Relatório Trabalho Prático

Guilherme Fidélis Freire/202320386
Hugo Prado Lima/202320987

Introdução

O objetivo deste trabalho é resolver uma variação do problema do Caixeiro Viajante (TSP - Traveling Salesman Problem), onde o foco principal não é minimizar a soma total das distâncias percorridas, mas sim minimizar a maior distância entre dois pontos consecutivos do percurso.

O código desenvolvido recebe como entrada o nome de um arquivo onde será gravado o percurso final, além dos pontos que devem ser visitados. Como saída, ele grava o percurso otimizado no arquivo especificado e imprime a maior distância entre dois pontos consecutivos na tela.

Esse problema tem aplicações práticas, como na otimização de rotas de transporte público, onde o objetivo é evitar trechos muito longos entre paradas de ônibus para melhorar a acessibilidade e eficiência do trajeto. Problemas dessa natureza são amplamente estudados na literatura e fazem parte da classe NP-difícil, sendo frequentemente abordados por heurísticas e meta-heurísticas devido à sua complexidade computacional (Gutin & Punnen, 2002).

Formulação

O problema pode ser representado como um grafo completo, onde:

- Cada ponto da instância é um vértice do grafo.
- Cada aresta representa a distância entre dois vértices.
- O peso da aresta é a distância entre os dois pontos e precisa ser calculado.

O objetivo é encontrar um ciclo Hamiltoniano (um percurso que visita cada ponto exatamente uma vez e retorna ao ponto inicial) onde a maior distância entre dois pontos consecutivos seja a menor possível.

Solução

Para resolver esse problema, exploramos diferentes abordagens:

1. Heurística do Vizinho Mais Próximo + 2-opt:
    - Inicialmente, utilizamos uma abordagem simples baseada na heurística do vizinho mais próximo para construir um percurso inicial.
    - Essa heurística funciona escolhendo sempre o próximo ponto mais próximo do atual.
    - Após obter o percurso, aplicamos a estratégia 2-opt, que tenta melhorar a solução invertendo segmentos do percurso para minimizar as maiores distâncias.
2. Implementação do GRASP (Greedy Randomized Adaptive Search Procedure):
    - Para melhorar a qualidade das soluções, implementamos o método GRASP, que permite explorar múltiplas soluções iniciais aleatórias em vez de seguir sempre a abordagem gulosa do vizinho mais próximo.
    - O GRASP modifica a construção do percurso inicial, iniciando-o de pontos aleatórios, aumentando a diversidade das soluções.
    - Em seguida, o refinamento com 2-opt é aplicado para otimizar cada solução encontrada.
    - Esse processo é repetido múltiplas vezes, e a melhor solução encontrada é mantida como resposta final.

O método GRASP é amplamente reconhecido na literatura como uma abordagem eficiente para resolver problemas combinatórios complexos, sendo aplicado com sucesso em variantes do TSP e problemas relacionados (Resende & Ribeiro, 2016).

Implementação

Bibliotecas Incluídas:

- **`<iostream>`**: Entrada e saída padrão.
- **`<vector>`**: Contêiner para vetores dinâmicos.
- **`<cmath>`**: Funções matemáticas (ex.: `sqrt`, `acos`, etc.).
- **`<limits>`**: Constantes e valores de limites para tipos numéricos.
- **`<fstream>`**: Manipulação de arquivos.
- **`<algorithm>`**: Funções padrão como `sort`, `reverse`, etc.
- **`<chrono>`**: Medição de tempo.
- **`<random>`**: Geração de números aleatórios.

Definições e Tipos:

- **`struct Ponto`**: Representa um ponto no espaço 2D com:
  - `id`: Identificador do ponto.
  - `x, y`: Coordenadas do ponto.
- **Função `calcularDistancia`**:
  - Calcula a distância euclidiana entre dois pontos.
- **`vector<vector<int>>`:**
  - Representa a matriz de distâncias entre pontos.

Variáveis e Estruturas recorrentes:

- **`vector<Ponto> pontos`**:
  - Armazena os pontos lidos da entrada.
- **`vector<vector<int>> distancias`**:
  - Matriz de distâncias entre pontos, calculada com `calcularMatrizDistancias`.
- **`vector<int> melhorPercurso`**:
  - Percurso que minimiza a maior distância entre dois pontos consecutivos.
- **`int menorMaiorDistancia`**:
  - Menor valor da maior distância entre dois pontos consecutivos em um percurso.

Funções

`calcularDistancia`

- **Descrição**: Calcula a distância euclidiana entre dois pontos.
- **Parâmetros**:
  - `const Ponto& a, const Ponto& b`: Pontos de entrada.
- **Retorno**:
  - Distância arredondada para o inteiro mais próximo.

`calcularMatrizDistancias`

- **Descrição**: Cria uma matriz contendo a distância entre todos os pares de pontos.
- **Parâmetros**:
  - `const vector<Ponto>& pontos`: Lista de pontos.
- **Retorno**:
  - Matriz de distâncias entre os pontos.

`maiorDistancia`

- **Descrição**: Calcula a maior distância entre dois pontos consecutivos em um percurso.
- **Parâmetros**:
  - `const vector<int>& percurso`: Ordem dos pontos no percurso.
  - `const vector<vector<int>>& distancias`: Matriz de distâncias.
- **Retorno**:
  - Maior distância entre dois pontos consecutivos no percurso.

`construirPercurso`

- **Descrição**: Constrói um percurso inicial baseado no GRASP.
- **Parâmetros**:
  - `const vector<Ponto>& pontos`: Lista de pontos.
  - `const vector<vector<int>>& distancias`: Matriz de distâncias.
  - `double alpha`: Controle da aleatoriedade.
- **Retorno**:
  - Um percurso inicial representado por uma lista de IDs de pontos.

`Opt`

- **Descrição**: Realiza a otimização do percurso usando a heurística 2-opt.
- **Parâmetros**:
  - `vector<int> percurso`: Percurso inicial a ser otimizado.
  - `const vector<vector<int>>& distancias`: Matriz de distâncias.
  - `int maxIter`: Número máximo de iterações.
- **Retorno**:
  - Percurso otimizado.

Main:

Descrição:

1. Lê os pontos da entrada padrão.
2. Calcula a matriz de distâncias entre os pontos.
3. Executa o algoritmo GRASP:
   - Gera soluções iniciais.
   - Otimiza cada solução com o 2-opt.
   - Mantém a melhor solução encontrada.
4. Grava o percurso otimizado em um arquivo de saída.
5. Imprime a maior distância no percurso final e o tempo total de execução.

Parâmetros Configuráveis

- **`int iteracoes = 10;`**
  - Número de soluções iniciais geradas pelo GRASP.
- **`double alpha = 0.3;`**
  - Controle da aleatoriedade na construção do percurso.
  - Valores menores são mais gulosos, valores maiores exploram mais possibilidades.
- **`int maxIterBuscaLocal = 100;`**
  - Número máximo de iterações da busca local (2-opt).

Obs: Como a primeira instância possui o tipo de distância diferente, sendo que a maioria das instâncias utiliza distância euclidiana e ela utiliza distância

geográfica, foi feito um código com alterações no cálculo de distâncias. Abaixo estão as explicações da partes diferentes:

Bibliotecas Incluídas:

As bibliotecas são as mesmas do código anterior, mas com destaque para:

- **`<cmath>`**: Usada para funções trigonométricas como `cos` e `acos`.

Definições e Constantes:

- **`const double RRR = 6378.388;`**
  - Define o raio da Terra em quilômetros.
- **`const double PI = 3.141592;`**
  - Define o valor de Pi para os cálculos trigonométricos.

Funções

`converterParaRadianos`

- **Descrição**: Converte uma coordenada geográfica no formato DDD.MM (graus e minutos) para radianos, necessários para o cálculo da distância geográfica.
- **Parâmetros**:
  - `double coordenada`: Coordenada no formato DDD.MM.
- **Retorno**:
  - Valor da coordenada convertido para radianos.

`calcularDistancia`

- **Descrição**: Calcula a distância geográfica entre dois pontos na superfície da Terra, considerando o modelo esférico idealizado com raio de 6378.388 km.
- **Parâmetros**:
  - `const Ponto& a`: Ponto de origem com coordenadas latitude e longitude.
  - `const Ponto& b`: Ponto de destino com coordenadas latitude e longitude.
- **Retorno**:
  - Distância entre os pontos arredondada para o inteiro mais próximo.

Resultados

Os testes foram realizados em um notebook de configurações: processador Intel I5-1135G7 2.42GHz e com 8 GB de RAM.

Abaixo estão os resultados dos testes feitos com o código inicial, utilizando a heurística de vizinho mais próximo junto com a estratégia de 2-opt, e dos testes feitos com a solução final implementando o método Grasp e algumas melhorias.

Instância	Valor Solução Inicial (SI)	Tempo computacional SI (s)	Valor Solução Final (SF)	Tempo computacional SF (s)	Desvio percentual de SF para SI	Valor Referência	Desvio percentual de SF para referência
1	10769	4	5114	2	52,51	3986	-28,30
2	1579	38	1289	45	18,37	1289	0,00
3	1547	22	1476	79	4,59	1476	0,00
4	1133	55	1133	172	0,00	1133	0,00
5	662	13	662	47	0,00	546	-21,25
6	585	25	552	68	5,64	431	-28,07
7	348	13	135	48	61,21	219	38,36
8	287	8	243	27	15,33	266	8,65
9	31	2	36	8	-16,13	52	30,77
10	349	38	234	107	32,95	237	1,27

Inicialmente a solução apresentou alguns resultados bons em relação aos valores de referência, porém em outros casos a resolução não se mostrava eficiente devido o algoritmo de vizinho mais próximo combinado ao 2-opt, considerava o melhor caso local.

A estratégia do GRASP resultou em uma melhoria significativa na solução encontrada, permitindo minimizar ainda mais a maior distância entre dois pontos. No entanto, isso teve um custo computacional, pois o tempo de execução aumentou devido à necessidade de gerar e refinar múltiplas soluções.

Conclusão

Este trabalho abordou uma versão modificada do Problema do Caixeiro Viajante, priorizando a minimização da maior distância entre dois pontos consecutivos. A heurística inicial baseada no vizinho mais próximo e 2-opt obteve bons resultados, mas apresentava limitações.

A implementação do GRASP permitiu uma melhoria significativa ao testar múltiplas soluções iniciais aleatórias, resultando em percursos otimizados. O principal desafio foi o aumento do tempo de execução, mas a qualidade da solução final se mostrou superior na maioria dos casos.

Referências bibliográficas

Gutin, G., & Punnen, A. P. (2002). **The Traveling Salesman Problem and Its Variations**. Springer.
Resende, M. G. C., & Ribeiro, C. C. (2016). **Optimization by GRASP: Greedy Randomized Adaptive Search Procedures**. Springer.