

Introdução a inteligência artificial - 2022/2023

ISEC – Instituto Superior de Engenharia de Coimbra

**Trabalho prático II**

**Professores:**

**Nuno Alexandre Cid Martins**

**Carlos Manuel Jorge da Silva Pereira**

**Realizado por:**

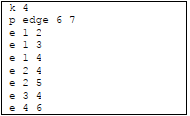
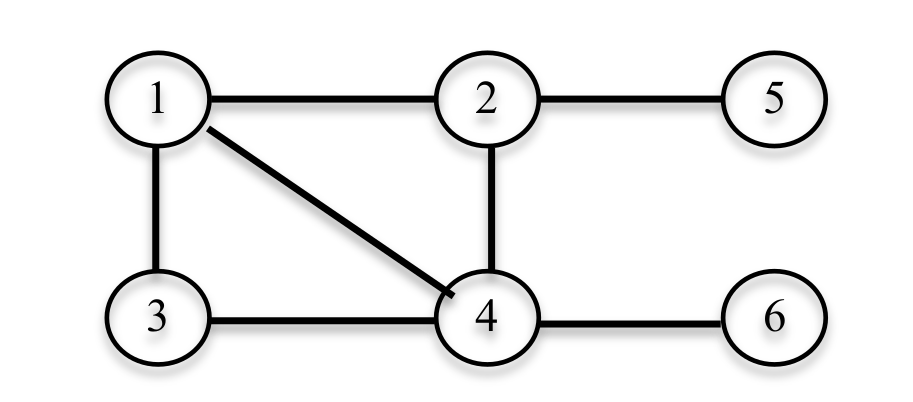
2021110042 - Jorge Ricardo Marques Duarte

2021146383 - João Alexandre Caravela Marques

**Introdução**

O objetivo do trabalho prático consiste em conceber, implementar e testar métodos de otimização que encontrem soluções de boa qualidade para o Maximum Edge Subgraph Problem. Este problema consiste em, dado um grafo não direcionado G = (V, A) e um inteiro k, encontrar um subconjunto de vértices S de tamanho k que maximize o número de arestas dentro de S. As instâncias para teste do problema são grafos com até 500 vértices, representados em ficheiros de texto com a informação armazenada na forma de lista de adjacências.

Queremos, portanto, encontrar métodos eficientes para resolver o Maximum Edge Subgraph Problem, de forma a obter soluções de qualidade para diferentes instâncias do problema.

O grafo para análise, vai ser gerado por base na leitura de um ficheiro que contém o número de vértices-k da solução e as ligações, arestas, que unem certos e determinados vértices.

**Algoritmo de pesquisa local**

O algoritmo de trepa-colinas é conhecido por ser eficiente em encontrar soluções ótimas para problemas com estruturas simples, mas é limitado em problemas mais complexos.

A nossa implementação começa com uma solução inicial gerada aleatoriamente e, em seguida, gera soluções vizinhas (com duas trocas aleatórias de elementos entre o conjunto de vértices selecionados e o conjunto de vértices não selecionados) e avalia se essas soluções são melhores que a solução atual. Se encontrar uma solução melhor, ela é aceite como a nova solução atual e o processo é repetido. Caso contrário, o algoritmo para. O número de iterações é limitado pelo parâmetro *iterationsHillClimbing*. A função *hill\_climbing* retorna o custo da solução final encontrada. A função *code\_execution* armazena a solução final como a melhor solução encontrada até o momento, se ela for melhor que a solução anteriormente armazenada como a melhor. O algoritmo também imprime a solução final e o custo associado, se o parâmetro *displaySolutions* for verdadeiro.

**Estratégia de validação de soluções inválidas**

A estratégia de validação de soluções inválidas é uma técnica utilizada para garantir que a solução gerada para um problema é válida e atende aos requisitos e restrições do problema.

Desta forma, no nosso algoritmo de pesquisa local estamos a garantir que não temos soluções inválidas na função que gera a solução inicial. A solução inicial é gerada de forma aleatória, escolhendo k números aleatórios entre 0 e 1 vértices e atribuindo o valor 1 a eles na solução. Depois, a função verifica se a solução inicial tem exatamente k elementos com valor 1. Se não tiver, a função gera novos números aleatórios até que a solução inicial tenha exatamente k vértices com valor 1. Dessa forma, a função garante que a solução inicial gerada é válida e atende aos requisitos do problema.

### **Problemas do algoritmo**

Algumas das limitações da implementação em relação ao problema são:

O número de iterações é fixo e pré-definido pelo parâmetro *iterationsHillClimbing*. Isto pode não ser suficiente para encontrar a solução ótima em instâncias mais complexas do problema.

A geração de soluções vizinhas é realizada com duas trocas aleatórias de elementos entre o conjunto de vértices selecionado e o conjunto de vértices não selecionados. Isso pode não ser a estratégia mais eficiente para gerar soluções vizinhas para o Maximum Edge Subgraph Problem.

O código está limitado a grafos com até 500 vértices. Isso pode não ser suficiente para instâncias mais complexas do problema.

O código não implementa nenhum método de escape de uma região local ótima. Se o algoritmo ficar preso em uma região local ótima, ele não será capaz de encontrar uma solução global ótima.

**Variantes**

A função *hill\_climbing* é um algoritmo de otimização que utiliza um processo iterativo para encontrar uma solução ótima para um problema. A cada iteração, a função gera uma solução vizinha da solução atual, avalia a qualidade da nova solução e, se ela for melhor, torna a nova solução a solução atual. Este processo é repetido até atingir um número máximo de iterações ou até que não sejam encontradas soluções vizinhas melhores.

***Trepa colinas com vizinhança 1***

A função *generate\_neighbor*gera uma solução vizinha da solução atual de maneira aleatória, escolhendo dois elementos da solução atual e trocando os seus valores. Por exemplo, se a solução atual for [1, 0, 1, 0], uma solução vizinha gerada pela função *generate\_neighbor* poderia ser [1, 0, 0, 1].

***Trepa colinas com vizinhança 2***

A função *generate\_neighbor2*também gera uma solução vizinha da solução atual de maneira aleatória, mas difere da função *generate\_neighbor*no sentido em que escolhe 4 elementos da solução atual e troca os valores dos pares de elementos escolhidos. Por exemplo, se a solução atual for [1, 0, 1, 0,1, 0, 1, 0], uma solução vizinha gerada pela função *generate\_neighbor2*poderia ser [0, 1, 1, 0,1, 0, 1, 0].

Ao utilizar a função *generate\_neighbor2*em vez da função *generate\_neighbor*, o algoritmo pode gerar uma maior variedade de soluções vizinhas, o que pode aumentar a probabilidade de encontrar uma solução ótima. No entanto, também pode levar a uma maior complexidade computacional, pois gera e avalia mais soluções vizinhas.

## 

## **Resultados experimentais**

| Trepa-Colinas com Vizinhança 2 | | | | | | Melhor solução possível | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| G = (# vértices, # arestas) | 100 execuções | 100 it | 1000 it | 5000 it | 10000 it |
| Teste 1: teste.txt  G=(6, 7) | Melhor | 5 | 5 | 5 | 5 | 5 | |
| MBF | 5,00 | 5,00 | 5,00 | 5,00 |
| Teste 2: file1.txt  G=(28, 210) | Melhor | 20 | 20 | 20 | 20 | 21 | |
| MBF | 20,00 | 20,00 | 20,00 | 20,00 |
| Teste 3: file2.txt  G=(64, 704) | Melhor | 15 | 15 | 15 | 15 | 16 | |
| MBF | 15,00 | 15,00 | 15,00 | 15,00 |
| Teste 4: file3.txt  G=(70, 1855) | Melhor | 111 | 112 | 112 | 112 | 112 | |
| MBF | 109,00 | 112,00 | 112,00 | 112,00 |
| Teste 5: file4.txt  G=(200, 1534) | Melhor | 60 | 79 | 79 | 79 | 79 | |
| MBF | 52,30 | 75,10 | 79,00 | 79,00 |
| Teste 6: file5.txt  G=(500, 4459) | Melhor | 51 | 83 | 93 | 93 | 98 | |
| MBF | 38,50 | 79,00 | 91,40 | 91,60 |

| Trepa-Colinas com Vizinhança 1 | | | | | | Melhor solução possivel | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| G = (# vértices, # arestas) | 100 execuções | 100 it | 1000 it | 5000 it | 10000 it |
| Teste 1: teste.txt  G=(6, 7) | Melhor | 5 | 5 | 5 | 5 | 5 | |
| MBF | 5,00 | 5,00 | 5,00 | 5,00 |
| Teste 2: file1.txt  G=(28, 210) | Melhor | 20 | 20 | 20 | 20 | 21 | |
| MBF | 20,00 | 20,00 | 20,00 | 20,00 |
| Teste 3: file2.txt  G=(64, 704) | Melhor | 15 | 15 | 15 | 15 | 16 | |
| MBF | 15,00 | 15,00 | 15,00 | 15,00 |
| Teste 4: file3.txt  G=(70, 1855) | Melhor | 112 | 112 | 112 | 112 | 112 | |
| MBF | 110,6 | 112,00 | 112,00 | 112,00 |
| Teste 5: file4.txt  G=(200, 1534) | Melhor | 64 | 79 | 79 | 79 | 79 | |
| MBF | 53,60 | 79,00 | 79,00 | 79,00 |
| Teste 6: file5.txt  G=(500, 4459) | Melhor | 52 | 98 | 98 | 98 | 98 | |
| MBF | 41,00 | 93,30 | 98,00 | 98,00 |

**Análise dos resultados**

Os resultados da tabela mostram que há uma diferença na solução ótima encontrada e na média dos valores de fitness na execução do algoritmo trepa-colinas com a função de *generate\_neighbor* em relação à função de *generate\_neighbor2*.

Em geral, a função de *generate\_neighbor2* resulta em soluções ótimas mais elevadas e valores de fitness médios mais elevados do que a função de gera vizinhança 1.

A explicação para esta diferença é que a função de gera vizinhança 2 gera uma vizinhança mais ampla, o que pode levar a uma busca mais intensiva pelo espaço de soluções. Isto pode aumentar a probabilidade de encontrar soluções ótimas mais elevadas, mas também pode levar a uma execução mais lenta do algoritmo devido à necessidade de avaliar mais soluções na vizinhança. Por outro lado, a função de gera vizinhança 1 gera uma vizinhança mais restrita, o que pode levar a uma busca menos intensiva pelo espaço de soluções. Isto diminui a probabilidade de encontrar soluções ótimas mais elevadas, mas pode resultar em uma execução mais rápida do algoritmo devido à necessidade de avaliar menos soluções na vizinhança.

**Algoritmo evolutivo**

O algoritmo evolutivo é uma técnica de otimização que se inspira no processo de seleção natural presente na natureza. O funcionamento do algoritmo baseia-se na criação de uma população inicial de soluções, avaliando-as de acordo com um critério de aptidão (fitness) e selecionando as melhores para serem os pais da próxima geração. A nova geração é gerada a partir da recombinação dos pais selecionados, com a introdução de mutações aleatórias com uma determinada probabilidade. Este processo é repetido sucessivamente até que se atinja um critério de paragem, podendo ser atingido um resultado ótimo ou um resultado satisfatório. Além disso, o algoritmo evolutivo é um método robusto, pois é capaz de encontrar soluções ótimas mesmo em problemas com espaços de busca muito grandes e com pouco conhecimento prévio sobre o problema.

A nossa implementação do algoritmo gera uma população de soluções e, em seguida, é avaliada e classificada com base em sua aptidão (número de arestas no conjunto). O algoritmo então seleciona os indivíduos mais aptos da população atual para serem os pais da próxima geração e gera novos indivíduos através da recombinação e mutação dos pais selecionados. O processo é repetido até que uma solução aceitável seja encontrada ou o número máximo de gerações seja atingido. O algoritmo também mantém a melhor solução encontrada até o momento.

A função *generateInitialPopulation*gera a população inicial de soluções aleatoriamente. A função *evaluate\_population*avalia cada solução da população e a classifica em ordem crescente de aptidão. A função *tournament\_selection*seleciona de forma aleatória dois indivíduos da população e comparando o seu fitness, seleciona um deles para a população de pais da próxima geração. A função *crossover*os indivíduos selecionados, gerando uma nova geração de soluções. A função *mutation*aplica mutações aleatórias às soluções da nova geração. A função *code\_execution*é o loop principal do algoritmo evolutivo, que repete o processo de seleção, reprodução e mutação até atingir o número de gerações especificado.

**Estratégia de validação de soluções inválidas**

A estratégia de validação de soluções inválidas no algoritmo evolutivo acaba por ser a mesma função utilizada no algoritmo do trepa colinas. Para cada indivíduo dentro da população é sempre gerada uma solução válida. Isto significa que se o tamanho da população for 1000, iremos ter no momento de inicialização 1000 soluções válidas. Em cada geração são efetuadas mutações, recombinações e torneios que podem afetar a solução, mas, estas contêm soluções de reparação, evitando desta forma soluções inválidas.

**Problemas do algoritmo**

Um dos problemas do algoritmo evolutivo é que ele pode ser computacionalmente intensivo, pois precisa gerar e avaliar uma grande quantidade de soluções. Além disso, o algoritmo evolutivo pode ser propenso a ficar preso em mínimos locais (soluções subótimas) e pode ter dificuldade em encontrar soluções ótimas para problemas com muitas restrições.

**Variantes**

***Recombinação por um ponto de corte (crossover)***

Sendo que o crossover é um operador genético para gerar novas soluções em algoritmos evolutivos, utilizamos esta variante na nossa análise com o objetivo de melhorar a qualidade das soluções na população.

Esta recombinação consiste em escolher aleatoriamente o código genético dos pais, os genes à esquerda do ponto de corte são copiados para o primeiro pai, enquanto que os genes à direita são copiados para o segundo pai.

***Recombinação por dois pontos de corte***

Outra variante utilizada para análise presente no trabalho foi a recombinação por dois pontos de corte, utilizando a função *recombinacao\_dois\_pontos\_corte*, que que realiza a recombinação de dois indivíduos de uma população através de um processo de troca em dois pontos de corte. Isso significa que, para os dois indivíduos selecionados para recombinação, são escolhidos dois pontos de corte aleatórios na sequência de genes de cada indivíduo. Todos os genes entre esses dois pontos de corte são então trocados entre os dois indivíduos.

***Recombinação por troca***

Utilizamos este operador de forma a conseguir uma mistura de genes mais flexível entre as soluções, permitindo assim inserir uma ampla variedade de soluções possíveis. É semelhante à recombinação com um ou dois pontos de corte, mas ao invés de simplesmente copiar os genes de um pai para outro, esta troca efetivamente os genes entre os pais.

***Recombinação uniforme***

No nosso algoritmo, foi utilizada a função *recombinacao\_uniforme*, que é uma função que realiza a recombinação de dois indivíduos de uma população através de um processo de troca uniforme. Isso significa que, para cada gene (característica) dos dois indivíduos selecionados para recombinação, há uma probabilidade igual de que o gene seja trocado entre os dois indivíduos.

***Mutação binária***

Nesta variante, aplicamos a mutação em um indivíduo de uma população e consiste em alterar aleatoriamente um ou mais genes de um indivíduo. A probabilidade de mutação é um parâmetro que é definido na estrutura *SimulationState*previamente definida e que indica a probabilidade de um gene ser mutado. Esta variante foi utilizada no algoritmo com o objetivo de introduzir novas soluções na população e, assim, aumentar a diversidade genética.

***Mutação por troca***

A mutação por troca é uma variante da mutação que consiste em trocar aleatoriamente dois genes de um indivíduo. Esta variante da mutação é utilizada em algoritmos genéticos que resolvem problemas em que a ordem dos genes é importante.

**Resultados experimentais**

|  |  | Algoritmo base (Recombinação de 1 ponto de corte + Mutação binária + Reparação) | | | | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | file1.txt | | file2.txt | | file3.txt | | file4.txt | | file5.txt | | Teste.txt | |
| Parâmetros Fixos | Parâmetros a variar | Best | MBF | Best | MBF | Best | MBF | Best | MBF | Best | MBF | Best | MBF |
| ger = 2500 | pr = 0.3 | 20 | 20,00 | 15 | 15,00 | 112 | 112,00 | 71 | 65,62 | 41 | 33,139 | 5 | 5,00 |
| pop = 100 | pr = 0.5 | 20 | 20,00 | 15 | 15,00 | 112 | 112,00 | 75 | 67,949 | 43 | 35,189 | 5 | 5,00 |
| pm = 0.01 | pr = 0.7 | 20 | 20,00 | 15 | 15,00 | 112 | 112,00 | 78 | 68,79 | 45 | 36,0299 | 5 | 5,00 |
| ger = 2500  pop = 100  pr = 0.7 | pm = 0.0 | 20 | 20,00 | 15 | 15,00 | 112 | 108,629 | 79 | 63,84 | 93 | 57,209 | 5 | 5,00 |
| pm = 0.001 | 20 | 20,00 | 15 | 15,00 | 112 | 111,91 | 79 | 77,379 | 98 | 92,4 | 5 | 5,00 |
| pm = 0.01 | 20 | 20,00 | 15 | 15,00 | 112 | 112 | 75 | 69,15 | 42 | 35,43 | 5 | 5,00 |
| pm = 0.05 | 20 | 20,00 | 15 | 15,00 | 112 | 107,25 | 35 | 31,87 | 25 | 21,719 | 5 | 5,00 |
| pr = 0.7  pm = melhor valor obtido | pop = 10 (ger = 25K) | 20 | 20,00 | 15 | 15,00 | 112 | 112 | 79 | 76,1 | 98 | 92,169 | 5 | 5,00 |
| pop = 50 (ger = 5K) | 20 | 20,00 | 15 | 15,00 | 112 | 112 | 79 | 77,32 | 98 | 91,19 | 5 | 5,00 |
| pop = 100 (ger = 2.5K) | 20 | 20,00 | 15 | 15,00 | 112 | 112 | 79 | 76,849 | 98 | 92,459 | 5 | 5,00 |

|  |  | Algoritmo base (Recombinação de 1 ponto de corte + Mutação binária + Reparação) com torneio 10 | | | | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | file1.txt | | file2.txt | | file3.txt | | file4.txt | | file5.txt | | Teste.txt | |
| Parâmetros Fixos | Parâmetros a variar | Best | MBF | Best | MBF | Best | MBF | Best | MBF | Best | MBF | Best | MBF |
| ger = 2500 | pr = 0.3 | 20 | 20,00 | 15 | 15,00 | 112 | 112,00 | 79 | 77,250 | 66 | 33,139 | 5 | 5,00 |
| pop = 100 | pr = 0.5 | 20 | 20,00 | 15 | 15,00 | 112 | 112,00 | 79 | 77,250 | 70 | 35,189 | 5 | 5,00 |
| pm = 0.01 | pr = 0.7 | 20 | 20,00 | 15 | 15,00 | 112 | 112,00 | 79 | 77,665 | 71 | 36,0299 | 5 | 5,00 |
| ger = 2500  pop = 100  pr = 0.7 | pm = 0.0 | 20 | 20,00 | 15 | 15,00 | 112 | 104,899 | 51 | 33,825 | 48 | 57,209 | 5 | 5,00 |
| pm = 0.001 | 20 | 20,00 | 15 | 15,00 | 112 | 112,00 | 79 | 77,25 | 98 | 92,4 | 5 | 5,00 |
| pm = 0.01 | 20 | 20,00 | 15 | 15,00 | 112 | 112,00 | 79 | 77,65 | 72 | 35,43 | 5 | 5,00 |
| pm = 0.05 | 20 | 20,00 | 15 | 15,00 | 112 | 107,724 | 51 | 43,125 | 28 | 21,719 | 5 | 5,00 |
| pr = 0.7  pm = melhor valor obtido | pop = 10 (ger = 25K) | 20 | 20,00 | 15 | 15,00 | 112 | 112,00 | 55 | 44,5 | 86 | 92,169 | 5 | 5,00 |
| pop = 50 (ger = 5K) | 20 | 20,00 | 15 | 15,00 | 112 | 112,00 | 79 | 78 | 98 | 91,19 | 5 | 5,00 |
| pop = 100 (ger = 2.5K) | 20 | 20,00 | 15 | 15,00 | 112 | 112,00 | 79 | 76,4 | 98 | 92,459 | 5 | 5,00 |

|  |  | Algoritmo com recombinação com 2 pontos de corte + mutação binária + reparação | | | | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | file1.txt | | file2.txt | | file3.txt | | file4.txt | | file5.txt | | Teste.txt | |
| Parâmetros Fixos | Parâmetros a variar | Best | MBF | Best | MBF | Best | MBF | Best | MBF | Best | MBF | Best | MBF |
| ger = 2500 | pr = 0.3 | 20 | 20,00 | 15 | 15,00 | 112 | 112 | 71 | 65,57 | 42 | 32,57 | 5 | 5,00 |
| pop = 100 | pr = 0.5 | 20 | 20,00 | 15 | 15,00 | 112 | 112 | 75 | 67,08 | 39 | 33,36 | 5 | 5,00 |
| pm = 0.01 | pr = 0.7 | 20 | 20,00 | 15 | 15,00 | 112 | 112 | 75 | 67,86 | 42 | 35,349 | 5 | 5,00 |
| ger = 2500  pop = 100  pr = 0.7 | pm = 0.0 | 20 | 20,00 | 15 | 15,00 | 112 | 112 | 79 | 65,68 | 79 | 57,23 | 5 | 5,00 |
| pm = 0.001 | 20 | 20,00 | 15 | 15,00 | 112 | 112 | 79 | 77,62 | 98 | 91,23 | 5 | 5,00 |
| pm = 0.01 | 20 | 20,00 | 15 | 15,00 | 112 | 112 | 79 | 66,41 | 43 | 34,52 | 5 | 5,00 |
| pm = 0.05 | 20 | 20,00 | 15 | 15,00 | 112 | 112 | 36 | 31,959 | 26 | 21,42 | 5 | 5,00 |
| pr = 0.7  pm = melhor valor obtido | pop = 10 (ger = 25K) | 20 | 20,00 | 15 | 15,00 | 112 | 112 | 79 | 75,699 | 98 | 92,01 | 5 | 5,00 |
| pop = 50 (ger = 5K) | 20 | 20,00 | 15 | 15,00 | 112 | 112 | 79 | 76,82 | 98 | 91,25 | 5 | 5,00 |
| pop = 100 (ger = 2.5K) | 20 | 20,00 | 15 | 15,00 | 112 | 112 | 79 | 77,44 | 98 | 92,459 | 5 | 5,00 |

|  |  | Algoritmo com recombinação com 2 pontos de corte + mutação por troca + reparação | | | | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | file1.txt | | file2.txt | | file3.txt | | file4.txt | | file5.txt | | Teste.txt | |
| Parâmetros Fixos | Parâmetros a variar | Best | MBF | Best | MBF | Best | MBF | Best | MBF | Best | MBF | Best | MBF |
| ger = 2500 | pr = 0.3 | 20 | 20,00 | 15 | 15,00 | 112 | 109,919 | 79 | 68,15 | 64 | 46,439 | 5 | 5,00 |
| pop = 100 | pr = 0.5 | 20 | 20,00 | 15 | 15,00 | 112 | 110,54 | 79 | 71,4 | 93 | 56,259 | 5 | 5,00 |
| pm = 0.01 | pr = 0.7 | 20 | 20,00 | 15 | 15,00 | 112 | 110,73 | 79 | 72,91 | 93 | 66,73 | 5 | 5,00 |
| ger = 2500  pop = 100  pr = 0.7 | pm = 0.0 | 20 | 20,00 | 15 | 15,00 | 112 | 108,68 | 79 | 63,04 | 85 | 55,29 | 5 | 5,00 |
| pm = 0.001 | 20 | 20,00 | 15 | 15,00 | 112 | 109,239 | 79 | 65,599 | 90 | 60,279 | 5 | 5,00 |
| pm = 0.01 | 20 | 20,00 | 15 | 15,00 | 112 | 111,15 | 79 | 74,019 | 98 | 69,089 | 5 | 5,00 |
| pm = 0.05 | 20 | 20,00 | 15 | 15,00 | 112 | 111,199 | 79 | 76,449 | 98 | 81,129 | 5 | 5,00 |
| pr = 0.7  pm = melhor valor obtido | pop = 10 (ger = 25K) | 20 | 20,00 | 15 | 15,00 | 112 | 112,00 | 79 | 76,68 | 90 | 65,97 | 5 | 5,00 |
| pop = 50 (ger = 5K) | 20 | 20,00 | 15 | 15,00 | 112 | 111,55 | 79 | 76,18 | 98 | 76,62 | 5 | 5,00 |
| pop = 100 (ger = 2.5K) | 20 | 20,00 | 15 | 15,00 | 112 | 111,37 | 79 | 76,809 | 98 | 81,559 | 5 | 5,00 |

|  |  | Algoritmo com recombinação uniforme + mutação por troca + reparação | | | | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | file1.txt | | file2.txt | | file3.txt | | file4.txt | | file5.txt | | Teste.txt | |
| Parâmetros Fixos | Parâmetros a variar | Best | MBF | Best | MBF | Best | MBF | Best | MBF | Best | MBF | Best | MBF |
| ger = 2500 | pr = 0.3 | 20 | 20,00 | 15 | 15,00 | 112 | 110,25 | 79 | 71,529 | 93 | 62,27 | 5 | 5,00 |
| pop = 100 | pr = 0.5 | 20 | 20,00 | 15 | 15,00 | 112 | 110,769 | 79 | 76,139 | 93 | 75,779 | 5 | 5,00 |
| pm = 0.01 | pr = 0.7 | 20 | 20,00 | 15 | 15,00 | 112 | 111,199 | 79 | 76,51 | 98 | 84,779 | 5 | 5,00 |
| ger = 2500  pop = 100  pr = 0.7 | pm = 0.0 | 20 | 20,00 | 15 | 15,00 | 112 | 110,13 | 79 | 75,04 | 98 | 82,26 | 5 | 5,00 |
| pm = 0.001 | 20 | 20,00 | 15 | 15,00 | 112 | 110,45 | 79 | 75,449 | 98 | 82,33 | 5 | 5,00 |
| pm = 0.01 | 20 | 20,00 | 15 | 15,00 | 112 | 110,93 | 79 | 76,61 | 98 | 83,169 | 5 | 5,00 |
| pm = 0.05 | 20 | 20,00 | 15 | 15,00 | 112 | 111,37 | 79 | 76,65 | 98 | 88,54 | 5 | 5,00 |
| pr = 0.7  pm = melhor valor obtido | pop = 10 (ger = 25K) | 20 | 20,00 | 15 | 15,00 | 112 | 111,93 | 79 | 75,599 | 95 | 67,94 | 5 | 5,00 |
| pop = 50 (ger = 5K) | 20 | 20,00 | 15 | 15,00 | 112 | 111,419 | 79 | 76,5 | 98 | 83,11 | 5 | 5,00 |
| pop = 100 (ger = 2.5K) | 20 | 20,00 | 15 | 15,00 | 112 | 111,33 | 79 | 76,82 | 98 | 87,949 | 5 | 5,00 |

**Análise dos resultados**

De acordo com os resultados apresentados, é possível observar que, em geral, o algoritmo evolutivo tem um bom desempenho para resolver o problema, conseguindo atingir as melhores soluções possíveis em vários casos.

É notável que o uso do torneio com tamanho 10, em alguns casos, pode levar a resultados ainda melhores, sendo mais constante em atingir a melhor solução possível e obtendo valores estatisticamente mais estáveis. No entanto, também é possível ver que a mudança de alguns parâmetros, como o aumento da probabilidade de mutação e o tamanho da população, pode afetar significativamente os resultados finais.

É interessante observar que a mudança na probabilidade de mutação tem um impacto significativo no desempenho do algoritmo. Quanto maior a probabilidade de mutação, mais diversidade é introduzida na população, o que pode levar a melhores soluções. No entanto, também é possível observar que uma probabilidade muito alta de mutação pode levar a soluções piores, provavelmente devido ao fato de que muitas mutações podem levar a soluções inválidas. Por outro lado, uma probabilidade muito baixa de mutação pode levar a um algoritmo menos eficiente, pois a população pode ficar presa em um ótimo local. Portanto, é importante encontrar um equilíbrio adequado entre a diversidade e a eficiência.

Outra análise que concluímos é que existem poucas diferenças significativas entre a utilização da mutação binária e a mutação por troca. Ambas as mutações parecem ter resultados semelhantes em todos os casos de teste, com exceção do File4.txt, onde a mutação por troca apresenta resultados ligeiramente melhores.

Ao comparar os resultados obtidos com a recombinação de 2 pontos de corte com os resultados obtidos com a recombinação de 1 ponto de corte, verificamos que a recombinação de 2 pontos de corte tende a ter resultados mais estáveis e consistentes em relação às melhores soluções possíveis para o problema. Além disso, a recombinação de 2 pontos de corte tende a ter valores de MBF ligeiramente mais altos, indicando que a solução encontrada é mais próxima da melhor solução possível. No entanto, é importante notar que os resultados finais podem variar dependendo dos outros parâmetros utilizados no algoritmo e do tamanho e complexidade do problema em questão.

**Algoritmo híbrido**

O algoritmo híbrido é um algoritmo que combina duas ou mais técnicas de solução de problemas diferentes para resolver um determinado problema.

Uma vantagem de se utilizar um algoritmo híbrido é que ele pode aproveitar os pontos fortes de cada algoritmo utilizado, o que pode resultar em uma solução mais eficiente para o problema em questão. No entanto, o uso de um algoritmo híbrido pode ser mais complexo e pode requerer mais tempo de processamento do que o uso de um único algoritmo.

O nosso algoritmo combina o algoritmo evolutivo com o algoritmo trepa-colinas. A estrutura do código inclui a definição de duas estruturas: *Individual* e *SimulationState*. A primeira é usada para armazenar informações sobre uma solução individual, enquanto a segunda é usada para armazenar informações sobre o estado atual da simulação. O código inclui funções para gerar uma população inicial, avaliar indivíduos e populações, selecionar os mais aptos para reprodução, recombinar indivíduos para formar novas soluções e mutar soluções. O algoritmo trepa-colinas é usado para tentar melhorar as soluções individuais ao longo do tempo. A função *code\_execution* é a função principal do código e é responsável por controlar o fluxo do algoritmo híbrido.

**Estratégia de validação de soluções inválidas**

Como mencionado no algoritmo evolutivo e no trepa colinas as implementações das soluções inválidas no algoritmo híbrido são as mesmas. As soluções iniciais da população são geradas aleatoriamente baseadas na variável *k*, sendo que caso não contenha o tamanho correto, gera automaticamente o valor 1 em posições aleatórias até se tornar uma solução válida. Esta implementação evita a necessidade da utilização de funções *flip* e por sua vez de funções de *reparação*.

**Problemas do Algoritmo**

Existem alguns problemas comuns que podem surgir ao se utilizar um algoritmo híbrido:

Complexidade: combinar vários algoritmos pode tornar o código mais complexo e difícil de entender. Isso pode ser um problema se você precisar depurar ou modificar o código mais tarde.

Desempenho: o desempenho de um algoritmo híbrido pode ser pior do que o de um algoritmo individual, pois há mais operações sendo realizadas. Isso pode ser especialmente verdadeiro se os algoritmos individuais tiverem desempenhos muito diferentes.

Dificuldade de otimização: ao combinar vários algoritmos, é mais difícil otimizar o desempenho do algoritmo híbrido. É preciso equilibrar as vantagens e desvantagens de cada um dos algoritmos individuais para obter o melhor resultado possível.

Dificuldade de escolha dos algoritmos: escolher os algoritmos certos para combinar pode ser um desafio. É preciso considerar como cada algoritmo se comporta em diferentes situações e quais são as suas vantagens e desvantagens.

Dificuldade de adaptação: um algoritmo híbrido pode ser menos adaptável do que um algoritmo individual, pois está dependente de vários outros algoritmos. Se um dos algoritmos individuais não for adequado para um determinado problema, isso pode afetar todo o algoritmo híbrido.

**Variantes**

A implementação do algoritmo híbrido, consiste na combinação dos operadores genéticos do algoritmo evolutivo com melhores resultados, juntamente com o algoritmo de pesquisa local ( Trepa Colinas ), isto com o objetivo de aumentar a precisão dos resultados, aumentar a complexidade e aleatoriedade das soluções geradas e ao mesmo tempo manter a estabilidade do algoritmo.

**Trepa colinas isolado + algoritmo evolutivo base + recombinação uniforme + mutação por troca + reparação + Torneio 2**

Ao combinar os dois algoritmos no híbrido, é possível obter as vantagens de ambos. O Trepa Colinas é utilizado para ajudar a encontrar rapidamente uma solução inicial na função *code\_execution*, que após serem geradas todas as soluções iniciais da população, é aplicado o algoritmo do trepa colinas em cada uma, enquanto o evolutivo, por outro lado, é um algoritmo de busca global que trabalha gerando uma população de soluções e aplicando operações de cruzamento e mutação para gerar novas soluções a partir das existentes, explorando uma ampla área do espaço de soluções.

Ao utilizar esta variante o processamento computacional necessário é o trepa colinas e o evolutivos juntos, mas a possibilidade de encontrar a melhor solução possível é maior.

**Algoritmo evolutivo base + recombinação uniforme + mutação por troca + reparação + Torneio 2 + trepa colinas por geração**

A diferença desta variante comparada com a do trepa colinas isolado é que o trepa colinas é executado em cada geração. Após cada geração são aplicadas as operações de cruzamento, torneio e mutações e é definido a melhor solução atual encontrada e de seguida executado o trepa colinas. Isto significa um processamento computacional muito mais elevado do que o trepa colinas isolado. Imaginando que temos 100 gerações, o trepa colinas é executado 100 vezes.

**Resultados experimentais**

|  | **Hibrido 1** | | **Hibrido 2** | |
| --- | --- | --- | --- | --- |
| **Ficheiro** | **Best** | **MBF** | **Best** | **MBF** |
| **file1** | **20** | **20,00** | **20** | **20,00** |
| **file2** | **15** | **15,00** | **15** | **15,00** |
| **file3** | **112** | **110.60** | **112** | **112,00** |
| **file4** | **79** | **76.14** | **79** | **79,00** |
| **file5** | **98** | **89,90** | **98** | **96,90** |

**Análise dos resultados**

Apesar dos resultados serem semelhantes para os 4 primeiros testes, para o *file5* é evidente a melhoria, conseguindo agora atingir constantemente a melhor solução possível. Podendo assim verificar o impacto do Trepa Colinas estar a ser chamado dentro de cada geração. Um ponto negativo que surge é o esforço computacional necessário para que sejam geradas as 100 soluções de forma a serem obtidos os resultados mais estatisticamente corretos.

Como previsto, o tempo necessário de computação no híbrido 2 é extremamente elevado, demorando com as configurações presentes nos resultados experimentais de 3 minutos por execução. Este tempo é extremamente elevado, mas concluímos que o MBF é também desta forma muito mais elevado do que o do híbrido normal e expectável.

**Conclusão**

Ao trabalhar com o Maximum Edge Subgraph Problem e a implementação de diferentes algoritmos para resolvê-lo, foi possível aprender sobre as vantagens e desvantagens de cada um deles. O algoritmo de trepa-colinas, por exemplo, é simples de implementar e rápido, mas tem a desvantagem de ficar preso em regiões locais ótimas e, portanto, pode não ser capaz de encontrar uma solução global ótima.

O algoritmo evolutivo, por outro lado, é mais robusto e capaz de explorar uma ampla gama de soluções, mas pode ser mais lento e exigir mais recursos computacionais.

Foi também importante perceber as possibilidades de implementação de diferentes algoritmos híbridos, que combinam as vantagens de diferentes algoritmos e podem ser mais eficientes em certas instâncias do problema. Por exemplo, um algoritmo híbrido que combina o algoritmo de trepa-colinas com o algoritmo evolutivo pode ser capaz de explorar uma ampla gama de soluções enquanto evita ficar preso em regiões locais ótimas.