSN 0167-5060. 17

MACCARTHY, B. L.; LIU, J. Addressing the gap in scheduling research: a review of optimization and heuristic methods in production scheduling. *https://doi.org/10.1080/00207549308956713*, Taylor & Francis Group, v. 31, n. 1, p. 59–79, 2007. Disponível em: <https://www.tandfonline.com/doi/abs/10.1080/00207549308956713>. 17

THONGKRAIRAT, S.; CHUTCHAVONG, V. A Time Improvement PSO Base Algorithm Using Multithread Programming. In: *2019 4th International Conference on Communication and Information Systems (ICCIS)*. [S.l.]: IEEE, 2019. ISBN 978-1-7281-6297-3. 25

WARI, E.; ZHU, W. A survey on metaheuristics for optimization in food manufacturing industry. *Applied Soft Computing*, Elsevier, v. 46, p. 328–343, sep 2016. ISSN 1568-4946. 12, 21

XHAFA, F.; ABRAHAM, A. *Metaheuristics for Scheduling in Industrial and Manufacturing Applications Studies in Computational Intelligence , Volume 128*. Springer Berlin Heidelberg, 2008. v. 128. 345 p. (Studies in Computational Intelligence, v. 128). ISBN 9783540776116. Disponível em: <http://portal.acm.org/citation.cfm?id=1481665>. 12