Kristin Schaefer

HW8 : NP-Completeness and Heuristic Algorithms

Sources:

Problem 1
1. CLRS, Section 34.5.4
2. https://en.wikipedia.org/wiki/Complete_graph
3. https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/sat.pdf
Problem 2
4. http://www.cs.toronto.edu/~ashe/ham-path-notes.pdf
5. http://personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/AproxAlgor/TSP/tsp.htm

Problem 1.

Given a graph G with V vertices and E edges, determine if the graph has a TSP solution consisting of a cost k.
Prove that the above stated problem is NP-Complete.

1. Show that TSP belongs to NP.

Given an instance of a TSP problem where its graph G has V vertices and E edges, we can verify that a sequence of V vertices in the tour contains each vertex exactly once, sums the cost of the edges, and checks whether the sum is at most k. To verify that a sequence of V vertices in the tour contains each vertex exactly once will take O(V) since we only need to verify that there are no duplicate vertices and that each vertex exists in the tour. To sum the cost of the edges and check if the sum equals at most k is O(E), since in the worst case we need to add all of the edges in the graph G. Hence, it takes O(V+E), so this process can be done in polynomial time and thus TSP ∃ NP.

2. Show that TSP is NP-Hard

To show that TSP is NP-Hard, we will reduce Hamiltonian cycle to TSP, so Hamiltonian cycle $\leq_p$ TSP. Hamiltonian cycle reduces to TSP because the output of TSP is the shortest tour of it's n vertices, which is thus the minimum Hamiltonian cycle of the n vertices.

Now take a graph G = (V, E) that is an instance of Hamiltonian cycle, and let us then construct a complete graph G' = (V, E') that is an instance of TSP. Note that a complete graph is defined as a simple, undirected graph in which every pair of distinct vertices is connected by a unique edge.

For the graph G', we can say that:

$E' = \{(i, j) : i, j \exists V, i \neq j\}$

We will define cost as:

$c(i, j) = 0$, if $(i, j) \exists E$

$c(i, j) = 1$, if $(i, j) \nexists E$

Graph G has a Hamiltonian cycle iff G' has a tour with a cost that is 0. Now suppose that graph G has a Hamiltonian cycle *h*. Since graph G is undirected and has no self-loops, then each edge in *h* ∃ E. Thus the total cost of *h* in G' is 0, so we can say that G' has a tour *h* with cost 0.

Now suppose that G' has a tour *h'* with a cost that is at most 0. Then each edge of *h'* must also be 0, so then all of the edges of *h'* ∃ E. Therefore *h'* is a Hamiltonian cycle in graph G.

Thus TSP is NP-Hard.

As we have proven that TSP belongs to NP and is NP-Hard, then TSP is NP-Complete.

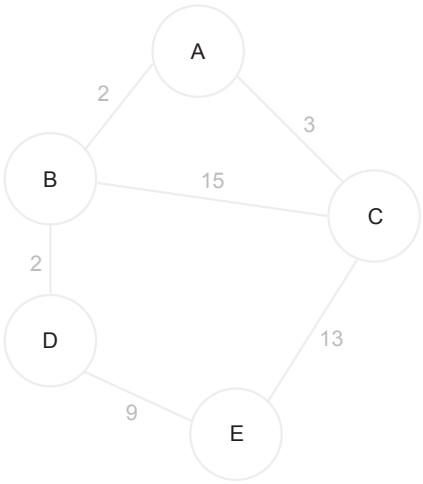
Kristin Schaefer

HW8 : NP-Completeness and Heuristic Algorithms

Problem 2.

	A	B	C	D	E
A	0	2	3	0	0
B	2	0	15	2	0
C	3	15	0	0	13
D	0	2	0	0	9
E	0	0	13	9	0

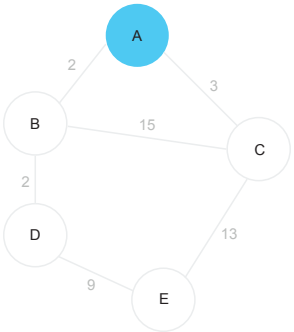
matrix which represents the distance of 5 cities



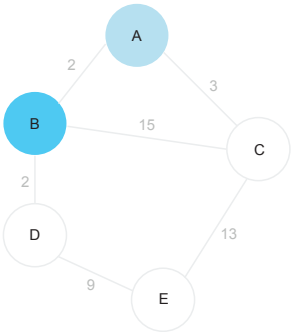
matrix represented in the form of a graph

b. Nearest-neighbour heuristic / closest-point heuristic

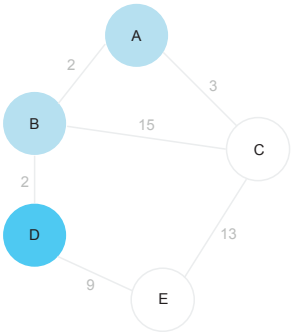
step 1: pick an arbitrary start point
visited: A
total cost: 0



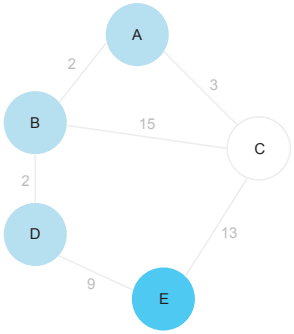
step 2: pick B
visited: A -> B
total cost: 0 + 2 = 2



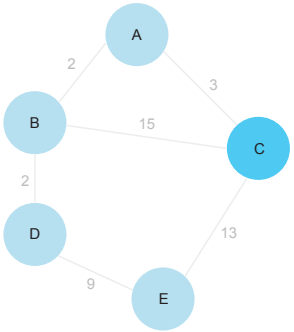
step 3: pick D
visited: A -> B -> D
total cost: 0 + 2 + 2 = 4



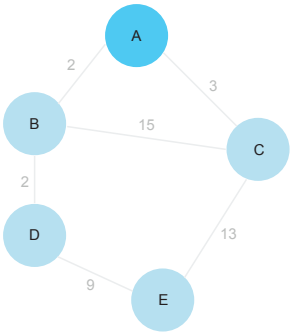
step 4: pick E
visited: A -> B -> D -> E
total cost: 0 + 2 + 2 + 9 = 13



step 5: pick C
visited: A -> B -> D -> E -> C
total cost: 0 + 2 + 2 + 9 + 13 = 26



step 6: return to A
visited: A -> B -> D -> E -> C -> A
total cost: 0 + 2 + 2 + 9 + 13 + 3 = 29



Kristin Schaefer

HW8 : NP-Completeness and Heuristic Algorithms

Problem 2.

c. Accuracy ratio of the approximate solution

C = 29

C* = 29

$\rho(n)$ = C/C* = 29/29 = 1

d. Pseudocode for nearest-neighbour heuristic

```
nearest_neighbour_tsp (G) :

        # create a set to store unvisited vertices
        unvisited = {vertices in G}

        # create a list to store the path of visited vertices
        visited_path = [ ]

        # select a random starting vertex and update visited and unvisited sets
        starting_vertex = select a random vertex v in G
        add starting_vertex to visited
        remove starting_vertex from visited_path

        current_vertex = null
        total_cost = 0

        while unvisited:
                # initialize current_vertex to starting vertex at beginning of path
                if current vertex == null:
                        current_vertex = starting vertex

                # find nearest neighbour
                from the vertices connected to current_vertex which are not in visited_path[ ] find the vertex v with the minimum edge cost k
                append v to visited_path[]
                remove v from unvisited
                add cost k of edge (current_vertex, v) to total_cost
                current_vertex = v

        # return to starting point
        append starting_vertex to visited_path[ ]

        return visited_path[ ], total_cost
```