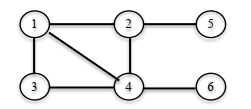
**Problema**

Foram-nos disponibilizados vários ficheiros com dados para caracterizar um determinado grafo.

O problema consiste em encontrar um subconjunto de k-vertices tal que o número de arestas dentro desse subconjunto seja máximo.

Na imagem seguinte, são representados os dados para leitura e o grafo respetivo.

Uma imagem com texto

Descrição gerada automaticamente

Para k = 4, considere três soluções possíveis

S1 = {1, 2, 5, 6} – Número de arestas = 2

S2 = {1, 2, 3, 4} – Número de arestas = 5

S3 = {2, 3, 5, 6} – Número de arestas = 1

**Algoritmo de Pesquisa Local (Trepa Colinas)**

O algoritmo trepa Colinas (“Hill Climbing”) vai procurar encontrar o subconjunto de k-vértices com o máximo de arestas possível. Para isso, parte de uma solução atual (conjunto de vértices) e explora as soluções vizinhas e ao comparar a qualidade das duas soluções, decide se atualiza ou não o estado atual. Aplicado a este problema, irão ser aceites soluções com qualidade melhor ou igual, tentando desta forma fugir a máximos locais e “planaltos”.

Este algoritmo procura encontrar soluções satisfatórias explorando os estados próximos ao estado inicial gerado.

**Problemas do algoritmo Trepa Colinas**

Este algoritmo pode ficar preso em um mínimo local/planalto, visto que este algoritmo considera apenas as soluções vizinhas e não explora todo o espaço de busca. O algoritmo em si, por vezes pode ser lento, visto que a solução inicialmente gerada tem muito impacto na busca pela solução ótima.

**Resultados experimentais**

**Trepa Colinas com Vizinhança 2**

Uma imagem com mesa

Descrição gerada automaticamente

Após análise dos resultados, é possível verificar que para o **Teste 2** e para o **Teste 3** que é apresentado o mesmo padrão de resultados: Sendo a melhor solução igual ao MBF, o que nos faz acreditar que para estes dois ficheiros, o algoritmo esteja a ficar preso em máximos locais.

Para o **Teste 4** podemos verificar que o algoritmo é bastante rápido a chegar á melhor solução possível, apenas com um pequeno desvio para as 100 interações.

Já no **teste 5** verificamos que para as 100 iterações, a gama de resultados é bastante discrepante, isto pode-se dever ao elevado número de vértices no grafo e a toda a aleatoriedade ao gerar a solução inicial.

**Trepa Colinas com Vizinhança 1**

Uma imagem com mesa

Descrição gerada automaticamente

Verificamos resultados semelhantes para os primeiros **4** testes mas ao contrário do que inicialmente esperávamos, para o **Teste 6** o máximo local foi ultrapassado , chegando agora á melhor solução possível e no **Teste 5** foram encontrados mais rapidamente resultados mais satisfatórios. O que nos leva a acreditar que sendo que na vizinhança 1 é apenas considerada uma alteração na solução atual, isto significa que ao ser mais restrita, por vezes leva a soluções melhores.

**Algoritmo Evolutivo**

**Para a implementação do algoritmo evolutivo, partimos dos projetos desenvolvidos no decorrer das aulas práticas.**

**Uma imagem com texto

Descrição gerada automaticamente**

**Uma imagem com texto

Descrição gerada automaticamente**

Estas duas estruturas vão armazenar tantos as soluções como também os dados principais para a implementação do algoritmo, sendo a maioria dos valores inicializados na função Main.

**Validação das soluções inválidas**

Inicialmente o algoritmo não aceita soluções inválidas, mas com a implementação da mutação e da reprodução, podem ser introduzidas soluções inválidas. Para corrigir temos a uma função reparação que vai validar se o número de vértices k é igual ao número de 1’s na solução.

**Funções mais relevantes**

**GenerateInitialPopulation**

Esta função vai inicializar uma população de indivíduos para simulação do algoritmo, para isso vai alocar memória para um array de indivíduos e vai e vai preencher cada elemento do array com uma solução inicial gerada pela função **generateInitialSolution**.

**Evaluate\_population**

Esta função é responsável por avaliar cada individuo da solução. Para cada solução é chamada a função ***evaluate\_individual*** que vai verificar se a solução é valida, se for chama a função ***calculate\_fit*** para calcular o fitness da solução. Se a solução for inválida, o fitness é definido como -1.

Depois de avaliar todos as soluções da população, é chamada função ***qsort*** que vai ordenar por ordem crescente de fitness. A função ***compare\_individual*** é passada como parâmetro para a função ***qsort*** e é usada para comparar duas soluções e determinar qual delas tem o melhor fitness. A função ***Evaluate\_Population*** recebe como parâmetro a matriz de adjacências para ser usada na função ***calculate\_fit*** no cálculo do fitness da solução.

**Vizinhança 1 e Vizinhança 2**

**Uma imagem com texto

Descrição gerada automaticamente**