Introdução à Inteligência Artificial

Trabalho Prático nº 2- Problema de Optimização

Docente:

Carlos Pereira

Alunos:

Paulo Henrique Figueira Pestana de Gouveia nº 2020121705 – LEI

João Filipe Silva de Almeida nº 2020144466 – LEI

Coimbra, 9 de Novembro de 2021

Índice

1. Introdução

Neste relatório vamos analisar o Problema do Conjunto Estável Máximo.
Dado um grafo não direcionado G = (V, A), composto por um conjunto V de vértices ligados entre si por arestas A, um subconjunto $S \subseteq V$ é chamado de conjunto estável quando não há nenhuma aresta entre os vértices de S. O objetivo do problema é encontrar um conjunto estável S tal que a sua cardinalidade (ou seja, o número de vértices que contém) seja máxima.
O objetivo do problema do conjunto estável máximo é portanto de maximização.

2. Algoritmo de Pesquisa Local (Trepa Colinas)

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 possíveis soluções.
O Trepa Colinas, algoritmo usado neste trabalho, 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 interação a partir do melhor atual e faz este procedimento até chegar a um estado em que todos os vizinhos têm qualidade inferior ao atual.

2.1 Funções usadas

2.1.1 read_file

Função criada para extrair e guardar os dados dos ficheiros de texto bem como, inicializar os dados da simulação com os valores desejados.
A função guarda as ligações entre os vértices numa matriz bidimensional de zeros e uns com um número de colunas e linhas igual ao número de vértices do nosso problema.

2.1.2 gera_sol_inicial

Umas das primeiras funções a ser chamadas na nossa resolução do problema, no caso de não haver penalização, irá apenas gerar soluções válidas. Está encarregue de criar um array aleatório de zeros e uns com dimensão igual ao número de vértices do problema onde cada booleano representa um vértice e nos diz se este faz parte da solução ou não.

2.1.3 trepa_colinas

Função principal do nosso algoritmo. Recebe a primeira solução gerada aleatoriamente, os dados dpo nosso problema e os parâmetros de execução, está encarregue de iterar uma quantidade de vezes igual ao número de iterações definidas e ir melhorando as soluções de forma a que no final possa retornar um array de uma solução mais adequada.

2.1.4 verifica_validade

Função encarregue de verificar se a solução recebida é válida ou não. As suas utilizações mais frequentes são para verificar se uma nova solução deve ser gerada caso não se esteja a aplicar penalização e se aceitem apenas soluções válidas, ou para vitrificar se a solução gerada é inválida e seguidamente a penalizar de acordo com o número de arestas nela presente.

2.1.5 calcula_fit

Função que nos irá ajudar a verificar o quão boa uma dada solução é. Função indispensável para o funcionamento do programa pois é a partir dela que sabemos se o nosso método está a se aproximar de uma solução mais ou menos desejável.

2.2 Peso das Repetições

Hipótese:
Supusemos que quanto maior o número de repetições efetuadas maior seria a precisão dos dados obtidos. Esperámos portanto obter nos testes com menores repetições valores mais aleatórios e afastados dos valores reais, que iriam convergir para um valor mais preciso e previsível nos testes com mais repetições.

Com mil iterações.

Ficheiro		1 repetição	10 repetições	25 repetições	50 repetições	100 repetições
file4.txt	Solução	28 vértices	33 vértices	36 vértices	38 vértices	36 vértices
	MBF	28	28.9	30.799999	30.139999	30.379999

Conclusão:
Observando os resultados obtidos podemos dizer com um grau elevado de certeza que a nossa suposição estava correta.
Nos resultados mais mais primordiais os valores de MBF obtidos são muito mais esporádicos e difíceis de prever, enquanto que à medida que se aumentam as repetições, convergem para um valor bastante mais previsível.

2.3 Peso das Vizinhanças

2.3.1 Vizinhança 1

Hipótese:
Supusemos que quanto mais iterações fossem efetuadas mais lento seria o processo de execução do programa mas em troca receberíamos resultados que se aproximariam mais do número de soluções máximo do problema.

Ficheiro		10 iterações	100 iterações	10^3 iterações	10^4 iterações	10^5 iterações
file4.txt	Solução	14 vértices	22 vértices	38 vértices	39 vértices	39 vértices
	MBF	5.47	16.889999	32.560001	35.650002	38.299999

Conclusão:
Observados os resultados obtidos pudemos concluir que a nossa suposição era verdadeira. Na maior parte dos casos, a melhor solução foi encontrada quando corrido o algoritmo com um milhão de iterações. E é possível observar o melhoramento em média das soluções encontradas no MBF (mean best fit), que também aumenta com o número de iterações.

2.3.2 Vizinhança 2

Hipótese:
Supusemos que com uma vizinhança menor seria um pouco mais custoso chegar a um valor de fitness melhor que os anteriores uma vez que teria que verificar o dobro das vizinhanças que antes. Mas iríamos obter resultados ligeiramente melhores devido ao número aumentado de vizinhanças e a escolher a melhor delas.

Ficheiro		10 iterações	100 iterações	10^3 iterações	10^4 iterações	10^5 iterações
file4.txt	Solução	17 vértices	26 vértices	37 vértices	37 vértices	37 vértices
	MBF	5.7	18.83	30.639999	30.51	30.52

Conclusão:
Não foram detectadas alterações no tempo de execução do algoritmo a olho nu, e os resultados também não diferiram muito à exceção dos resultados obtidos no ficheiro 4, o que nos diz que em certas situações uma segunda vizinhança pode ter um peso bastante significativo.

2.4 Peso dos Custos

Hipótese:
Segundo as nossas discussões, acreditamos que quer aceitássemos ou não soluções de custo igual não se iria observar uma diferença muito notável nos resultados devido à natureza de terem o mesmo custo.
Foi proposta no entanto a hipótese de aumentar a sua eficácia pois soluções de custo igual não são necessariamente iguais às últimas soluções e poderiam estar a propor diferentes soluções para o problema de uma perspetiva diferente, funcionando como uma espécie de mutação.

2.4.1 Aceitar Custo igual

Ficheiro		10 iterações	100 iterações	10^3 iterações	10^4 iterações	10^5 iterações
file4.txt	Solução	11 vértices	25 vértices	39 vértices	39 vértices	39 vértices
	MBF	5.49	19.1	33.450001	36.75	38.790001

2.4.2 Não Aceitando Custo igual

Ficheiro		10 iterações	100 iterações	10^3 iterações	10^4 iterações	10^5 iterações
file4.txt	Solução	14 vértices	22 vértices	38 vértices	39 vértices	39 vértices
	MBF	5.47	16.889999	32.560001	35.650002	38.299999

Conclusão:
Observados os resultados obtidos podemos concluir que a aceitação de soluções com ou sem custo igual faz uma grande diferença nos resultados obtidos. Parecendo ser benéfica a não aceitação uma vez que regra geral nos fornecia MBFs mais adequados.

2.5 Peso das Penalizações

2.5.1 Sem Penalização

Hipótese:
Supusemos que um método com penalização baseado na quantidade de vértices que a solução inválida tem de forma a saber quanto esta devia ser penalizada, iria tirar proveito das das soluções inválidas mas próximas de uma solução ótima. E que ao aceitar tais soluções e as penalizando conseguimos nos aproximar de uma solução ótima muito mais rapidamente.

Gerando apenas soluções Válidas

Ficheiro		10 iterações	100 iterações	10^3 iterações	10^4 iterações	10^5 iterações
file4.txt	Solução	14 vértices	22 vértices	38 vértices	39 vértices	39 vértices
	MBF	5.47	16.889999	32.560001	35.650002	38.299999

2.5.2 Com Penalização

Aplicando uma penalização de um de fitness sempre que um vizinho inválido é gerado.

Ficheiro		10 iterações	100 iterações	10^3 iterações	10^4 iterações	10^5 iterações
file4.txt	Solução	13 vértices	23 vértices	37 vértices	37 vértices	37 vértices
	MBF	5.19	16.620001	30.440001	30.120001	30.299999

Conclusão:
Como esperado foi observado que em todos os casos testados a aplicação de métodos de penalização aumentou a eficácia do nosso algoritmo permitindo chegar a melhores soluções mais rápido.

3. Algoritmo Evolutivo

(Proponha forma de lidar com soluções inválidas: evitando que surjam na população, ou se surgirem, comparar estratégias de penalização ou de reparação.)
Fitness: Percorrer o vetor, verificar se algum dos vértices a 1 tem uma aresta que liga com outro vértice a 1

3.1 Funções usadas

3.1.1 read_file

Foi utilizada a mesma função aplicada no algoritmo local.
Função criada para extrair e guardar os dados dos ficheiros de texto bem como, inicializar os dados da simulação com os valores desejados.
A função guarda as ligações entre os vértices numa matriz bidimensional de zeros e uns com um número de colunas e linhas igual ao número de vértices do nosso problema.

3.1.2 init_pop

Função encarregue de gerar as soluções iniciais no algoritmo evolutivo. Devolverá um ponteiro para o espaço na memória onde alocou e guardou um array de soluções aleatoriamente geradas para o problema proposto.

3.1.3 evaluate

Função encarregue de gerar um fitness para cada solução do array que lhe é dada, faz isto através da função auxiliar, eval_individual. Irá calcular e guardar no parâmetro fitness de cada solução o valor adequado do fitness dessa solução com base na quantidade de vértices da solução, apenas aceitará soluções válidas caso não se esteja a aplicar penalizações, e às inválidas dará um fitness de zero.

3.1.4 verifica_validade

Foi utilizada a mesma função aplicada no algoritmo local.
Função encarregue de verificar se a solução recebida é válida ou não. As suas utilizações mais frequentes são para verificar se uma nova solução deve ser gerada caso não se esteja a aplicar penalização e se aceitem apenas soluções válidas, ou para vitrificar se a solução gerada é inválida e seguidamente a penalizar de acordo com o número de arestas nela presente.

3.1.5 tournament

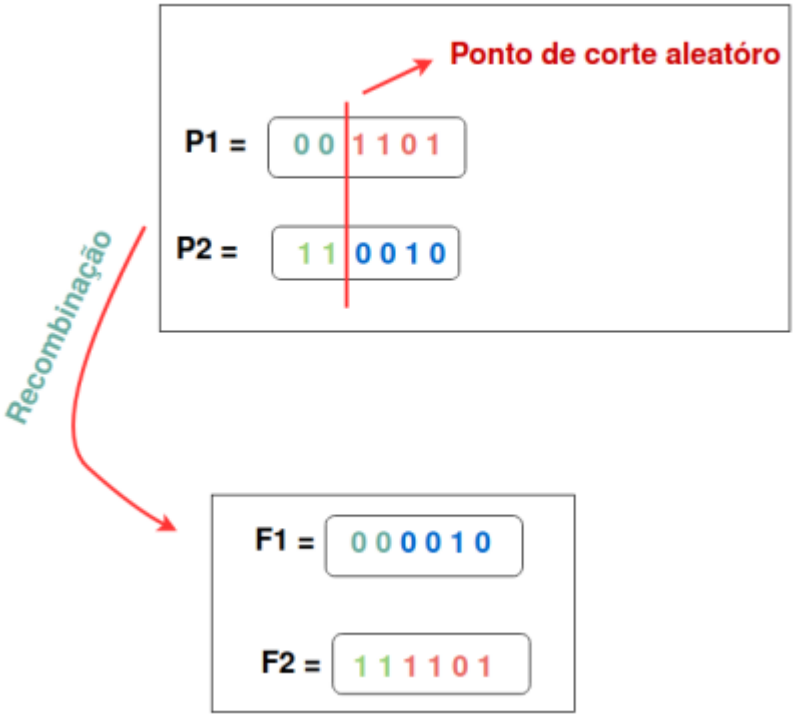
Guarda num novo array de soluções *parents,* uma série de soluções um pouco melhores que as anteriores, onde é possível que algumas das soluções anteriores sejam repetidas.

3.1.6 genetic_operators

Aplica mutação e crossover com ponto de corte às soluções que lhe são dadas.

3.1.7 crossover

Aplica o ponto de corte às soluções, aleatoriamente misturando os bits do parent e do offspring criando soluções com base na probabilidade de recombinação, onde metade dos bits são das soluções criadas pelo tournament e metade são das soluções originais.



3.1.8 mutation

Com base na probabilidade de mutação irá mudar os bits das soluções aleatoriamente. Esta função irá apenas ajudar ao escape de máximos locais.

3.2 File4.txt

Hipótese:

Recombinação:

Supusemos que uma maior recombinação iria ajudar a que fossem criadas soluções válidas mais rapidamente. No entanto concordámos na ideia que com uma combinação muito alta os valores começaram a se afastar dos ótimos devido à criação de soluções inválidas por recombinação em demasia.

Mutação:

Acreditamos que um toque de mutação poderia grandemente melhorar os resultados, mas ambos concordámos que caso fosse um pouco mais alto que o que deveria ser a mutação começaria a gerar soluções inválidas em demasia e a alterar negativamente as nossas soluções.

População:

Quanto maior a população mais tempo demoraria a gerar os resultados e iria ser mais custoso chegar a resultados, mas devido ao tamanho aumentado de objetos iremos também chegar a soluções melhores mais rapidamente quanto maior for a população.

		Algoritmo base sem penalização		Algoritmo base com penalização	
Parâmetros fixos	Parâmetros a variar	Melhor	MBF	Melhor	MBF
gerações = 5000 população = 100 prob. mutação = 0.1	prob. recombinação = 0.3	36	29.969999	37	30.16
	prob. recombinação = 0.5	38	29.43	38	30.57
	prob. recombinação = 0.7	38	29.2	38	29.15
gerações = 5000 população = 100 prob. recombinação = 0.3	prob. mutação = 0.01	28	20.67	26	19.92
	prob. mutação = 0.1	37	29.98	36	30.25
	prob. mutação = 0.5	39	31.139999	37	30.959999
prob. recombinação = 0.3 prob. mutação = 0.1	população = 100 (gerações = 2500)	35	27.36	32	27.549999
	população = 50 (gerações = 2500)	31	22.75	32	22.66
	população = 10 (gerações = 1000)	15	4.69	17	6.32

Conclusão:

Recombinação:

Como esperado na nossa hipótese, a recombinação quando é feita com um valor probabilístico muito alto começa a não ser benéfico para os nossos resultados.

Mutação:

Analisando os resultados podemos ver que mesmo aplicando uma mutação com valores bastante altos de probabilidade, o resto do código é resistente à alta mutação e somos capazes de obter resultados credíveis de qualquer forma.

População:

Como esperado, com uma população muito pequena, não foram efetuados testes e alterações suficientes para chegar a uma solução válida muito alta. Quanto maior a população maior o custo computacional, mas melhores são as soluções obtidas.

4. Algoritmo Híbrido

Este algoritmo é uma combinação do algoritmo Trepa Colinas com o ALgoritmo Evolutivo visando utilizar ambos para chegar à solução ótima

Criamos o algoritmo híbrido de 2 maneiras diferentes, nas seguintes tabelas inspecionamos as suas qualidades, os algoritmos funcionam da seguinte maneira:
Híbrido I: Consiste em aplicar para o número de iterações pedidas o algoritmo evolutivo, e de seguida, traduz as soluções para poderem ser lidas e utilizadas pelo algoritmo local que é corrido de seguida para o número de iterações pedidas. Dando um resultado onde a primeira metade de iterações foi corrida com o algoritmo evolutivo e a segunda metade, com o algoritmo do trepa colinas.
Híbrido II: Elaboramos um segundo algoritmo por termos proposto a possibilidade de haver uma preferência por um dos métodos aplicando primeiro um e depois o outro. Portanto criámos um sistema que alterna entre ambos os métodos traduzindo os dados de um para o outro assim que preciso.

Resultados:

Ficheiro file4.txt					
		Híbrido (I)		Híbrido (II)	
Parâmetros Fixos	Parâmetros a variar	Best	MBF	Best	MBF
ger = 1000	pr = 0.3	36	31.9	34	31.299999
pop = 100	pr = 0.5	35	29.95	37	31.700001
pm = 0.1	pr = 0.7	39	33.349998	39	33.200001
ger = 1000	pm = 0.01	35	31.200001	32	29.5
pop = 100	pm = 0.1	39	32.549999	36	32.200001
pr = 0.3	pm = 0.5	37	31.25	39	32.099998
pr = 0.3	pop = 50 (ger = 1K)	36	32.200001	37	32.099998
pm = 0.1	pop = 10 (ger = 1K)	33	26	36	29.6

Conclusão:

Comparando aos valores obtidos no algoritmo evolutivo, notamos um ligeiro aumento na média de maior número de vértices que encontra em geração mas não muito significativo, exceto que encontrava melhores soluções em muito menos repetições.

Em relação à diferença entre os algoritmos híbridos, são muito similares quanto aos valores, apesar de funcionarem de maneiras diferentes, pelo que podemos inferir que a ordem de execução dos algoritmos de pesquisa local e de evolução não trazem grandes alterações aos resultados.

5. Conclusão

No final achámos este trabalho benéfico para as nossas vidas académicas devido ao que nos ensinou sobre a capacidade de algoritmos serem capazes de resolver problemas que exigiriam uma quantidade de tempo indefinido para resolver com algoritmos lineares.

Aumentou-nos os horizontes no que toca ao conhecimento de formas de resolver problemas que à primeira vista parecem impossíveis e à aplicação de métodos que poderiam ser aplicados e generalizados para muitos tipos de problemas, mesmo problemas reais que a maior parte das vezes provêm de situações em que a quantidade de dados a processar é demasiado grande para qualquer algoritmo não aplicado de uma das formas estudadas.

O trabalho foi realizado em linguagem C, utilizando-se ferramentas como o Visual Studio Code, Git e Github desktop, Google Docs, Discord, para colaboração.

6. Referências

- Materiais Práticos dados por Mateus Mendes e materiais teóricos dados em contexto da cadeira de Introdução à Inteligência artificial do ISEC gerida por Carlos Manuel Jorge da Silva Pereira