Relatório - Trabalho Prático nº2 - Problema de Otimização

Departamento de Engenharia Informática e de Sistemas (DEIS)

Bruno Teixeira(2019100036), Rafael Ribeiro(2019131989)

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

Neste relatório vamos analisar o Problema da Diversidade Máxima de Grupos.

Este problema tem por objetivo particionar um conjunto de **(M)** elementos em **(G)** subconjuntos menores, chamados grupos. Cada um destes subconjuntos deve ter o mesmo número de elementos **(N)**.
O objetivo da otimização é encontrar uma divisão que maximize a diversidade dos elementos pertencentes ao mesmo conjunto.
A diversidade de um subconjunto **(Gi)** com **(N)** elementos é igual à soma das distâncias entre todos os pares de elementos que o constituem.

$$div(G_1) = \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} dist(e_i, e_j)$$

Figura 1.1: Formula

Capítulo 2

Desenvolvimento

Neste capítulo são apresentadas todas as propostas e alterações implementadas, assim como estratégias criadas para a simulação. São descritas algumas experiências realizadas e conclusões retiradas a partir das mesmas.

2.1 Inicialização da Matriz

Foi criada uma matriz em espelho **M** * **M** contendo os valores das distâncias que foram disponibilizadas nos ficheiros de teste.

2.2 Algoritmo de Pesquisa Local

Um algoritmo de Pesquisa Local em termos gerais é um algoritmo que recebe um problema como entrada e retorna uma solução válida para o mesmo, depois de resolver um certo número de possiveis soluções.

O algoritmo de Pesquisa Local usado neste trabalho 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 é o melhor. Se o melhor vizinho é melhor do que o atual, aceita o mesmo, uma vez que estamos num problema de maximização. É iniciada uma nova iteração a partir do melhor atual e faz este procedimento até chegar a um estado em que todos os vizinhos têm qualidade inferior ao atual.

2.2.1 Funções mais relevantes

geraSolInicial

Usámos um array auxiliar com o tamanho de **G** que é responsável por contar quantas vezes já foram inseridos um certo número de subconjuntos. Usámos também um array que guarda a solução, sendo este chamado de **solucao**.

É gerado um número aleatório entre **0** e o **número de subconjuntos**.Verificamos no array **auxiliar[número aleatório]** se o valor é menor do que **N**, caso seja verdade, significa que ainda podemos inserir elementos daquele subconjunto.
É então incrementada uma unidade na posição do número aleatório no array auxiliar e inserido no array de solução o número aleatório.
Esta função termina quando o array auxiliar tem todas as suas posições preenchidas com o valor igual a **N**.

calculaFit

Usámos um array auxiliar com o tamanho de **N** que é responsável por guardar os indices das distâncias de cada subconjunto.
Esse array serve como indice para a matriz criada anteriormente para que seja feita a soma dos valores das distâncias. O auxiliar é depois limpo para receber os indices do próximo subconjunto.

Esta função termina quando o calculo das distâncias de todos os subconjuntos estiverem somadas, retornado assim a variavel **soma**.

geraVizinho

É gerado um **p_ponto** aleatorio entre **0** e **M-1** e de seguida é feito o mesmo para um **s_ponto**. O **s_ponto** é continuamente gerado enquanto for igual ao **p_ponto** garantido que ambos sejam indices diferentes. Quando ambos são diferentes, é feita uma substituição dos valores dos respetivos indices, trocando o **p_ponto** pelo **s_ponto**, gerando assim um novo vizinho.

2.2.2 Resultados

Trepa-colinas com vizinhança 1 - 15 rondas						
Ficheiro		100 iterações	1000 iterações	2500 iterações	5000 itrações	10000 iterações
n010.txt	Melhor	1228	1228	1228	1228	1228
	MBF	1228.000000	1228.000000	1228.000000	1228.000000	1228.000000
n012.txt	Melhor	1000	1000	1000	1000	1000
	MBF	956.799988	988.599976	982.533325	994.666687	983.599976
n030.txt	Melhor	4818	5110	5156	5156	5168
	MBF	4587.799805	4986.200195	5052.533203	5072.200195	5072.200195
n060.txt	Melhor	16242	18178	18387	18369	18468
	MBF	15837.933594	17625.800781	18022.066406	18127.066406	18157.400391
n120.txt	Melhor	37468	42049	43418	44428	44564
	MBF	36676.867188	41387.132812	42842.132812	43656.933594	44226.265625
n240.txt	Melhor	122440	136484	139840	143205	145619
	MBF	120801.531250	134562.937500	139112.203125	141890.062500	144836.593750

Figura 2.1: Trepa Colinas - 1

Nos ficheiros mais pequenos (n010.txt e n012.txt) conseguimos perceber que o **Trepa Colinas** lidava bastante bem com o problema e em todos os conjuntos de iterações o melhor valor foi sempre atingido.

Nos restantes ficheiros, é percetivel que quanto maior for o número de iterações mais próximo o algoritmo fica de chegar ao melhor valor.No entanto os ficheiros n120.txt e n240.txt precisariam de mais iterações para chegarem a um melhor valor, uma vez que os resultados ainda se encontram um pouco longe do ótimo.

Trepa-colinas com vizinhança 1 e aceitando soluções de custo igual - 15 rondas						
Ficheiro		100 iterações	1000 iterações	2500 iterações	5000 iterações	10000 iterações
n010.txt	Melhor	1228	1228	1228	1228	1228
	MBF	1228.000000	1228.000000	1228.000000	1228.000000	1228.000000
n012.txt	Melhor	1000	1000	1000	1000	1000
	MBF	972.466675	984.666687	991.799988	976.466675	996.799988
n030.txt	Melhor	4702	5126	5143	5194	5173
	MBF	4523.200195	4996.066895	5034.399902	5071.866699	5080.066895
n060.txt	Melhor	16160	17725	18295	18221	18362
	MBF	15663.533203	17500.466797	17910.666016	18083.000000	18120.933594
n120.txt	Melhor	37978	42295	43391	44017	44893
	MBF	36482.867188	41529.199219	42894.601562	43604.398438	44308.667969
n240.txt	Melhor	122512	136243	140507	142674	145822
	MBF	120904.132812	134453.796875	139277.000000	142020.671875	144678.531250

Figura 2.2: Trepa Colinas - 2

Nesta experiência fizemos com que o Trepa Colinas aceitasse soluções de custo igual, de modo a percebermos se isso iria ou não influenciar nos resultados. Comparativamente às experiências anteriores não se notou nenhuma diferença em relação aos ficheiros n010.txt e n012.txt
Nas restantes experiências nota-se um aumento pouco significativo, o que faz com que possamos concluir que aceitando soluções de custo igual, não iria alterar em muito os resultados, uma vez que o objetivo é chegar ao melhor valor.

2.3 Algoritmo Evolutivo

Este é um tipo de algoritmo baseado na Teoria da Evolução de Darwin.
O principio básico do mesmo é, partir com um conjunto de soluções aleatórias para um problema, dando a este conjunto o nome de **população**.
A partir da população selecionamos os melhores que irão ser chamados de **progenitores**. Estes progenitores depois podem ou não ser **recombinados** (probabilisticamente) gerando descendentes, descendentes estes que probabilisticamente podem ser **mutados**.

Nestas experiências o torneio escolhido foi sempre com o tamanho de **2**. Para fazermos este torneio, selecionamos duas soluções aleatórias e comparavamos as suas **fitness**, a solução que tivesse maior **fitness** iria ser a solução escolhida para as próximas iterações.
Na recombinação foi usado um ponto de corte aleatório que consistia em dividir duas **soluções pai** em dois e dividir uma parte para um **filho** e a outra parte para outro **filho** como mostra no exemplo em baixo. Depois da recombinação, existia tambem uma probabilidade de haver uma mutação, que alterava apenas um valor numa solução.

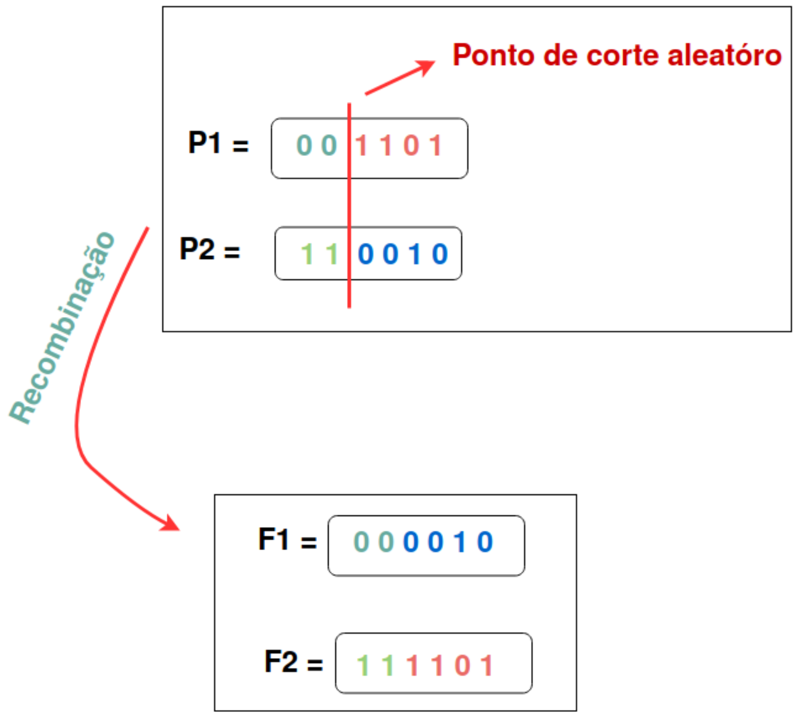

Figura 2.3: Exemplo de Recombinação

2.3.1 Funções mais relevantes

initPop

Esta função baseia-se na **gerasolInicial**, no entanto como estamos a usar várias soluções, usamos uma estrutura para nos auxiliar na construção das mesmas.Ou seja, o algoritmo é exatamente o mesmo, no entanto cada solução é inserida num array que pertence a uma estrutura.

avaliaIndividual_1

Nesta função usamos os mesmos principios da **calculaFit**, uma vez que é responsável por calcular a fitness de uma solução numa população.
Como existe a **recombinacao** e a **mutacao**, há uma probabilidade da solução a ser avaliada ser inválida, logo foi preciso criar uma função chamada **verificaValida** que tem como objetivo validar uma solução e caso esta seja válida o programa prossegue, caso contrário, é marcada como inválida e sai da **avaliaIndividual_1**.

avaliaIndividual_2

Contráriamente à **avaliaIndividual_1**, a **avaliaIndividual_2** aplica uma penalização a todas as soluções que forem inválidas usando a função **verificaValidaPen**. Caso a solução seja inválida, é então aplicada uma penalização e a mesma é reparada usando a função **reparacao**.
O valor a ser penalizado é a soma de todo os subconjuntos com **N** elementos errados. Por exemplo, tendo o **M = 6, G = 2**, sabemos que o **N = 3**. Então, se a solução for a seguinte **S = [0 0 0 1 0 1]**, a sua penalidade é **6** uma vez que o **subconjunto 0** tem 4 valores e o **subconjunto 1** tem 2 valores, sendo que para a solução ser válida, ambos os subconjuntos deviam de ter **3 elementos**.

reparacao

Para esta função usamos alguma aleatoriedade para tentar gerar uma solução nova. Distribuimos os subconjuntos certos para um array e os subconjuntos errados para outro.
Dentro de um ciclo, **geramos um número aleatório**.
Se for igual a 0, vamos ao array dos valores certos. Verificamos se existe algum valor válido no array, e **voltamos gerar um número aleatório** que é um indice usado para aceder a uma certa posição desse array para posteriormente copiar esse valor para a solução. Depois de feita esta cópia, o valor da posição do numero aleatorio no array de certos passa a ser **-1** para sabermos que já foi copiado.
Se o primeiro número aleatorio gerado for igual a 1, fazemos exatamente o mesmo mas com o array dos valores errados.
No fim, temos uma solução válida em que os valores lá contidos foram colocados de maneira aleatória e com o número de subconjuntos correto.

2.3.2 Resultados

		Ficheiro n030.txt			
		Algorítmo base sem penalização		Algorítmo base com penalização e reparação	
Parâmetros fixos	Parâmetros a variar	Melhor	MBF	Melhor	MBF
gerações = 2500	prob. recombinação = 0.3	5028	4857.666504	4964	4786.933105
população = 100	prob. recombinação = 0.5	5007	4838.666504	4628	4464.733398
prob. mutação = 0.01	prob. recombinação = 0.7	4773	4249.399902	4631	4467.866699
gerações = 2500	prob. mutação = 0.0	4199	4040.066650	4577	4443.000000
população = 100	prob. mutação = 0.001	4356	4057.133301	4475	4424.000000
prob. recombinação = 0.7	prob. mutação = 0.5	4311	4130.533203	4476	4375.066895
prob. recombinação = 0.7	população = 20 - gerações = 5000	4033	3859.46653	4198	420.000
prob. recombinação = 0.7	população = 50 - gerações = 7000	4274	4050.733398	4488	4282.866699

Figura 2.4: Algoritmo Evolutivo - n030.txt

Com o **algoritmo base sem penalização**, fixando uma **população** de 100, uma **geração** de 2500 e uma **probabilidade de mutação** de 0.01 variando apenas a **probabilidade de recombinação** conseguimos perceber que quanto maior é a probabilidade de recombinação mais longe o algoritmo fica de atingir a solução ótima.
Com os mesmos valores anteriores, fixando apenas a **probabilidade de recombinação** e variando a **probabilidade de mutação** os resultados tornam-se ainda piores, porque a probabilidade de recombinação fixa é alta e a probabilidade de mutação não favorece os resultados.
Com uma população muito baixa, os resultados foram os piores, mesmo aumentando o número de gerações, o que leva a concluir que uma maior população influencia num bom resultado.

No caso do **algoritmo base com penalização e reparação**, uma vez que este aceita soluções inválidas e posteriormente as repara, havia uma possibilidade de encontrar uma solução ótima, no entanto isso não aconteceu o que fez com que os resultados não tenham sido muito diferentes comparativamente ao primeiro algoritmo.

		Ficheiro n060.txt			
		Algorítmo base sem penalização		Algorítmo base com penalização e reparação	
Parâmetros fixos	Parâmetros a variar	Melhor	MBF	Melhor	MBF
gerações = 2500	prob. recombinação = 0.3	14605	14356.466797	15392	15221.266602
população = 100	prob. recombinação = 0.5	15133	14401.200192	15441	15192.400391
prob. mutação = 0.01	prob. recombinação = 0.7	14710	14388.933594	15330	15154.799805
gerações = 2500	prob. mutação = 0.0	14949	14415.666992	15472	15131.466797
população = 100	prob. mutação = 0.001	14644	14380.266602	15489	15216.000000
prob. recombinação = 0.7	prob. mutação = 0.5	14719	14369.400391	14969	14783.266602
prob. recombinação = 0.7	população = 20 - gerações = 5000	14565	14033.266602	14669	14317.400391
prob. recombinação = 0.7	população = 50 - gerações = 7000	14772	14211.400391	15353	14847.5333203

Figura 2.5: Algoritmo Evolutivo - n060.txt

As conclusões do **algoritmo base sem penalização** nesta tabela, são parecidas à tabela inicial pois mostra pequenas variâncias entre os valores.

No **algoritmo base com penalização e reparação** já se notou um aumento das qualidades das soluções evidenciado na comparação do **MBF** entre os dois algoritmos, o que leva a concluir que a reparação contribuiu para uma melhoria dos resultados.

		Ficheiro n120.txt			
		Algorítmo base sem penalização		Algorítmo base com penalização e reparação	
Parâmetros fixos	Parâmetros a variar	Melhor	MBF	Melhor	MBF
gerações = 2500	prob. recombinação = 0.3	35043	34374.601562	35817	35557.199219
população = 100	prob. recombinação = 0.5	34804	34426	35874	35544.867188
prob. mutação = 0.01	prob. recombinação = 0.7	34872	34413.000000	36018	35587.867188
gerações = 2500	prob. mutação = 0.0	34736	34242.667969	35934	35559.199219
população = 100	prob. mutação = 0.001	34996	34525.066406	36619	35658.066406
prob. recombinação = 0.7	prob. mutação = 0.5	34965	34401.464844	35369	34688.933594
prob. recombinação = 0.7	população = 20 - gerações = 5000	34835	33963.867188	34959	34382.332031
prob. recombinação = 0.7	população = 50 - gerações = 7000	34893	34253.132812	35643	35036.464844

Figura 2.6: Algoritmo Evolutivo - n120.txt

No ficheiro n120.txt usando o **algoritmo base sem penalização** conclui-se mais uma vez que uma menor **probabilidade de recombinação** faz com que os resultados sejam melhores. Como mostra a tabela, sempre que a probabilidade de recombinação aumenta, os valores dos resultados diminiuem.

Novamente o **algoritmo base com penalização e reparação** ajudou a obter resultados melhores não aumentando drásticamente os mesmos. A melhor maneira de perceber isso é comparando o **MBF** dos dois algoritmos vendo que as diferenças não são muito grandes.

		Ficheiro n240.txt			
		Algorítmo base sem penalização		Algorítmo base com penalização e reparação	
Parâmetros fixos	Parâmetros a variar	Melhor	MBF	Melhor	MBF
gerações = 2500	prob. recombinação = 0.3	118997	117124.070312	119894	119356.468750
população = 100	prob. recombinação = 0.5	118358	117336.070312	119946	119456.796875
prob. mutação = 0.01	prob. recombinação = 0.7	118488	117596.734375	119751	119146.335938
gerações = 2500	prob. mutação = 0.0	119130	117420.664062	120429	119735.664062
população = 100	prob. mutação = 0.001	119394	117412.531250	120625	119917.132812
prob. recombinação = 0.7	prob. mutação = 0.5	118592	117144.468750	118266	117202.000000
prob. recombinação = 0.7	população = 20 - gerações = 5000	117731	116656.601562	118697	117408.132812
prob. recombinação = 0.7	população = 50 - gerações = 7000	117601	116901.265625	119756	118496.000000

Figura 2.7: Algoritmo Evolutivo - n240.txt

Nesta tabela, no **algoritmo base sem penalização**, observou-se que usando parâmetros variáveis na mutação comparativamente com os parâmetros variáveis na recombinação, houve um pequeno aumento na qualidade das soluções.

No **algoritmo base com penalização e reparação** notou-se uma ligeira melhoria em relação ao algoritmo sem penalização.

Mais uma vez é possível constatar que uma população pequena gera soluções de baixa qualidade mesmo tendo um número elevado de gerações.

2.4 Algoritmo Hibrido

Este algoritmo combina o Trepa Colinas com o Algoritmo Evolutivo de modo a aproximar-se de uma solução ótima.

Usamos o algoritmo hibrido de duas maneiras diferentes. A primeira foi usada aquando da criação da população inicial, ou seja, em vez de começarmos com uma população aleatória, melhoramos a população inicial com o Trepa Colinas e esta serviu de ponto de partida para o algoritmo evolutivo.
A segunda maneira foi deixar executar o algoritmo evolutivo primeiro e no final usar o trepa colinas na população final e vendo se era possivel ainda melhorar um pouco mais a mesma.

2.4.1 Resultados

	Ficheiro n060.txt - 10000 iterações			
	Algoritmo base híbrido i)		Algoritmo base híbrido ii)	
Parâmetros	Melhor	MBF	Melhor	MBF
população = 100 - gerações = 2500 prob. mutação = 0 prob. recombinação = 0.3	18619,0	18556.800781	18584,0	18546.732422
prob. mutação = 0.0001 prob. recombinação = 0.3 tsize = 2	18606,0	18544.333984	18668,0	18563.732422

Figura 2.8: Algoritmo Hibrido - n060.txt

	Ficheiro n120.txt - 10000 iterações			
	Algoritmo base híbrido i)		Algoritmo base híbrido ii)	
Parâmetros	Melhor	MBF	Melhor	MBF
população = 100 - gerações = 2500 prob. mutação = 0 prob. recombinação = 0.3	45079,0	44934.000000	45116,0	44950.000000
prob. mutação = 0.0001 prob. recombinação = 0.3 tsize = 2	45166,0	44997.398438	45160,0	44721.745418

Figura 2.9: Algoritmo Hibrido - n120.txt

	Ficheiro n240.txt - 10000 iterações			
	Algoritmo base híbrido i)		Algoritmo base híbrido ii)	
Parâmetros	Melhor	MBF	Melhor	MBF
população = 100 - gerações = 2500 prob. mutação = 0 prob. recombinação = 0.3	146778,0	146137.796875	146621,0	146185.593750
prob. mutação = 0.0001 prob. recombinação = 0.3 tsize = 2	146477,0	146242.000000	146384,0	145474.895422

Figura 2.10: Algoritmo Hibrido - n240.txt

Nas 3 tabelas anteriores, conseguimos perceber que o **algoritmo hibrido** foi uma mais valia uma vez que comparativamente ao **algoritmo evolutivo** registámos um aumento significativo o que leva a concluir que o algoritmo hibrido foi o mais eficaz dos três.

Em relação ao **algoritmo base hibrido 1** e ao **algoritmo base hibrido 2** não se nota resultados muitos significativos o que leva a concluir que ambos foram bons comparado com o **algoritmo evolutivo**.

Capítulo 3

Conclusão

Existem alguns conclusões que podemos tirar depois de analisadas as tabelas e os resultos das mesmas.

Uma das conclusões é que o **trepa colinas** conseguiu apenas os melhores resultados em ficheiros pequenos, quando era deparado com um grande numero de subconjuntos a sua eficácia desceu consideravelmente.

Outra conclusão bastante importante e fácil de perceber é que quanto maior era a probabilidade de recombinação , no **algoritmo evolutivo**, pior eram os resultados uma vez que as soluções iriam ser recombinadas mais vezes logo havia uma maior probabilidade das mesmas serem inválidas.
Reparamos que quanto menor era a população, e mesmo aumentando o número de gerações, o algoritmo era pior, **concluindo que o número de populações é um fator importante para atingir soluções ótimas**.

Com a mutação não existiu grandes alterações nos resultados, apenas se verificou ocasionalmente algumas descidas da qualidade muito pouco significativas.

Uma notória melhoria nas soluções foi aquando da realização do **algoritmo hibrido**. Este conseguiu melhorar todas as experiências realizadas anteriormente chegando muito perto dos valores ótimos.
Para que fossem atingidos os valores ótimos, usando o **algoritmo hibrido** seria necessário aumentar o número de iterações a realizar.

DEIS Departamento de Engenharia Informática e de Sistemas