Departamento de Engenharia Informática e Sistemas (DEIS)

Introdução à Inteligência Artificial 2023/2024

Trabalho Prático nº 2

Problema de Otimização

André Dias – a2021140917

João Pascoal – a2017009420

ÍNDICE

Introdução 3
Algoritmo de pesquisa local 4
Inicialização da matriz 4
Algoritmo 4
Funções mais importantes 4
Resultados 5
Algoritmo Evolutivo 7
Funções mais importantes 7
Resultados 8

Introdução

Neste relatório vamos analisar o Subconjunto de custo mínimo.

O desafio do "Subconjunto de Custo Mínimo" em grafos não direcionados busca identificar um grupo de vértices de tamanho k, interconectados por arestas, minimizando o custo total interno. Esta busca pela eficiência máxima requer métodos de otimização capazes de encontrar soluções de alta qualidade para diferentes cenários do problema. Este estudo visa conceber, implementar e testar tais métodos, oferecendo soluções de qualidade excecional para este desafio complexo da teoria dos grafos.

Algoritmo de pesquisa local

Inicialização da matriz

- Foi criada uma matriz V * V contendo os valores que foram disponibilizados nos ficheiros de teste.

Algoritmo

- Um algoritmo de pesquisa local, em termos genéricos, é um algoritmo que recebe um problema como entrada e retorna uma solução válida para o mesmo, depois de resolver um certo número de possíveis soluções.
- Neste trabalho o algoritmo que utilizámos foi o trepa-colinas. O trepa-colinas parte de um estado inicial aleatório, define um critério de vizinhança, avalia todos os seus vizinhos e vê qual o melhor, se houver algum melhor ele aceita o mesmo. De seguida é iniciada uma nova iteração e repete esse procedimento até chegar a um estado em que todos os vizinhos sejam de qualidade inferior para o contexto do problema apresentado.

Funções mais importantes

"**initDados**" nesta função é onde vamos ler os ficheiros de texto e preencher a matriz com os respetivos valores, além de atribuir o custo na posição correspondente.

"**geraSolInicial**" esta função recebe um array onde vamos guardar as soluções, o número de vértices e o valor de "k", a função vai depois preencher esse mesmo array.

"**calculaFit**" esta função vai calcular o valor de ajuste dado uma solução representada pelo array "a". Primeiro inicia-se com o valor 0 o array "con" que vai servir para verificar se um vértice está conectado a outro vértice. De seguida percorremos todos os vértices do grafo e verificamos o vértice "i" está incluído na solução e depois verificamos as suas conexões. Se o vértice "j" também estiver incluído na mesma solução("a[j] == 1") e se houver uma aresta entre i e j ("mat[i * vert +j] !=0") então ambos os vértices são marcados como conectados("con[i] = 1" e "con[j] = 1") e o custo da aresta é adicionado á

variável total. A função vai devolvar o valor de total/2 para evitar a contagem duplicada de arestas. Nesta função é ainda ou se vai implementar as penalizações e os mecanismos de reparação.

"**geraVizinho**" é gerado um ponto p1 aleatório entre 0 e n-1, e de seguida acontece o mesmo com um ponto p2. O p2 vai ser gerado num ciclo enquanto for igual a p1, ou seja, vai garantir que sejam índices diferentes. Quando se der essa situação é feita uma substituição dos valores dos respetivos índices, trocando p1 com p2 gerando assim um novo vizinho.

Resultados

Trepa-Colinas com Vizinhança 1

NUM VERT		100 it	1000 it	10000 it	100000 it
file1.txt	Melhor	57	45	45	45
	MBF	74.699997	57.299999	61.599998	57.200001
file2.txt	Melhor	23	12	8	14
	MBF	64.400002	25.100000	24.500000	20.500000
file3.txt	Melhor	459	346	336	336
	MBF	497.899994	379.200012	390.899994	363.100006
file4.txt	Melhor	1000000	27	8	8
	MBF	0.000000	28.000000	9.800000	9.300000
file5.txt	Melhor	1000000	1000000	22	9
	MBF	0.000000	0.000000	2.200000	10.400000

Podemos observar que o trepa-colinas lida bem com o problema e em todos os ficheiros conseguimos atingir a solução ideal, ou então chegar relativamente perto, possivelmente até atingindo com mais iterações.

Trepa-Colinas com Vizinhança 1 e aceitando soluções de custo igual

NUM VERT		100 it	1000 it	5000 it	10000 it
file1.txt	Melhor	56	50	45	45
	MBF	78.000000	56.299999	55.099998	58.299999
file2.txt	Melhor	36	14	15	14
	MBF	66.099998	25.900000	21.799999	21.799999
file3.txt	Melhor	438	336	336	336
	MBF	498.399994	374.799988	381.299988	363.399994
file4.txt	Melhor	98	14	8	8
	MBF	32.400002	28.500000	9.500000	9.500000
file5.txt	Melhor	1000000	1000000	25	9
	MBF	0.000000	0.000000	15.500000	10.400000

Nesta experiência fizemos com que o trepa-colinas aceita-se soluções de custo igual, mas não se notou nenhuma diferença significativa, portanto podemos concluir que não alterou em muito os resultados.

NUM VERT		100 iterações Prob = 0.01	5000 iterações Prob = 0.01	100 iterações Prob = 0.0005	5000 iterações Prob = 0.0005
file1.txt	Melhor	56	45	50	48
	MBF	75.500000	54.700001	71.599998	62.000000
file2.txt	Melhor	40	15	24	13
	MBF	61.799999	25.100000	64.699997	20.900000
file3.txt	Melhor	448,0	336,0	441,0	336,0
	MBF	489.899994	383.000000	490.100006	375.299988
file4.txt	Melhor	72,0	9,0	82,0	8,0
	MBF	16.100000	12.000000	30.600000	12.200000
file5.txt	Melhor	1000000	1000000	1000000	48
	MBF	0.000000	0.000000	0.000000	11.900000

De seguida, além de aceitar situações de custo igual transformamos o trepa-colinas num trepa-colinas probabilístico, mas também não houve uma melhoria significativa nos resultados.

Algoritmo Evolutivo

Funções mais importantes

"**initDados**" nesta função é onde vamos ler os ficheiros de texto e preencher a matriz com os respetivos valores, além de atribuir o custo na posição correspondente. Nesta situação os resultados foram todos guardados numa estrutura de dados do tipo "info".

"**geraSolInicial**" esta função recebe um array onde vamos guardar as soluções, o número de vértices e o valor de "k", a função vai depois preencher esse mesmo array.

"**calculaFit**" esta função vai calcular o valor de ajuste dado uma solução representada pelo array "a". Primeiro inicia-se com o valor 0 o array "con" que vai servir para verificar se um vértice está conectado a outro vértice. De seguida percorremos todos os vértices do grafo e verificamos o vértice "i" está incluído na solução e depois verificamos as suas conexões. Se o vértice "j" também estiver incluído na mesma solução("a[j] == 1") e se houver uma aresta entre i e j ("mat[i * vert +j] !=0") então ambos os vértices são marcados como conectados("con[i] = 1" e "con[j] = 1") e o custo da aresta é adicionado á variável total. A função vai devolver o valor de total/2 para evitar a contagem duplicada de arestas. Nesta função é ainda ou se vai implementar as penalizações e os mecanismos de reparação.

"**initPop**" esta função recebe uma estrutura que contém os parâmetros do problema. Vamos ter uma variável "indiv" que é um ponteiro para uma estrutura do tipo "chrom" que representa um individuo na população, depois vamos alocar dinamicamente essa variável para armazenar a população de indivíduos. Depois temos um ciclo sobre cada individuo em que cada individuo vai chamar a função "geralSolInicial" para preencher o seu "gene" com valores binários.

Foram ainda implementados 2 métodos de seleção diferentes (Tournament e Tournament2), 2 operadores de recombinação diferentes (crossover e crossover2) a diferenças entre as recombinações são:

A função crossover utiliza um ponto aleatório enquanto a crossover2 utiliza dois pontos aleatórios, mas garante que sejam diferentes e ordena-os. Além disto temos ainda 3 operadores de mutação diferentes (Mutation, Mutation2 e Mutation_por_troca.

A Mutation inverte independentemente os valores dos genes, a Mutation2 inverte os valores entre uma posição de valor 0 e uma posição de valor1, e a Mutation_por_troca troca os valores entre duas posições.

Resultados

PopSize = 250 numGenerations = 5000 ; OPERADORES USADOS : CROSSOVER + MUTATION					
Prob. do operador pm=1 pr = 1 ; Metodo de seleção Tournament					
Ficheiro		Melhor solução	10 runs	15 runs	20 runs
teste.txt (V = 6; A = 7 ; K = 4)	Melhor	6	6	6	6
	MBF		6.00	6.00	6.00
file1.txt (V = 28; A =210 ; K = 8)	Melhor	45	48.0	45.0	45.0
	MBF		49.799999	48.533333	49.049999
file2.txt (V = 64; A =704 ; K = 7)	Melhor	8	24.0	23.0	21.0
	MBF		24.700001	25.400000	25.299999
file3.txt.txt (V = 70 A = 1855 ; K = 16)	Melhor	336	454.0	434.0	458.0
	MBF		484.700012	490.333344	490.950012
file4.txt.txt (V = 200 A = 1534 ; K = 14)	Melhor	7	32.0	25.0	29.0
	MBF		39.299999	39.933334	38.950001
file5.txt (V = 500 A = 4459 ; K = 15)	Melhor	11	161.0	178.0	178.0
	MBF		233.000000	226.066666	226.149994

Nesta experiência testamos para um número diferente de runs os operadores genéticos crossover(recombinação) e mutation(mutação) com as suas probabilidades ao máximo. Podemos observar que sem penalização ou reparação nos ficheiros maiores os valores ficam longe do desejado.

PopSize = 250 numGenerations = 5000 ; OPERADORES USADOS : CROSSOVER + MUTATION_por_TROCA					
Prob. do operador pm=0.05 pr=0.5 ; Metodo de seleção Tournament					
Ficheiro		Melhor solução	10 runs	15 runs	20 runs
teste.txt (V = 6; A = 7 ; K = 4)	Melhor	6	6	6	6
	MBF		6.00	6.00	6.00
file1.txt (V = 28; A =210 ; K = 8)	Melhor	45	45.0	45.0	45.0
	MBF		45.000000	45.000000	45.000000
file2.txt (V = 64; A =704 ; K = 7)	Melhor	8	8.0	8.0	8.0
	MBF		13.000000	13.133333	12.400000
file3.txt.txt (V = 70 A = 1855 ; K = 16)	Melhor	336	336.0	336.0	336.0
	MBF		349.100006	348.866669	340.750000
file4.txt.txt (V = 200 A = 1534 ; K = 14)	Melhor	7	9.0	9.0	8.0
	MBF		11.500000	12.133333	11.600000
file5.txt (V = 500 A = 4459 ; K = 15)	Melhor	11	46.0	40	58.0
	MBF		135.399994	123.999992	134.399994

Nesta experiência baixámos os valores das probabilidades de recombinação e mutação, e utilizámos a mutação por troca, e podemos ver que os valores especialmente nos ficheiros mais pequenos chegaram aos valores ideais, e no "file4.txt" chegou bastante perto podendo até assumir que chegaria lá com um número mais elevado de runs.

PopSize = 250 numGenerations = 5000 ; OPERADORES USADOS : CROSSOVER2 + MUTATION2					
Prob. do operador pm=0.05 pr=0.5 ; Metodo de seleção Tournament; penalização					
Ficheiro		Melhor solução	10 runs	15 runs	20 runs
teste.txt (V = 6; A = 7 ; K = 4)	Melhor	6	6	6	6
	MBF		6.00	6.00	6.00
file1.txt (V = 28; A =210 ; K = 8)	Melhor	45	45	45	45
	MBF		50.799999	47.00000	45.00000
file2.txt (V = 64; A =704 ; K = 7)	Melhor	8	15.0	11.0	10.0
	MBF		22.200001	22.133333	22.799999
file3.txt.txt (V = 70 A = 1855 ; K = 16)	Melhor	336	336.0	336.0	
	MBF		382.500000	378.933319	378.950012
file4.txt.txt (V = 200 A = 1534 ; K = 14)	Melhor	7	10.0	8.0	8.0
	MBF		212.199997	77.599998	130.949997
file5.txt (V = 500 A = 4459 ; K = 15)	Melhor	11	500.0	500.0	276.0
	MBF		500.000000	500.000000	488.799988

Nesta experiência além de usarmos um método de recombinação e mutação alternativos, utilizámos também a penalização, e embora para os ficheiros mais pequenos os valores sejam próximos do ideal, podemos observar que para o ficheiro “file5.txt” os valores ficaram bastante longes do esperado, só melhorando com 20 runs, mas longe do esperado.

PopSize = 250 numGenerations = 5000 ; OPERADORES USADOS : CROSSOVER2 + MUTATION					
Prob. do operador pm=0.05 pr=0.5 ; Metodo de seleção Tournament2					
Ficheiro		Melhor solução	10 runs	15 runs	20 runs
teste.txt (V = 6; A = 7 ; K = 4)	Melhor	6	6	6	6
	MBF		6.00	6.00	6.00
file1.txt (V = 28; A =210 ; K = 8)	Melhor	45	45	45	45
	MBF		45.00000	45.00000	45.00000
file2.txt (V = 64; A =704 ; K = 7)	Melhor	8	16.0	12.0	12.0
	MBF		21.200001	19.866667	19.933333
file3.txt.txt (V = 70 A = 1855 ; K = 16)	Melhor	336	45.0	432.0	423.0
	MBF		459.399994	460.799988	450.733333
file4.txt.txt (V = 200 A = 1534 ; K = 14)	Melhor	7	25.0	21.0	20.0
	MBF		35.700001	37.133335	34.44234
file5.txt (V = 500 A = 4459 ; K = 15)	Melhor	11	161.0	100.0	93
	MBF		211.800003	194.000000	154.12491

Enquanto que nesta experiência utilizámos um método de seleção alternativo (Tournament2) para verificar o impacto que teria nos resultados, e podemos verificar que os resultados não melhoraram como alguns até pioraram portanto concluído assim que não trás vantagens para a resolução do problema.

Conclusão

Uma conclusão que podemos tirar no algoritmo evolutivo é que baixando as probabilidades de recombinação e de mutação os valores melhoram, assim como quanto maior o número de runs melhores os resultados. Também podemos tirar a conclusão de que a mutação por troca é aquela que gera melhores resultados.

Quanto ao trepa-colinas, podemos observar que teve uma boa eficácia, mas que nem aceitar soluções de custo igual, nem aumentar o número de vizinhanças, nem transformar em trepa-colinas probabilístico afeta muito os resultados, mas o numero de iterações afeta sim proporcionalmente o resultado.