MO824 - Algoritmos Genéticos

Adolfo Aires Schneider (234991)
Ieremies Romero (217938)
Lucas Guesser Targino da Silva (203534)

25 de junho de 2022

1 Descrição do problema

No problema *Maximum Quadratic Binary Function* (MAX-QBF), proposto em [1], temos que maximizar a função $f : \mathbb{B}^{|x|} \to \mathbb{R}$, onde $|x|$ é a dimensão do problema, descrita como:

$$\begin{aligned} max \quad & \sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}x_i x_j \\ & x_i \in \{0,1\} \qquad \text{para todo i } \in \{1,..,n\}, \end{aligned}$$

em que $a_{i,j} \in \mathbb{R}$ $(i, j = 1, \ldots, n)$ são os coeficientes da função, dados como parâmetros do problema.

A este problema adicionamos a restrição

$$\sum_{i=1}^{n} w_i x_i \leq W,$$

onde w_i é dito o peso da variável x_i. Nessa nova versão chamada *Maximum Knapsack quadratic binary function* (MAX-KQBF), continuamos tendo por objetivo maximizar a função f, mas desta vez também precisamos satisfazer a restrição que a soma dos pesos das variáveis na solução não exceda W. Veja que podemos ver esse problema como um conjunto de itens x, onde cada item $i \in x$ tem um peso w_i, nos quais devem ser empacotados em uma mochila de capacidade W. Neste caso, as variáveis de decisão x_i, consiste em escolher se o item i está ou não na mochila (que representa a solução), onde $x_i = 1$ se o item i estiver na solução.

Durante esse relatório, nos referiremos a $\delta(x_i)$ como o custo que o elemento x_i incumbe a atual solução X. Caso x_i não esteja na solução X, então $\delta(x_i) = f(X \cup \{x_i\}) - f(X)$. Caso x_i já esteja na solução X, então $\delta(x_i) = f(X \setminus \{x_i\}) - f(X)$.

2 Metodologia

Nessa atividade, utilizaremos **Algoritmos Genéticos** (GA) como nossa meta-heurística.

3 Codificação de uma solução

Uma característica interessante de GA é sua separação da representação e suas variáveis. Tomando como inspiração a biologia, o **genótipo** de é uma representação codificada das variáveis. Assim, o vetor

x é representado por uma string s de tamanho l feita de símbolos no alfabeto A usando o mapeamento c.

O comprimento da string depende tanto do tamanho do alfabeto como do tamanho do espaço de busca do problema. Cada elemento da string é dito **gene** e seus possíveis valores são chamadas **alelos**.

Assim, na prática, precisamos encontrar

$$\arg \min_{s \in A^l} g(s)$$

tal que $g(s) = f(c(s))$, onde f é a função a ser minimizada no espaço de busca original.

Dessa forma, tomando como inspiração o comportamento de seleção natural, os descendentes de uma população (conjunto de soluções) devem ter as características mais "desejáveis". Estas, são determinadas por favorecer aqueles cromossomos (strings acima referenciadas) com maior aptidão, uma função real de f, monotônica, não negativa. A estratégia de escolha de tal função tem por objetivo favorecer aqueles cromossomos que fornecem as melhores soluções para o problema original.

Tais descendentes, acima mencionados, são gerados a partir do cruzamento de cromossomos e mutação. Para o operador de cruzamento (*crossover*), é comum utilizarmos a troca de um ou mais alelos entre os parentes. Já para a mutação, é escolhido um conjunto de alelos randomicamente para terem seus valores alterados.

Podemos repetir esse processo e, a cada iteração, produzir uma nova geração de cromossomos até que estejamos satisfeitos. Um algoritmo de alto nível, retirado de [2], é apresentado no Algoritmo 1.

Algorithm 1 Genetic-Algorithm

```
escolher a população inicial de cromossomos
while condição de parada não é satisfeita do
    while Não há descendentes suficientes do
        if condição de cruzamento é satisfeita then
            Selecionar cromossomos pai
            Selecionar parâmetros de cruzamento
            Executar cruzamento
        if condição de mutação é satisfeita then
            Selecionar pontos de mutação
            Executar mutação
        Calcular adaptação (fitness) dos descendentes
    Selecionar nova população
return Melhor cromossomo da população
```

4 População inicial

É possível perceber que o tamanho da população utilizada irá diretamente influenciar a qualidade das soluções e o tempo de execução. Populações grandes significam uma busca mais ampla e, possivelmente, melhores soluções ao custo de um tempo maior de execução. Assim, precisa-se encontrar um balanço.

Como proposto em [2], podemos utilizar a ideia de que todo o espaço de busca deve ser acessível pelo *crossover* da população inicial. Para tal, cada valor do nosso alfabeto (alelo) deve aparecer em cada posição da 'string' ('locus').

Uma vez determinado o tamanho, é necessário ainda escolher a população inicial. De forma geral, a literatura aponta que uma simples amostragem sem repetição é suficiente, mas formas mais refinadas

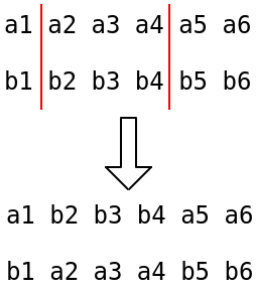

Figura 1: Representação esquemática do ***two point crossover*** (`2X`).

já foram propostas com objetivo de garantir a cobertura do espaço de busca, mencionado acima. Além disso, podemos "semear" a população inicial com boas soluções já obtidas por outros métodos.

5 Seleção e Reprodução

Para produzirmos a próxima geração, precisamos selecionar aqueles indivíduos que serão "reproduzidos". Fazemos isso pelo método estilo **torneio**, onde dois cromossomos são escolhidos aleatoriamente e o melhor dos dois (em relação à função de aptidão) é escolhido para continuar enquanto o outro é descartado. Assim, nossas populações entre gerações mantém os melhores indivíduos em comum, enquanto inserimos novos para ampliar a busca.

Visto que a população conterá duplicatas, precisamos reconstruí-la utilizando o *crossover*. No nosso caso, utilizamos ***two point crossover*** (`2X`) como o método de produzir novas soluções. Para dois cromossomos, ditos pais, construímos dois descendentes. Cada um é a cópia de um dos pais com um intervalo do locus do outro pai, conforme exemplificado na Figura 1. As duas posições, em vermelho na figura, são escolhidas aleatoriamente.

É importante, nesse ponto, ressaltar que nem todo cromossomo gerado corresponde a uma solução viável ao problema original. Porém, estes serão descartados já que a nossa função de aptidão retorna menos infinito para soluções inviáveis ao problema.

6 Mutação

Como discutido anteriormente, é importante que haja uma taxa de mutação a cada reprodução para ser mantida a diversidade da população. Implementamo-na a partir da **taxa de mutação** M que representa a chance de cada alelo em cada descendente de uma geração sofrer uma mutação. Caso isso ocorra, o valor dele é alterado. Iremos experimentar com diferentes valores para esta taxa.

7 Critério de parada

Critérios de parada mais simples, e também mais comuns na literatura, são:

1. limitar o tempo de execução;
2. limitar o número de gerações geradas;

Porém, critérios que consideram os resultados obtidos pelo algoritmo também podem ser utilizados:

1. parar quando a diversidade da população é baixa;
2. parar quando a solução não melhora após algumas gerações;

8 Estratégias evolutivas alternativas

As estratégias evolutivas alternativas utilizadas no desenvolvimento foram *Steady-State* $\lambda + \mu$ e *Manutenção de diversidade*, ambas são descritas abaixo.

8.1 Steady-State $\lambda + \mu$

Uma das estratégias alternativas implementadas é uma variante da estratégia *Steady-State*, a estratégia $\lambda+\mu$. Nessa variante, em cada iteração, μ *offsprings* são gerados, os melhores λ indivíduos do conjunto união dos *parents* e *offsprings* são escolhidos para compor a nova população. Na implementação feita tanto λ quanto μ foram definidos como o tamanho da população. O objetivo dessa abordagem consiste em não perder soluções consideradas boas de uma geração para a próxima, porém podendo levar o algoritmo apresentar uma convergência prematura e uma baixa diversidade. Para balancear esse fator, a próxima estratégia evolutiva alternativa utilizada procura aumentar a diversidade do algoritmo genético.

8.2 *Manutenção de diversidade*

A diversidade da população é um fator importante para um algoritmo genético conseguir explorar de forma ampla o espaço de soluções. Diversos métodos podem ser aplicados com objetivo de aumentar essa diversidade, como criar uma população de maneira mais uniforme, aumentar o tamanho da população ou taxa de mutação e também evitar gerar indivíduos repetidos entre gerações. Para essa última abordagem existe a estratégia alternativa chamada de Manutenção de Diversidade, cujo possui o objetivo de evitar que os *offsprings* sejam meras duplicatas dos indivíduos que as formaram.

Para evitar duplicatas, uma forma seria a comparação dos indivíduos com os novos candidatos, adicionando um esforço computacional maior e mais perceptível em grandes populações. A estratégia utilizada limita os possíveis *crossover-points* no intervalo $[L, R]$ com L e $R \in \{1, ..., n\}$ e $R >= L$ de uma forma a tentar diminuir a chance de ocorrerem duplicatas, onde L é o primeiro locus com alelo diferentes entre os pais e R o último. Assim, as chances do descendente gerado ser igual são reduzidas.

9 Implementação

Partimos de uma configuração padrão:

1. Tamanho da população: $P_1 = 100$;
2. Taxa de mutação: $M_1 = 1\%$;
3. População inicial construída aleatoriamente[1]
4. ***two point crossover*** (`2X`);
5. Critério de parada: 1000 gerações;

[1] Criamos um cromossomo com todos os alelos desativados e ativamos alguns deles aleatoriamente, respeitando as restrições do problema em questão.

Estudamos 5 configurações, alterando apenas um aspecto por vez[2]:

Configuração 1 PADRÂO: Algoritmo genético com tamanho de população P_1, taxa de mutação M_1 e construção aleatória da população.

Configuração 2 PADRÃO + POP: Algoritmo genético PADRÃO, mas com tamanho de população P_2.

Configuração 3 PADRÃO + MUT: Algoritmo genético PADRÃO, mas com taxa de mutação M_2.

Configuração 4 PADRÃO + EVOL1: Algoritmo genético PADRÃO, mas com estratégia evolutiva alternativa 1 (X).

Configuração 5 PADRÃO + EVOL2: Algoritmo genético PADRÃO, mas com estratégia evolutiva alternativa 2 (X).

Selecionamos para os experimentos $P_2 = 300$ e $M_2 = 0.5\%$.

10 Análise dos resultados

10.1 Taxa de mutação

Comparando as Tabelas 1 e 2, observamos que uma taxa de mutação menor levou a resultados um pouco melhores. Na verdade, a diferença é mais evidente em instâncias maiores, como a 200 e 400, já que em instâncias menores ambos chegaram ao mesmo resultado. A mutação, de certa forma, representa uma busca local, no sentido que são feitas modificações nas soluções conhecidas a fim de explorar a vizinhança. Assim, taxas de mutações maiores estão relacionadas a maior diversificação, enquanto que taxas menores estão relacionadas a intensificação. Assim, faz-se necessário escolher um valor para a taxa de mutação que consiga dosar os dois corretamente.

Por causa da forma com que foi implementada, esse balanço entre intensificação e diversificação controlado pela taxa de mutação é influenciado pelo número de genes de um cromossomo: cromossomos maiores sofrem, estatisticamente, mais modificações do que cromossomos menores.

Voltando para a comparação dos resultados, vemos que para instâncias grandes, a taxa de mutação está muito alta, não permitindo o algoritmo acumular características boas de uma solução para a outra. Por isso, seria interessante em trabalhos futuros investigar uma taxa de mutação adaptativa, que considerasse o número de genes de um cromossomo.

10.2 Tamanho da população

Comparando as Tabelas 1 e 3 observamos melhores resultados em geral ao custo de um aumento proporcional no tempo computacional (aproximadamente três vezes maior em cada instância).

A melhora da solução obtida pode ser explicada pelo fato do algoritmo explorar mais o domínio de solução.

Em alguns casos houve piora da solução entretanto. Populações maiores reduzem as chances de indivíduos bons se reproduzirem, e caso a diversidade da população estiver alta, isso pode ser um problema. Nesses casos, utilizar soluções como a estratégia evolutiva alternativa $\lambda + \mu$ seja uma boa alternativa.

[2]Quando não explicitado, os parâmetros se mantém igual à configuração padrão.

10.3 *Manutenção de diversidade*

Comparando as Tabelas 1 e 5 observamos que praticamente não houve modificação nos resultados. A estratégia evolutiva alternativa *Manutenção de diversidade* mostrou-se equivalente à proposta original.

Uma explicação para tal resultado pode estar na similaridade entre as duas. Apesar de haver diferença em como os pontos de cruzamento são selecionados, em ambos eles ainda são bastante similares.

10.4 Steady-State $\lambda + \mu$

Comparando as Tabelas 1 e 4, observamos melhora dos resultados, principalmente nas instâncias maiores.

O algoritmo genético é baseado e gerar descendentes que combinam características boas dos pais. Isso significa que quanto melhores forem as características dos pais, melhores serão as dos filhos. A estratégia original propõe a próxima geração ser todos os filhos (exceto o prior deles) mais a melhor solução até então encontrada. Mas nem todos os filhos têm características boas. Pode ser que alguns filhos combinem apenas as características ruins dos pais. Dessa forma, é interessante considerar manter os pais ao invés dos filhos quando for vantajoso, que é justamente o que a estratégia Steady-State $\lambda + \mu$ faz.

Por outro lado, observamos um aumento consideravel no tempo computacional. Isso pode ser explicado pela operação necessária para selecionar os melhores elementos entre os pais e filhos, que envolve uma operação de ordenação.

Referências

[1] G. Kochenberger, J.-K. Hao, F. Glover, M. Lewis, Z. Lü, H. Wang, and Y. Wang, "The unconstrained binary quadratic programming problem: a survey," *Journal of combinatorial optimization*, vol. 28, no. 1, pp. 58–81, 2014.

[2] C. R. Reeves, "Genetic algorithms," in *Handbook of metaheuristics*, pp. 109–139, Springer, 2010.

[3] L. G. T. da Silva, "Mo824a-combinatorial-optimization," 2022.

Apêndice A Implementação e execução dos experimentos

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 17 e Gradle 7.4.

O desenvolvimento da solução do problema foi feito em Java, baseado no framework disponibilizado pelo professor, e o código pode ser encontrado em [3].

Apêndice B Instâncias

As instâncias utilizadas nos experimentos foram disponibilizadas pelo professor da disciplina e diferem essencialmente na quantidade de variáveis $|x|$, havendo uma progressão no tamanho da mochila W com o aumento de $|x|$.

Apêndice C Resultado dos experimentos computacionais

- *n. gen*: número de gerações, utilizado como critério de parada;
- *mut rate*: taxa de mutação;
- *pop size*: tamanho da população;
- *k. capac.*: capacidade da mochila;
- *sol. w*: peso da solução, para verificarmos que a restrição de limite de peso é satisfeita;
- *exec t.*: tempo de execução em segundos;
- *sol. cost*: valor da função objetivo (custo da solução encontrada);

	instances	n. gen	mut rate	pop size	k. capac.	sol. w	exec t. [s]	sol. cost
0	20	1000	0.01	100	64	54	0.918	120
1	40	1000	0.01	100	138	132	2.648	303
2	60	1000	0.01	100	221	221	5.082	470
3	80	1000	0.01	100	288	286	9.090	733
4	100	1000	0.01	100	346	345	13.646	1249
5	200	1000	0.01	100	680	671	54.607	3613
6	400	1000	0.01	100	1344	1338	221.373	7884

Tabela 1: Solução obtida com o algoritmo genético não modificado para cada configuração e instância do problema.

	instances	n. gen	mut rate	pop size	k. capac.	sol. w	exec t. [s]	sol. cost
0	20	1000	0.005	100	64	54	0.974	120
1	40	1000	0.005	100	138	132	3.200	303
2	60	1000	0.005	100	221	217	5.529	481
3	80	1000	0.005	100	288	288	9.627	766
4	100	1000	0.005	100	346	345	15.386	1237
5	200	1000	0.005	100	680	678	62.351	3714
6	400	1000	0.005	100	1344	1344	245.488	9200

Tabela 2: Solução obtida com o algoritmo genético não modificado para cada configuração e instância do problema.

	instances	n. gen	mut rate	pop size	k. capac.	sol. w	exec t. [s]	sol. cost
0	20	1000	0.01	300	64	54	2.411	120
1	40	1000	0.01	300	138	133	7.634	307
2	60	1000	0.01	300	221	212	16.005	491
3	80	1000	0.01	300	288	282	27.461	783
4	100	1000	0.01	300	346	346	41.520	1208
5	200	1000	0.01	300	680	679	165.724	3550
6	400	1000	0.01	300	1344	1339	723.375	8240

Tabela 3: Solução obtida com o algoritmo genético não modificado para cada configuração e instância do problema.

	instances	n. gen	mut rate	pop size	k. capac.	sol. w	exec t. [s]	sol. cost
0	20	1000	0.01	100	64	56	3.015	120
1	40	1000	0.01	100	138	136	10.860	305
2	60	1000	0.01	100	221	217	23.954	453
3	80	1000	0.01	100	288	288	46.665	781
4	100	1000	0.01	100	346	346	76.459	1187
5	200	1000	0.01	100	680	679	254.404	3889
6	400	1000	0.01	100	1344	1344	1204.936	10227

Tabela 4: Solução obtida com o algoritmo genético com estratégia adaptativa alternativa Steady-State $\lambda + \mu$ para cada configuração e instância do problema.

	instances	n. gen	mut rate	pop size	k. capac.	sol. w	exec t. [s]	sol. cost
0	20	1000	0.01	100	64	54	1.033	120
1	40	1000	0.01	100	138	138	2.968	279
2	60	1000	0.01	100	221	221	5.415	481
3	80	1000	0.01	100	288	285	9.852	776
4	100	1000	0.01	100	346	341	14.252	1166
5	200	1000	0.01	100	680	679	59.862	3457
6	400	1000	0.01	100	1344	1339	257.981	7717

Tabela 5: Solução obtida com o algoritmo genético com com estratégia adaptativa alternativa *Manutenção de diversidade* para cada configuração e instância do problema.

Apêndice D Resultados esperados

Abaixo está uma tabela que indica o intervalo no qual está a solução ótima para cada instância do problema.

$\|x\|$	MAX-KQBF
20	[80,151]
40	[275,429]
60	[446, 576]
80	[729,1000]
100	[851, 1539]
200	[3597, 5826]
400	[10846, 16625]

Tabela 6: Tabela com os intervalos de resultado esperados para cada instância.