Otimização de Matching Máximo de Custo Mínimo utilizando Meta-Heurística M^2S (Multi-neighbourhood MultiStart)

Bruno Araújo Lima[1], **Diego Alysson Braga Moreira**[1], **Flávio Alves dos Santos**[1]

[1]Mestrado Acadêmico em Ciência da Computação
Universidade Estadual do Ceará (UECE)
Av. Dr. Silas Munguba, 1700. Itaperi. CEP 60.714.903 – Fortaleza - CE – Brasil

{bruno.araujo, diego.alysson, flavio.alves}@aluno.uece.br

Resumo. *Um emparelhamento envolve a formação de pares disjuntos de vértices de um grafo, casando-os dois a dois. O emparelhamento é perfeito quando todos os vértices do grafo estiverem casados de forma que cada vértice faça parte de exatamente um par. O emparelhamento simples de custo mínimo é um problema que consiste em achar o emparelhamento simples que minimize o custo total das arestas do subconjunto. Neste trabalho foi em implementada e avaliada a Meta-Heurística M^2S - Multi-neighbourhood Multistart, que consiste de uma heurística híbrida de resolução de problemas por meio de soluções estocásticas e buscas locais. Se utilizando do artífico de se gerar soluções iniciais aleatórias, dentro do espaço viável, é possível realizar pertubações locais a fim de se encontrar uma solução de melhor qualidade.*

1. Introdução

Encontrar um emparelhamento máximo em um grafo é um problema clássico no estudo de algoritmos, com muitas aplicações, como a designação de tarefas, determinação de rotas de veículos, problema do carteiro chinês, determinação de percursos mínimos, e muitos outros. O artigo histórico de [Edmonds 1965] que descreveu um algoritmo $O(n^4)$ para o problema geral de emparelhamento, na verdade, introduziu a noção de algoritmo de tempo polinomial. Implementações mais eficientes do algoritmo de Edmonds foram apresentadas por vários pesquisadores como Lawler (algoritmo $O(n^3)$), Hopcroft e Karp (algoritmo $O(m\sqrt{n})$ para o caso bipartido), Micali e Vazirani (algoritmo $O(m\sqrt{n})$ para o caso geral). Problemas como Cobertura de Vértices de um Grafo, Subpartição de Grafos, Circuito Hamiltoniano, Carteiro Chinês, entre outros, onde a busca exaustiva determinística é um problema de decisão são considerados como sendo NP-Completos, ou seja, o universo de busca é o conjunto de todos os elementos do grafo combinados entre si. Geralmente, possui tempo computacional na ordem $O(n!)$, ou maior.

A utilização de meta-heurísticas como métodos aproximativos de otimização consistem em gerar soluções candidatas às soluções ótimas, mas com um tempo computacional menor. *Heurística* é qualquer método ou técnica criada, ou desenvolvida, para resolver um determinado tipo de problema. As *Meta-Heurísticas* são consideradas heurísticas de uso geral ou uma heurística das heurísticas [Viana 1998]. As meta-heurísticas são usadas na resolução dos problemas em que o tempo computacional é inviável para se obter um valor ótimo. Geralmente, elas são usadas como heurísticas de busca local estocástica, ou gulosa, a fim de se encontrar um valor ótimo para o problema sem que haja a necessidade de realizar uma busca em todo o conjunto de soluções. Estas meta-heurísticas levam

em consideração a exploração do espaço alternando entre dois momentos que são o *exploring* e o *exploiting*, onde realizam uma busca global (geralmente obtendo-se soluções aleatórias dentro do espaço viável) e através de perturbações é feita uma busca local pela melhor solução, respectivamente.

O problema considerado neste trabalho é o problema do emparelhamento de vértices em grafos, que é um subproblema de Cobertura de Vértices, que tem sua complexidade do tipo NP-difícil. É utilizada a meta-heurística M^2S (Multi-neighbourhood MultiStart) para a resolução do problema.

2. Problema de Emparelhamento

Informalmente, um emparelhamento envolve a formação de pares disjuntos de vértices de um grafo, ”casando-os”dois a dois. O emparelhamento é perfeito quando todos os vértices do grafo estiverem ”casados”de forma que cada vértice faça parte de exatamente um par.

Definição: Dado um grafo $G = (V, E)$, um emparelhamento M em G é um subconjunto de arestas de E tal que quaisquer duas arestas distintas em M não possuem extremos em comum (ou seja, não são adjacentes) ou, equivalentemente, tal que vértice $v \in V$ seja incidente em, no máximo, uma aresta em M. Diz-se que um vértice $v \in V$ está emparelhado se, e somente se, ele é incidente em alguma aresta em M, e é dito não emparelhado caso contrário.

A Figura 1 ilustra um emparelhamento em um grafo com quatro vértices e seis arestas. As arestas do emparelhamento são mostradas em vermelho e as arestas que não estão no emparelhamento são mostradas em azul.

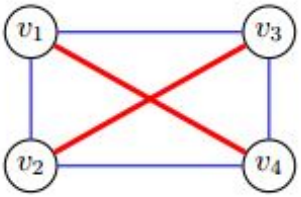

Figura 1. Exemplo de emparelhamento

Um emparelhamento, M, em um grafo G é um conjunto de arestas. Logo, a cardinalidade (ou tamanho) de M corresponde ao número de arestas que ele contém. O emparelhamento da Figura 1 é perfeito.

Observe que todo emparelhamento perfeito é um emparelhamento de cardinalidade máxima, mas nem todo emparelhamento de cardinalidade máxima é perfeito. Quando um emparelhamento de cardinalidade máxima, M, não é perfeito, o grafo não admite um emparelhamento perfeito. Caso contrário, haveria um emparelhamento (isto é, o emparelhamento perfeito) de cardinalidade maior que $|M|$, o que contradiz o fato da cardinalidade, $|M|$, de M ser máxima. Observe também que um subgrafo gerador, G', de um grafo G, é um $1 - fator$ de G se, e somente se, o conjunto de arestas de G' é um emparelhamento perfeito em G.

2.1. Aplicações

Vários problemas podem ser modelados como problemas de emparelhamento em grafos. A seguir, são descritos alguns destes.

2.1.1. Problema da atribuição de pessoal

O problema de atribuição ou de alocação de pessoal tem como dados n funcionários e n posições numa empresa. Cada funcionário está qualificado para uma ou mais posições. É possível atribuir uma posição a cada funcionário, de modo que cada funcionário ocupe exatamente uma posição na empresa?

O problema de atribuição ou de alocação ótima de pessoal pode apresentar como dados, adicionalmente, uma função que atribui a cada funcionário um valor numérico, correspondente à sua eficiência para ocupar determinada posição na empresa. O objetivo agora é encontrar uma atribuição ou uma alocação que maximize a eficiência total dos unficonários.

O primeiro caso corresponde ao problema de emparelhamento perfeito em grafos bipartidos, enquanto que o segundo caso é o problema do emparelhamento máximo em grafos bipartidos com pesos.

2.1.2. Problema dos casamentos

Dada uma matriz onde cada entrada é 0 ou 1, um conjunto de entradas é independente se não temos duas entradas na mesma linha ou na mesma coluna. Pede-se encontrar um conjunto indepentende de entradas de valor 1 que tem cardinalidade máxima. Podemos interpretar as linhas da matriz como sendo rapazes e as colunas como sendo moças. Uma entrada i, j com valor 1 seria o caso de rapaz e moça compatíveis. O objetivo é casar um número máximo de casais compatíveis. Este problema corresponde ao problema de emparelhamento de cardinalidade máxima em grafos bipartidos.

2.1.3. Problema de construção de amostras

Um vendedor de brinquedos educativos possui em estoque brinquedos de várias formas geométricas (cubos, pirâmides, etc.), cada qual fabricado em várias cores. O vendedor quer carregar consigo o menor número de objetos tal que cada cor e cada forma estejam representadas pelo menos uma vez.

O vendedor constrói o seguinte grafo: cada forma e cada cor estão representados individualmente por um vértice e existe uma aresta ligando um vértice-forma a um vértice-cor, caso aquela forma geométrica seja fabricada naquela cor.

O número mínimo de objetos que o vendedor precisa carregar é igual ao número de arestas numa cobertura de cardinalidade mínima: um conjunto de arestas C tal que cada vértice do grafo é extremo de alguma aresta em C, e este mesmo conjunto C de arestas é o de menor cardinalidade mínima e o problema de emparelhamento de cardinalidade máxima.

3. A Meta-Heurística M^2S - Multi-neighbourhood Multistart

O algoritmo M^2S consiste de uma heurística híbrida de resolução de problemas por meio de soluções estocásticas e buscas locais. Se utilizando do artifício de se gerar soluções iniciais aleatórias, dentro do espaço viável, é possível realizar perturbações locais a fim de se encontrar uma solução de melhor qualidade.

3.1. Busca Local

As técnicas de Busca Local são não-exaustivas no sentido de que não garantem encontrar uma solução factível (ou ótima), mas realiza uma busca não-sitemática até um critério de parada específico ser satisfeito [Di Gaspero and Schaerf 2002].

Dada uma instância p da busca, ou otimização de um problema P, associamos um espaço de busca S a ele. Cada elemento de $s \in S$ corresponde a uma solução potencial de p, e é chamada $estado$ de p. A busca local depende de uma função N que descreve cada $s \in S$ como a vizinhança $N(s) \subseteq S$. Cada $s' \in N(s)$ é chamado vizinho de s [Di Gaspero and Schaerf 2002]. A definição de vizinhança pode ser ilustrada através da Figura 2.

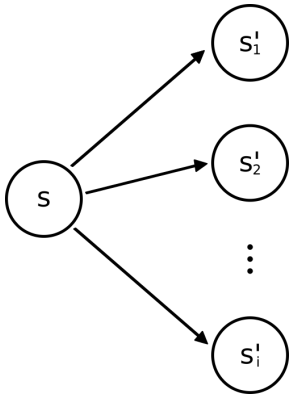

Figura 2. Definição de vizinhança

O algoritmo é inicializado a partir de um estado inicial s_0 que pode ser gerado por outra técnica, como geração de uma solução aleatória, ou uma solução gulosa e navega dentro do espaço de busca gerando, por meio de perturbações, estados s_i, de um de seus vizinhos s_{i+1}.

3.2. Multi-neighbouhood - Multivizinhança

De acordo com [Viana 1998], um procedimento que realiza transformações de uma cofiguração s, em uma configuração s' ($s' = P[s]$) com $s, s' \in S$, onde S é considerado o espaço de busca. As novas configurações s' obtidas a partir de s introduzem o conceito de Vizinhanças, cujo tamanho depende do procedimento de perturbação.

Considerando um conjunto k de vizinhos e sua função de vizinhança N_1, ..., N_k definida em S. E dado um conjunto de n tecnicas de busca local (perturbações), podemos definir $k \times s$ diferentes algoritmos de busca pela combinação de qualquer técnica com qualquer função de vizinhança.

Neste trabalho são são consideradas duas vizinhanças distintas. A primeira consiste em gerar dois número aleatorios i e k entre 1 e n, onde n é o número de vértices,

onde uma solução candidata é um vetor conjunto os vértices do matching. Assim, ao se selecionar dois números aleatórios i e k, faz-se a permutação entre eles no conjunto, ou seja, removemos a aresta v_{ij}, onde j é o vértice par de i e adicionamos a aresta v_{kj} e removemos a aresta v_{kl}, sendo l o vértice par de k e adicionamos a aresta v_{il}. Com isto, dizemos que nossa função de vizinhança N que equivale a uma perturbação P, é dada por

$$N(s) = P(s, i, k).$$

Nossa segunda função de vizinhança é descrita por um deslocamento, onde geramos dois números aleatórios distintos i e k e fazemos o deslocamento do vértice v_i para a posição v_{i+k}. Caso $i + k > n$, fazemos v_{n-k+i}. Assim, descrevemos o modelo assim

$$N(s) = \begin{cases} P(s, i, i + k), \text{ se } i + k \leq n \\ P(s, i, n - k + i), \text{ caso contrário.} \end{cases}$$

4. Descrição do problema

Considere um grafo não direcionado que consiste de um conjunto N de n nós particionados em $N^{\leq} \cup N^{=} \cup N^{\geq} \cup N^{\circ}$ (onde algumas desses subconjuntos podem ser vazios), um conjunto E de m arestas, e um vetor de custo C. O Emparelhamento simples (1-matching) é um subconjunto de arestas que satisfazem as seguintes propriedades [Perin 1981]:

1. Todos os nós de $N^{\leq}$ são incidentes com, no máximo, uma aresta do subconjunto.
2. Todos os nós de $N^{=}$ são incidentes com, exatamente, uma aresta do subconjunto.
3. Todos os nós de $N^{\geq}$ são incidentes com, pelo menos, uma aresta do subconjunto.
4. Não há restrições para um nó de N°.

O emparalhamento simples de custo mínimo é um problema que consiste em achar o emparelhamento simples que minimize o custo total das arestas do subconjunto.

Considere um grafo não direcional G = (N, E, c) que consiste em um conjunto finito e não vazio de n nós $N = \{i : i = 1 \rightarrow n\}$, um conjunto de m arestas $E = \{(i, j) : i \neq j \in N\}$, e um vetor de custo $c = [c_{ij}]$. Dada a partição $N^{\leq} \cup N^{=} \cup N^{\geq} \cup N^{\circ}$ do conjunto de nós onde alguns dos sub-conjuntos podem ser vazios. Seja $x = [x_{ij}]$ um vetor [0-1] definido pelo conjunto de arestas. O Emparelhamento simples (1-matching) de custo mínimo é formulado da seguinte forma:

$$min \sum_{ij} [c_{ij} x_{ij} : (i, j) \in E]$$

sujeito às restrições do nó

$$x(i) \begin{cases} \leq 1, \text{ para } i \in N^{\leq} \\ = 1, \text{ para } i \in N^{=} \\ > 1, \text{ para } i \in N^{>} \\ irrestrito, \text{ para } i \in N^{0} \end{cases}$$

restrito a

$$x_{ij} = \{0, 1\}, \text{ para } (i, j) \in E$$

onde

$$x(i) = \sum_j [x_{ij} : (i, j) \in E], \text{ para } i \in N$$

5. Implementação da Meta-Heurística

A meta-heurística foi implementada na linguagem C++, sendo que as instâncias são arquivos de texto plano disponibilizados em um repositório público chamado OR-Library. Os testes foram realizados em uma máquina com processador Intel Core i3 com memória DDR3 com 4GB de capacidade.

5.1. Representação da Solução Candidata

As soluções candidatas são representadas, a nível de código, como um vetor indexado por inteiros em que cada item do vetor é um vértice do grafo, onde cada vértice de índice par e seu sucessor ímpar representam uma aresta entre eles. É importante lembrar que os índices do vetor estão dentro do intervalo $[0, n-1]$.

Representando o vetor como

$$S = [A, B, C, D, E, F]$$

temos o vértices A, B, C, D, E, F e as arestas AB, CD e EF no para o mathing.

A representação de uma aresta X_{AB}, onde A e B são vértices quaisquer, temos

$$X_{AB} : \begin{cases} A_i, \text{ onde } i\%2 == 0 \\ B_j, j = i + 1 \end{cases}$$

onde i e j são os índices do vetor que armazena os vértices.

5.2. Funções de Vizinhança

As funções de vizinhança são definidas por dois métodos de pertubação distintos. Nas duas é necessário gerar dois números aleatórios quaisquer.

Na primeira função de vizinhança, são gerados dois números aleatórios i e k dentro do intervalo $[1, n]$, onde n é o número de vértices do grafo. A função de perturbação é definida pela permutação entre os elementos das posições i e k no vetor de vertices.

$$\text{s' = N(s),}$$
$$\text{N(s) = P(i-1, k-1)}$$

Como, na linguagem C, os índices dos vetores variam no intervalo $[0, n-1]$, devemos adaptar a representação dos índices dos vértices, que são gerados no intervalo $[1, n]$.

No modelo também adotou-se que para cada elemento são gerados 5 novos vizinhos que são avaliados e a melhor solução entre eles assume a solução parcial da vizinhança, ou seja, as demais são descartadas e as novas perturbações são realizadas a partir do melhor vizinho encontrado.

5.3. Função Aleatória

A função geradora de números aleatórios (função randômica - rand()) da linguagem C++ é uma função homogênea descrita de acordo com a *ISO C standard*. A função gera números pseudo-aleatórios inteiros dentro do intervalo $[0, RAND_MAX]$, em que $RAND_MAX$ é uma constante implementada na linguagem e deve ser no máximo 32767. Quando não

Tabela 1. Comparação do TSP58 com diferentes configurações iniciais

Gerações	Inicializações	Melhor Resultado	Tempo
1000000	1	35410	58,8 min
100000	1	34900	6,11 min
10000	10	38000	6,06 min
1000	100	38190	6,09 min
100	1000	36698	7,17 min
10	10000	37447	7,4 min
1	100000	36463	8,43

fixada no código-fonte do software que a utiliza, retorna valores reais proporcionais no intervalo $[0, 1]$.

Para que a geração de números pseudo-aleatórios seja o mais aleatória possível, é necessário informar uma boa semente que não seja, diretamente, conhecida. Neste trabalho utilizamos como semente o timestamp local da máquina como semente, informada como parâmetro através da função *srand()*.

Feito isso, são gerados números pseudo-aleatórios dentro do intervalo $[0, 1]$ e estes são mapeados para valores inteiros no intervalo $[1, n]$ de acordo com a expressão

$$randBetween(min, max) = (rand()\%(max + 1 - min)) + min.$$

5.4. Algoritmo

Como descrito nos tópicos anteriores, o algoritmo descreve a Meta-Heurística M^2S-Multi-neighbourhood Multistart uma Heurística híbrida de resolução de problemas por meio de soluções estocásticas e buscas local. Alguns critérios foram definidos para regular, a quantidade de vizinhos que seriam criados na fase de busca local, como o número de vizinhos e a quantidade de vezes que é reiniciado com uma nova entrada. Desta forma, percorre-se todas as abordagens propostas pela Meta-Heurística.

```
#define NVIZINHOS 3364
#define NVIZINHANCAS 1000
#define INICIALIZACOES 100
```

Cada elemento declarado está relacionada com a quantidade de vezes em que cada um dos fatores é realizado. Temos que, quanto maior o número de vizinhos, uma maior variação dentro de uma determinada busca deverá ocorrer. Porém, a partir de uma certa quantidade de execuções, as permutações são feitas de forma repetidas.

Quanto maior for o número de vizinhos, teremos soluções candidatas mais variadas dentro do espaço de busca, logo, tenta-se achar um ótimo local a partir do ultimo ótimo encontrado.

O número de inicializações dá ao algoritmo a chance de fazer buscas em outros locais, desta forma percorre-se uma maior quantidade de possobilidades. A Tabela 1 mostra os resultados obtidos com a instância TSP58, com variadas configurações de número de vizinhos, número de vizinhança e inicializações.

As variáveis apresentadas controlam as seguintes funções:

```
void melhorVizinhanca(int *vetor, Graph grafo, int tipo)
{
    int *melhorVetor;
    int melhor = calcularMatching(vetor,grafo);
    int atual;

    for(int i = 0; i < NVIZINHOS ; i++)
    {
        permutarVetor(vetor,grafo.GetVerticesNumber());
        atual = calcularMatching(vetor,grafo);
        if ( atual < melhor )
        {
            melhor = atual;
            melhorVetor = vetor;
        }
    }

}
```

A partir desta função, gera-se múltiplos vizinhos, por meio do método de permutação escolhido e seleciona-se o vizinho que obteve uma melhor solução em relação aos demais. O vizinho escolhido será, então, utilizado para se comparar às melhores gerações de vizinhanças.

```
int melhorGeracaoRandom(int matching[], Graph grafo)
{
    int nVertices = grafo.GetVerticesNumber();
    //Valor grande o bastante.
    int valorMatching = ((nVertices/2)*9999)+1;

    int melhorMatchingPermutado[nVertices];
    criaVetorRandom(melhorMatchingPermutado,nVertices);
    int valorMelhorMatchingPermutado;

    for(int i = 0 ; i < NVIZINHANCAS ; i++)
    {
        melhorVizinhanca(melhorMatchingPermutado,
                         grafo,0);
        valorMelhorMatchingPermutado = calcularMatching
                              (melhorMatchingPermutado,grafo);

        if(valorMelhorMatchingPermutado < valorMatching)
        {
            valorMatching = valorMelhorMatchingPermutado;
            copiarVetor(melhorMatchingPermutado,
                         matching,nVertices);
        }
```

```
    }
    return valorMatching;
}
```

Desta forma, obtem-se qual das gerações já avaliadas obteve um melhor resultado. De acordo com a busca local. Este passo representa o processo de Multi-neighbourhood, processo ao qual obtem-se várias vizinhanças.

E para determinar outras buscas locais, processo que determina a fase Multistart, temos:

```
for(int k = 0 ; k < INICIALIZACOES ; k++)
{
    cout<<"-------------------------------------------------"<<endl;
    valor = melhorGeracao(matching,grafo);
    cout<<endl;
    cout<<"Geracao: "<<k<<endl;
    cout<<"-------------------------------------------------"<<endl;
}
```

Ao fim deste processo obtem-se o melhor valor de matching encontrado, assim como qual o vetor que representa o matching.

6. Resultados

Pode-se observar através da Tabela 2 os resultados obtidos com a Meta-Heurística. Levando em consideração que a mesma é baseada somente em algoritmos aleatórios, observamos que não foram obtidos bons resultados. Para instâncias que apresentavam um número de vertices pequenas, achou-se o valor ótimo, visto que pode-se cobrir a grande maioria das possibilidades de combinações dos matching existentes. Porém, à medida que a quantidade de vertice cresce, é nescessário uma quantidade de permutações muito maior para obter-se bons resultados, o que demanda custo um computacional muito grande.

Tabela 2. Resultados

Instâncias Comparativas				Solução	Solução Obtida			Aproximação
Nº	Referência	n	m	"Ótima"	Inicial	Tempo(s)	Melhor	Gap
1	K5x5	10	25	50	26716	1 min 49s	50	1
2	Fign16m35	16	35	23	69997	2 min 41s	23	1
3	TSP58	58	1682	9464	65743	6 min 46s	34900	3,69
Média das aproximações com resultados conhecidos →								
Nº	Referência	n	m	inicial	Máxima	Média	Melhor	Tempo(s)
4	Groestchel	442	97682	39375	40807	37838	34870	37 min 11s
6	Rinaldi	2392	2860832	37868	39117	37311	35505	1h 23min
7	BillCook	20726	214783583	38203	Após uma tarde inteira, não achou resultado			

O Gráfico da Figura 3 mostra o comportamento de procura do melhor matching, neste podemos observar o momento em que há trocas de gerações e a procura de um vizinho melhor.

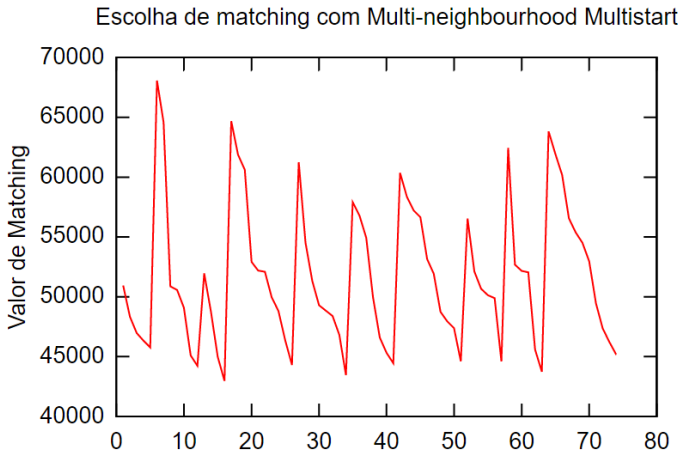

Figura 3. Comportamento do Multi-neighbourhood Multistart

Os resultados apresentados acima foram obtidos através da execução em um computador com processador Intel Core i3 2ª Geração com clock de 2.8GHz e 4GB de memória RAM com frequência de 1333MHZ.

A seguir, a Tabela 3 mostra os resultados obtidos utilizando a instância Groestchel (instância número 4) em um computador com processador Intel Core i7 4ª Geração clock de 1.8GHz, 12GB de memória RAM com clock de 1600MHz. As configurações utilizadas para a comparação foram: Número de inicializações: 10; Número de Viinhanças (perturbações por função de vizinhança): 1000.

Tabela 3. Instância Groestchel utilizando diferentes funções randômicas (n=442; m=97682)

Função	inicial	Máxima	Média	Melhor	Tempo(s)
C++ Standard	35048	40759	34778	34154	4 min 48s
Park Miller	35251	41318	34977	32723	4 min 43s
/dev/urandom	33977	40276	34571	33977	13 min 24s

As execuções da Meta-Heurística M^2S é inteiramente dependente da uma função aleatória. A inicialização das soluções candidatas é feita de maneira aleatória e as perturbações utilizam parâmetros que são definidos por um valor aleatório. A função padrão do C++ (C++ Standard Random Function) é uma função aleatória com 15 bits de resolução, sendo 32.768 o valor máximo que ela gera. Já a função de Park Miller utiliza 32 bits de resolução, onde 4.294.967.296 é o valor máximo que ela consegue gerar. A terceira função aleatoria utilizada é uma função implementada em hardware por processadores Intel, onde pode ser acessada através do arquivo ”/dev/urandom”.

As duas primeiras funções são definidas por software e dependem de uma semente para se calcular um valor aleatório. Para a primeira função se utilizou o *timestamp* da máquina como semente. Para a segunda função, utilizou-se como semente a expressão

$$semente = timestamp \ \% \ rand()*rand()$$

sendo uma boa alternativa a ser utilizada como semente, mas que depende de outra função aleatória, para gerar a sequência aleatória. A terceira função não necessita

da atribuição de uma semente. A geração de um número aleatório por esta é feita por meio de operações matemáticas definidas pelas variação de valores nos registradores do processador. A resolução dos valores aleatórios retornados por esta função é definida pela quantidade de bytes a serem recuperados do arquivo que a torna disponível. Neste trabalho utilizou-se a leitura de 32 bytes do arquivo.

Pode-se perceber que a terceira função utiliza cerca de 3 vezes mais tempo que as anteriores. Isto deve-se à funcionalidade de leitura dos dados no disco do computador, onde é redurecionada ao registrador que a posição de memória que a armazena. As duas primeiras são executadas em tempos muito próximos, mas é importante observar que a função geradora de Park Miller [Payne et al. 1969] (algumas vezes citadas na literatura como Função Geradora de Números Aleatório de Lehmer) possui um resultado melhor por conta da sua resolução. Devido a isto, é uma função com melhor uniformidade e também é homogênea.

Podemos observar que a ordem de complexidade do M^2S é pequena. Tomando n como o número de vértices, m sendo o número de arestas, p, o número de inicializações aleatórias e q, o número de permutações por vizinho, inferimos que a ordem de complexidade do algoritmo que inicializa as soluções é $O[p]$, onde p é um número inteiro (descritos neste trabalho como 1, 10, 10^2, 10^3, 10^4 e 10^5). A ordem de complexidade do número de permutações é $O[q]$, onde q é um inteiro dado por $q = n^2$. e a ordem do algoritmo se dá por $O[2pq]$, onde 2 é a quantidade de vizinhanças implementadas (2 funções de vizinhança), sendo que as duas possuem seus tempos computacionais muito próximos.

7. Trabalhos Futuros

Como trabalhos futuros pretende-se realizar o experimento utilizando uma meta-heurística para a inicialização das soluções candidatas. Ao invés de se utilizar soluções puramente aleatórias, utilizar uma técnica que gere valores de qualidade melhor para que as perturbações possam convergir para um melhor resultado e que dispenda menos iterações até ateingí-lo.

São também válidos a implementação de uma função aleatória de melhor qualidade. A função rand() da linguagem C++ é uma boa função, quando se inicaliza uma boa semente, mas não tão satisfatória quanto a função equivalente da linguagem Java. As duas são homogêneas, porém o método da linguagem Java possui um uniformidade melhor que a função da linguagem C++.

Um outro ponto a ser tratado como trabalho futuro é a análise dinâmica das soluções como retroalimentação da entrada, ou seja, as melhores soluções serem utilizadas como entrada do algoritmo, gerando saídas cada vez melhores.

8. Conclusão

A Meta-Heurística M^2S implementada não conseguiu ótimos resultados, condizente com o que já era esperado, visto que sua inicialização depende, exclusivamente, de algoritmos aleatórios. É necessário fazer um ajuste no número de inicializações e número de perturbações para se ter melhores resultados, ou seja, o ajuste destes fatores são extremamente importantes e cruciais para se ter um bom desempenho. Neste trabalho, o ajuste foi feito de forma manual, destes valores, mas pode-se usar uma outra meta-heurística para

fazer este ajuste automaticamente, a fim de se ter melhores soluções e minimizar o custo do algoritmo.

Referências

Di Gaspero, L. and Schaerf, A. (2002). Multi-neighbourhood local search with application to course timetabling. In *International Conference on the Practice and Theory of Automated Timetabling*, pages 262–275. Springer.

Edmonds, J. (1965). Maximum matching and a polyhedron with 0, l-vertices. *J. Res. Nat. Bur. Standards B*, 69(1965):125–130.

Payne, W., Rabung, J. R., and Bogyo, T. (1969). Coding the lehmer pseudo-random number generator. *Communications of the ACM*, 12(2):85–86.

Perin, C. (1981). Technical report 81-3 department of industrial and operations engineering the university of michigan ann arbor, michigan 48109.

Viana, G. V. R. (1998). *Meta-Heurísticas e Programação Paralela em Otimização Combinatória*. Edições UFC, 1st edition.