Maze Simulation and Analysis of Graph Algorithms

Aman Gupta and Arpit Chauhan

A report submitted for the course
COP290 Design Practices
Supervised by: Prof. Rijurekha Sen
Indian Institute of Technology Delhi

May 2021

Except where otherwise indicated, this report is our own original work.

Aman Gupta and Arpit Chauhan
21 May 2021

Abstract

Simulation of real world events is an important task of computers, and in this subtask we simulate something in a maze environment. We place 6 infinity stones at random positions in a randomly generated maze and then try to find the shortest walk picking up all the stones . This is a special kind of Travelling Salesman Problem called Steiner TSP where a subset of nodes needs to be visited using the shortest possible path. In this task , along with the detailed analysis and simulation of Travelling Salesman Problem , we also explored Steiner TSP and Close Enough TSP both of which can either be reduced to a smaller TSP or be solved using other heuristics some of which have been discussed here .These Problems find a lot of real world applications and hence obtaining an optimal solution is of prime importance.

Contents

1. Travelling Salesman Problem

We are given a list of cities and distances between each pair of cities. Our goal is to find a path that covers all the cities and covering the least distance. We need to start and end the path at the same city.

1.1 Introduction

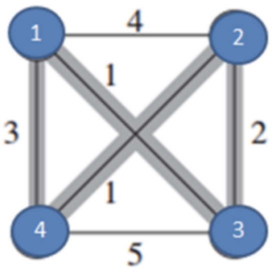
We are given a complete undirected graph G = (V,E) that has a non-negative integer cost c(u,v) associated with each edge (u,v) $\in$ E, and we want ro find a **hamiltonian cycle** (a tour) of G with minimum cost. As an extension of our notation, let c(A) denote the total cost of the edges in the subset A $\subseteq$ E:

$$c(A) = \sum_{(u,v)\in A} c(u,v) \ .$$

In most practical situations the cost function c(u,v) satisfies the triangle inequality,

$$c(u,w) \le c(u,v) + c(v,w) \ .$$

1.2 Graphical Representation



$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 4 & 1 & 3 \\ 2 & 4 & 0 & 2 & 1 \\ 3 & 1 & 2 & 0 & 5 \\ 4 & 3 & 1 & 5 & 0 \end{array}$$

An adjacency matrix data structure is used to represent the cost function between any two nodes in a TSP problem. All the diagonal elements are shown as zero because there is no self-loop from one node to itself.

1.3 Heuristics/Approach to get a near-optimal solution of TSP

The traveling-salesman problem is **NP-complete** even if we require that the cost function satisfy the triangle inequality. Thus, we should not expect to find a polynomial-time algorithm for solving this problem exactly. Instead, we look for good approximation algorithms.

1.3.1 Brute Force / Naive Approach

This method involves looking at all possible cycles to connect all the nodes. It iterates over all cycles, calculates the total cost function in each cycle, and then finds the global minimum of these total weights to obtain the most optimal solution.

Run Time Analysis

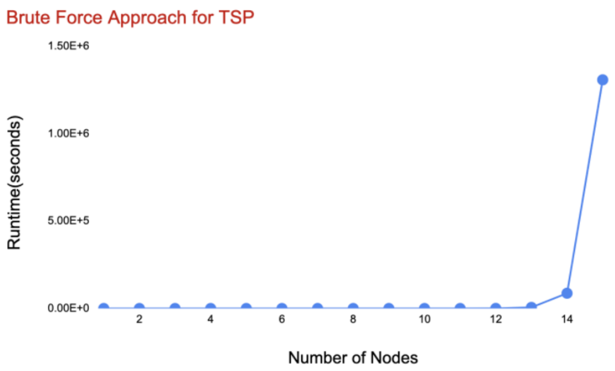
The approach checks all possible cycles and calculates the total path length in each of the cycles.

Therefore, Total Time = Total number of possible cycles X Time taken to calculate weight of one cycle.

Fixing the source node the total number of permutations for the number of possible cycles is n-1 x n-2 x n-3 . . . . . . x 2 x1, (n being the total number of nodes in the graph) = (n-1)!

Hence the total time = (n-1)! X Time taken to calculate weight of one cycle.

Even if we assume that we can find the total time in constant time, the overall time complexity of the problem is still **O(n!)**, which is considered to be too poor and is practically not feasible to perform this many numbers of computations.



Correctness Even though the brute force approach is computational very expensive it provides the global optimal solution to the problem as it checks all the possibilities.

1.3.2 Approx TSP Tour

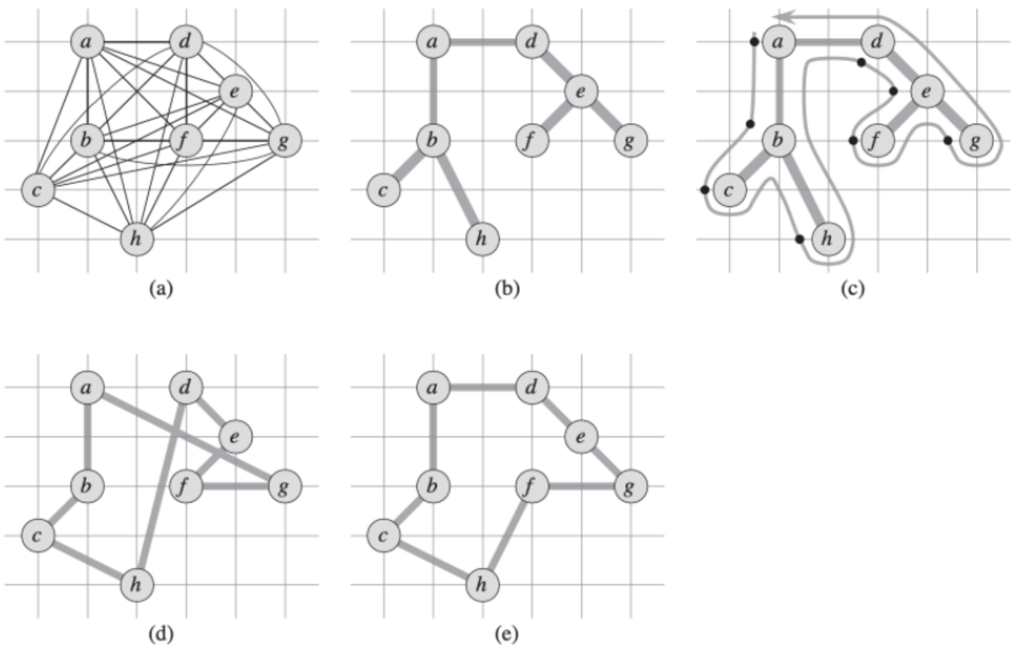
This approach is used to find a lower bound to the global optimal TSP solution. It uses the Minimum spanning tree and the triangle inequality of the cost functions to find the bound.

Algorithm :

The following is the pseudo-code or the set of steps followed in this approach:

```
APPROX-TSP-TOUR($G, c$)
1  select a vertex $r \in G.V$ to be a "root" vertex
2  compute a minimum spanning tree $T$ for $G$ from root $r$
       using MST-PRIM($G, c, r$)
3  let $H$ be a list of vertices, ordered according to when they are first visited
       in a preorder tree walk of $T$
4  **return** the hamiltonian cycle $H$
```

The graphical representation of the algorithm is as follows :



(a)The complete graph of all the nodes of the TSP problem (b)Minimum Spanning Tree of the graph with node 'a' as root (c)The preorder traversal of the MST tree. (d)Cycle made from the preorder traversal (e)The global optimal solution of the TSP.

Time Complexity The code performs the following operations:

- Finding the minimum spanning tree: $O(n^2)$ Using Prims Algorithm
- Preorder traversal and computation of weight: O(n)

Hence, the overall complexity for this procedure is just $O(n^2)$ but it does not provide the global optimal solution, it just provides a lower bound to the minimum weighted cycle. Here, n is equal to the total number of nodes.

Correctness

Let H^* be the global optimal solution of the TSP. Since H^* is a cycle we would need to remove an edge from H* to obtain the TSP. Since all the edge weights are positive we get:
$c(T) < c(H^*) \ldots\ldots\ldots\ldots (i)$
Let W be an Eulers walk around the Tree T, and we know that each edge would be covered twice in the walk so,
$c(W) = 2c(T) \ldots\ldots\ldots\ldots (ii)$

Removing some of the redundant edges from W so that none of the vertexes is visited twice we obtain a closed cycle W′ then,
$c(W') < c(W) = 2c(T) < 2c(H^*)$
Hence this provides a lower bound to the TSP optimal solution.

1.3.3 Bellman-Held-Karp Dynamic programming algorithm

This approach uses the principles of Dynamic programming to obtain the optimal solution of the TSP. This is very effective and has a non factorial run-time. The bottom-down approach starts with a smaller problem and builds up the function call tree to reach the optimal solution.

Recursion Used:

$$D_{\mathrm{T}}(S, v, w) = \begin{cases} 0 & \text{if } v = w \wedge S = \{v\} \\ c(v, w) & \text{if } v \neq w \text{ and } S = \{v, w\} \\ \max\limits_{u \in S \setminus \{v, w\}} D_{\mathrm{T}}(S \setminus \{w\}, v, u) + c_t(u, w) & \text{otherwise} \end{cases}$$

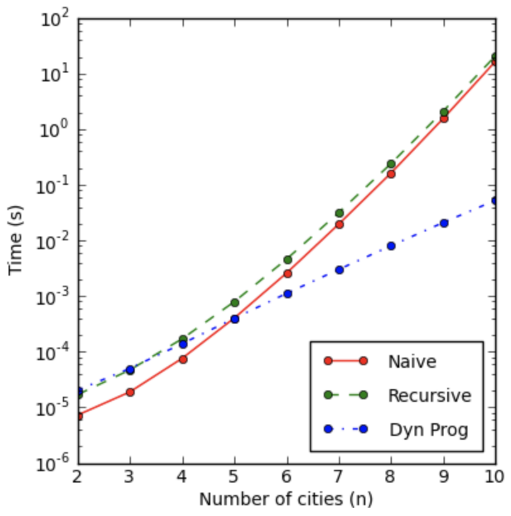
Here, $D_{TSP}(S, u)$ represents the the solution that starts at the origin node visits all the nodes in the sub-graph S and ends at the node u.

Algorithm/PsuedoCode

```
Algorithm 1: Dynamic Programming algorithm for the original TSP
  Data: A set of locations $V$, an arbitrary location $v_0 \in V$ and cost function $c$
  Result: A shortest tour that visits all locations in $V$
1 Initialize $D_{\text{TSP}}$ with values $\infty$ ;
2 Initialize a table $P$ to retain predecessor arcs ;
3 foreach $w \in V$ do
4     $D_{\text{TSP}}(\{w\}, w) \leftarrow c(v_0, w)$ ;
5 for $i = 2, \dots, |V|$ do
6     for $S \subseteq V$ where $|S| = i$ do
7         foreach $w \in S$ do
8             foreach $u \in S \setminus \{w\}$ do
9                 $v \leftarrow D_{\text{TSP}}(S \setminus \{w\}, u) + c(u, w)$ ;
10                if $v < D_{\text{TSP}}(S, w)$ then
11                    $D_{\text{TSP}}(S, w) \leftarrow v$ ;
12                    $P(S, w) \leftarrow (u, w)$ ;
13 return path obtained by backtracking over arcs in $P$ starting at $P(V, v_0)$ ;
```

Runtime Analysis

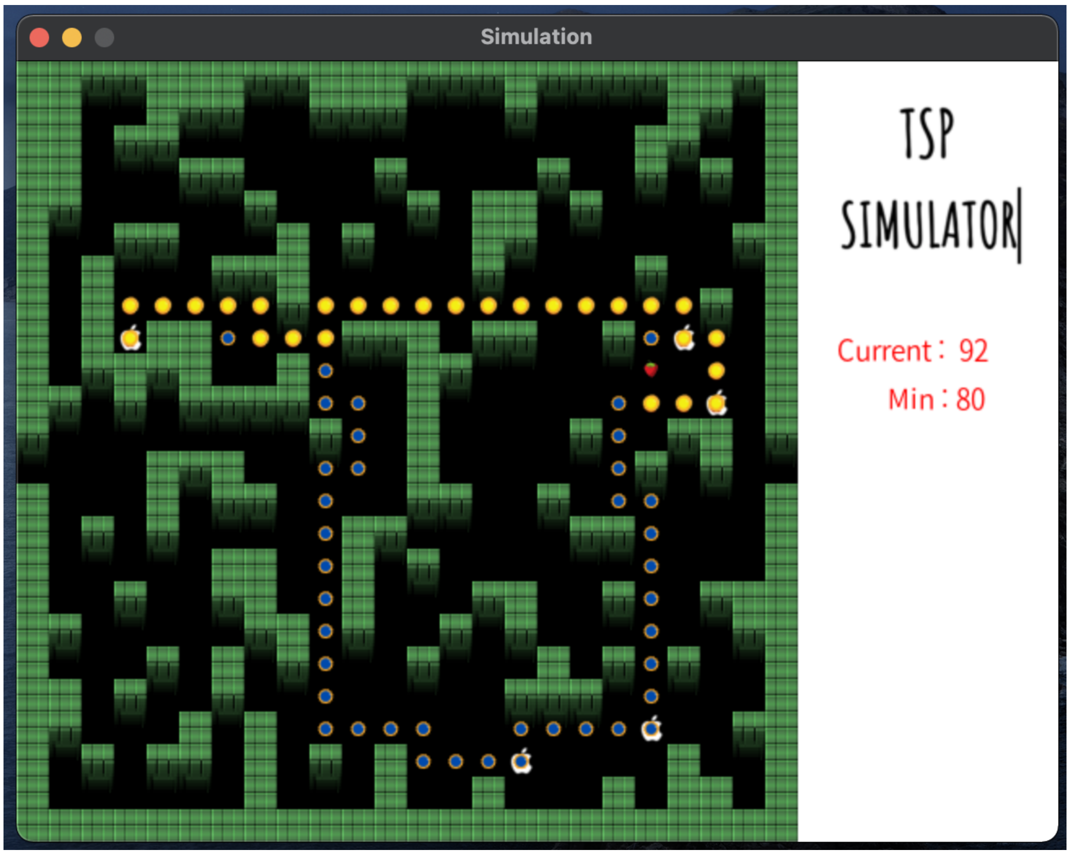
We can see that there are $n.2^n$ sub-problems to solve here, as S can be any subset of V. For each sub-problem in the recurrence relation we only consider at most n smaller sub-problems, and as a result this algorithm can be run in $O(n^2.2^n)$ time. Even though it has exponential time complexity, it is still much better than the brute force factorial complexity.



1.4 Simulation

We have used SDL library of C++ and random seed to generate a maze. We then randomly select 6 points on this connected maze graph. We call them the **Steiner Nodes** and they represent infinity stones. Our task is to find the **shortest hamiltonian walk** that starts and ends at the same infinity stone and passes through all the stones atleast once.

For the purpose of simulation we have used the brute force approach wherein we cover all the possibilities and check that which one among them has the shortest distance covered.



The yellow path represents the path currently being executed by the algorithm. The blue path represents the shortest path found by the algorithm till now. The right plate displays the weight of the current path and the weight of minimum path found till now. Big Apples represents the Steiner Nodes/ Infinity stones and strawberry represents the head of the path currently being executed.

We have introduced appropriate delays so that the animation is aesthetic to watch. Random maze is generated and the six infinity stones are also placed randomly every time the program is executed.

1.5 Data Structures Used

The following data structures were used to implement the simulation of program to find the optimal solution of the TSP:

- Adjacency matrix: This is used in the implementation of the Dijkastra's algorithm to path along the minimum distance between two Steiner nodes. This is a n X n matrix that holds the value for the distance from the starting node.
- Queue: This First-in-First-out (FIFO) data structures is used to iteratively implement the Breadth-First-Search (BFS) from a particular node.
- Point : This is a C++ structure that we defined to represent a node in the graph. It has two key parameters as its X and Y coordinate.
- Array/Vectors : These were used to store and reuse data.

The code for the simulation can be found at this github repository.
The compilation and demonstration of the simulation is present here

2. Steiner TSP

2.1 Problem Statement

Suppose we have a maze with M infinity stones. The maze has definite Entry and Exit points .We need to go through the maze , collecting all the infinity stones taking the shortest possible route. This situation can be mapped to a Steiner TSP , which is a variant of the TSP that assumes that only a given subset of nodes must be visited by a shortest route. Each block in the maze represents a node in the graph.

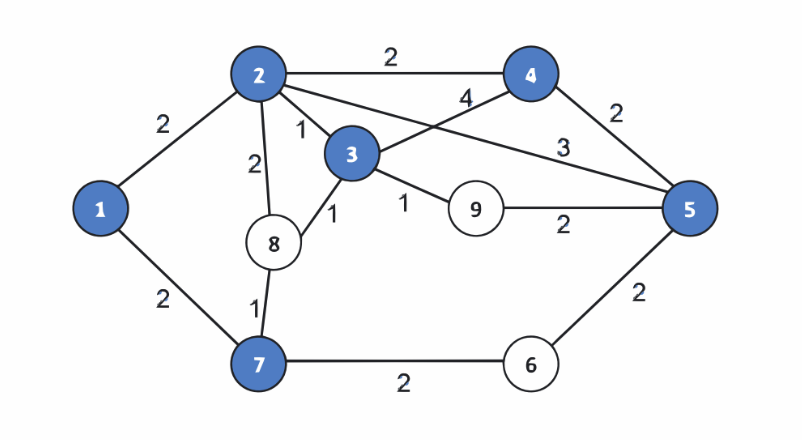


Fig. 1. Instance of Steiner TSP with nine nodes: the six required nodes are represented in blue.

2.2 Heuristics

Greedy Approach

To solve this problem greedily , the following steps are followed -
=>First , the STSP instance is reduced to a TSP instance in a complete graph which is defined by the set of required vertices only.
=> Next , any of the heuristics/algorithms discussed in the Chapter 1 of this document is applied to solve TSP in this new and complete graph.
=> In the final step , the obtained solution for TSP on the new graph is converted back to a STSP solution by expanding every edge by the corresponding shortest path between the two consecutive required nodes.
But this algorithm gives poor results when the original STSP instance is a

sparse graph , so instead we use an adaptation of the nearest neighbour TSP adaptive greedy approach to solve the STSP.

Algorithm 1 Nearest neighbor adaptive greedy heuristic for STSP.

```
1:  Select initial required node $i \in V_R$;
2:  $\mathcal{N} \leftarrow \{i\}$;
3:  $\mathcal{P} \leftarrow \{i\}$;
4:  $current \leftarrow i$;
5:  while $\mathcal{N} \neq V_R$ do
6:      $next \leftarrow$ closest node to $current$ among all those in $V_R \setminus \mathcal{N}$;
7:      $\mathcal{P}' \leftarrow$ shortest path from $current$ to $next$;
8:      $\mathcal{P} \leftarrow \mathcal{P} \oplus \mathcal{P}'$;
9:      $\mathcal{N} \leftarrow \mathcal{N} \cup \{next\}$;
10:     $current \leftarrow next$;
11: end while;
12: $\mathcal{P}' \leftarrow$ shortest path from $current$ to initial node $i$;
13: $\mathcal{P} \leftarrow \mathcal{P} \oplus \mathcal{P}'$;
14: return $\mathcal{P}$.
```

Local Search

Local search heuristics are used to iteratively improve the initial solution obtained by using constructive heuristics . Effective update functions are used which require minimal recalculations in obtaining a better solution.

```
procedure Local_Search(Solution)
1      while Solution is not locally optimal do
2              Find $s' \in N(\texttt{Solution})$ with $f(s') < f(\texttt{Solution})$;
3              Solution $\leftarrow s'$;
4      end;
5      return Solution;
end Local_Search.
```

GRASP with path-relinking heuristic

GRASP stands for greedy randomized adaptive search procedures in which we repeatedly perform construction and local search .
Path-relinking is usually carried out between the initial solution and other guiding solution . A path that connects these solutions is constructed in the search for better solutions.Considering only a restricted neighbourhood, Path-relinking tries to preserve common characteristics of good walks/common subpaths. Path-relinking matches the positions of the largest common sub-path to the initial and guiding solutions and then swaps the positions of nodes that do not belong to this common subpath.

Algorithm 2 Path-relinking algorithm for STSP.

```
1: $S, S^* \leftarrow S_i$;
2: $f^* \leftarrow cost(S_i)$;
3: while $S \neq S_g$ do
4:     $S' \leftarrow$ best solution in the restricted neighborhood of $S$;
5:     if $cost(S') < f^*$ then
6:         $S^* \leftarrow S'$;
7:         $f^* \leftarrow cost(S')$;
8:     end if;
9:     $S \leftarrow S'$;
10: end while;
11: return $S^*$.
```

Algorithm 3 GRASP+PR algorithm for STSP

```
1: $\mathcal{E} \leftarrow \emptyset$;
2: while stopping criterion not satisfied do
3:     $S \leftarrow RandomizedGreedy$;
4:     $S \leftarrow LocalSearch(S)$;
5:     if $|\mathcal{E}| > 0$ then
6:         Select solution $S'$ at random from $\mathcal{E}$;
7:         $S \leftarrow AdjustRepresentation(S, S')$
8:         $S \leftarrow PathRelinking(S, S')$;
9:         $S \leftarrow LocalSearch(S)$;
10:     end if;
11:     $UpdateEliteSet(S, \mathcal{E})$;
12: end while;
13: Return the best solution $S$ in $\mathcal{E}$.
```

2.3 Analysis

GRASP vs GRASP+PR, 200 pure GRASP iterations, instances with 1/3 of required nodes.

	Greedy ($\alpha = 0$)	GRASP (200 iterations)		GRASP+PR (same running time)		
Nodes	value	value	seconds	value	seconds	iterations
50	906	**789**	0.359	**789**	0.359	207
75	1181	**830**	0.782	**830**	0.797	191
100	1030	**919**	1.406	**919**	1.406	191
125	1306	**1145**	2.343	1148	2.344	188
150	1429	1121	3.469	**1108**	3.469	191
175	1606	**1272**	4.828	**1272**	4.828	193
200	1595	**1295**	6.500	**1295**	6.500	187
225	1625	1416	8.594	**1401**	8.609	195
250	1956	1531	10.859	**1491**	10.860	186
300	2030	1693	15.750	**1657**	15.797	191

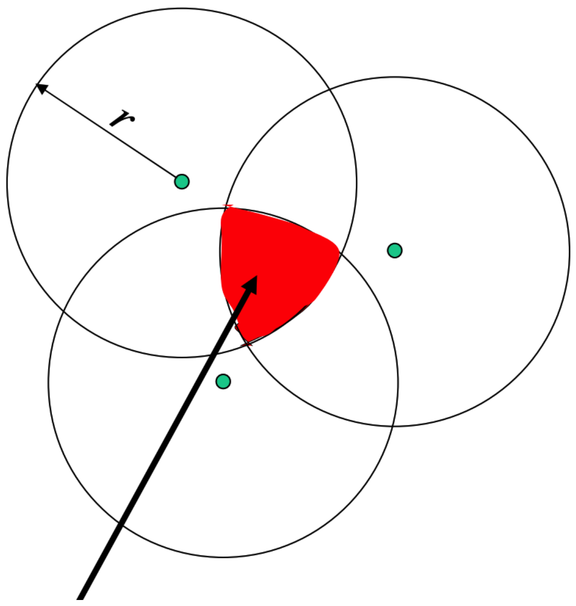
3. Close Enough TSP

3.1 Problem Statement

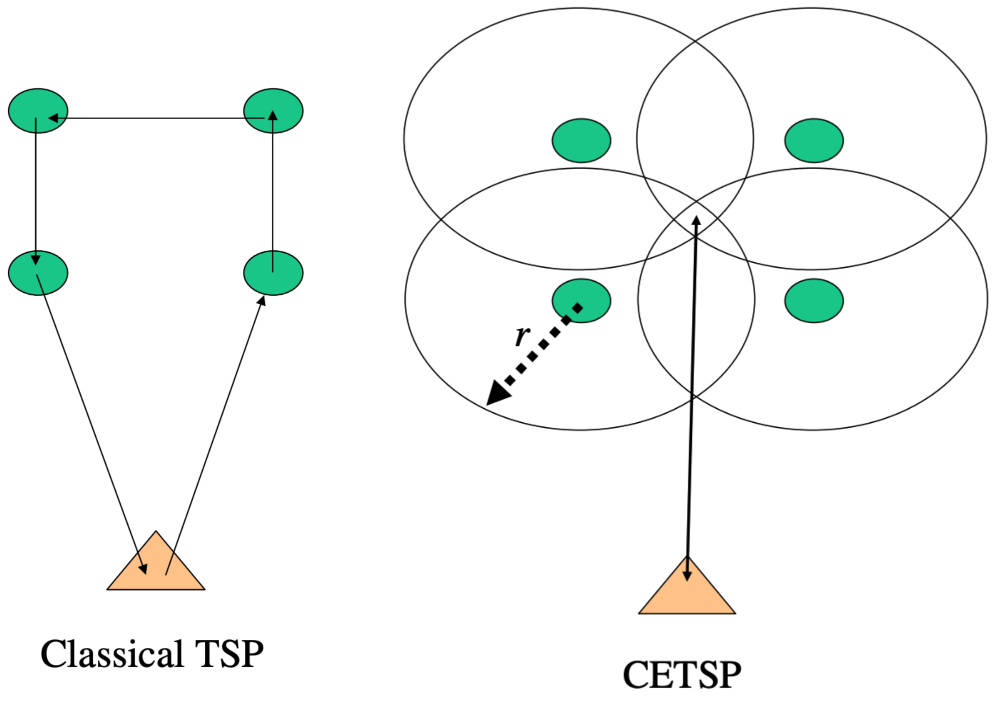
We know that in classical Travelling Salesman Problem, the salesman visits each node in the graph . Now consider a situation where a person has to visit within r distance from every node in the graph. That is if a disc of radius r is placed around every node , then the tour must touch every disc atleast once . This is called the Close Enough Travelling Salesman Problem. For our case , the mapping is such that every block in the maze represents a node in the graph and then we traverse it using the Close Enough TSP. This problem finds important applications in **Ship tracking** , **Aerial forest fire detection** , **Aircraft route planning** etc.

3.2 Heuristics

To solve this problem , let us first define **Steiner Zones**.



A Steiner zone of degree k is defined as a region in which any point is simultaneously close enough to k nodes. The above image shows a SZ of degree 3 marked in red .

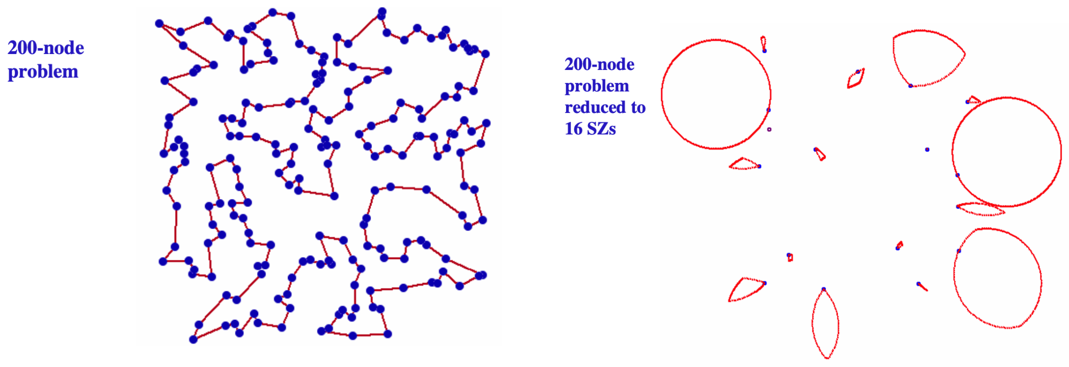


Consider the above example. We can observe a steiner zone of degree 4 in the right diagram. The required paths for both the problems on the same graph, are also shown.

Steiner Zone Heuristic

Solving the problem using Steiner Zones involves 3 key steps -

Graph Reduction - In this step , certain SZs are selected such that the union of member nodes of all the SZs is the set of all nodes . That is , we build a Supernode Set which has significantly lesser nodes than the original graph.



As we can see in the above example , A 200-node CETSP problem reduced to 16 SZs using this step.

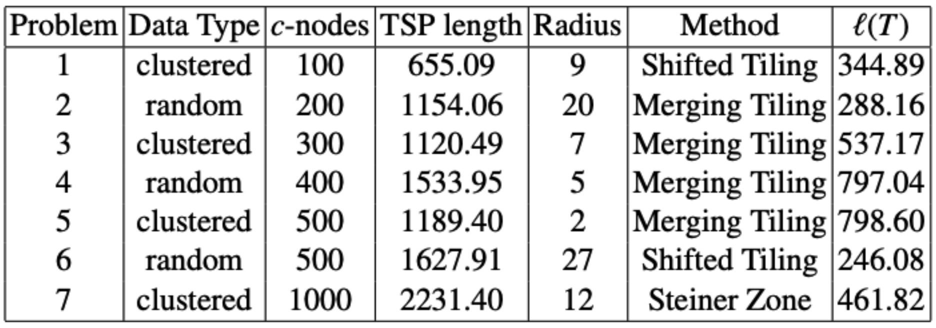
A Naive method for this reduction invloves the following steps -

=> Computing all the Steiner Zones with respect to a particular node in the graph and storing them in descending order by degree.
=>Remove all the member nodes of th SZ with the highest degree.
=>Repeat the 1st step in case any node remains uncovered in the first 2 steps.

TSP Solving - After obtaining a feasible Supernode set by replacing each SZ by a representative node (possibly the centroid) , we need to find a TSP tour on this set using the various heuristics mentioned in Chapter 1 of this document .
For example , the 200 node CETSP above boils down to solving a 16 node TSP problem, whose heuristics we already know.

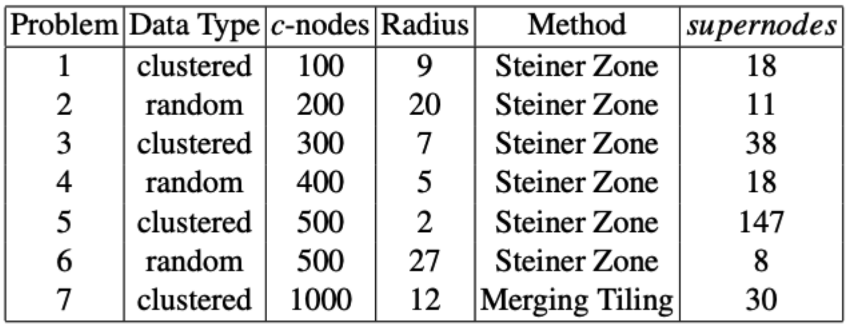
Economization - In the final and last step for obtaining an optimal solution for this problem, we optimize the TSP tour with respect to the Steiner Zones by minimizing the marginal cost of visiting a given node in the tour.

3.3 Analysis



Problem	Data Type	c-nodes	TSP length	Radius	Method	$\ell(T)$
1	clustered	100	655.09	9	Shifted Tiling	344.89
2	random	200	1154.06	20	Merging Tiling	288.16
3	clustered	300	1120.49	7	Merging Tiling	537.17
4	random	400	1533.95	5	Merging Tiling	797.04
5	clustered	500	1189.40	2	Merging Tiling	798.60
6	random	500	1627.91	27	Shifted Tiling	246.08
7	clustered	1000	2231.40	12	Steiner Zone	461.82

Table 1. Shortest tour lengths $\ell(T)$ generated by heuristics discussed.



Problem	Data Type	c-nodes	Radius	Method	*supernodes*
1	clustered	100	9	Steiner Zone	18
2	random	200	20	Steiner Zone	11
3	clustered	300	7	Steiner Zone	38
4	random	400	5	Steiner Zone	18
5	clustered	500	2	Steiner Zone	147
6	random	500	27	Steiner Zone	8
7	clustered	1000	12	Merging Tiling	30

Table 2. Fewest supernodes generated by heuristics discussed.

4. Resources

- Introduction to Algorithms is a book on computer programming by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
- Close Enough Traveling Salesman Problem: A Discussion of Several Heuristics - Damon J. Gulczynski, Jeffrey W. Heath, Carter C. Price
- The Close Enough Traveling Salesman Problem with Time Window - Soukaina Semami, Hamza Toulni, Abdeltif ElByed
- Close Enough Traveling Salesman Problem - William Mennell
- A GRASP heuristic using path-relinking and restarts for the Steiner traveling salesman problem - Ruben Interian
- Dynamic Programming Approaches for the Traveling Salesman Problem with Drone Paul Bouman, Niels Agatz and Marie Schmidt
- Dynamic Programming Solution to the Travelling Salesman Problem