PROBLEMAS DE OTIMIZAÇÃO

Maximum Edge Subgraph Problem

Carolina Pimenta 2015015938 // Carolina Veloso 2021140780

Índice

INTRODUÇÃO

Este trabalho consiste em conceber, implementar e testar métodos de otimização que encontrem soluções de boa qualidade para diferentes instâncias relativamente ao *Maximum Edge Subgraph Problem*.

O *Maximum Edge Subgraph Problem* é um problema de otimização que envolve encontrar o subconjunto máximo de arestas de um grafo que satisfaz algumas restrições. O objetivo é encontrar o subconjunto de arestas que maximize alguma medida de interesse, como o tamanho ou o peso total das arestas.

Através dos ficheiros disponibilizados pelos docentes, iremos submetê-los a diferentes algoritmos de otimização, maximização.

1.ALGORITMOS DE OTIMIZAÇÃO

Neste trabalho irão ser abordados 3 algoritmos de otimização:

- Algoritmo de pesquisa local – trepa colinas;
- Algoritmo evolutivo;
- Algoritmo híbrido – junção do trepa colinas com o evolutivo.

1.1 Algoritmo de pesquisa local – trepa colinas

O algoritmo trepa colinas (Hill Climbing) é um algoritmo de pesquisa local que tenta encontrar a solução ótima de um problema por meio de tentativa e erro. Ele começa com uma solução inicial e, em seguida, tenta melhorá-la, um passo de cada vez, até que não seja mais possível fazê-lo.

Este é baseado no princípio de que, se uma solução é boa, então soluções ligeiramente melhores devem ser ainda melhores. Portanto, a cada passo, o algoritmo gera uma lista de soluções ligeiramente melhores e escolhe a melhor delas como a próxima solução. Isto é repetido até que não seja mais possível encontrar soluções melhores.

O algoritmo é simples de implementar e pode ser eficiente para encontrar soluções ótimas em problemas com espaços de busca pequenos e estruturados. No entanto, ele tem a desvantagem de poder ficar preso em ótimos locais ou em ótimos globais que não são a solução ótima real. Além disso, ele pode ser muito lento para problemas com espaços de pesquisas grandes e não estruturados.

1.1 Algoritmo evolutivo

Um algoritmo evolutivo é um tipo de algoritmo que se baseia nos princípios da seleção natural para resolver problemas. Algoritmos evolutivos utilizam técnicas da biologia evolutiva, como reprodução e mutação, para gerar e avaliar uma série de soluções possíveis para um determinado problema. Estes

Algoritmos são frequentemente usados para resolver problemas complexos para os quais não existe uma solução ótima conhecida.

Para implementar um algoritmo evolutivo, é criada uma população inicial de soluções possíveis e cada uma é avaliada para determinar qual é a melhor. Em seguida, as soluções mais aptas são selecionadas para se reproduzir e são combinadas para criar soluções. Esses processos de seleção e reprodução são repetidos várias vezes, permitindo que a população evolua para soluções cada vez melhores para o problema em questão.

1.2 Algoritmo híbrido – evolutivo e trepa colinas

Um algoritmo híbrido é um tipo de algoritmo que combina duas ou mais técnicas de solução de problemas diferentes para resolver um problema de otimização. No caso do algoritmo híbrido que combina o algoritmo trepa_colinas com o algoritmo evolutivo, o algoritmo combinaria as técnicas de otimização desses dois algoritmos para encontrar uma solução para o problema em questão.

Por exemplo, o algoritmo híbrido poderia começar por usar o algoritmo trepa_colinas para encontrar um ponto de partida na superfície de custo, e em seguida, usar o algoritmo evolutivo para fazer pequenas alterações na solução a partir desse ponto inicial, a fim de melhorar ainda mais a solução. Este tipo de algoritmo híbrido pode ser muito eficaz para resolver problemas de otimização complexos que não podem ser resolvidos por um algoritmo de otimização individual.

2. ESTUDO EXPERIMENTAL

2.1 Trepa Colinas

2.1.1 Funções mais relevantes

gera_sol_inicial - A solução inicial é representada por um vetor de inteiros chamado "sol", que deve ter pelo menos "v" elementos.
A função começa inicializando todos os elementos do vetor "sol" como zero. Em seguida, sorteia aleatoriamente 8 posições no vetor e atribui o valor 1 a elas. Isso é feito pelo loop "for (i = 0; i < 8; i++)".

A função "random_l_h(l, h)" é usada para sortear um número inteiro aleatório entre "l" e "h", inclusive. O loop "do...while" garante que uma posição já sorteada não é sorteada novamente.

Portanto, ao final da execução desta função, o vetor "sol" terá 8 elementos com o valor 1 e os demais elementos terão o valor 0.

calcula_fit - Este código calcula a "fitness" (aptidão) de uma determinada solução para um problema. A solução é representada por um vetor de inteiros chamado "a", que deve ter pelo menos "vert" elementos. Existe também um segundo parâmetro, "mat", que é a matriz de adjacência que representa um grafo.

A função começa a variável "total" como zero. Em seguida, percorre cada elemento "i" do vetor "a" e verifica se ele é igual a 1. Se for, então percorre os elementos "j" do vetor "a" que vêm depois de "i" (ou seja, j = i+1, i+2, etc.) e verifica se o elemento "j" também é igual a 1 e se há uma aresta entre os vértices "i" e "j" no grafo (isso é verificado pela matriz de adjacência "mat"). Se ambas as condições forem verdadeiras, então incrementa "total" em 1.

Ao final da execução desta função, a variável "total" terá como valor o número de arestas presentes no grafo que são conectadas a pelo menos um vértice que está presente na solução (isto é, cujo índice está marcado com 1 no vetor "a").

gera_vizinho - Esta função gera uma solução vizinha para uma solução dada. Ele faz isso criando uma cópia da solução dada, selecionando duas posições aleatórias na cópia e trocando os valores nessas posições se um dos valores for 0 e o outro for 1.

A função recebe três argumentos:

- a: uma matriz que representa a solução atual
- b: uma matriz que armazenará a solução vizinha gerada
- n: o tamanho das matrizes a e b

A função primeiro copia a solução atual em a para a solução vizinha b usando um loop for. Em seguida, seleciona duas posições aleatórias em b usando a função random_l_h e troca os valores nessas posições se um dos valores for 0 e o outro for 1.

Max Best Fitness (MBF) é uma expressão comumente usada em algoritmos genéticos e em outras áreas da computação para se referir ao melhor fitness (desempenho) que foi alcançado por um indivíduo ou por uma população de indivíduos em algum momento durante um processo de otimização. No contexto de algoritmos genéticos, o fitness de um indivíduo é uma medida da qualidade de sua solução para um determinado problema. O MBF é usado como uma forma de manter o registo do melhor desempenho alcançado até o momento durante o processo de otimização, para que possa ser comparado com o desempenho atual da população e para avaliar o progresso que está sendo feito.

2.1.2 RESULTADOS

Trepa Colina FIRST CHOICE com vizinhança 1 e aceitando solucoes com custo igual - 30 RUNS

NUM VERT		100 it	1000 it	5000 it	10000 it	K	BEST
28	Melhor	20	20	20	20	8	21
	MBF	19.799999	20.0	20.0	20.0		
64	Melhor	15	15	15	15	7	16
	MBF	14.633333	15.0	15.0	15.0		
70	Melhor	109	112	112	112	16	112
	MBF	105.833336	111.666664	112.0	112.0		
200	Melhor	34	79	79	79	14	79
	MBF	29.566668	71.733330	77.266670	75.233330		
500	Melhor	34	85	98	98	15	98
	MBF	20.1000	67.066666	88.433334	90.566666		

Nos ficheiros mais pequenos (file1.txt e file2.txt) conseguimos perceber que o TREPA COLINAS lidava bastante bem com o problema e em todos os conjuntos de iterações apesar de o melhor valor não ter sido atingido, tivemos sempre valores iguais e muito próximos da best solutions.

Nos outros ficheiros (file3.txt, file4.txt e file5.txt) apesar de os valores não serem sempre iguais em todas as iterações, conseguimos sempre pelo menos duas vezes obter a melhor solução para cada ficheiro, respetivamente.

Trepa Colina FIRST CHOICE com vizinhança 2 e aceitando solucoes com custo igual - 30 RUNS

NUM VERT		100 it	1000 it	5000 it	10000 it	K	BEST
28	Melhor	20	20	20	20	8	21
	MBF	19.166666	19.966667	20.0	20.0		
64	Melhor	15	15	15	15	7	16
	MBF	13.833333	15.0	15.0	15.0		
70	Melhor	106	112	112	112	16	112
	MBF	104.033333	108.866669	111.866669	112.0		
200	Melhor	42	72	76	79	14	79
	MBF	29.266666	58.50	68.599998	10.400002		
500	Melhor	42	74	85	87	15	98
	MBF	21.766666	57.366665	77.133331	80.50		

Nos ficheiros mais pequenos (file1.txt e file2.txt) conseguimos perceber que o TREPA COLINAS lidava bastante bem com o problema e em todos os conjuntos de iterações apesar de o melhor valor não ter sido atingido, tivemos sempre valores iguais e muito próximos da best solutions.

Nos ficheiros (file3.txt e file4.txt) conseguimos obter a best solution pelo menos uma vez. Já no último ficheiro (file5.txt) não obtivemos os melhores resultados, nunca conseguimos chegar ao best solution e os valores também nunca dão o mesmo apesar das mudanças de iterações.

2.2 Algoritmo evolutivo

2.2.1 Funções mais relevantes

eval_individual - Esta função avalia a aptidão de um indivíduo.

O valor de aptidão é calculado da seguinte maneira: a função itera sobre cada elemento sol[i] do array e, se ele for igual a 0, conta o número de conexões (especificadas pelo valor 1 no array mat) entre o elemento i e outros elementos que também são iguais a 0. A contagem resultante é armazenada na variável total.

Se total for igual a 0, significa que não há conexões entre os elementos que são iguais a 0 e a função define o valor apontado por v como 1 e retorna o número de elementos iguais a 0 no array sol, que é armazenado na variável zeros.

Caso contrário, se total não for 0, a função define o valor apontado por v como 0 e retorna o valor negativo de total.

Eval_individual_reparado1 - Esta função é similar à anterior, mas possui alguma lógica adicional para reparar um indivíduo que tem conexões entre os seus elementos iguais a 0.

Se o total não for igual a 0, isso significa que há conexões entre elementos iguais a 0 no array sol e a função entra num loop. Dentro do loop, ela seleciona aleatoriamente um elemento do array sol que é igual a 0 e altera o seu valor para 1. Em seguida, ela recalcula o total verificando as conexões entre elementos iguais a 0 no array sol modificado. O loop continua até que não haja mais conexões entre elementos iguais a 0.

Após o loop terminar, a função conta o número de elementos iguais a 0 no array sol reparado, armazena o resultado na variável zeros, define o valor apontado por v como 1 e retorna zeros.

tournament - Essa função realiza um torneio de seleção de pais em uma população de indivíduos, representada pelo array pop de estruturas chrom, com tamanho d.popsize, para gerar uma nova população de pais, representada pelo array parents de estruturas chrom, também com tamanho d.popsize. O contexto do problema é especificado pela estrutura info d.

A função itera sobre o array parents e, para cada posição, gera dois índices aleatórios x1 e x2 no intervalo de 0 a d.popsize - 1 (inclusive). Ela

então seleciona o indivíduo da população original (pop) com o maior valor de fitness e o copia para a população de pais (parents). O valor de fitness é um medidor da qualidade da solução que um indivíduo representa e é armazenado no campo fitness da estrutura chrom.

O torneio de seleção é um tipo de seleção de pais em que os indivíduos da população original são divididos em grupos e, dentro de cada grupo, dois indivíduos são selecionados aleatoriamente para competir entre si. O indivíduo vencedor é selecionado para fazer parte da próxima geração e o perdedor é descartado. O torneio de seleção é um tipo de seleção de pressão de sobrevivência, pois os indivíduos mais aptos são mais propensos a sobreviver e passar seus genes para a próxima geração. Ele é usado com frequência em algoritmos genéticos porque é um método rápido e eficiente de seleção de pais. No entanto, ele também pode ser menos preciso do que outros métodos de seleção, como a roleta, pois a escolha do indivíduo vencedor é aleatória e, portanto, pode não refletir a aptidão real dos indivíduos.

tournament_geral - A função tournament_geral é uma variante da função tournament, que seleciona os indivíduos que irão participar do processo de reprodução (crossover e mutação) de uma população de indivíduos. A diferença entre as duas funções é que, enquanto tournament seleciona apenas dois indivíduos para participar de um torneio e escolhe o que tiver maior aptidão para ser um dos pais, tournament_geral seleciona um conjunto de tsize indivíduos para participar de um torneio e escolhe o que tiver a maior aptidão entre eles para ser um dos pais.

A função começa declarando um ponteiro para um vetor de inteiros chamado pos, que será utilizado para armazenar as posições dos indivíduos selecionados para participar de cada torneio. Em seguida, um laço for é utilizado para realizar popsize torneios. Dentro do laço, outro laço for é utilizado para selecionar tsize indivíduos diferentes para participar do torneio. Para cada indivíduo, uma posição é sorteada aleatoriamente na população e, em seguida, é verificado se essa posição já foi escolhida anteriormente. Se sim, o laço sorteia outra posição até encontrar uma que ainda não tenha sido escolhida. Quando todos os indivíduos para o torneio são selecionados, a posição do indivíduo com maior aptidão é armazenada em best e o indivíduo com essa posição é escolhido como um dos pais.

2.2.1.1 Operadores genéticos

Recombinação com um ponto de corte - Esta função realiza o crossover em uma população de indivíduos selecionados, representados pelo array parents de estruturas chrom, e armazena os descendentes resultantes no array offspring, ambos com um tamanho d.popsize. O contexto do problema é especificado pela estrutura info d.

A função itera sobre o array parents em pares de dois e, em cada par, gera um valor aleatório entre 0 e 1 usando a função rand_01(). Se o valor gerado for menor que a probabilidade de crossover d.pr, ela realiza um crossover de um ponto no par selecionando um ponto de crossover aleatório point entre 0 e d.numGenes-1 e trocando o material genético (o campo p) dos dois indivíduos do lado direito do ponto de crossover. Se o valor gerado for maior ou igual a d.pr, ela simplesmente copia os dois indivíduos do para o array offspring.

Essa função é comumente usada em algoritmos genéticos como uma forma de produzir novas gerações de indivíduos a partir de indivíduos selecionados da geração anterior. Ela funciona selecionando dois indivíduos da população e combinando algumas de suas características (representadas pelo material genético) para produzir dois novos indivíduos. Isso pode ajudar a manter a diversidade genética da população e a produzir indivíduos com características de ambos os pais.

Recombinação com dois pontos de corte - Esta função realiza o crossover de dois pontos numa população de indivíduos selecionados, representados pelo array parents de estruturas chrom, e armazena os descendentes resultantes no array offspring, ambos com um tamanho d.popsize. O contexto do problema é especificado pela estrutura info d.

A função itera sobre o array parents em pares de dois e, em cada par, gera um valor aleatório entre 0 e 1 usando a função rand_01(). Se o valor gerado for menor que a probabilidade de crossover d.pr, ela realiza um crossover de dois pontos no par selecionando dois pontos de crossover aleatórios point1 e point2 entre 0 e d.numGenes-1 de modo que point1 seja menor que point2. Em seguida, ela troca o material genético (o campo p) dos dois indivíduos entre os dois pontos de crossover. Se o valor gerado for maior ou igual a d.pr, ela simplesmente copia os dois indivíduos do par para o array offspring.

Essa função é semelhante à função de crossover de um ponto, mas permite a troca de mais material genético entre os dois indivíduos. Isso pode potencialmente produzir descendentes com uma maior variedade

de características, mas também tem um maior risco de introduzir traços indesejáveis na população.

recombinacao_uniforme - Essa função realiza o crossover uniforme numa população de indivíduos selecionados, representados pelo array parents de estruturas chrom, e armazena os descendentes resultantes no array offspring, ambos com um tamanho d.popsize. O contexto do problema é especificado pela estrutura info d.

A função itera sobre o array parents em pares de dois e, em cada par, gera um valor aleatório entre 0 e 1 usando a função rand_01(). Se o valor gerado for menor que a probabilidade de crossover d.pr, ela realiza o crossover uniforme no par. Para cada gene (o campo p) do par, ela gera um novo valor aleatório usando a função flip(). Se o valor gerado for 1, ela copia o gene do primeiro indivíduo do par para o primeiro descendente e o gene do segundo indivíduo para o segundo descendente. Se o valor gerado for 0, ela copia o gene do segundo indivíduo para o primeiro descendente e o gene do primeiro indivíduo para o segundo descendente. Se o valor gerado pelo rand_01() for maior ou igual a d.pr, ela simplesmente copia os dois indivíduos do par para o array offspring.

O crossover uniforme funciona selecionando aleatoriamente os genes dos indivíduos pais para serem incluídos nos descendentes. Isso pode produzir indivíduos com uma ampla variedade de características, mas também pode levar a resultados imprevisíveis.

Mutação binária - Essa função realiza a mutação binária em uma população de indivíduos, representada pelo array offspring de estruturas chrom, com tamanho d.popsize. O contexto do problema é especificado pela estrutura info d.

A função itera sobre o array offspring e, para cada indivíduo e cada gene (o campo p), gera um valor aleatório entre 0 e 1 usando a função rand_01(). Se o valor gerado for menor que a probabilidade de mutação d.pm, a função inverte o valor do gene, trocando 0 por 1 e 1 por 0. Caso contrário, ela não faz nada.

A mutação binária é um tipo de operação de mutação que modifica aleatoriamente os valores de um ou mais genes de um indivíduo. No caso da mutação binária, isso é feito inverter o valor dos genes, mas outras formas de mutação também são possíveis, como alterar o valor do gene

para outro aleatório ou adicionar ou remover genes inteiros. A mutação é um mecanismo importante em algoritmos genéticos, pois pode introduzir novas características na população e evitar que ela fique presa em soluções locais ótimas. No entanto, a mutação também pode introduzir características prejudiciais à população e, por isso, é necessário ajustar cuidadosamente a probabilidade de mutação para garantir um bom equilíbrio entre a exploração de novas soluções e a preservação de características úteis.

mutacao_por_troca - Essa função realiza a mutação por troca numa população de indivíduos, representada pelo array offspring de estruturas chrom, com tamanho d.popsize. O contexto do problema é especificado pela estrutura info d.

A função itera sobre o array offspring e, para cada indivíduo, gera um valor aleatório entre 0 e 1 usando a função rand_01(). Se o valor gerado for menor que a probabilidade de mutação d.pm, a função escolhe duas posições aleatórias pos1 e pos2 no vetor de genes (o campo p) do indivíduo. A pos1 é escolhida aleatoriamente entre todas as posições do vetor que contenham o valor 0 e a pos2 é escolhida aleatoriamente entre todas as posições do vetor que contenham o valor 1. Em seguida, ela troca o valor do gene na posição pos1 pelo valor do gene na posição pos2. Caso contrário, ela não faz nada.

A mutação por troca é um tipo de operação de mutação que modifica aleatoriamente os valores de um ou mais gene de um indivíduo trocando-os por outros valores. No caso da mutação por troca, isso é feito trocando aleatoriamente um valor 0 por um valor 1 e vice-versa, mas outras formas de mutação também são possíveis, como alterar o valor do gene para outro aleatório ou adicionar ou remover genes inteiros. A mutação é um mecanismo importante em algoritmos genéticos, pois pode introduzir novas características na população e evitar que ela fique presa em soluções locais ótimas. No entanto, a mutação também pode introduzir características prejudiciais à população e, por isso, é necessário ajustar cuidadosamente a probabilidade de mutação para garantir um bom equilíbrio entre a exploração de novas soluções e a preservação de características úteis.

2.2.2 RESULTADOS

file1.txt - 28 vértices											
		Algoritmo base (Recombinação de 1 ponto de corte + Mutação binária + Penalização cega)		Recombinação de 1 ponto de corte + Mutação binária + Reparação1 (aleatória)		Recombinação de 1 ponto de corte + Mutação por troca + Reparação1		Recombinação com 2 pontos de corte + mutação binária + reparação1		Recombinação uniforme + mutação binária + reparação1	
Parâmetros Fixos	Parâmetros a variar	Best	MBF	Best	MBF	Best	MBF	Best	MBF	Best	MBF
pop = 100 (ger = 2500)	pr = 0.3	7	7	7	7	7	7	7	7	7	7
pm = 0.01	pr = 0.5	7	7	7	7	7	7	7	7	7	7
tsize = 2	pr = 0.7	7	7	7	7	7	7	7	7	7	7
pop = 100	pm = 0.0	6	4.3	7	5.2	6	4.4	6	4.6	7	5.9
ger = 2500	pm = 0.001	7	6.6	7	7	7	6.5	7	7	7	7
pr = 0.7	pm = 0.01	7	7	7	7	7	6.6	7	7	7	7
tsize = 2	pm = 0.05	7	6.6	7	6.966667	7	7	7	6.666667	7	6.866667
pr = 0.7	pop = 10 (ger = 25K)	7	7	7	7	7	7	7	7	7	7
pm = melhor valor obtido	pop = 50 (ger = 5K)	7	7	7	7	7	7	7	7	7	7
tsize = 2	pop = 100 (ger = 2.5K)	7	7	7	7	7	7	7	7	7	7

		Algoritmo base		Com Reparação1 (aleatória)		Com Mutação por troca + Reparação1		Com Recombinação com 2 pontos de corte + Mutação Binária + Reparação1		Com Recombinação Uniforme + Mutação Binária + Reparação1	
Parâmetros Fixos	Parâmetros a variar	Best	MBF	Best	MBF	Best	MBF	Best	MBF	Best	MBF
pop = 100 (gen = 2500)	tsize = 3	7	7	7	7	7	6.5	7	7	7	7
pr = 0.7	tsize = 10	7	6.7	7	7	6	7	7	7	7	7
pm = 0.001	tsize = 50	7	6.7	7	7	7	6.5	7	7	7	7

Ao observar a tabela podemos concluir, que mesmo alterando parâmetros e métodos, os resultados são sempre semelhantes. O mesmo acontece com todos os ficheiros, ainda que por vezes existam resultados negativos. Posto isto, concluímos que o número final de 0's é quase sempre 7, no file1.txt, que tem 28 vértices, o que significa que existem no máximo 7 conexões.

2.3 Algoritmo híbrido

2.3.1 Resultados

		Algoritmo base híbrido i)		Algoritmo base híbrido ii)		Algoritmo base híbrido iii)		
Parâmetros Fixos	Parâmetros a variar	Best	MBF	Best	MBF	Best	MBF	
pop = 100 (gen = 2.5k)	PROBGERAVIZ 1	39	36.0333	40	37.9002	38	35.1667	i) a mutação1 e a eval_individual
pr = 0.7								ii) a mutação1 e eval_reparado1
pm = 0.001	PROBGERAVIZ. 0.8	38	36.0333	40	37.7999	36	34.0666	iii) a mutação2 e a eval_reparado1
tsize = 2								

Como podemos verificar, o algoritmo híbrido obteve melhores resultados ainda que não muito diferentes. Neste file, file5.txt, os valores aumentaram, mas nos files onde os vértices são menores, os valores são iguais tanto no evolutivo como no híbrido.

CONCLUSÃO

Existem algumas conclusões que podemos tirar depois de analisadas as tabelas e os resultados das mesmas.

Uma das conclusões é que o trepa colinas conseguiu os melhores resultados em comparação com o ficheiro fornecido pelos professores.

Uma melhoria nas soluções foi aquando da realização do algoritmo híbrido. Este conseguiu melhorar algumas experiências realizadas anteriormente chegando perto dos valores ótimos.

Para que fossem atingidos os valores ótimos, usando o algoritmo híbrido seria necessário aumentar o número de iterações a realizar.