Trabalho Prático IIA
META 2

Trabalho Realizado por:

- *Pedro Rodrigues Jorge nº2021142041*
- *Luís Travassos nº2021136600*

Problema proposto

Foi nos proposto conceber, implementar e testar métodos de otimização que encontrem soluções de boa qualidade para diferentes instâncias do mesmo problema.

Em relação ao problema foi nos dado 6 ficheiros, cada um com um grafo e um valor inteiro k, com o objetivo de aplicar sobre eles o "Maximum Edge Subgraph Problem" que consiste em encontrar um subconjunto de k-vértices tal que o número de arestas dentro desse mesmo seja máximo.

Em relação ao métodos de otimização foram implementados os seguintes: um algoritmo de pesquisa local, um algoritmo evolutivo e um algoritmo híbrido (uma mistura das melhores partes do algoritmo local com o evolutivo).

Algoritmo de Pesquisa Local

O algoritmo de Pesquisa Local começa por inicializar os dados do ficheiro de teste escolhido pelo utilizador e para isso é necessário criar as seguintes variáveis: nPoints (nº de vértices), nLines (nº de arestas), kValue (nº de vértices por grupo de teste/solução), grid (array com o mapa de ligações dos vértices), sol (grupo de teste/solução "inicial"), bestSol (melhor grupo de teste/solução).

Após isso será inicializada a grupo de teste/solução inicial na função "geraSolIni( )", a qual irá preencher o array sol com kValue 1´s e o resto com 0's (exemplo, no ficheiro teste.txt, o array sol será algo como o seguinte [011101]).

De seguida o código entra na função "trepaColinas( )", a qual irá gerar um (ou dois) vizinho(s) á solução inicial, estes serão avaliados e será retirado o seu custo (nº de ligações) e este será comparado com o custo da solução inicial, sendo preservada a solução com maior custo (ou igual). Todo este ciclo será repetido por numIter vezes.

A criação dos vizinhos é responsável pela função "geraVizinho( )", a qual começa por replicar a solução inicial para uma nova solução e de seguida troca um valor de sitio (muda um 1 de índice) na nova solução.

O apuramento do custo de cada solução é realizado pela função "calculaFit( )", a qual irá usar os índices da solução com que está a trabalhar para procurar na grid se estes têm ligação entre si, se sim aumenta o custo e no final do processo devolve-o.

Após a função "trepaColinas( )" é guardado a melhor solução e custo e este comparados com a melhor solução e custo do programa todo, sendo o melhor preservado na variável solBest e bestCost, após isto o processo será repetido denovo runs vezes, onde no final é apresentada a média das soluções do programa e a melhor solução de todas.

Algoritmo Evolutivo

O algoritmo Evolutivo começa por inicializar os dados do ficheiro de teste escolhido pelo utilizador e para isso existem as seguintes estruturas: struct info (responsável por armazenar o tamanho da população (popsize), a probabilidade de mutação (pm), a probabilidade de recombinação (pr), o tamanho do torneio (tsize), o número de objectos que cabem na "mochila"/array (numMochila) e o número de gerações (numGenerations)), struct individual (responsável por armazenar o array/mochila (mochila), o número de ligações do array (fitness) e se a mochila é válida(valido)), kValue (o número de vértices por solução/mochila).

Após isso será inicializada a população de teste/soluções, sendo cada array/mochila da população constituído por kValue 1's e o resto 0's (exemplo, no ficheiro teste.txt, o array sol será algo como o seguinte [011101]).

De seguida o código entra na função "evaluate( )", a qual define o fitness de cada individuo dentro da população, usando os índices da solução com que está a trabalhar para procurar na grid se estes têm ligação entre si e se sim aumenta o custo e no final do processo devolve-o.

Após isso será determinado a melhor solução com a função "get_best( )", a qual devolve o individuo com melhor fitness. Com esse valor definido entramos no "tournament( )", o qual irá buscar tsize valores de fitness à população e com o maior irá criar um parent, estrutura igual à população mas preenchida com os valores de parent.

De seguida a população passa pela função "genetic_operators( )", a qual é responsável por lançar a função "crossover( )" e função "mutation( )". A função "crossover( )" irá, após a verificação da probabilidade de recombinação, criar um "ponto" de divisão no array, após o qual irá formar "sons" a partir da primeira parte (do ponto para trás) do "parente[i]" e a segunda parte do "parente[i+1]". A função "mutation( )" irá, após a verificação da probabilidade de mutação, copiar os valores alternados do "parent" para o "son" (0's passam para 1's e vice-versa).

Após a realização destas alterações é lançado a função "correction( )", a qual irá verificar o número de 1's em cada individuo e tirar ou colocar 1's até o valor de kValue ser alcançado.

De seguida é determinado denovo a melhor solução com a função "get_best( )", a qual devolve o individuo com melhor fitness. Será guardado este "individuo" e o processo todo repetido por runs vezes, após isso é apresentada a média das soluções e a melhor solução de todas.

Algoritmo Híbrido

O algoritmo Híbrido começa por inicializar os dados do ficheiro de teste escolhido pelo utilizador e para isso existem as seguintes variáveis: nPoints (nº de vértices), nLines (nº de arestas), kValue (nº de vértices por grupo de teste/solução), grid (array com o mapa de ligações dos vértices), sol (grupo de teste/solução "inicial"), sol2 (2º grupo de teste/solução "inicial"), bestSol (melhor grupo de teste/solução), probRec (probabilidade de recombinação), probMut (probablidade de mutação).

Após isso será inicializada ambos os grupos de teste/soluções iniciais na função "geraSolIni( )", a qual irá preencher o array sol e sol2 com kValue 1´s e o resto com 0's (exemplo, no ficheiro teste.txt, o array sol será algo como o seguinte [011101]).

De seguida o código entra na função "trepaColinas( )", a qual irá gerar um vizinho á solução inicial, estes serão avaliados e será retirado o seu custo (nº de ligações) e este será comparado com o custo da sol e a sol2, sendo preservada a solução com maior custo, sendo primeiro substituída a sol e só se não for possível substitui-se a sol2, ficando assim sol com o valor do melhor vizinho e sol2 com o valor do 2º melhor vizinho. Todo este ciclo será repetido por numIter vezes.

De seguida a população passa pela função "geneticTrepaColinas( )", a qual é responsável por criar vizinhos a partir de sol e, de seguida, submete-los á função "crossover( )" e á função "mutation( )", caso a probabilidade destas se verifique. A função "crossover( )" irá criar um "ponto" de divisão no sol e sol2, após o qual irá reformar o vizinho a partir da primeira parte (do ponto para trás) do "sol" e a segunda parte do "sol2". A função "mutation(_)" irá copiar os valores alternados do "sol" para o vizinho (0's passam para 1's e vice-versa).

Após a realização destas alterações é lançado a função "correction( )", a qual irá verificar o número de 1's em cada individuo e tirar ou colocar 1's até o valor de kValue ser alcançado. De seguida irá ser verificado o custo do vizinho e será dado o mesmo processo de substituição de sol e sol2 da função "trepaColinas( )". Todo este ciclo será repetido por numIter vezes.

Após a função "geneticTrepaColinas( )" é guardado a melhor solução e custo (neste caso será o sol) e este comparados com a melhor solução e custo do programa todo, sendo o melhor preservado na variável solBest e bestCost, após isto o processo será repetido denovo runs vezes, onde no final é apresentada a média das soluções do programa e a melhor solução de todas.

Análise Excel

No Excel responsável pela análise do algoritmo de Pesquisa Local é bastante notório que quaisquer medidas que aumentem o número de vizinhos gerados e testados, como por exemplo o nº de iterações ou nº de vizinhos gerados por iteração, iram ter como consequência melhores resultados.

No Excel responsável pela análise do algoritmo Evolutivo notasse que, devido à necessidade de corrigir os resultados obtidos pela recombinação e mutação, assim como o número gigante de dados criados, em comparação ao algoritmo de Pesquisa Local, e a "fraca" seleção das melhores soluções, os resultados são muito aquém do esperado, sendo apenas o aumento do tamanho o torneio a única mudança que se mostra significante na maioria do experimento.

No Excel responsável pela análise do algoritmo Hibrido é de notar que a maioria dos resultados tende a ser sempre o mais alto, mostrando assim uma ótima capacidade de decifração, mas devido ao número de vizinhos gerado e ao número de testes submetidos a estes, o algoritmo tende a tornar-se pesado quando têm que lidar com várias iterações de uma única vez, mas sendo mais do que capaz de chegar ao resultado máximo com bastantes poucas iterações.

Conclusão

Este trabalho foi bastante importante e enriquecedor no ensino e utilização de algoritmos de inteligência artificial, sendo um trabalho que nos mostrou como se cria, usa e se retiram conclusões sobre problemas que manualmente seriam, basicamente, impossíveis.

Estes ensinamentos serão, sem dúvida, bastante uteis na nossa vida profissional e educativa.