Approximation Algorithms and Local Search for CVRP

CSCI2951O: Foundations of Prescriptive Analytics

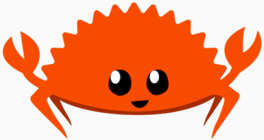
Hammad Izhar (carlisle)

May 13th, 2024

Brown University

Implementation Overview

- ∼700 lines of Rust code
- TSP Approximation Algorithm to determine an initial feasible solution
- Naive Local Search Heuristics



Christofides–Serdyukov Algorithm

Christofides–Serdyukov Algorithm

- $\frac{3}{2}$-Approximation Algorithm for TSP

Christofides–Serdyukov Algorithm

- $\frac{3}{2}$-Approximation Algorithm for TSP

```
pub fn chirstofides(&self) -> Vec<ClientId> {
  // Step 1: Compute the minimum spanning tree of the graph
  let mst = self.mst();
  let odd_clients = extract_odd_degree_vertices(&mst);
```

Christofides–Serdyukov Algorithm

- $\frac{3}{2}$-Approximation Algorithm for TSP

```
pub fn chirstofides(&self) -> Vec<ClientId> {
  // Step 1: Compute the minimum spanning tree of the graph
  let mst = self.mst();
  let odd_clients = extract_odd_degree_vertices(&mst);
  // Step 2: Find a minimum weight perfect matching of the odd degree vertices
  let matching = self.find_minimum_weight_matching(&odd_clients);
```

Christofides–Serdyukov Algorithm

- $\frac{3}{2}$-Approximation Algorithm for TSP

```
pub fn chirstofides(&self) -> Vec<ClientId> {
  // Step 1: Compute the minimum spanning tree of the graph
  let mst = self.mst();
  let odd_clients = extract_odd_degree_vertices(&mst);
  // Step 2: Find a minimum weight perfect matching of the odd degree vertices
  let matching = self.find_minimum_weight_matching(&odd_clients);
  // Step 3: Find an Eulerian tour of the MST-matching multigraph
  let eulerian_tour =
    VehicleRoutingGraph::find_eulerian_tour(&mst, &matching);
```

Christofides–Serdyukov Algorithm

- $\frac{3}{2}$-Approximation Algorithm for TSP

```rust
pub fn chirstofides(&self) -> Vec<ClientId> {
  // Step 1: Compute the minimum spanning tree of the graph
  let mst = self.mst();
  let odd_clients = extract_odd_degree_vertices(&mst);
  // Step 2: Find a minimum weight perfect matching of the odd degree vertices
  let matching = self.find_minimum_weight_matching(&odd_clients);
  // Step 3: Find an Eulerian tour of the MST-matching multigraph
  let eulerian_tour =
    VehicleRoutingGraph::find_eulerian_tour(&mst, &matching);
  // Step 4: Convert the Eulerian tour into a Hamiltonian circuit
  let circuit =
    VehicleRoutingGraph::convert_eulerian_tour_to_circuit(&eulerian_tour);
  circuit
}
```

Christofides–Serdyukov Algorithm

- $\frac{3}{2}$-Approximation Algorithm for TSP

```
pub fn chirstofides(&self) -> Vec<ClientId> {
  // Step 1: Compute the minimum spanning tree of the graph
  let mst = self.mst();
  let odd_clients = extract_odd_degree_vertices(&mst);
  // Step 2: Find a minimum weight perfect matching of the odd degree vertices
  let matching = self.find_minimum_weight_matching(&odd_clients);
  // Step 3: Find an Eulerian tour of the MST-matching multigraph
  let eulerian_tour =
    VehicleRoutingGraph::find_eulerian_tour(&mst, &matching);
  // Step 4: Convert the Eulerian tour into a Hamiltonian circuit
  let circuit =
    VehicleRoutingGraph::convert_eulerian_tour_to_circuit(&eulerian_tour);
  circuit
}
```

- Runs in $\mathcal{O}(mn^2)$ time, where n is the the number of vertices in the graph and m is the number of edges

Christofides–Serdyukov Algorithm

- $\frac{3}{2}$-Approximation Algorithm for TSP

```rust
pub fn chirstofides(&self) -> Vec<ClientId> {
  // Step 1: Compute the minimum spanning tree of the graph
  let mst = self.mst();
  let odd_clients = extract_odd_degree_vertices(&mst);
  // Step 2: Find a minimum weight perfect matching of the odd degree vertices
  let matching = self.find_minimum_weight_matching(&odd_clients);
  // Step 3: Find an Eulerian tour of the MST-matching multigraph
  let eulerian_tour =
    VehicleRoutingGraph::find_eulerian_tour(&mst, &matching);
  // Step 4: Convert the Eulerian tour into a Hamiltonian circuit
  let circuit =
    VehicleRoutingGraph::convert_eulerian_tour_to_circuit(&eulerian_tour);
  circuit
}
```

- Runs in $\mathcal{O}(mn^2)$ time, where n is the the number of vertices in the graph and m is the number of edges
- Limited by the runtime of determining a perfect matching

Minimum Spanning Tree Algorithm

For dense graphs

Minimum Spanning Tree Algorithm

For dense graphs, Prim's algorithm computes MST faster compared to Kruskal's

Minimum Spanning Tree Algorithm

For dense graphs, Prim's algorithm computes MST faster compared to Kruskal's

```
fn mst() {
  let mut included_clients = HashSet::new([0]);
  let mut pq = PriorityQueue::new();
  let mut tree = Vec::new();
  // Push into pq all of the incident edges to the depot
  ...
  while included_clients.len() < self.clients.len() {
      let (edge, _) = pq.pop().unwrap();
      // The fringe vertex we include by adding this edge
      let new_client = ...
      ...
      tree.push(edge);
      for client in 0..self.clients.len() {
        // Add edges to vertices not yet visited in the tree
      }
  }
  tree
}
```

Minimum Weight Perfect Matching

Minimum Weight Perfect Matching

Claim: The number of vertices with odd degree in the minimum spanning tree is even

Minimum Weight Perfect Matching

Claim: The number of vertices with odd degree in the minimum spanning tree is even

Proof.
By the Handshaking Lemma,

$$\sum_{v \in V(G)} \deg v = 2|E(G)|$$

□

Minimum Weight Perfect Matching

Claim: The number of vertices with odd degree in the minimum spanning tree is even

Proof.
By the Handshaking Lemma,

$$\sum_{v \in V(G)} \deg v = 2|E(G)|$$

□

- For complete graphs, this can be done in $\mathcal{O}(mn^2)$ time using the Blossom Algorithm

Minimum Weight Perfect Matching

Claim: The number of vertices with odd degree in the minimum spanning tree is even

Proof.
By the Handshaking Lemma,

$$\sum_{v \in V(G)} \deg v = 2|E(G)|$$

$\square$

- For complete graphs, this can be done in $\mathcal{O}(mn^2)$ time using the Blossom Algorithm
  - At a high-level, find augmenting paths while contracting odd cycles (a.k.a blossoms) along the way

Minimum Weight Perfect Matching

Claim: The number of vertices with odd degree in the minimum spanning tree is even

Proof.
By the Handshaking Lemma,

$$\sum_{v \in V(G)} \deg v = 2|E(G)|$$

$\square$

- For complete graphs, this can be done in $\mathcal{O}(mn^2)$ time using the Blossom Algorithm
    - At a high-level, find augmenting paths while contracting odd cycles (a.k.a blossoms) along the way
- Can be viewed as a generalization of the Ford-Fulkerson Max-Flow algorithm to find maximum matchings in bipartite graphs

Eulerian Tour (Hierholzer's) Algorithm

```
pub fn find_eulerian_tour(mst: &[Edge], matching: &[Edge]) -> Vec<ClientId> {
    let mut vertex_to_edges = HashMap::new();
    // Associate each edge to its endpoints
    ...
    let mut eulerian_tour = Vec::new();
    let mut stack = vec![mst[0].first];
    while !stack.is_empty() {
        let edges = vertex_to_edges.get_mut(stack.last()).unwrap();
        if edges.is_empty() {
            eulerian_tour.push(stack.pop().unwrap());
        } else {
            let edge = *edges.iter().next().unwrap();
            // The next vertex in the path
            let next_vertex = ...
            stack.push(next_vertex);
        }
    }
    eulerian_tour
}
```

Shortcutting and Partitioning

- The computed Eulerian tour most likely revisits vertices many times, however, our TSP (and VRP) tours should not!

Shortcutting and Partitioning

- The computed Eulerian tour most likely revisits vertices many times, however, our TSP (and VRP) tours should not!
- Since our graph is complete, we can move from any one vertex to any other vertex

Shortcutting and Partitioning

- The computed Eulerian tour most likely revisits vertices many times, however, our TSP (and VRP) tours should not!
- Since our graph is complete, we can move from any one vertex to any other vertex
- Since we are working in Euclidean space, by the triangle inequality shortcutting our Eulerian tour isn't increasing the length of the tour!

Shortcutting and Partitioning

- The computed Eulerian tour most likely revisits vertices many times, however, our TSP (and VRP) tours should not!
- Since our graph is complete, we can move from any one vertex to any other vertex
- Since we are working in Euclidean space, by the triangle inequality shortcutting our Eulerian tour isn't increasing the length of the tour!
- The TSP tour isn't a valid routing plan for our vehicles

Shortcutting and Partitioning

- The computed Eulerian tour most likely revisits vertices many times, however, our TSP (and VRP) tours should not!
- Since our graph is complete, we can move from any one vertex to any other vertex
- Since we are working in Euclidean space, by the triangle inequality shortcutting our Eulerian tour isn't increasing the length of the tour!
- The TSP tour isn't a valid routing plan for our vehicles
- The instances are too constrained to use a naive partitioning approach

Shortcutting and Partitioning

- The computed Eulerian tour most likely revisits vertices many times, however, our TSP (and VRP) tours should not!
- Since our graph is complete, we can move from any one vertex to any other vertex
- Since we are working in Euclidean space, by the triangle inequality shortcutting our Eulerian tour isn't increasing the length of the tour!
- The TSP tour isn't a valid routing plan for our vehicles
- The instances are too constrained to use a naive partitioning approach
- We pack the clients into k "bins" where k is the number of vehicles we have access to using a first-fit strategy

Shortcutting and Partitioning

- The computed Eulerian tour most likely revisits vertices many times, however, our TSP (and VRP) tours should not!
- Since our graph is complete, we can move from any one vertex to any other vertex
- Since we are working in Euclidean space, by the triangle inequality shortcutting our Eulerian tour isn't increasing the length of the tour!
- The TSP tour isn't a valid routing plan for our vehicles
- The instances are too constrained to use a naive partitioning approach
- We pack the clients into k "bins" where k is the number of vehicles we have access to using a first-fit strategy
- Works for all but a couple of instances! For these remaining couple, a random solution is generated

Local Search Heuristics

- Inter-Route Swap Heuristic picks two random clients in two different and swaps them

Local Search Heuristics

- Inter-Route Swap Heuristic picks two random clients in two different and swaps them
- The vertices are placed in the position that minimizes the cost of the new route

Local Search Heuristics

- Inter-Route Swap Heuristic picks two random clients in two different and swaps them
- The vertices are placed in the position that minimizes the cost of the new route
- Only neighboring solutions with strictly better costs are ever accepted

Local Search Heuristics

- Inter-Route Swap Heuristic picks two random clients in two different and swaps them
- The vertices are placed in the position that minimizes the cost of the new route
- Only neighboring solutions with strictly better costs are ever accepted
- Runs for all remaining available time