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, implementando uma estratégia de buscar os próximos pontos com um pouco de aleatoriedade, para que diferentes caminhos sejam abordados.
    - 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.

Constantes:

- `const double RRR = 6378.388;`: Raio médio da Terra em quilômetros.
- `const double PI = 3.141592;`: Valor de Pi.

Funções Auxiliares:

- `auxLeitura`: Remove espaços em branco extras de uma string.
- `converterParaRadianos`: Converte coordenadas de graus para radianos.
- `calcularDistanciaEuclidiana`: Calcula a distância euclidiana entre dois pontos.
- `calcularDistanciaGeografica`: Calcula a distância geográfica entre dois pontos, considerando a curvatura da Terra.

Funções Principais:

calcularMatrizDistancias

- **Descrição**: Cria uma matriz contendo a distância entre todos os pares de pontos com base no tipo de métrica especificada (`EUC_2D` para distância euclidiana e `GEO` para distância geográfica).
- **Parâmetros**:
  - `const vector<Ponto>& pontos`: Lista de pontos.
  - `const string& tipoPeso`: Tipo de métrica de distância.
- **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.

construirPercursoGRASP

- **Descrição**: Constrói um percurso inicial baseado no algoritmo 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**:
  - Lê os pontos da entrada padrão.
  - Calcula a matriz de distâncias entre os pontos.
  - Executa o algoritmo GRASP:
    - Gera soluções iniciais.
    - Otimiza cada solução com o 2-opt.
    - Mantém a melhor solução encontrada.
  - Grava o percurso otimizado em um arquivo de saída.
  - 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).

Resultados

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

A tabela a seguir apresenta os resultados dos testes feitos com a implementação do método GRASP. A solução inicial é obtida pelo primeiro percurso gerado pelo código, sem a aplicação da estratégia 2-opt. A solução final é obtida após o algoritmo executar todas as iterações do GRASP, refinando o percurso inicial e abordando diferentes caminhos. O algoritmo armazena e retorna o percurso onde a maior distância entre dois pontos consecutivos é minimizada.

Instância	Valor Solução Inicial (SI)	Tempo computacional SI (ms)	Valor Solução Final (SF)	Tempo computacional SF (ms)	Desvio percentual de SF para SI (%)	Valor Referência	Desvio percentual de SF para referência (%)
1	18879	267	5114	2931	72,91	3986	-28,30
2	4929	3638	1289	40658	73,85	1289	0,00
3	4786	10139	1476	96653	69,16	1476	0,00
4	5204	20680	1133	215148	78,23	1133	0,00
5	2882	4692	651	52619	77,41	546	-19,23
6	1953	8032	503	78918	74,24	431	-16,71
7	2685	5237	130	51954	95,16	219	40,64
8	3047	2649	243	28423	92,02	266	8,65
9	574	798	32	8340	94,43	52	38,46
10	2693	11027	255	120637	90,53	237	-7,59

A partir dos testes, é possível observar que a construção do primeiro percurso inicial não é eficiente, pois resulta em valores altos para a maior distância entre dois pontos consecutivos.

Porém após as iterações do algoritmo GRASP, a solução final apresenta uma melhoria significativa para a sua solução final, reduzindo drasticamente a maior distância entre dois pontos do percurso.

Conclusão

Este trabalho abordou uma variação do Problema do Caixeiro Viajante, priorizando a minimização da maior distância entre dois pontos consecutivos do percurso.

A implementação do método GRASP permitiu explorar múltiplas soluções iniciais com um pouco de aleatoriedade e aplicar refinamentos,

resultando em percursos otimizados na maioria dos casos. Embora o tempo de execução tenha aumentado consideravelmente para instâncias maiores, a qualidade das soluções finais foi superior em relação às soluções iniciais.

Os resultados indicam que a abordagem GRASP é eficiente para resolver esse tipo de problema, especialmente quando combinada com técnicas de refinamento como o 2-opt. No entanto, futuras melhorias podem considerar técnicas adicionais para reduzir o tempo de execução, como a paralelização do algoritmo ou a adoção de outras heurísticas e meta-heurísticas.

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.