Problema da Mochila Relacional com PLI

André Almeida *164047*
MC658 - Lab 04

I. O problema

São dados n itens, $N = 1, ..., n$, e uma mochila de capacidade C. Cada item $i \in N$ tem um peso $s_i > 0$ e valor v_i e cada par de itens $i, j \in N$ tem uma relação r_{ij} (não necessariamente positiva). O objetivo é encontrar um subconjunto $S \subseteq N$ tal que $\sum_{i \in S} s_i \leqslant C$ e $\sum_{i \in S} v_i + \sum_{i \in S} \sum_{j \in S, j > i} r_{ij}$ é máximo.

A entrada do problema é dada por:

- Número de itens;
- Matriz de relações de itens;
- Vetor de valor de itens;
- Vetor de peso de itens;
- A capacidade da mochila e
- O tempo máximo disponível para o cálculo da solução

Todos os resultados mostrandos foram executados em uma máquina com processador Intel Core i7-3537U CPU @ 2.00GHz, rodando com GNU/Linux 4.15. O compilador usado foi o GCC 7.3.0 e o Gurobi 7.5.2 como o *solver* do PLI, usando as *flags* de compilação do Makefile disponibilizado.

II. Programação Linear Inteira

A programção linear inteira é um jeito otimizado de resolver sistemas lineares onde as soluções são dadas por números inteiros. O algoritmo que resolve esse

sistema faz parte da classe de problemas **NP-Completos**. Para definirmos nosso sistema linear, precisamos definir as varíveis do sistema, o que queremos maximizar/minizar e as restrições as quais nossas variáveis estão sujeitas.

a. Definição do modelo

Nesse problema, temos duas variáveis:
- x_i, $i \in N$: define se o objeto i está na solução S;
- y_{ij}, $i, j \in N, j > i$: define se a relação entre os objetos i e j deve ser aplicada, ou seja, se i e j fazem parte da solução S. Fazemos com que $j > i$ para evitar pegar a mesma relação duas vezes, só utilizando a matriz diagonal superior direita.

Queremos então máximizar o valor dos objetos escolhidos mais a soma das relações:

$$max \sum_{i \in S} v_i + \sum_{i \in S} \sum_{j \in S, j > i} r_{ij}$$

As nossas restrições são:

- $\sum_{i \in S} s_i \leqslant C$: a soma de todos os pesos dos objetos na solução não devem ser maior que a capacidade;
- $2 * y_{ij} \leqslant x_i + x_j$: define que y_{ij} não deve ser 1 se ambos objetos não estão na solução;
- $1 + y_{ij} \geqslant x_i + x_j$: define que y_{ij} deve ser 1 se ambos objetos estão na solução;

b. Resultados

Com estes dados aplicados no código usando o Gurobi, foi possível chegar nestes resultados (com o tempo máximo definido como 120 segundos). Essa solução utiliza apenas o *presolver* do Gurobi, sem usar nenhuma heurística personalizada. A precisão é definida pela divisão entre o valor obtido e o valor esperado:

Teste	Precisão (%)	Tempo (s)
1	100,00%	0.0142
2	100,00%	0.112
3	100,00%	0.0130
4	100,00%	0.0197
5	100,00%	0.0215
6	100,00%	0.0190
7	100,00%	0.0466
8	100,00%	0.0571
9	100,00%	0.0706
10	100,00%	0.0850
11	100,00%	0.4466
12	100,00%	18.1200
200	100,00%	72.0984
300	5,61%	120
400	3,31%	120
500	0,79%	120
600	1,27%	120

III. Heurística

Foi criada no experimento passado uma heurística para resolver o problema de forma aproximada. A heurística ordena decrescentemente os itens pelo valor relativo, como definido a seguir:

$$valor\_relativo = (somaTodasRelaÇÕes(item) + valor[item])/peso[item]$$

Após a ordenação, ela insere o maior item não inserido que cabe na mochila até que ela esteja cheia e toma estes itens como a solução. Essa heurística foi escolhida porque privilegia os itens com maior potencial para contribuirem com uma soma de valores próxima da ótima. A divisão pelo peso garante que o valor relativo decresça conforme seu volume aumenta.

a. Implementação

A implementação da heurística é dada por:

> *01.* Inicia o alarme com o tempo máximo e ordena decrescentemente o vetor `itens` com os `n` itens pelo seu *valor relativo*
> *02.* Inicializa `soma` como 0
> *04.* Inicializa `capacidade_parcial` com o valor da `capacidade`
> *05.* Inicializa `solucao` como um vetor vazio
> *06.* Para `ì`, de `0` até `n`:
> *07.* Se o alarme acabou, retorna a soma e solução atual
> *08.* Se `peso[i] <= capacidade_parcial`:
> *09.* Coloca `i` em `solucao`
> *10.* Adiciona `valor[i]` em `soma`
> *11.* Adiciona `valorRelacoes(i, solucao)` em `soma`
> *12.* Subtrai `peso[i]` de `capacidade`
> *13.* Retorna `soma` e `solucao`

b. Resultados

Então, aplicamos o valor encontrado pela heurística como um valor inicial para o Gurobi e desabilitamos o *presolver* dele, para usar apenas o nosso. Todos os resultados obtiveram 100% de precisão, por isso esse número foi omitido.

Caso	Tempo (s)
1	0.0094
2	0.0082
3	0.0105

Caso	Tempo (s)
4	0.0162
5	0.0139
6	0.0169
7	0.0152
8	0.0444
9	0.0586
10	0.0745
11	0.0795
12	0.4466
200	5.1246
300	11.5292
400	12.6679
500	15.2679
600	25.7337

IV. Conclusão

Os resultados obtidos sem a heurística foram fracos para os casos grandes. O *presolver* do Gurobi se mostrou pouco eficiente para este problema. Quando havia o corte por tempo, a solução apresentada estava muito longe da solução esperada. É preferível utilizar a solução aproximada do laboratório anterior para as entradas grandes.

Já os resultados com a heurística foram muito positivos. Em menos de 30 segundos o algoritmo consegue resolver com exatidão qualquer entrada. Isso se deve ao fato da heurística cortas ramos da árvore de solução que com certeza não irão gerar

resultados próximos ao ótimo. Quanto mais perto da solução exato o resultado da heurística estiver, mais cortes serão feitos e melhor será o desempenho. Como pode ser visto no relatório do experimento anterior, os casos grandes tinham heurísticas de 99% de precisão, o que com certeza garantiu um bom desempenho. Essa solução foi a melhor encontrada até o momento para as entradas grandes, já que retorna o resultado 100% preciso em tempo hábil. Para as entradas menores, o algoritmo exato do laboratório anterior e a solução de PLI com heurística tem custos computacionais muito próximos, sendo necessários mais testes para chegar a uma conclusão de qual dos dois é o melhor para entradas pequenas em geral.