Universidade Federal do Rio Grande do Norte
Departamento de Informática e Matemática Aplicada

Disciplina: DIM0406 — Algoritmos Avançados
Docente: Sílvia Maria Diniz Monteiro Maia
Discente: Felipe Cortez de Sá

GRASP-VNS aplicado ao problema de Steiner com rotulação mínima

1 Introdução

Neste relatório são apresentados os algoritmos *Greedy Randomized Adaptative Search Procedure* e *Variable Neighbourhood Search* para solução do problema da árvore de Steiner com rotulação mínima. Os princípios de cada técnica são descritos, assim como a maneira em que foram utilizados para resolver o problema. É realizada uma análise de complexidade em tempo para as implementações, identificando os gargalos. Explica-se como foram gerados os casos de teste e são apresentados resultados comparando o resultado das execuções para cada técnica, incluindo o algoritmo exato desenvolvido na segunda unidade. Além disso, é feito um adendo para o relatório do primeiro trabalho, apresentando os resultados e conclusão previamente faltantes.

2 Metaheurísticas utilizadas

As definições seguintes foram adaptadas do *Handbook of Metauheuristics* [4].

2.1 GRASP

O *Greedy Randomized Adaptative Search Procedure* é comumente utilizado em problemas de otimização combinatória. A cada iteração, é realizada uma fase de construção, em que se gera uma solução para o problema e posteriormente uma fase de busca local, procurando um mínimo local na vizinhança da solução gerada. Se a melhor solução global é encontrada na iteração, atualiza-se a variável que contém a melhor solução. É um algoritmo de inicialização múltipla, ou seja, as duas fases são repetidas até o critério de parada ser satisfeito, podendo ser o número de iterações ou o tempo de execução, por exemplo.

Na fase de construção, é criada uma lista de candidatos restritos, possuindo os candidatos cujos elementos adicionados minimizam os custos incrementais. O elemento é selecionado aleatoriamente da RCL.

2.2 VNS

O *Variable Neighbourhood Search* faz uso de múltiplas estruturas de vizinhança, explorando comumente espaços cada vez mais distantes e maiores, portanto mais custosos. Para fugir de mínimos locais, o algoritmo possui uma fase de agitação, em que a solução encontrada pode ser trocada por uma pior a fim de diversificar a busca, explotando melhor o espaço.

3 Metaheurística aplicada ao problema

3.1 GRASP

No problema, a fase de construção gera uma lista de candidatos restritos calculando argmin(comp(*col*)) para cada cor não utilizada, onde comp(*col*) é o número de componentes conexos do grafo com a coloração *col* que incluem pelo menos um nó básico. Após a segunda repetição da inicialização múltipla, o primeiro rótulo a ser adicionado é totalmente aleatório, ou seja, a lista de candidatos é inicializada como $1, 1, ..., 1$, explotando melhor o espaço de busca em vez de escolher sempre rótulos que minimizam comp(*col*).

Após a fase de construção, é feita uma busca local, que consiste em tentar remover cores da solução e verificar se ainda obtém-se um grafo conexo, configurando outra solução válida.

Na implementação, percebeu-se que para grafos com **DENSITY** acima de 50, a cardinalidade da coloração atinge o mínimo global identificado pelo algoritmo exato nas primeiras iterações. Logo, para deixar o algoritmo mais rápido, o critério de parada é a falta de melhoria em três iterações. Para densidades menores, a variação entre iterações é maior.

```
grasp(limite) {
    col* = {}
    int no_improv = 0

    while(no_improv < limite) {
        col = {}
        construct(col)
        local(col)
        if(card(col) < card(col*)) {
            col* = col
            no_improv = 0
        } else {
            ++no_improv
        }
    }
}

construct() {
    if(iteration > 2) {
        rcl = {1, 1, ... , 1}
        col[random()] = 1
    }

    while(comp(c) > 1) {
        rcl = argmin comp(c)
        col = col ∪ rcl[random()]
    }
}
```

Listing 1: Pseudocódigo para GRASP

3.2 VNS

Quando o algoritmo é executado independentemente, a configuração das cores inicial é totalmente aleatória, isto é, cada elemento do vetor é inicializado para 0 ou 1 seguindo a distribuição uniforme.

Na fase de agitação, cores de *col* são removidas e adicionadas dependendo da cardinalidade do conjunto. A remoção de cores pode tornar o grafo desconexo, então é realizado o mesmo procedimento do GRASP para conectar o grafo escolhendo $c \in \text{argmin}(\text{comp}(col \cup c))$ repetidamente.

Em seguida, o mesmo procedimento de busca local utilizado pelo GRASP é realizado, tentando descartar cores desnecessárias.

```
vns(col, kmax)
col2 = col
k = 1
while(k <= kmax) {
    shaking(col2, k)
    local(col2)
    if(card(col2) < card(col)) {
        col = col2
        k = 1
    } else {
        ++k
    }
}

shaking(col, k) {
    col2 = col
    for(i in 1..k) {
        if(i <= card(col)) {
            col2[random(cores utilizadas em col)] = 0
        } else {
            col2[random(cores nao utilizadas em col)] = 1
        }
    }

    while(comp(col2) > 1) {
        melhores = argmin comp(c)
        col2[random(melhores)] = 1
    }
}
```

Listing 2: Pseudocódigo para VNS

4 Complexidade

4.1 GRASP

4.2 VNS

5 Casos teste utilizados

Os casos teste utilizados são gerados automaticamente por um programa **`generate.c`** de acordo com parâmetros de entrada. Os parâmetros são **`SIZE`**, a quantidade de nós do grafo, **`COLORS`**, o número de rótulos, **`DENSITY`**, a proporção de arestas para cada nó, e **`BASIC`**, a quantidade de nós básicos. **`DENSITY`** funciona percorrendo a matriz de adjacência que representa o grafo e de acordo com a probabilidade definida (sendo 0 e 100 equivalentes a 0% e 100%, respectivamente) adicionando ou não uma aresta de rotulação aleatória ligando dois nós. O arquivo gerado é então passado para o programa principal.

A fim de comparar os resultados com o trabalho realizado por Cerulli [2], os parâmetros dos testes são os mesmos, isto é, tem-se uma combinação entre **`SIZE`** $\in \{50, 100\}$, **`COLORS`** $\in \{0.25n, 0.5n, n, 1.25n\}$, **`DENSITY`** $\in \{0.2, 0.5, 0.8\}$ e **`BASIC`** $\in \{0.2n, 0.4n\}$. Cada caso teste é executado dez vezes diferentes e são apresentadas a média, melhor e pior casos e mediana.

O código que gera os arquivos de caso teste para os parâmetros desejados está em **`generate.py`**.

6 Resultados

7 Experimentos comparativos

7.1 Comparação com algoritmo exato

Como pode ser visto nos plots e na tabela, o algoritmo exato funciona bem para instâncias pequenas do problema, porém

7.2 Comparação com literatura

Ao contrário do que se verifica na literatura, o GRASP funciona melhor para os casos teste, apresentando menos variação e a maioria dos resultados Verifica-se que a densidade é o fator mais importante na execução do algoritmo exato

8 Conclusões

9 Correções do primeiro trabalho

9.1 Técnica utilizada

O *branch-and-bound* é um algoritmo de otimização que explora o espaço de busca de maneira mais eficiente que uma enumeração total de soluções possíveis por força-bruta. Atualizando o limite inferior continuamente, é possível eliminar a exploração de regiões não-promissoras do espaço de busca.

9.2 Resultados

O tempo de execução para cada instância do caso teste está na tabela 1. Experimentalmente percebeu-se que se o algoritmo demora mais que cinco segundos, ele não retornará uma resposta dentro de um minuto. O algoritmo funciona mais rapidamente para casos em que a densidade é maior. Para `SIZE` $= 50$, costuma falhar na maioria dos casos em que `DENSITY` ≤ 20. Para tamanhos maiores, falha também quando `DENSITY` ≤ 50.

9.3 Conclusões

Neste trabalho verificou-se que o algoritmo exato implementado com branch-and-bound funciona rapidamente para instâncias pequenas e altas densidades, mas falha para o restante dos casos.

9.4 Considerações adicionais

O código referente ao algoritmo exato foi modificado para aceitar entradas de um caso de teste, foi comentado mais extensivamente e agora é cronometrado para possibilitar a análise de resultados.

10 Tabelas e figuras

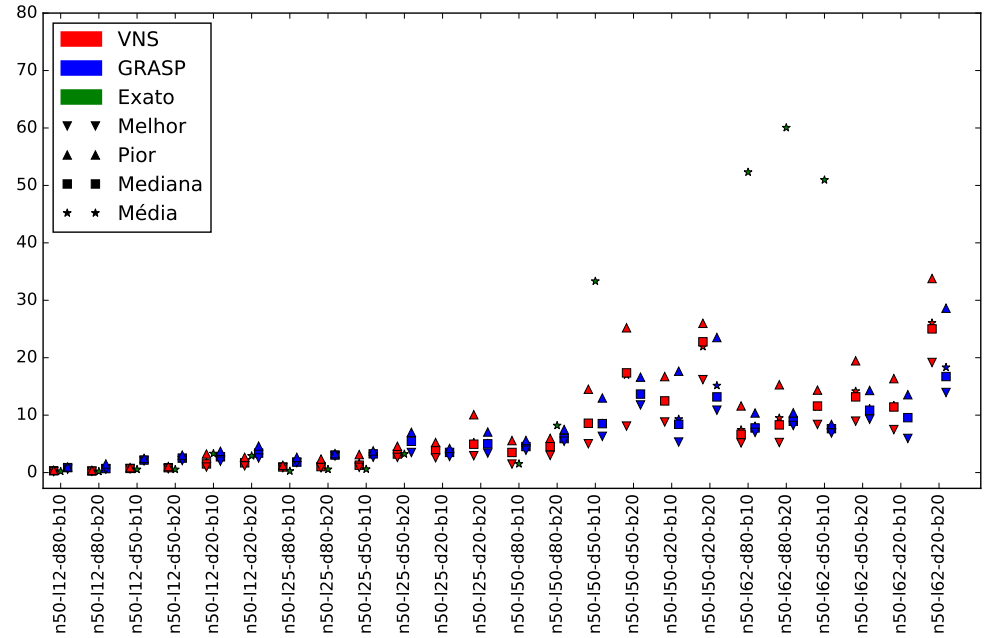

Figura 1: Plot para **SIZE** = 50

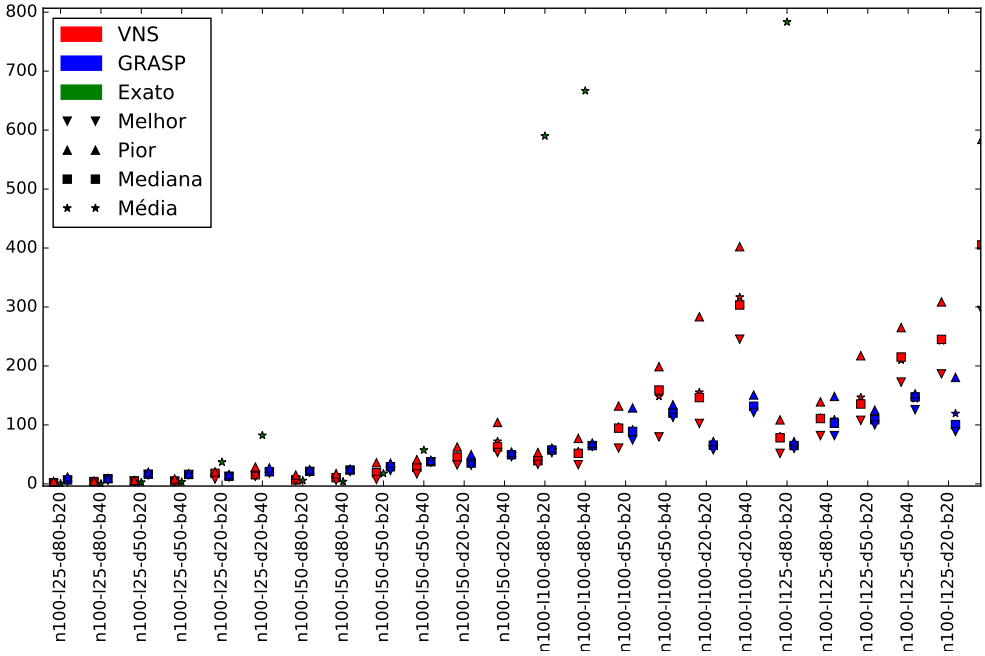

Figura 2: Plot para **SIZE** = 100

n	l	d	b	Exato	GRASP	VNS
100	25	80	20	0.41	7.11	1.95
	25	80	40	0.24	8.33	3.77
	25	50	20	2.81	16.70	4.97
	25	50	40	3.31	16.20	5.28
	25	20	20	36.97	13.04	16.66
	25	20	40	82.28	21.49	17.47
	50	80	20	5.60	21.73	7.52
	50	80	40	3.87	23.37	10.47
	50	50	20	17.61	29.25	19.35
	50	50	40	57.33	37.99	28.40
	50	20	20	-1.00	36.90	46.14
	50	20	40	-1.00	49.40	72.14
	100	80	20	590.00	57.09	42.44
	100	80	40	666.56	65.54	54.84
	100	50	20	-1.00	92.08	96.86
	100	50	40	-1.00	121.62	148.25
	100	20	20	-1.00	65.57	155.33
	100	20	40	-1.00	132.17	316.71
	125	80	20	783.32	65.42	80.41
	125	80	40	2071.99	109.48	112.10
	125	50	20	-1.00	110.95	146.82
	125	50	40	-1.00	144.25	209.87
	125	20	20	-1.00	119.30	242.61
	125	20	40	-1.00	192.82	406.93

Tabela 1: Tempo (ms)

n	l	d	b	Exato	GRASP	VNS
100	25	80	20	1.00	1.00	1.00
	25	80	40	1.00	1.00	1.90
	25	50	20	2.00	2.00	2.00
	25	50	40	2.00	2.00	2.00
	25	20	20	3.00	3.00	3.00
	25	20	40	3.00	3.10	3.80
	50	80	20	2.00	2.00	2.00
	50	80	40	2.00	2.00	2.00
	50	50	20	2.00	2.00	2.00
	50	50	40	2.00	2.00	2.40
	50	20	20	-1.00	4.70	4.70
	50	20	40	-1.00	5.00	5.30
	100	80	20	3.00	3.00	3.00
	100	80	40	3.00	3.00	3.10
	100	50	20	-1.00	3.30	3.50
	100	50	40	-1.00	4.00	4.10
	100	20	20	-1.00	6.00	6.30
	100	20	40	-1.00	8.00	8.10
	125	80	20	3.00	3.00	3.00
	125	80	40	3.00	3.10	3.30
	125	50	20	-1.00	4.00	4.10
	125	50	40	-1.00	4.00	4.30
	125	20	20	-1.00	6.60	6.40
	125	20	40	-1.00	7.30	7.50

Tabela 2: Resultado ($|C|$)

Referências

[1] S. Consoli, K. Darby-Dowman, N. Mladenovic, J.A. Moreno-Perez. *Variable neighbourhood search for the minimum labelling Steiner tree problem.* Annals of Operations Research, 2009.

https://www.researchgate.net/publication/225327721_Variable_neighbourhood_search_for_the_minimum_labelling_Steiner_tree_problem

[2] R. Cerulli, A. Fink, M. Gentili e S. Voß. *Extensions of the minimum labelling spanning tree problem.* Journal of Telecommunications and Information Technology, 2006.

https://www.researchgate.net/publication/228668519_Extensions_of_the_minimum_labelling_spanning_tree_problem

[3] *Graphviz — Graph Visualization Software.*

http://www.graphviz.org/

[4] Glover, F., Kochenberger, G. A. et al. *Handbook of Metaheuristics.* Kluwer Academic Publishers

[5] *Stack Overflow — Execution time of a C program.*

http://stackoverflow.com/questions/5248915/execution-time-of-c-program