UM ALGORITMO GRASP COM FASE DE DIVERSIFICAÇÃO PARA PROBLEMA DE LAYOUT DE FACILIDADES EM FILA ÚNICA

Gildasio Lecchi Cravo
Universidade Federal do Espírito Santo - UFES
Av. Fernando Ferrari, 514 - Goiabeiras, Vitória - ES, 29075-910
Programa de Pós-graduação em Informática – PPGI

Faculdades Integradas de Aracruz - FAACZ
Rua Professor Berilo Basílio dos Santos, 180 - Centro, Aracruz - ES, 29194-910
gildasio@fsjb.edu.br

André Renato Sales Amaral
Universidade Federal do Espírito Santo - UFES
Av. Fernando Ferrari, 514 - Goiabeiras, Vitória - ES, 29075-910
amaral@inf.ufes.br

RESUMO

Este trabalho considera o problema de layout de facilidades em fila única (SRFLP – *single-row facility layout problem*). O SRFLP é um problema NP-difícil que consiste em arranjar facilidades ao longo de uma linha reta, tendo como objetivo a minimização da soma ponderada das distâncias entre todos os pares de facilidades. Uma implementação da meta-heurística GRASP (*Greedy Randomized Adaptive Search Procedure*) é apresentada para resolver o problema. A novidade da implementação proposta é que esta foi dotada de uma fase de diversificação. O algoritmo foi testado utilizando 53 instâncias consideradas de grande porte disponíveis na literatura e os resultados obtidos foram comparados com os melhores disponíveis. Nos testes computacionais foram obtidas soluções com boa qualidade para as instâncias testadas.

PALAVARAS CHAVE. SRFLP, Meta-heurística, GRASP.

Área principal (Meta-heurística, Otimização Combinatória, PO na Indústria)

ABSTRACT

This paper considers the single-row facility layout problem (SRFLP). The SRFLP is a NP-hard problem that consists of arranging facilities along a straight line in order to minimize the weighted sum of distances between all pairs of facilities. An implementation of the Greedy Randomized Adaptive Search Procedure (GRASP) meta-heuristic is presented to solve the problem. The novelty of the proposed implementation is that it was endowed with a phase of diversification. The algorithm was tested using 53 large instances available in the literature and the results obtained were compared to the best available. In the computational tests, solutions with good quality were obtained for the tested instances.

KEYWORDS. SRFLP, Meta-heuristic, GRASP.

Main area (Meta-heuristics, Combinatorial Optimization, PO in Industry)

1. Introdução

O problema de layout de facilidades em fila única (*single row facility layout problem* – SRFLP) foi proposto primeiramente por [Simmons 1969] e possui uma grande gama de aplicações práticas como, por exemplo, arranjo de departamentos ao longo de um corredor em supermercados, hospitais ou escritórios [Simmons 1969], arranjo de livros em uma prateleira [Picard e Queyranne 1981] entre outros. Formalmente pode-se definir o problema como a seguir [Kothari e Ghosh 2012]:

Encontrar uma permutação $\prod=\{\pi_1, \pi_2, ..., \pi_n\}$ do conjunto de facilidades $N=\{1, 2, ..., n\}$, que forneça o menor valor para a expressão:

$$\sum_{1\le i<j\le n} c_{\pi_i\pi_j} d_{\pi_i\pi_j} \quad (1)$$

Onde $d_{\pi_i\pi_j} = \frac{l_{\pi_i} + l_{\pi_j}}{2} + \sum_{i<k<j} l_{\pi_k}$ é a distância entre os centros das facilidades π_i e π_j de N dispostas na permutação $\prod$. Temos que l_k é o comprimento da facilidade k; c_{ij} o fluxo entre as facilidades i e j; n é a quantidade de facilidades.

A figura 1 mostra um exemplo de aplicação do SRFLP em uma linha de produção.

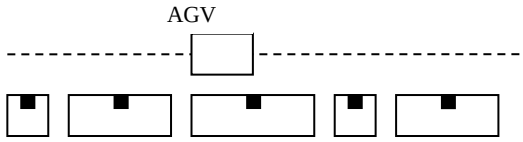

Figura 1 – arranjo de facilidades em fila única

Na figura 1, o veículo (*automated guided vehicle - AVG*) desloca-se em linha reta e cada retângulo representa uma estação de trabalho, onde o veículo deverá entregar ou coletar uma ferramenta ou equipamento.

Ao longo dos anos o SRFLP vem recebendo considerável atenção dos pesquisadores. [Simmons 1969] propôs um algoritmo *branch-and-bound*; [Picard e Queyranne 1981] apresentaram um algoritmo de programação dinâmica; [Amaral 2009] propôs um algoritmo de planos de corte; [Hungerländer e Rendl 2013] testaram uma abordagem baseada em programação semidefinida; e [Amaral e Letchford 2013] propuseram um algoritmo *branch-and-cut*. Outros autores apresentaram formulações de programação inteira [Love e Wong 1976], [Heragu e Kusiak 1991], [Amaral 2006, 2008a]. Os métodos exatos mais eficientes são os propostos por [Amaral 2009] e [Hungerländer e Rendl 2013]. Entretanto, o maior tamanho de instância que pôde ser resolvida eficientemente, até a presente data, possui no máximo cerca de 40 facilidades. Para problemas de maior porte torna-se proibitiva a aplicação de métodos exatos já que o SRFLP um problema NP-Difícil [Amaral 2006]. Métodos para encontrar limitantes inferiores para o SRFLP são descritos em [Anjos et al. 2005], [Amaral 2009], [Anjos e Yen 2009], [Hungerländer e Rendl 2013] e [Amaral e Letchford 2013].

Diversas abordagens heurísticas foram propostas para o SRFLP, tais como, heurística gulosa [Kumar et al. 1995]; *simulated annealing* [Heragu e Alfa 1992], [De Alvarenga et al. 2000]; *enhanced local search* [Amaral 2008b]; *tabu search* [De Alvarenga et al. 2000], [Samarghandi e Eshghi 2010], [Kothari e Ghosh 2013]; algoritmos genéticos [Ozcelik 2011], [Datta et al. 2011], [Kothari e Ghosh 2014a]; *Scatter search* [Kumar et al. 2008], [Kothari e Ghosh, 2014b]; colônia de formigas [Solimanpur et. al. 2005]; p*article swarm* [Samarghandi et al. 2010] e GRASP [Cravo e Amaral 2015].

Mais recentemente, [Palubeckis 2015] introduziu técnicas de buscas locais rápidas, as quais permitiram-lhe aplicar *Variable Neighborhood Search* (VNS) para instâncias com até 300

facilidades em tempo razoável. [Guan e Lin 2016] apresentaram um algoritmo híbrido entre VNS e colônia de formigas onde, também, técnicas rápidas para explorar a vizinhança nas buscas locais são aplicadas. [Palubeckis 2016] propôs um algoritmo *multi-start simulated annealing*. Os melhores resultados até o momento são reportados em [Palubeckis 2016].

O presente trabalho apresenta uma implementação de uma meta-heurística GRASP com uma etapa adicional de diversificação bem conhecida na literatura sobre busca tabu. Tal diversificação é baseada em uma memória de freqüência que favorece a incorporação de atributos que foram menos incorporados no passado.

O algoritmo GRASP desenvolvido neste estudo foi aplicado a instâncias de grande porte do SRFLP, com tamanhos de 110 a 1000 facilidades, e os resultados obtidos foram comparados os melhores resultados da literatura.

O restante do trabalho será dividido da seguinte forma: a seção 2 apresenta uma breve revisão sobre a meta-heurística GRASP. Na seção 3 é apresentada a implementação proposta, em seguida, na seção 4, são expostos os testes computacionais e os resultados encontrados. Na seção 5 estão as conclusões obtidas, seguidas das referências bibliográficas utilizadas.

2. Meta-heurística GRASP

O GRASP, proposto por [Feo e Resende 1995], é um método iterativo constituído de duas fases: a de construção e a de busca local. Na fase de construção, uma solução é construída elemento a elemento. Gera-se uma *lista de candidatos* (LC) contendo elementos de uma solução do problema, ordenados de acordo com sua contribuição na função objetivo (ou de acordo com algum critério estabelecido). Os melhores elementos de LC formam uma lista, denominada *lista de candidatos restrita* (LCR). A construção de uma solução é probabilística, pois a escolha de cada elemento que será retirado da LCR para ser adicionado à solução é feita aleatoriamente. A heurística também é adaptativa, pois, a cada iteração da fase de construção, são atualizadas as contribuições dos elementos restantes na LC a fim de refletir as mudanças ocasionadas pela seleção do elemento adicionado à solução parcial na iteração anterior. Um parâmetro importante no GRASP é o *tamanho* da lista LCR (LCR_tam) que determina o quão guloso ou aleatório será o procedimento.

Provavelmente, uma solução gerada ao final da fase de construção, não será localmente ótima. Daí a importância da segunda fase do GRASP, a busca local, que tenta melhorar a solução constituída na fase de construção, explorando a sua vizinhança.

Existem na literatura diversas aplicações práticas do método GRASP em problemas de otimização na indústria. Dentre esses estão inclusos problemas de roteirizarão, lógica, particionamento, localização, teoria dos grafos, transporte, telecomunicações, projeto VLSI, problema de rotulação cartográfica de pontos [Cravo et al. 2008], projeto de rede de transmissão [Faria et al. 2005]. Outras aplicações do GRASP podem ser vistas em [Resende e Ribeiro 2005].

3. Procedimento proposto para o SRFLP

O algoritmo GRASP proposto é uma extensão daquele desenvolvido por [Cravo e Amaral 2015]. Em [Cravo e Amaral 2015], o cálculo do ganho na função objetivo dos movimentos nas estruturas de vizinhanças exigia o cálculo completo da função objetivo. Em um trabalho recente, [Guan e Lin 2016] apresentaram técnicas para o cálculo do ganho, na função objetivo, sem a necessidade do cálculo completo. Essas técnicas foram utilizadas no algoritmo GRASP aqui proposto. Além disso, uma etapa de diversificação foi adicionada na tentativa de aumentar o espaço de busca. A diversificação é baseada em uma memória de freqüência comumente utilizada na literatura sobre *tabu search*: uma matriz F de tamanho $n \times n$ é mantida e atualizada, onde o elemento $F(i, p)$ da matriz contém a frequência com que a facilidade i ocupou a posição p nas soluções geradas até o momento. Dessa forma, atributos mais raros podem ser identificados e, portanto, favorecidos.

3.1. Fase Construtiva do GRASP

Foi utilizado o procedimento disponível em [Cravo e Amaral 2015] para a geração da solução inicial para o GRASP. Uma lista LC com os possíveis pares de facilidades que poderão compor uma solução a partir da posição livre mais a esquerda *p* e da posição livre mais a direta *q* na solução parcial corrente.

Quando uma solução *S* está vazia, o custo w_{ij} do par de facilidades *i* e *j*, tal que *i* será colocada na posição livre p=1 e *j* será colocada na posição livre q=n, é calculado como:

$$w_{ij} = c_{ij}\left(\frac{l_i + l_j}{2} + \sum_{1\le k\le n; k\ne i, k\ne j} l_k\right),\ (1 \le i < j \le n) \qquad (2)$$

Assim, a LC é criada, ordenada pelo custo w_{ij}. Os *LCR_tam* melhores elementos de LC definem a LCR e então, um elemento da lista LCR é sorteado e adicionado à solução *S* vazia nas posições *p* e *q*. A partir deste momento, como a solução não é mais vazia, os custos w_{ij} serão recalculados, levando em consideração os elementos já pertencentes à solução *S*, pela equação:

$$w_{ij} = c_{ij}\left(\frac{l_i + l_j}{2} + \sum_{k\ne i,\, k\ne j,\, k\notin S} l_k\right) + \sum_{k\in S'}(d_{ki}c_{ki} + d_{kj}\,c_{kj}) + \sum_{k\in S''}(d_{ki}c_{ki} + d_{kj}\,c_{kj}),\quad \forall i, j \notin S \qquad (3)$$

S' é o conjunto dos elementos à esquerda da próxima posição livre *p* na solução *S*; e *S"* o conjunto dos elementos à direita da próxima posição livre *q* na solução *S*.

Na equação (3), para um par candidato (*i*, *j*), $i, j \notin S$, será escolhida a melhor configuração, pois tem-se a possibilidade de colocar a facilidade *i* na posição *p* e a facilidade *j* na posição *q* ou *i* na posição *q* e *j* na posição *p*. A Figura 2 ilustra a aplicação das expressões (2) e (3) para um problema com 6 facilidades. A figura 3 apresenta o pseudocódigo da etapa de construção.

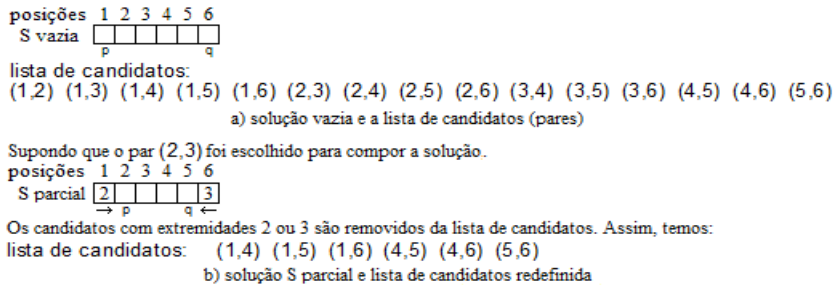

Figura 2 – aplicação das expressões (2)-(3) na geração de uma solução inicial [Cravo e Amaral 2015]

```
Procedimento ConstruçãoGulosaAleatoria
Dados: n, LCR_Tam //Tamanho da lista de candidatos restrita
Resultado: S
01: Alocados = {"F", ...,"F"} // F- não alocado, V-alocado
02: S ← {} //Lista de facilidades, ou seja, a solução
03: p ← 1, q ← n // posições livres
04: LC ← CriarLC_Inicial();
05: Enquanto ( LC ≠ {} ) Faça
06:    par ← Sortear (LCR_tam, LC);
07:    S(p) ← par.i;
08:    S(q) ← par.j;
09:    p ← p + 1;
10:    q ← q - 1;
11:    Alocados(par.i) ← Alocados(par.j) ← " V ";
12:    RemoverConflitos(par, LC);
13:    AtualizarListaCandidatos(S, LC);
14:    OrdenarListaCandidatos(LC);
15: Fim-Enquanto
16:    Se (n é ímpar) Então
17:       S((n/2)+1) ← BuscaNãoAlocado (Alocados);
18:    Fim-Se
19: Retorne S;
20: Fim-ConstruçãoGulosaAleatoria
```

Figura 3 – pseudocódigo para a fase de construção do GRASP [Cravo e Amaral 2015]

O procedimento construtivo mostrado na Figura 3 recebe *n* (quantidade de facilidades) e *LCR_tam* (quantos elementos da LC fazem parte da LCR). Na linha 1, 2 e 3 são inicializados: o vetor "*Alocados*" que controla as facilidades já alocadas na solução *S* colocando "F" (Falso) para todas as facilidades, a solução *S* como vazia e as posições livres *p* e *q*, sendo *p* a posição mais esquerda e *q* a posição mais a direita na solução *S*. Na linha 4, a LC é criada conforme a expressão (2).

O laço *Enquanto*, definido nas linhas de 5 até 15, executa até que não se tenha mais candidatos na LC. Na linha 6, o candidato para compor a solução é sorteado e, são adicionadas em *S* as facilidades nas posições livres (linha 7 e 8); incrementa-se a posição livre *p* e decrementa-se a posição livre *q* (linhas 9 e 10) e marcam-se como alocadas as facilidades *i* e *j* do par selecionado, na linha 11. Na linha 12, removem-se os pares que têm extremidade conflitante; na linha 13, atualizam-se os custos dos candidatos remanescentes na LC e na linha 14 ordena-se a LC novamente pelos custos (atualizados conforme a expressão 3).

O bloco da linha 16 verifica se a instância possui uma quantidade ímpar de facilidades, pois, se possuir, a solução deverá ser acertada, buscando a facilidade ainda não alocada para colocá-la na posição central da permutação *S*.

O procedimento *ConstruçãoGulosaAleatoria* é um algoritmo guloso, aleatório e a propriedade adaptativa reside no fato de que a cada par de facilidades adicionado à solução *S*, os elementos conflitantes são removidos e os custos dos elementos restantes são atualizados, sendo recalculados levando-se em consideração a solução parcial *S* que está sendo construída. Ao final dessa etapa uma solução viável inicial *S* é devolvida (linha 19).

Como a construção da permutação *S* é feita de forma gulosa e aleatória é provável que a solução não seja localmente ótima. A fim de melhorar a solução inicial encontrada nessa primeira fase, uma busca local é aplicada.

3.2. Fase de busca local do GRASP

Na implementação presente, a segunda fase do GRASP aplica não uma, mas cinco estratégias de busca local, baseadas nas estruturas de vizinhanças *2OPT*, *inserção e swap*. Essas estruturas são bastante usadas na literatura para o SRFLP [Ozcelik 2011], [Kothari e Ghosh 2013], [Palubeckis 2015]. A Figura 4 ilustra o movimento *2OPT*.

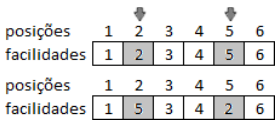

Figura 4 – Movimento *2OPT* [Cravo e Amaral 2015]

A Figura 4 mostra um exemplo do movimento *2OPT*. A facilidade que estava na posição 2 (facilidade 2) foi trocada com a facilidade que estava na posição 5 (facilidade 5). Três estruturas de vizinhança *2OPT* são utilizadas: *2_opt_par_aleatorio*, *2_opt_par_sequencial* e *swap*. A estrutura *2_opt_par_aleatorio* verifica todos os pares de posições, sorteando as posições (*p, q)* de uma lista com os possíveis pares de posições. A estrutura *2_opt_par_sequencial* explora sequencialmente os pares de posições, ou seja, todos os pares de posições (*p, q*) com $p = 1, ..., n$; $q = i, ..., n$; $p \neq q$; são explorados. A estrutura de vizinhança *swap* aplica o movimento *2OPT* para pares de posições consecutivas sendo $p = 1, ..., n-1$ e $q = p+1$ e em seguida $p=n, ..., 2$ e $q = p - 1$. São aplicados somente os movimentos que melhoram a função objetivo.

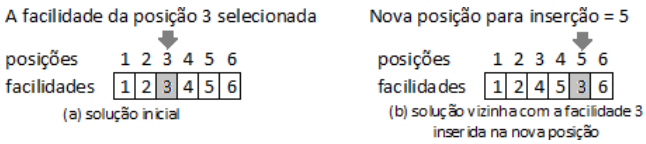

Figura 5 – movimento das estruturas de vizinhança por inserção [Cravo e Amaral 2015]

A Figura 5 mostra um exemplo do movimento de *inserção*. A facilidade que estava na

posição 3 da solução foi selecionada e inserida na posição 5. Para isso, as facilidades das posições 4 e 5 foram deslocadas para que a facilidade selecionada fosse inserida na posição 5. Duas estruturas de vizinhança por inserção são utilizadas: *inserção_par_aleatorio* e *inserção_par_sequencial*. A estrutura *inserção_par_aleatorio*, sorteia de uma lista com os possíveis pares de posições, o par de posições (p, q), enquanto que a *inserção_par_sequencial* é feita sequencialmente percorrendo todas as posições p=1,..., n da solução, tentando inserir a facilidade da posição p nas posições q = 1,..., n com $q \neq p$. São aplicados somente os movimentos que melhoram a função objetivo.

Casa uma das cinco estruturas exploram todas as possibilidades antes de retornar se a solução corrente melhorou ou não.

Os movimentos *2OPT, inserção* e *swap* das estruturas de vizinhança, utilizam para a avaliação da função objetivo, as técnicas apresentadas no trabalho de [Guan e Lin 2016]. Os autores apresentam técnicas para o cálculo do ganho na função objetivo ocorrido pela troca de posições entre duas facilidades (ou a inserção de uma facilidade em uma nova posição) sem que seja necessária uma reavaliação completa da solução, ocasionando assim um ganho no tempo computacional para a exploração da vizinhança. A Figura 6 mostra o pseudocódigo da busca local.

```
Procedimento BuscaLocal
Dados: S
Resultado: S
01:   Faça
02:         melhorou ← 2_opt_par_aleatorio(S)
03:         melhorou ← melhorou + inserção_par_aleatorio(S)
04:         melhorou ← melhorou + swap(S)
05:         melhorou ← melhorou + 2_opt_sequencial(S)
06:         melhorou ← melhorou + inserção_par_sequencial(S)
07:   Enquanto (melhorou > 0)
08: Retorne S
09: Fim-BuscaLocal
```

Figura 6 – pseudocódigo da fase de busca local do GRASP

Na figura 6, a busca local aplica sistematicamente as cinco estruturas de vizinhança à solução corrente *S*, recebida como entrada. O laço das linhas 1 à 7 é repetido enquanto houver melhora na solução corrente *S*. As estruturas de vizinhanças são aplicadas na seguinte ordem: *2_opt_par_aleatorio* (linha 2), *inserção_par_aleatorio* (linha 3), *swap* (linha 4) *2_opt_par_sequencial* (linha 5) e *inserção_par_sequencial* (linha 6). Ao final do laço 1-7 a solução *S* modificada pela busca local é devolvida (linha 8).

3.3. Fase de diversificação

A fase de diversificação baseia-se na matriz *F* de frequência e na definição da contribuição de cada facilidade para o valor objetivo corrente.

3.3.1. Matriz de frequência

A matriz *F* de frequência contém n x n elementos inteiros, sendo n o número de facilidades. Cada linha da matriz *F* representa uma facilidade e cada coluna da matriz representa uma posição na solução. O valor do elemento (i, p) da matriz fornece o número de vezes que em que uma facilidade i foi colocada na posição p em iterações anteriores. Assim, a matriz de frequência representa uma memória de longo prazo, que servirá para realizar diversificação. A matriz *F* de frequência é atualizada a partir da solução *S* fornecida pela Busca Local em cada iteração do algoritmo GRASP. A cada atualização, o elemento (facilidade, posição) da matriz de frequência *F* é incrementado de uma unidade.

3.3.2. Contribuição da facilidade para o valor objetivo corrente

Na solução corrente, cada posição *p* é ocupada por uma facilidade que contribui com um dado montante m_p para o valor objetivo corrente.

O montante m_p, em um primeiro momento, é obtido quando do cálculo do valor objetivo para a nova solução *S* gerada na fase construtiva do GRASP e pode ser expresso, para cada posição *p* da solução como:

$$m_p = \sum_{j=1, j\neq p}^{n} (c_{\pi_p \pi_j} d_{\pi_p \pi_j}) \quad (4)$$

onde π_k é a facilidade da posição *k* na solução *S*. Com a aplicação da busca local, os valores de m_p serão atualizados à medida que ocorrem melhorias no custo da solução corrente.

Para a atualização dos pesos m_p à cada troca que resulte em ganho no valor objetivo corrente, utilizaram-se dois procedimento para a atualização dos pesos m_p. No movimento *2OPT* ou *swap*, envolvendo as facilidades das posições *p* e *q*, resultando em um ganho w_{pq} no valor objetivo, atualizam-se os valores de m_p *e* m_q com o valor médio (w_{pq}/n) do ganho. Já no movimento de *Inserção*, que realiza a inserção da facilidade *p* na posição *q*, se for obtido um ganho w_{pq} no valor objetivo, os valores de m_k com $k=p,..., q$ serão atualizados também com o valor médio (w_{pq}/n) do ganho. Portanto, sendo as atualizações feitas dessa maneira, apenas facilidades diretamente envolvidas nos movimentos terão os pesos atualizados.

3.3.3. Diversificação

A diversificação é feita tomando-se *k* facilidades e alterando-se suas posições na solução. Vimos que se uma facilidade ocupa a posição *p* na solução corrente, então essa facilidade contribui um montante m_p para o valor objetivo corrente. Então, identifica-se um conjunto *K*, contendo as *k* facilidades com as menores contribuições para o valor objetivo corrente e toma-se, aleatoriamente, uma facilidade *i* do conjunto *K*, até que *K* esteja vazio.

Para cada facilidade $i \in K$, tem-se da linha *i* da matriz *F*, a frequência com que a facilidade *i* ocupou cada posição de 1 a *n* no passado. Forma-se, então uma lista contendo as posições ordenadas por frequência, sendo as posições mais raramente ocupadas pela facilidade *i* primeiro. Um parâmetro *LCR_diver* é utilizado para limitar o tamanho da lista de posições destino, de forma semelhante à que é feita na fase construtiva do GRASP. Sorteia-se a nova posição para a facilidade *i* dentre as primeiras *LCR_diver* posições candidatas.

3.4. O procedimento GRASP proposto

A figura 7 apresenta o pseudocódigo para o algoritmo GRASP proposto. O algoritmo recebe como entrada o tamanho da instância *n*, o valor de *LCR_tam*, que define o tamanho da *LCR* da fase construtiva do GRASP; o valor *Iter_nova*, que é o número de iterações sem melhora na melhor solução encontrada *S** e define quando deve-se gerar uma nova solução do GRASP; o parâmetro *k* que define quantos elementos serão realocados na etapa de diversificação; e *LCR_diver*, que limita o tamanho da lista das posições destino no procedimento de diversificação.

```
Procedimento GRASP
Dados: n, LCR_tam, Iter_nova, k, LCR_diver
Resultado: S*
01: f* ←Inf
02: Iter ← 0
03: iterUltimaMelhora ← 0
04: S ← ConstruçãoGulosaAleatoria(n, LCR_tam)
05:   Enquanto (Critério de Parada Não Satisfeito)
06:         Se ((Iter - iterUltimaMelhora) > Iter_nova)
07:               S ← ConstruçãoGulosaAleatoria(n, LCR_tam)
08:         Senão Se (Iter > 0)
09:               AplicaçãoDiversificação(F, S, k, LCR_diver)
10:         Fim-Se
11:         S ← BuscaLocal(S)
12:         AtualizarFrequencias(S, F)
13:         Se (f(S) < f*)
14:               f* ← f(S)
15:               S* ← S
16:               iterUltimaMelhora ← Iter
17:         Fim-Se
18:   Iter ←Iter + 1
19:   Fim-Enquanto
20: Retorne S*
21: Fim-GRASP
```

Figura 7 – Pseudocódigo do algoritmo GRASP proposto

Nas linhas 1 a 4 são inicializadas as variáveis locais, incluindo a primeira solução inicial (linha 4). No laço das linhas 5 à 19 é definido o processo iterativo do GRASP, que é repetido até que um critério de parada seja satisfeito. Nas linhas 6 a 10, o algoritmo verifica se o número de iterações sem melhora na função objetivo foi atingido (*Iter_nova*). Caso tenha sido atingido o número máximo de iterações sem melhora, o algoritmo constrói uma nova solução *S* (linha 7). Caso contrário, para *iter* > 0, é aplicada uma diversificação na solução *S* corrente (linha 9).

Na linha 11, aplica-se uma busca local na solução *S* (seção 3.2). Na linha 12, é feita a atualização da matriz *F* de frequências utilizando como informação a solução corrente *S*. A matriz de frequências é utilizada na fase de diversificação do algoritmo (linha 9).

Na linha 13, verifica-se se a nova solução é melhor que a melhor solução visitada até o momento. Caso afirmativo, a solução global e a variável *iterUltimaMelhora* são atualizadas.

Ou seja, sempre que o número de iterações sem melhora de *S** é atingido, o algoritmo passa a construir uma solução inicial *S* (linha 7) e aplicar a busca local na solução *S* (linha 11). Quando uma melhora em *S** é atingida (linha 13), a variável *iterUltimaMelhora* é atualizada (linha 19), e o algoritmo aplica o procedimento de diversificação na solução *S* (linha 9) em vez de gerar uma nova solução pela fase construtiva.

4. Experimentos Computacionais

Nos experimentos, foi utilizado um computador desktop com processador Intel i7-7700k com 16GB de memória. O algoritmo GRASP proposto para o SRFLP foi implementado na linguagem de programação C e foi usado o compilador Microsoft Visual Studio *Community* 2017. Para a validação do algoritmo, foram executados testes com três instâncias de [Amaral e Letchford 2011], todas com tamanho com n=110; 20 instâncias de [Palubeckis 2015], uma para cada n=110, 120, ..., 300*;* e 30 instâncias de [Palubeckis 2016], uma para cada n=310, 320, ..., 500, 550, 600, ..., 1000.

O GRASP tem um importante parâmetro que é o tamanho da lista de candidatos restrita (*LCR_Tam*, na Figura 7). Para a calibração desse parâmetro foram utilizadas as três instâncias com n=110 de [Amaral e Letchford 2011]. Nos testes iniciais o GRASP apresentou um comportamento mais robusto para o parâmetro *LCR_tam* = (10%)·n, obtendo a melhor solução conhecida para as instâncias com um tempo de 100s, conforme mostrado na Tabela 1. Outros parâmetros são necessários para a fase de diversificação, o parâmetro *Iter_nova*, *k* e *LCR_diver* que são a quantidade de iterações sem melhora, a quantidade de facilidades que será realocada e o tamanho da lista de candidatos para as novas posições, respectivamente, na fase de diversificação. Nos testes

realizados foram utilizados os valores de *Iter_nova*=10, *k*= (10%)·*n* e valor de *LCR_diver*= (10%)·*n*. O algoritmo GRASP foi executado 10 (dez) vezes para cada instância.

Como critério de parada foi utilizado um tempo limite de execução, em segundos. Para as instâncias de [Amaral e Letchford 2011] o tempo limite de execução foi de 100s. Para as instâncias de [Palubeckis 2015] o tempo limite de execução foi de 600s. Acima desse valor os resultados pareciam não apresentar melhoras. Para as instâncias de [Palubeckis 2016] os tempos limites foram: 1200s para n={310,...400}, 1800s para n={410, ..., 500} e 3600s para $n \geq 510$.

A Tabela 1 apresenta a seguinte estrutura: A coluna "Instância" mostra o nome da instância, a coluna "n" o tamanho da instância, seguida da coluna "Palubeckis (2016)" que contém os melhores valores de solução conhecidos na literatura. A coluna "$GRASP_{Melhor\ FO}$" é a melhor solução do GRASP e "$GRASP_{Média\ FO}$"é a média das soluções em dez execuções do GRASP. A coluna "DP" mostra o desvio padrão. A coluna "Iter" apresenta o total médio de iterações realizadas, seguido do tempo médio. A coluna "Erro (%)" mostra o erro percentual entre a melhor solução do GRASP em relação à melhor solução conhecida na literatura.

Tabela 1 – Soluções para as instâncias n=110 [Amaral e Letchford 2011]

Instância	n	Palubeckis (2016)	$GRASP_{Melhor\ FO}$	$GRASP_{Média\ FO}$	DP	Iter	Tempo(s)	Erro (%)
SRFLP-110-1	110	144296664,50	144296664,50	144296912,2	355,25	592	100	0,00000
SRFLP-110-2	110	86050037,00	86050037,00	86050037	0,00	609	100	0,00000
SRFLP-110-3	110	2234743,50	2234743,50	2234751,7	13,34	596	100	0,00000

Para as instâncias de [Amaral e Letchford 2011], o GRASP encontrou a melhor solução conhecida para todas as instâncias. A robustez do algoritmo pode ser notada pela coluna "DP" onde os valores são relativamente baixos em relação aos valores médios da função objetivo.

A Tabela 2 mostra os resultados para o conjunto de instâncias com n={110, 120, ..., 300} de [Palubeckis 2015].

Tabela 2 – Soluções para as instâncias com $110 \leq n \leq 300$ [Palubeckis 2015]

Instância	n	Palubeckis (2016)	$GRASP_{Melhor\ FO}$	$GRASP_{Média\ FO}$	DP	Iter	Tempo(s)	Erro (%)
p110	110	4435868,0	4435868,0	4435868,0	0	1499	600	0,00000
p120	120	6282721,0	6282721,0	6282721,0	0	1180	600	0,00000
p130	130	7880929,5	7880929,5	7880960,3	76	752	600	0,00000
p140	140	9257162,0	9257162,0	9257165,6	8	672	601	0,00000
p150	150	10624389,5	10624389,5	10624775,1	604	429	601	0,00000
p160	160	14873277,0	14873277,0	14873398,1	57	376	602	0,00000
p170	170	16630187,0	16630187,0	16630464,4	510	272	601	0,00000
p180	180	18746031,5	18746048,5	18746532,8	619	237	602	0,00009
p190	190	24453272,0	24453272,0	24454743,5	906	172	604	0,00000
p200	200	27482649,5	27482653,5	27490576,7	8729	88	602	0,00001
p210	210	29512000,5	29512000,5	29520077,1	6833	109	602	0,00000
p220	220	37351850,5	37352625,5	37358331,8	4099	98	604	0,00207
p230	230	46744886,0	46744886,0	46747968,5	4693	75	605	0,00000
p240	240	46717781,0	46718092,0	46726508,3	8669	69	603	0,00067
p250	250	54526293,5	54528980,5	54545459,7	10521	56	611	0,00493
p260	260	63300360,5	63301683,5	63319048,2	11345	50	605	0,00209
p270	270	68960438,5	68960444,5	68977137,9	31631	41	607	0,00001
p280	280	73845821,0	73845834,0	73888284,8	31589	37	619	0,00002
p290	290	86255267,5	86255267,5	86291155,1	28075	30	611	0,00000
p300	300	95937735,0	95937759,0	95995144,9	39369	23	610	0,00003

Observando a coluna do erro percentual, coluna "Erro (%)", nota-se que a melhor solução encontrada pelo GRASP apresenta um erro relativo baixo em relação a melhor solução da literatura. Por exemplo, o erro máximo observado de $4{,}93 \times 10^{-5}$ (para a instância p250) é muito baixo. A coluna "DP" mostra valores relativamente baixos em relação aos valores médios da função objetivo.

A Tabela 3 mostra os resultados encontrados para as instâncias de [Palubeckis 2016]. Para essas instâncias, o GRASP atingiu valores muito próximos das melhores soluções conhecidas, conforme pode ser visto na coluna "Erro (%)". Ainda é possível notar que para instâncias com n>=550 o tempo máximo de execução é ultrapassado, pois o tempo de cada iteração é alto fazendo com que o algoritmo termine com um tempo final alto.

Tabela 3 – soluções para as instâncias n≥310 [Palubeckis 2016]

Instância	n	Palubeckis (2016)	$GRASP_{Melhor\ FO}$	$GRASP_{Média\ FO}$	DP	Iter	Tempo(s)	Erro (%)
p310	310	105754955,0	105755484,0	105763610,0	9038,7	80	1206	0,00050
p320	320	119522881,5	119522956,5	119563902,7	22254,7	71	1209	0,00006
p330	330	124891823,5	124891975,5	124942142,2	39735,2	63	1206	0,00012
p340	340	129796777,5	129796980,5	129818342,0	19791,8	56	1211	0,00016
p350	350	149594388,0	149595270,0	149630961,3	37673,0	47	1213	0,00059
p360	360	141122187,0	141146462,0	141172859,2	28712,9	42	1217	0,01720
p370	370	174663159,5	174712474,5	174753135,3	37090,8	39	1225	0,02823
p380	380	189452403,5	189492458,5	189573018,9	45794,9	35	1216	0,02114
p390	390	208688557,0	208705208,0	208774301,5	75705,2	31	1221	0,00798
p400	400	213768810,5	213794811,5	213907026,2	78949,1	28	1229	0,01216
p410	410	243494269,0	243505155,0	243620220,0	58252,1	37	1835	0,00447
p420	420	270756527,5	270815329,5	270904834,8	45765,2	34	1828	0,02172
p430	430	286334521,5	286346604,5	286441187,2	71192,0	31	1826	0,00422
p440	440	301067264,5	301104660,5	301191370,6	55491,9	28	1840	0,01242
p450	450	324488485,0	324548815,0	324690899,1	103460,5	25	1836	0,01859
p460	460	314884659,0	314970942,0	315118711,4	68481,8	22	1851	0,02740
p470	470	379529990,0	379596922,0	379789381,7	128099,3	21	1833	0,01764
p480	480	366821075,0	366844366,0	367039308,5	85213,5	19	1852	0,00635
p490	490	413901954,5	414010930,5	414115717,1	139108,0	18	1867	0,02633
p500	500	465570835,5	465664365,5	465834231,0	115550,8	17	1871	0,02009
p550	550	587090450,5	587423710,5	587509841,8	78409,1	20	3712	0,05676
p600	600	801567664,5	801910873,5	802096027,1	135587,4	13	3771	0,04282
p650	650	927512834,0	927933783,0	928189525,8	178166,1	9	3811	0,04538
p700	700	1158462340,0	1159262622,0	1159610593,3	164440,8	6	4026	0,06908
p750	750	1438408860,5	1439437111,5	1439917219,9	297429,4	4	3942	0,07149
p800	800	1861593391,0	1862882958,0	1863624392,7	639908,5	2	4404	0,06927
p850	850	2126675923,0	2128592903,0	2129355445,8	449642,2	2	4869	0,09014
p900	900	2600124305,0	2602604317,0	2603190254,6	578715,8	1	5240	0,09538
p950	950	2993153592,5	2995739830,5	2996906356,8	705615,6	1	6150	0,08641
p1000	1000	3426647267,0	3430852254,0	3431596081,7	474550,9	1	7068	0,12271

5. Conclusões

O algoritmo aqui proposto é uma extensão de [Cravo e Amaral 2015] com (*i)* adição de uma etapa de diversificação ao algoritmo para aumentar o espaço de busca; *(ii)* avaliação de movimentos sem a necessidade de cálculo completo da função objetivo, o que possibilitou uma melhora nos tempos computacionais.

O algoritmo GRASP, executado com tempo limitado, para várias instâncias com $110 \leq n \leq 1000$, atingiu os melhores valores conhecidos na literatura ou valores muito próximos dos melhores conhecidos. Consideradas 10 execuções do algoritmo, este obteve soluções robustas com baixo desvio padrão. Na literatura, há somente dois trabalhos que tratam instâncias com $n \geq 110$: [Palubeckis 2015] que trata problemas com $110 \leq n \leq 300$ e [Palubeckis 2016] com $110 \leq n \leq 1000$. Portanto, o algoritmo GRASP aqui proposto pode ser uma boa alternativa.

Referências

Amaral A. R. S. (2006). On the exact solution of a facility layout problem. *European Journal of Operational Research*, 173:508-518.

Amaral A. R. S. (2008a). An exact approach for the one-dimensional facility layout problem. *Operations Research,* 56:1026-1033.

Amaral, A.R.S. (2008b) . Enhanced local search applied to the single row facility layout problem. In *Anais do XL SBPO*, p. 1638–1647. João Pessoa. SOBRAPO.

Amaral A. R. S. (2009). A New Lower Bound for the Single Row Facility Layout Problem. *Discrete Applied Mathematics*, 157:183-190.

Amaral A. R. S. e Letchford A. N. (2011) . A polyhedral approach to the single row facility layout problem. Technical Report. Department of Management Science, Lancaster University.

Amaral A. R. S. e Letchford A. N. (2013). A polyhedral approach to the single row facility layout problem. *Mathematical Programming*, 141:453-477.

Anjos M. F., Kennings A. e Vannelli A. (2005). A semidefinite optimization approach for the single-row layout problem with unequal dimensions. *Discrete Optimization*, 2:113-122.

Anjos M. F. e Yen G. (2009). Provably near-optimal solutions for very large single row facility layout problems. *Optimization Methods and Software*, 24:805-817.

Chung J., e Tanchoco J. M. A. (2010). The double row layout problem. *International Journal of Production Research, 48*(3):709–727.

Cravo G. L., Ribeiro G.M. e Lorena L.A. N. (2008). A greedy randomized adaptive search procedure for the point-feature cartographic label placement. *Computer & Geosciences*, 34:373-386.

Cravo G. L. e Amaral A. R. S. (2015). Um algoritmo GRASP aplicado ao Problema de Layout de Facilidades em Fila Única. In: *Anais do XLVII SBPO*, Porto de Galinhas. SOBRAPO.

Datta D., Amaral A. R. S. e Figueira J. R. (2011). Single row facility layout problem using a permutation-based genetic algorithm. *European Journal of Operational Research*, 213:388-394.

De Alvarenga A. G., Negreiros-Gomes F. J. e Mestria M. (2000). Metaheuristic methods for a class of the facility layout problem. *Journal of Intelligent Manufacturing*, 11:421-430.

Faria H. Jr., Binato S., Resende M. G. C. e Falcão D. J. (2005). Transmission network design by a greedy randomized adaptive path relinking approach. *IEEE Trans Power Systems*, 20:43-49.

Feo, T. A. e Resende, M. G. C. (1995). Greedy Randomized Adaptive Search Procedures. *Journal of Global Optimization*, 6:109-133.

Guan, J., & Lin, G. (2016). Hybridizing variable neighborhood search with ant colony optimization for solving the single row facility layout problem. *European Journal of Operational Research*, 248(3):899–909.

Heragu, S. S. e Alfa, A. S. (1992). Experimental analysis of simulated annealing based algorithms for the layout problem. *European Journal of Operational Research*, 57:190-202.

Heragu S. S. e Kusiak A. (1988). Machine layout problem in flexible manufacturing systems. *Operations Research*, 36:258-268.

Heragu, S. S. e Kusiak, A. (1991). Efficient models for the facility layout problem. *European Journal of Operational Research*, 53:1-13.

Hungerländer P. e Rendl F. (2013). A Computational Study for the Single-Row Facility Layout Problem. *Computational Optimization and Applications*, 55:1-20.

Kothari R. e Ghosh D. (2012). The Single Row facility Layout Problem: state of the art. *OPSEARCH*, 49:442-462.

Kothari, R. e Ghosh, D. (2013). Tabu Search for the single row facility layout problem using exhaustive 2-opt and insertion neighborhoods. *European Journal of Operational Research*, 224:93-100.

Kothari, R. e Ghosh, D. (2014a). An efficient genetic algorithm for single row facility layout. *Optimization Letters*, 8:679-690.

Kothari, R. e Ghosh, D. (2014b). A Scatter search algorithms for the single row facility layout problem. *Journal of Heuristics*, 20:125-142.

Kumar K. R., Hadjinicola G.C. e Lin T.L. (1995). A heuristic procedure for the single-row facility layout problem. *European Journal of Operation Research*, 87:65-73.

Kumar, S., Asokan, P., Kumanan, S. e Varma, B. (2008). Scatter search algorithm for single row layout problem in FMS. *Advances in Production Engineering and Management*, 3:193-204.

Love, R. F. e Wong, J. Y. (1976). On solving a one-dimensional space allocation problem with integer programming. *INFOR*, 14:139-144.

Ozcelik, F. (2011). A hybrid genetic algorithm for the single row layout problem. *International Journal of Production Research*, 50:5872-5886.

Palubeckis, G. (2015). Fast local search for single row facility layout. *European Journal of Operational Research*, 246(3):800–814.

Palubeckis, G. (2016). Single row facility layout using multi-start Simulated Annealing. *Computers & Industrial Engineering*, 103:1–16.

Picard J. e Queyranne M. (1981). On the one dimensional space allocation problem. *Journal of Operation Research*, 29:371-391.

Samarghandi H. e Eshghi K. (2010). An Efficient Tabu Algorithm for the Single Row Facility Layout Problem. *European Journal of Operational Research*, 205:98-105.

Samarghandi H., Taabayanb P. e Jahantighc F. F. (2010). A particle swarm optimization for the single row facility layout problem. *Computers & Industrial Engineering*, 58:529–534.

Simmons D. M. (1969). One-dimensional space allocation: An ordering algorithm. *Operations Research*, 17:812-826.

Solimanpur M., Vrat P. e Shankar R. (2005). An Ant Algorithm for the Single Row Layout Problem in Flexible Manufacturing Systems. *Computers & Operations Research*, 32:583-598.