UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

DEPARTAMENTO DE INFORMÁTICA TEÓRICA
INF05010 - Otimização Combinatória - 2014/1
Prof. Dra. Luciana Salete Buriol
Aluno Germano Andersson
Aluno Ricardo Sturmer

Metaheurística GRASP aplicada ao problema Linear Arrangement

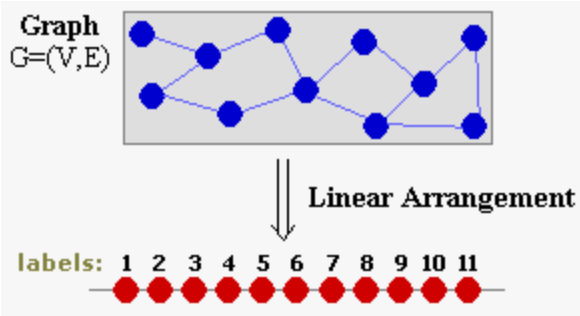
Relatório Trabalho Prático

Sumário

O problema

Descrição

O problema "Linear arrangement" tem o objetivo de arranjar os vértices de um grafo não-direcionado de modo linear. Cada vértice deve receber um label único, entre 1 e n, sendo n a quantidade de vértices deste grafo. O custo total deste grafo se dará pela soma do módulo das diferenças entre os labels dos vértices incidentes de todas arestas deste grafo. A imagem abaixo ilustra o objetivo deste problema:



Seu respectivo problema de minimização, conhecido como MINLA, busca otimizar o mapeamento de labels dos vértices de modo que o custo total do grafo seja o menor possível. Este problema de otimização é classificado como NP-Difícil.

Formulação Matemática

Dado um grafo (nao-direcionado) G = (V,A), queremos encontrar uma função bijetora f: V -> {1,2,...,|V|} tal que a distância total $\{u,v\} \in A$ |f(u)-f(v)| entre os vértices incidentes a cada aresta seja minimizada.

Algoritmo proposto

Dados de entrada

As instâncias fornecidas para geração deste relatório foram disponibilizadas em http://www.inf.ufrgs.br/~mrpritt/oc/la.zip. Cada instância está representada em um arquivo no seguinte formato:

número n;
número m;
n números representando os graus dos vertices δ1,...,δn;
2m numeros v1,...,v2m;

o número −1;
n + 1 numeros a1,...,an+1.
Os números satisfazem δi=ai+1 − ai . Os vertices do grafo sao 0,1,...,n−1. Os vizinhos do vertice
i são os vertices Vai,...,Vai+1 −1.

Principais estruturas de dados

Grafo -> lista de adjacência

Optamos em representar o grafo através de uma **lista de adjacência** pois o algoritmo fica otimizado se iterarmos sob as arestas. Para implementar a lista de adjacência utilizamos um array de array de inteiros.

Soluções -> array de inteiros

Para armazenar as soluções utilizamos um **array de inteiros**. O índice i do array representa o vértice i do grafo. O inteiro array[i] representa a ordem do vértice na arranjo linear.

Restricted Candidate List (RCL) -> array de inteiros

A RCL é uma estrutura de dados utilizada para adicionar aleatoriedade a heurística gulosa (greedy), nominada por *semi-greedy heuristic* (Hart and Shogan, 1987; Feo and Resende, 1989). Nesta heurística, os elementos selecionados pelo algoritmo guloso não vão diretamente para a solução, mas para uma lista prévia, a RCL. Ao final, os elementos da RCL são sorteados aleatoriamente para entrar na solução. Utilizamos um **array de inteiros** para implementar a RCL pois facilitaria na seleção randômica de seus itens, o que não é possível em outras estruturas mais elaboradas, com interface limitada a seleção apenas do primeiro ou último elemento.

Elementos selecionados -> queue

Para organizar os elementos selecionados para solução utilizamos uma **queue**. Escolhemos uma fila pela característica FIFO, garantindo a prioridade dada na seleção do elemento na RCL.

Heurística Construtiva

Soluções geradas aleatoriamente, na média, são de baixa qualidade (Johnson et al., 1988). Algoritmos gulosos produzem solução de melhor qualidade do que as geradas aleatoriamente. Nesta construção, o algoritmo seleciona um elemento da solução por vez, visando a cada passo a melhor solução. Apesar de gerar boas soluções iniciais, algoritmos gulosos não possuem capacidade de gerar uma grande variedade de soluções iniciais. GRASP

é uma heurística proposta para equilibrar o uso destas duas técnicas com o objetivo de atingir melhores soluções. Para isso, utiliza-se uma estrutura de dados intermediária entre a escolha do algoritmo guloso e a solução. Elementos selecionados gulosamente são colocados em uma lista de candidatos restrita, chamada RCL. Após a finalização do algoritmo guloso, aleatoriamente os elementos são selecionados, formando assim uma solução semi-gulosa (Hart and Shogan, 1987; Feo and Resende, 1989).

Para construção de soluções iniciais do problema MINLA (MInimum Linear Arrangement), optamos pelo método de busca por arestas, ou seja, dado um elemento inicial, construímos o restante da solução a partir dos vizinhos (elementos adjacentes) dos elementos já presentes na solução. Iniciamos selecionando o primeiro elemento aleatoriamente. Após, adicionamos seus vizinhos a RCL. Com a iteração finalizada, os elementos na RCL são sorteados aleatoriamente para compor a solução. Repete-se esta heurística até que todos vizinhos dos elementos que já estão na solução sejam selecionados. Caso existam vértices que não possuem vizinhos, são adicionados na solução ao final:

```
Construção():
        solucao = selecionados = RCL = 0
        semente = elemento aleatorio(1 a n)
        solucao.recebe(semente)
        selecionados.recebe(semente)
        enquanto selecionados não for vazio:
                para todos vizinhos de selecionados:
                        selecionados.recebe(vizinho)
                        RCL.recebe(vizinho)
                solucao.recebe(aleatorio(RCL))
        solucao.recebe(elementos sem vizinhos)
```

A complexidade de nossa função de construção é O(n).

Vizinhança e a estratégia de escolha dos vizinhos

Com múltiplas soluções iniciais geradas pela heurística semi-gulosa (GRASP), podemos agora buscar melhores resultados localmente, ou seja, buscar uma solução melhor que esteja próxima da solução inicial gerada através de uma operação elementar.

Para busca local no problema MINLA (MInimum Linear Arrangement), escolhemos como operação elementar a inversão de dois vértices vizinhos da solução. A função de busca local itera sobre os n elementos, invertendo o i-ésimo elemento com seu sucessor. Caso a inversão

resulte em uma melhor solução, a inversão é mantida para a próxima iteração. Caso contrário, a inversão é desfeita:

```
BuscaLocal():
    para todos elementos da solução menos o último:
        AtualizaSolucao(elemento,elemento+1)

AtualizaSolucao(v1,v2):
    diferenca_a_remover = InfluenciaSolucao(v1,v2)
    Inverte(v1,v2)
    diferenca_a_adicionar = InfluenciaSolucao(v1,v2)

    se diferenca_a_remover < diferenca_a_adicionar:
        Inverte(v1,v2)
    senão:
        solucao = solucao - diferenca_a_remover + diferenca_a_adicionar

InfluenciaSolucao(v1,v2):
    para cada vizinho de v1 e v2:
        influencia = influencia + absoluto(v1 - vizinho)
        influencia = influencia + absoluto(v2 - vizinho)
    retorna influencia
```

A complexidade de nossa função de busca local é O(n).

Parâmetros do método

Nosso método recebe como entrada, além dos arquivos de dados, a quantidade de iterações e um parâmetro para informar o uso ou não de aleatoriedade. Testamos as seguintes quantidades de iterações: 1, 10, 100, 1000, 10000, 100000. Cada iteração gera uma solução inicial e n soluções locais.

Criamos um script para automatizar a execução e geração dos dados necessários para análise dos resultados:

```
Script():
        aleatorio=verdadeiro
        execute 10 vezes:
                para iteração = {1, 10, 100, 1000, 10000, 100000}:
                        para cada arquivo de entrada:
                                la-grasp iteração aleatorio < arquivo de entrada
```

Resultados

Focaremos na análise de dois pontos a cerca dos resultados obtidos: melhor quantidade de iterações como critério de parada e aproximação do melhor resultado conhecido.

Serão apresentados gráficos e tabelas com informações consolidadas. Os dados completos podem ser visualizados através deste link: http://goo.gl/nSNr24

Critério de parada

Cada iteração de nossa implementação do GRASP gera uma solução inicial e n soluções locais, sendo n a quantidade de vértices. Analisando os resultados, observamos que o tempo de execução entre 100 iterações e 1000 iterações cresceu linearmente, ou seja, em ordem de grandeza 10, porém a melhora nos resultados na maioria dos casos foi insignificante. Os gráficos 1 e 2 apresentam a curva de melhora de resultado em função do tempo de dois problemas:

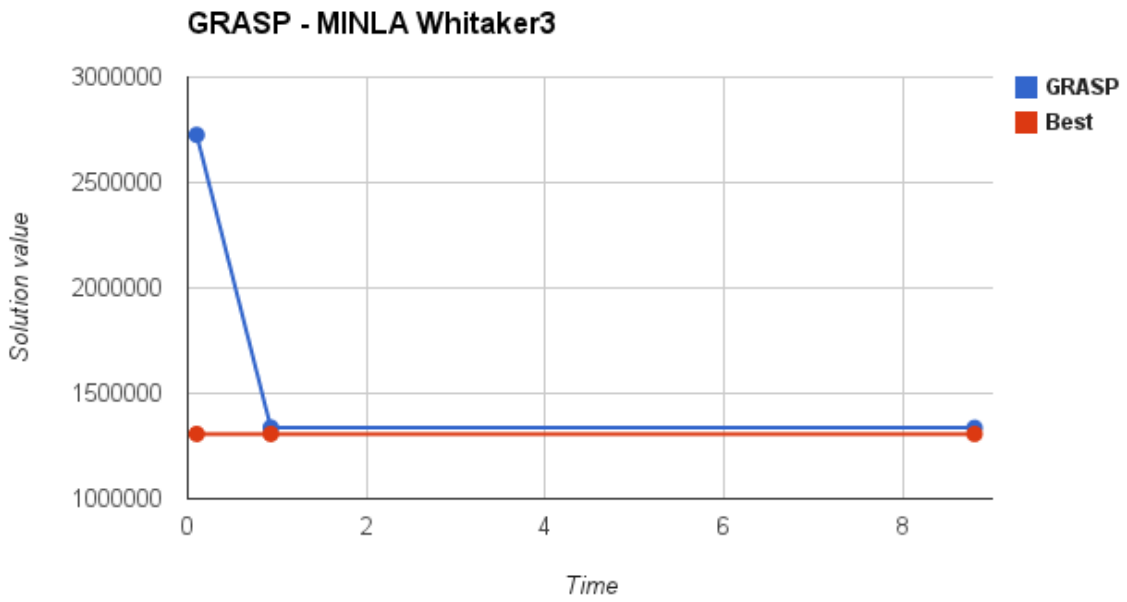


Gráfico 1 - GRASP x Whitaker3

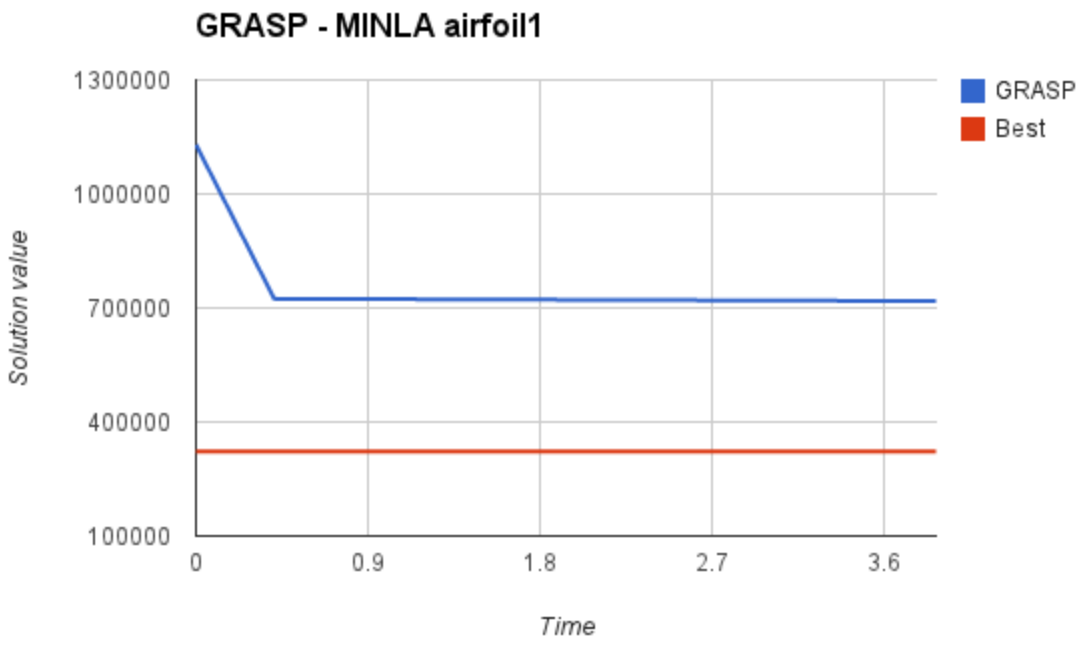


Gráfico 2 - GRASP x Whitaker3

Aproximação do melhor resultado conhecido

Em comparação com os melhores resultados conhecidos, de modo geral os resultados obtidos não foram bons. A solução através de GRASP pelo método de busca por arestas foi satisfatório para o problema whitaker3. Na tabela1 apresentamos a consolidação dos resultados, destacando o pior(vermelho) e melhor(verde) caso.

Input	First	Best 1000	Avg 1000	Time 1000	Best 100	Avg 100	Time 100	Best-know	Jpetit	SD
3elt	1146730	905274	907408.8	4.31607	903584	915322.1	0.4560	431737	7023967	52.2195
airfoil1	1129899	718139	721838.5	3.87143	724347	728658.8	0.4092	322611	1146768	55.4618
bintree10	136825	112382	112382.6	0.639697	112382	112412.8	0.0715	4267	210675	96.2031
c1y	269491	221639	222973.8	0.729738	223822	226652.4	0.0767	-	316477	-
c2y	331689	282242	284962.3	0.85797	286521	290872.9	0.0899	-	446717	-

c3y	520177	440169	445218.6	1.14811	449569	455669.8	0.1221	124117	828414	72.3920
c4y	503001	384365	389834.4	1.16648	397981	404119.6	0.1229	115114	867489	71.0755
c5y	435395	335317	340319.9	1.04924	341663	350118.4	0.1097	96952	657756	71.6235
gd95c	1359	797	805.3	0.087871	809	819.2	0.0093	-	1120	-
gd96a	317325	253510	256629.3	0.8579	252542	259731.1	0.0907	96253	397617	61.8863
gd96b	6180	2242	2250.2	0.121283	2239	2271.4	0.0128	-	3653	-
gd96c	851	688	691.4	0.083473	693	700.3	0.0087	-	1559	-
gd96d	10001	5003	5067.5	0.161754	5151	5211.4	0.0169	-	7814	-
hc10	923780	859474	863702.8	1.27106	863392	868411.4	0.1349	523776	1277994	39.3351
mesh33x33	64191	45776	46026.8	0.806718	45910	46405.7	0.0876	32703	495005	28.7672
randomA1	1388105	1229477	1237075.7	1.37318	1231895	1244865.3	0.1450	-	1349328	-
randomA2	8044778	7469522	7483129.5	4.26564	7486724	7504823	0.4490	-	7570309	-
randomA3	16367646	15560992	15580630.1	7.89588	15563643	15610043.9	0.8333	-	1560037	-
randomA4	2481450	2214636	2221320.2	1.84632	2226231	2235419.6	0.1950	-	2324446	-
randomG4	278462	258660	261106.9	1.81093	262538	263654	0.1931	-	1550172	-
small	18	14	14	0.034586	14	14	0.0035	-	-	-
whitaker3	2726359	1336727	1341253.7	8.80439	1337980	1359430.6	0.9291	1307540	57965229	2.2751

Tabela 1 - Resultados consolidados

Encontramos outro experimento similar a este porém com outras heurísticas envolvidas, conduzido pelo Dr. Jordi Petit da Universidade Politécnica de Cataluña. Nos resultados apresentados por esta pesquisa, a heurística gulosa randômica também apresenta resultados ruins, em comparação com outras técnicas como SA (Simulated Annealing). Comparando os dois trabalhos, o nosso obteve resultados mais satisfatórios, como pode ser visto na tabela 1.

Conclusões

Os resultados nos mostraram que alcançar bons resultados de aproximação para problemas NP-Hard como o MINAL depende fortemente da diversidade de características das instanciase da meta heurística escolhida. Nossos resultados mostraram que a solução através de GRASP pelo método de busca por arestas foi satisfatório para o problema whitaker3, porém longe de ser satisfatório para outros problemas, como o bintree10.

Referências

http://www.inf.ufrgs.br/~mrpritt/oc/la.zip
http://tracer.lcc.uma.es/problems/minla/minla.htm
http://www2.research.att.com/~mgcr/doc/grasp-hao.pdf
http://www.lsi.upc.edu/~jpetit/MinLA/Experiments/upperbounds.html