Atividade 4 - GRASP

Felipe Pereira RA 263808
Sandro Henrique Uliana Catabriga RA 219597
Lucas Guesser Targino da Silva RA 203534

14 de maio de 2022

1 Definições

Definição 1 (Conjunto Binário)**.** $\mathbb{B} = \{0, 1\}$

Definição 2 (Função Binária Quadrática (QBF))**.** É uma função $f : \mathbb{B}^n \to \mathbb{Z}$ da forma:

$$f(x) = \sum_{j=1}^{n} x_i \cdot a_{i,j} \cdot x_j = x^T \cdot A \cdot x$$

em que $a_{i,j} \in \mathbb{Z},\ \forall i, j \in \{1, \cdots, n\}$ e A é a matriz n por n induzida pelos $a_{i,j}$.

Definição 3 (Problema de Maximização de uma Função Binária Quadrática (MAX-QBF))**.** Dada uma QBF f, um MAX-QBF é um problema da forma:

$$\max_x f(x)$$

Fato 1. MAX-QBF é NP-difícil [**?**]

Definição 4 (Maximum knapsack quadractic binary function (MAX-KQBF))**.** Dada uma QBF f, um vetor $w \in \mathbb{Z}^n$[1], e um valor $W \in \mathbb{Z}$, um MAX-KQBF é um problema da forma:

$$\begin{aligned} \max \quad & f(x) \\ \text{subjected to} \quad & w^T x \leq W \\ & x \in \mathbb{B}^n \end{aligned}$$

2 Metodologia

Abordaremos nessa seção os itens básicos para podermos implementar os métodos de construção do GRASP: lista de candidatos e critérios de parada. Definidos esses pontos, apresentamos os três métodos implementados e comparados: heurística construtiva padrão, *random plus greedy*, e *sampled gredy*.

[1] O problema original foi definido com números reais. Decidimos aqui utilizar inteiros por dois motivos. Primeiro, todas as instâncias fornecidas possuem apenas valores inteiros para $a_{i,j}, w, W$. Garante-se que os valores são sempre inteiros pois $\mathbb{Z}$ é fechado nas operações envolvidas: adição e multiplicação. Segundo, simplifica a implementação e comparações (não é necessário fazer comparação de números em ponto flutuante).

2.1 Lista de Candidatos

A lista de candidatos é independente do método de construção utilizado. Diferente do problema QBF, no MAX-KQBF temos uma restrição relativa à capacidade da mochila que deve ser respeitada. Portanto, são candidatas a comporem a solução aquelas variáveis cujo peso, quando inseridas na mochila, não ultrapassa a capacidade definida.

Podemos definir a lista de candidatos CL como: $\text{CL} = \{x_i \mid W_{s'} + w_i \leq W, \forall i \in \{1, \ldots, n\} \backslash s'\}$, onde s' é a solução atual (que está em construção). Ou seja, são candidatas as variáveis que não fazem parte da solução atual, e quando incorporadas a ela não estouram a capacidade da mochila.

2.2 Critérios de Parada dos Métodos de Construtivos

Para todos os métodos de construção utilizados, os critérios de parada foram os mesmos. São eles:

1. Solução não melhora: quando a inserção de um candidato na solução não a melhora.
2. Lista de candidatos vazia: quando não há mais candidatos para serem inseridos na solução.

Dessa forma, as soluções são construídas iterativamente até que um dos critérios de parada ocorra.

2.3 Heurística Construtiva Padrão

Para a heurística construtiva padrão do GRASP, consideramos dois valores de α: 0.2 e 0.5. Para cada iteração, enquanto o critério de parada não era atingido, a lista restrita de candidatos RCL era construída a partir da lista de candidatos CL e do valor de α.

Seja $c(x_i)$ a função que mede a contribuição da variável x_i na função objetivo, c^{min} a menor contribuição dentre os candidatos, e c^{max} a maior contribuição. Podemos definir a lista restrita de candidatos RCL: $\text{RCL} = \{x_i \mid c(x_i) \in [c^{min}, c^{min} + \alpha(c^{max} - c^{min})]\}$. Ou seja, entram para a lista restrita de candidatos todos os candidatos que estão em um intervalo de qualidade definido por α.

A partir da RCL, seleciona-se então aleatoriamente um candidato para ser integrado à solução.

2.4 Método de Construção Alternativo 1

Para o método de construção *random plus greedy*, que busca juntar aleatoriedade e determinismo, utilizamos $p = n \times 0.8$, onde n é o número de variáveis no nosso problema. Dessa forma, buscamos utilizar aleatoriedade completa, com $\alpha = 1$, em aproximadamente 80% dos passos construtivos, e no restante utilizar determinismo, com $\alpha = 0$.

A construção da lista restrita de candidatos foi equivalente a da heurística construtiva padrão, mas, devido ao valor de p, nas primeiras p iterações utilizamos $\alpha = 1$, e nas restantes $\alpha = 0$. Quando $\alpha = 1$, o candidato selecionado para compor a solução foi escolhido aleatoriamente, e quando $\alpha = 0$, o candidato selecionado para compor a solução foi o melhor da lista de candidatos.

2.5 Método de Construção Alternativo 2

Para o *sampled greedy*, que também busca combinar aleatoriedade e determinismo, utilizamos $p = n \times 0.7$. Assim, a lista restrita de candidatos RCL foi composta por $min(p, |CL|)$ candidatos da lista de candidatos CL, selecionados aleatoriamente.

Portanto, a cada iteração, a RCL era construída selecionando aleatoriamente $min(p, |CL|)$ candidatos da CL. Para compor a solução, era selecionado o melhor candidato da RCL.

2.6 Operadores de Busca Local

Existem três Operadores: Inserção, Remoção e Troca.

Cada operador foi implementado para levar em consideração o peso atual da Mochila.

No caso do Operador de Inclusão, cada candidato da Lista de Candidatos é avaliado conjuntamente à solução atual disponível. O novo item é inserido na Mochila, e a Mochila possui seu peso avaliado, verificando se as restrições de peso mínimo e peso total da Mochila estão sendo atendidas.

Para o Operador de Remoção, são avaliados todos os itens da Mochila, e o item com menor contribuição para o custo da solução é retirado da Mochila.

Por fim, o Operador de Troca é implementado de tal forma que ele escolhe dois itens, um da solução e um da lista de candidatos, de forma a otimizar o custo da solução. O primeiro é removido enquanto que o segundo é inserido.

2.7 Métodos de Busca

Foram implementados dois métodos de busca: First-Improving e Best-Improving.

2.8 First-Improving

O método de First-Improving foi implementado de forma que a primeira coisa importante a acontecer é a avaliação do peso atual da Mochila e, logo em seguida, a atualização das Lista de Candidatos.

Para cada item da Lista de Candidatos atualizada, é feita uma primeira verificação para avaliar se o item pode ser inserido na Mochila sem violar as restrições do problema. Caso sim, o item é inserido com sucesso se a inserção deste item melhora o custo da solução.

Logo em seguida é avaliado a exclusão de algum item da lista. Se um candidato apropriado, e que melhore o custo da solução, é encontrado, ele é removido da solução corrente.

Por fim, para cada item da Lista de Candidatos, é avaliado se a troca de um item da Lista de Candidatos (adição à solução corrente) por um item da Mochila (exclusão da solução corrente) é benéfica para a solução como um todo. Se sim, a troca é feita.

O custo da solução é sempre levado em conta, através do cálculo da função objetivo, e as modificações só são operadas se e somente se: obedecem às restrições e reduzem o custo da função. Quando um item é inserido, removido ou trocado, o restante das possibilidades não são exploradas, tal qual o First-Improving exige.

2.9 Best-Improving

O algoritmo de Best-Improving tem uma implementação relativamente parecida, porém com uma diferença importante: os passos de Inserção, Exclusão e Troca são avaliados para cada item, porém só são executados de fato após todos os itens serem avaliados.

Ou seja, primeiramente avalia-se, da Lista de Candidatos, todos os itens candidatos a serem inseridos na Mochila.

Em seguida avalia-se, na Solução Corrente, todos os itens que podem ser excluídos. Por fim avaliam-se, para todos os itens da Lista de Candidatos e da Solução Corrente, quais são as trocas vantajosas para o custo da solução.

Instância	—x—	Número de possibilidades	MAX-KQBF (Z_*)
kqbf020	20	1.0e+06	[80, 151]
kqbf040	40	1.1e+12	[275, 429]
kqbf060	60	1.2e+18	[446, 576]
kqbf080	80	1.2e+24	[729, 1000]
kqbf100	100	1.3e+30	[851, 1539]
kqbf200	200	1.6e+60	[3597, 5826]
kqbf400	400	2.6e+120	[10846, 16625]

Tabela 1: Definição das instâncias utilizadas.

Apenas ao fim de todas estas avaliações, o algoritmo decide qual é o melhor movimento a ser feito e implementa apenas o melhor movimento a ser feito, seja ele de Inserção, Remoção ou Troca.

2.10 Critérios de Parada do GRASP

Para o GRASP, no método principal, realizamos sempre 1000 iterações. Esse foi o critério de parada estabelecido. Para as instâncias menores, era suficiente para encontramos uma boa solução. Para as instâncias maiores, era suficiente para executar o GRASP até o tempo limite máximo estabelecido, que foi de 30 minutos.

3 Experimentos Computacionais

3.1 Configurações da Máquina

O problema foi executado num ideapad S145 81S90005BR: Lenovo IdeaPad S145 Notebook Intel Core i5-8265U (6MB Cache, 1.6GHz, 8 cores), 8GB DDR4-SDRAM, 460 GB SSD, Intel UHD Graphics 620.

O sistema operacional foi o Fedora 35 executando o Java 11 e Gradle 6.8.

3.2 Instâncias

Foram utilizadas as instâncias fornecidas para este problema conforme a Tabela 1.

4 Análise

4.1 Custo das Soluções

Comparando os custos ótimos obtidos com os fornecidos na Tabela 1, verifica-se que os valores ótimos obtidos estão dentro do intervalo de referência. Isso dá alguma credibilidade para os resultados.

4.2 Tempo de Execução

4.2.1 *First Improving* e *Best Improving*

Observa-se que *Best Improving* é mais lento porém leva a soluções melhores do que o *First Improving*.

Tanto o *Best Improving* quanto o *First Improving* implementam um procedimento de troca: incluir um elemento e excluir outro da solução atual caso isso leve a uma melhora. A diferença é que no *Best Improving* ela sempre é executado, enquanto que no *First Improving* a troca só é feita quando ela é a única opção de melhora da solução.

Se a mochila está cheia, ou muito próxima de cheia, adicionar um elemento é quase inviável. Além disso, remover elementos, em geral, não é uma opção que leva a melhora de solução[2]. Portanto, soluções boas são obtidas com maior probabilidade quando trocas são feitas. Isso, pelo exposto no parágrafo anterior, leva a piora nos tempos computacionais. Pode-se inferir, então, que execuções muito rápidas terão soluções ruins pois não exploram tão bem a vizinhança.

Considerando a restrição de limite da mochila, seria interessante implementar buscas locais que permitissem trocas entre três elementos (remover dois e adicionar um, remover um adicionar dois). Em tal caso, o tempo computacional provavelmente aumentaria consideravelmente por causa do aumento da complexidade do algoritmo. Seria também interessante considerar exceder tal limite temporariamente a fim de explorar mais soluções interessantes (é fácil corrigir uma solução assim com uma estratégia gulosa: basta remover elementos com alto peso e baixa contribuição no objetivo). Novamente, isso está além do escopo da presente atividade.

4.3 Parâmetros Ótimos

Observa-se que os melhores resultados, tanto em tempo de execução quanto em custo da solução, foram obtidos com o seguintes parâmetros:

- $\alpha = 0.2$
- local search = *Best Improving*
- método de construção = default
- iterações = 1000

Comparando os resultados das configurações acima com $\alpha = 0.5$ observa-se que o custo ainda é bom mas o tempo de execução é muito prejudicado. Constata-se que valores muito altos para α requerem uma busca local muito intensa, o que explica o tempo computacional maior.

Já comparando os resultados das configurações acima com o método de busca local com *First Improving*, nota-se que eles não são muito diferentes, nem em tempo de execução nem em custo. De fato, ambos são bem parecidos quando caem no caso de troca de elementos na solução. Como esse é um algoritmo guloso, é muito provável que no momento da busca local, a única forma de melhorar a solução é com trocas e por isso ambas acabam tendo performances muito parecidas.

4.4 *Random Plus Greedy*

Observa-se que o método de construção *Random Plus Greedy* obtém custos bons para a busca local *Best Improving*, mas não para a *First Improving*. Nesse problema, tanto soluções puramente aleatórias quanto puramente gulosas demandam muito da busca local, o que explica os altos tempos de execução observados (comparando com o método *Padrão*).

Observando a performance do *First Improving*, pode-se obter três conclusões importantes:

[2]Isso é uma hipótese baseada na intuição dos autores. Seria interessante implementar contadores e monitorar o número de inserções, remoções, e trocas, para então verificar tal suposição. Isso está além do escopo da presente atividade entretanto.

1. as soluções obtidas pelo método de construção *Random Plus Greedy* são inicialmente bem ruins, demandando um método de busca local muito bom (se não fosse esse o caso, as soluções com a busca local *First Improving* deveriam ser melhores);

2. o método de busca local *Best Improving* é bastante bom para o problema, ele é capaz de encontrar ótimos locais bons mesmo partindo de soluções inicialmente ruins;

3. a qualidade da solução depende fortemente do método de busca local, enquanto que o tempo de execução depende fortemente do método de construção. O último é o que demanda menos tempo, de forma que quanto melhor ele for, menor o caminho que a busca local tem que fazer. Obviamente, um ótimo é obtido com uma busca local não tão custosa (e eventualmente menos poderosa) aliada a um método de construção bom.

4.5 *Sampled Greedy*

Os resultados do *Sampled Greedy* são muitíssimo parecidos com os do *Random Plus Greedy*. Na verdade, ambos os métodos são bastante parecidos, segundo [1] ambos visam combinar características gulosas e diversificadoras. Essa sofre do mesmo problema de excesso de diversificação descrito na Subseção 4.4: as soluções iniciais são ruins e dependem muito da qualidade da busca local.

4.6 Método de Construção - Conclusão

Conclui-se que ambos *Random Plus Greedy* e *Sampled Greedy* são ineficientes para o problema tratado nesse trabalho. Observa-se ainda que ajustar os parâmetros de tais métodos, considerando a análise feita, não indica nenhuma perspectiva de melhora. Portanto, o método de construção mais indicado é *Padrão*.

5 Resultados

	instances	construction	alpha	local search	iterations	best cost	weight	duration
0	020	default	0.2	first improving	1000	120	56	0.145
1	020	default	0.5	first improving	1000	120	56	0.042
2	020	default	0.2	best improving	1000	120	56	0.074
3	020	default	0.5	best improving	1000	120	54	0.1
4	020	random plus greedy	-	first improving	1000	57	49	0.086
5	020	random plus greedy	-	best improving	1000	120	54	0.145
6	020	sampled greedy	-	first improving	1000	84	62	0.09
7	020	sampled greedy	-	best improving	1000	120	54	0.138

Tabela 2: Número de possíveis soluções para cada tamanho de problema - parte 0.

	instances	construction	alpha	local search	iterations	best cost	weight	duration
8	040	default	0.2	first improving	1000	306	136	0.353
9	040	default	0.5	first improving	1000	269	132	0.196
10	040	default	0.2	best improving	1000	295	137	1.765
11	040	default	0.5	best improving	1000	286	133	1.627
12	040	random plus greedy	-	first improving	1000	92	95	0.063
13	040	random plus greedy	-	best improving	1000	224	137	1.806
14	040	sampled greedy	-	first improving	1000	89	76	0.061
15	040	sampled greedy	-	best improving	1000	277	137	1.616

Tabela 3: Número de possíveis soluções para cada tamanho de problema - parte 1.

	instances	construction	alpha	local search	iterations	best cost	weight	duration
16	060	default	0.2	first improving	1000	455	220	1.261
17	060	default	0.5	first improving	1000	380	216	0.493
18	060	default	0.2	best improving	1000	491	212	1.981
19	060	default	0.5	best improving	1000	491	212	6.399
20	060	random plus greedy	-	first improving	1000	77	68	0.148
21	060	random plus greedy	-	best improving	1000	482	215	7.876
22	060	sampled greedy	-	first improving	1000	83	96	0.134
23	060	sampled greedy	-	best improving	1000	482	215	7.149

Tabela 4: Número de possíveis soluções para cada tamanho de problema - parte 2.

	instances	construction	alpha	local search	iterations	best cost	weight	duration
24	080	default	0.2	first improving	1000	783	286	3.016
25	080	default	0.5	first improving	1000	598	271	1.184
26	080	default	0.2	best improving	1000	795	284	4.242
27	080	default	0.5	best improving	1000	802	284	25.248
28	080	random plus greedy	-	first improving	1000	118	69	0.27
29	080	random plus greedy	-	best improving	1000	802	284	20.344
30	080	sampled greedy	-	first improving	1000	151	69	0.249
31	080	sampled greedy	-	best improving	1000	791	287	19.385

Tabela 5: Número de possíveis soluções para cada tamanho de problema - parte 3.

	instances	construction	alpha	local search	iterations	best cost	weight	duration
32	100	default	0.2	first improving	1000	1213	335	6.056
33	100	default	0.5	first improving	1000	911	343	1.927
34	100	default	0.2	best improving	1000	1228	345	9.068
35	100	default	0.5	best improving	1000	1234	343	75.18
36	100	random plus greedy	-	first improving	1000	101	100	0.42
37	100	random plus greedy	-	best improving	1000	1234	343	85.987
38	100	sampled greedy	-	first improving	1000	113	99	0.393
39	100	sampled greedy	-	best improving	1000	1234	343	50.211

Tabela 6: Número de possíveis soluções para cada tamanho de problema - parte 4.

	instances	construction	alpha	local search	iterations	best cost	weight	duration
40	200	default	0.2	first improving	1000	3692	679	75.552
41	200	default	0.5	first improving	1000	2672	669	11.496
42	200	default	0.2	best improving	1000	3766	677	100.731
43	200	default	0.5	best improving	1000	3840	678	1280.062
44	200	random plus greedy	-	first improving	1000	71	86	1.563
45	200	random plus greedy	-	best improving	1000	3829	678	1422.645
46	200	sampled greedy	-	first improving	1000	133	105	1.379
47	200	sampled greedy	-	best improving	1000	3837	678	1424.165

Tabela 7: Número de possíveis soluções para cada tamanho de problema - parte 5.

	instances	construction	alpha	local search	iterations	best cost	weight	duration
48	400	default	0.2	first improving	1000	9756	1342	650.589
49	400	default	0.5	first improving	1000	7102	1343	77.348
50	400	default	0.2	best improving	1000	10213	1343	973.293
51	400	default	0.5	best improving	1000	10197	1343	1816.949
52	400	random plus greedy	-	first improving	1000	91	119	5.995
53	400	random plus greedy	-	best improving	1000	9833	1343	1815.805
54	400	sampled greedy	-	first improving	1000	147	118	6.574
55	400	sampled greedy	-	best improving	1000	9832	1343	1809.777

Tabela 8: Número de possíveis soluções para cada tamanho de problema - parte 6.

Referências

[1] M. G. Resende and C. C. Ribeiro, "Greedy randomized adaptive search procedures: advances and extensions," in *Handbook of metaheuristics*, pp. 169–220, Springer, 2019.