Heurística para o Problema do Caixeiro Viajante Assimétrico (PCVA)

Prof. Mateus Costa

27 de janeiro de 2025

O Problema do Caixeiro Viajante Assimétrico (PCVA)

- O PCVA é uma variação do Problema do Caixeiro Viajante (PCV), onde o custo de viajar de uma cidade i para outra cidade j pode ser diferente do custo de viajar de j para i.
- É um problema de otimização combinatória com aplicações em logística, roteamento e planejamento.
- Objetivo: Determinar a rota de menor custo que visita todas as cidades uma única vez e retorna à cidade de origem.

Heurística Desenvolvida

- Baseada no algoritmo de Johnson-McGeogh.
- Utiliza a heurística construtiva do **vizinho mais próximo** para construir uma rota inicial.
- Aplica a heurística de busca local **3-Opt** para melhorar a solução inicial.
- Complexidade: $O(N^3)$, onde N é o número de cidades.

Passos do Algoritmo

1. Construir uma rota inicial utilizando a heurística do **vizinho mais próximo**:
   - Para cada cidade inicial, construa uma rota considerando sempre a cidade mais próxima ainda não visitada.
   - Selecione a rota de menor custo como solução inicial.
2. Aplicar a heurística **3-Opt**:
   - Remova 3 arestas da rota.
   - Reorganize os caminhos resultantes de forma a reduzir o custo total.
   - Repita até que nenhuma melhoria adicional seja possível.

Construção da Rota Inicial

```
public List<Vertex> nearestNeighbor(
    DirectedSparseGraph<Vertex, Edge> graph, Vertex
    start) {
    List<Vertex> route = new ArrayList<>();
    Set<Vertex> visited = new HashSet<>();
    Vertex current = start;
    visited.add(current);
    route.add(current);

    while (visited.size() < graph.getVertexCount()) {
        Vertex next = null;
        double minCost = Double.MAX_VALUE;

        for (Edge edge : graph.getOutEdges(current)) {
            Vertex neighbor = graph.getOpposite(
                current, edge);
            if (!visited.contains(neighbor) && edge.
                getWeight() < minCost) {
                minCost = edge.getWeight();
                next = neighbor;
            }
        }
```

Construção da Rota Inicial

```
            if (next == null) break;

            visited.add(next);
            route.add(next);
            current = next;
        }

        route.add(start); // Complete the cycle
        return route;
    }
```

Heurística 3-Opt

```
public List<Vertex> threeOpt(List<Vertex> route,
    DirectedSparseGraph<Vertex, Edge> graph) {
     boolean improvement = true;
     while (improvement) {
          improvement = false;
          for (int i = 0; i < route.size() - 3; i++) {
               for (int j = i + 2; j < route.size() - 1;
                    j++) {
                    for (int k = j + 2; k < route.size();
                         k++) {
                         double originalCost =
                              calculateRouteCost(route,
                              graph);
                         List<Vertex> newRoute =
                              apply3OptSwap(route, i, j, k);
                         double newCost =
                              calculateRouteCost(newRoute,
                              graph);
```

Construção da Rota Inicial

```
                        if (newCost < originalCost) {
                            route = newRoute;
                            improvement = true;
                        }
                    }
                }
            }
        }
        return route;
    }
    private double calculateRouteCost(List<Vertex> route,
        DirectedSparseGraph<Vertex, Edge> graph) {
        double cost = 0.0;
        for (int i = 0; i < route.size() - 1; i++) {
            Edge edge = graph.findEdge(route.get(i), route
                .get(i + 1));
            if (edge != null) {
                cost += edge.getWeight();
            }
        }
        return cost;
    }
```

Construção da Rota Inicial

```
private List<Vertex> apply3OptSwap(List<Vertex> route,
     int i, int j, int k) {
    List<Vertex> newRoute = new ArrayList<>();
    newRoute.addAll(route.subList(0, i + 1));
    newRoute.addAll(route.subList(j + 1, k + 1));
    Collections.reverse(newRoute.subList(i + 1,
        newRoute.size()));
    newRoute.addAll(route.subList(k + 1, route.size())
        );
    return newRoute;
}
```

Conjunto de Testes

- Foram realizados testes utilizando as matrizes disponíveis no site TSPLIB.
- Exemplos de instâncias testadas:

Instância	**n**	**Ótimo**	**Heurística**	**Limite Superior**
br17.atsp	17	39	40	80.5
ftv33.atsp	33	1286	1457	3257.49
ftv53.atsp	53	6905	8462	19822.41
ft70.atsp	70	38673	41815	118717.42
ftv64.atsp	64	1839	2202	5547.73
rbg443.atsp	443	2720	2860	11958.19

Tabela: Resultados dos Testes com a TSPLIB

Resultados Detalhados

Teste: br17.atsp.gz

- Melhor resultado: 39
- Limite superior da heurística: 80.5
- Custo da menor rota antes da busca local: 56
- Custo da menor rota com a busca local: 40

Rota:

3 4 5 6 14 15 0 11 10 9 1 13 2 12 7 8 16

Resultados Detalhados

Teste: ftv33.atsp.gz

- Melhor resultado: 1286
- Limite superior da heurística: 3257.49
- Custo da menor rota antes da busca local: 1504
- Custo da menor rota com a busca local: 1457

Rota:

16 31 27 22 8 15 29 24 9 21 10 19 4 3 28 23 12 25 26 18 2 5 11 1 0 14 30 20 6 13 17 7 32

Resultados Detalhados

Teste: ftv53.atsp.gz

- Melhor resultado: 6905
- Limite superior da heurística: 19822.41
- Custo da menor rota antes da busca local: 8584
- Custo da menor rota com a busca local: 8462

Rota:

19 18 17 16 15 52 50 51 48 49 29 28 25 26 27 7 5 8 6 9 33 31 30 0 3 2 1 41 43 42 46 45 44 34 32 21 20 39 35 40 37 36 10 12 14 13 11 38 4 22 47 23 24

Estímulo para os Estudantes

- Os testes realizados no passado com a TSPLIB fornecem um benchmark importante.
- Os estudantes podem implementar a heurística em Java e comparar os resultados com os apresentados nesta tabela.
- Proposta: Testar instâncias menores para validar a implementação e, em seguida, executar em instâncias maiores, como `rbg443.atsp`.

Conclusão

- A heurística combina a simplicidade do **vizinho mais próximo** com a potência do **3-Opt**.
- Implementação em Java utilizando a biblioteca JUNG permite flexibilidade e extensibilidade.
- A abordagem pode ser expandida para outros problemas relacionados a roteamento.

Trabalho Prático Final

- Implementar uma heurística para o PCVA.
- Testar a heurística com os arquivos do TSPLIB.
- Mostrar os resultados graficamente.
- Comparar os resultados obtidos com os apresentados nestes slides.
- Elaborar um relatório em PDF contendo:
  - Algoritmos.
  - Código fonte.
  - Resultados dos testes.
- Base para o trabalho: TSPLIB e Heurística para PCVA.