ROB537: HW2

Ammar Kothari

October 14, 2017

1 Introduction

Optimization through search is an important part of modern engineering. With the advent of modern computers, people are able to generate and evaluate many solutions. Search has been used widely in fields like engineering and finance, but has also found uses in politics [1]. Search is an integral part of finding solutions to challenging problems today.

In this assignment, I apply three search algorithms to solve a Travelling Salesman Problem (TSP) with 15, 25, or 100 cities that are dispersed somewhat evenly through the map. A fourth scenario with 25 cities is structured in a way to make the solution easily identifiable. The three algorithms investigated are Simulated Annealing, Genetic Algorithms, and Monte Carlo Tree Search.

2 Problem Description

TSP is meant to replicate the challenge of a traveller trying to visit many places without revisiting any locations. Accordingly, the solution must contain every city only once, except for the start location. The path must start and end at the same location. The goal is to minimize the total cost of the path. In this case, the cost is the distance travelled.

3 Algorithm Explanation

3.1 Simulated Annealing

- s = InitialGuess()
- For i < TOTALITERATIONS:
  - sNew = NeighborSol(s)
  - s = TempSelect(s, sNew, Temp)
  - DecreaseTemp(Temp)

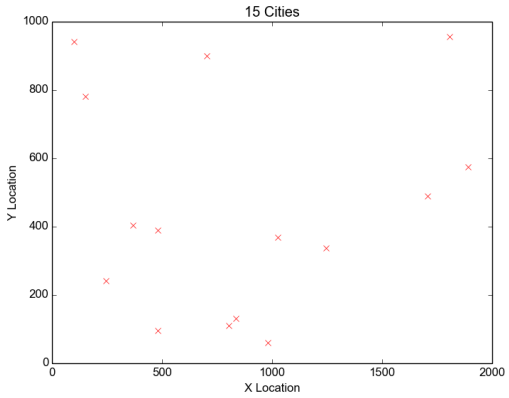

Figure 1: 15 cities plot

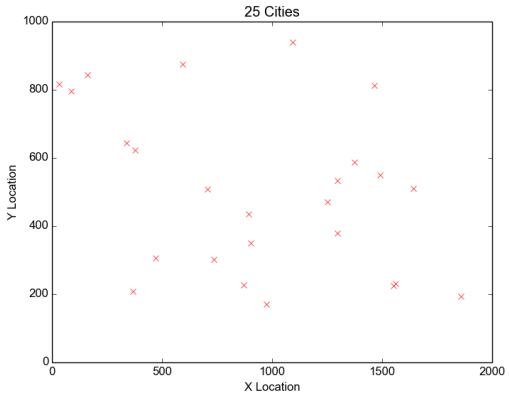

Figure 2: 25 cities plot

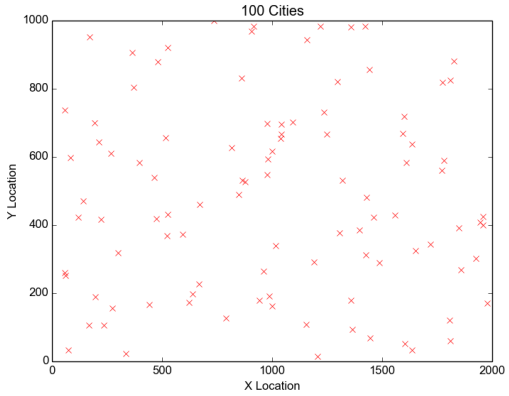

Figure 3: 100 cities plot

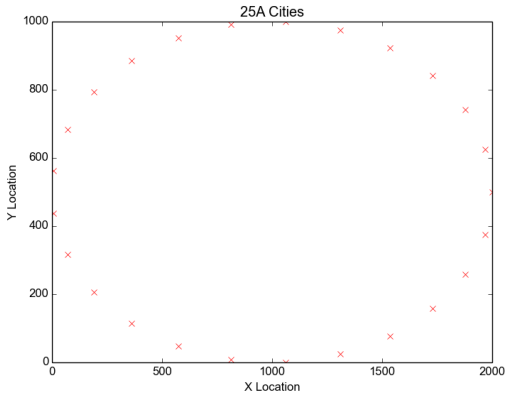

Figure 4: 25 cities plot

- FinalSolution = s

InitialGuess – Intial solution generated randomly
TOTALITERATIONS – Number of solutions to try
NeighborSol – Generate a succesor state to s by switching the order of two adjacent cities in s
TempSelect – Selects a solution between the two presented based on the temperature. The best solution is probabilistically choosen based on the temperature. The higher the temperature the more likely the next solution is choosen at random.
DecreaseTemp – Decrases the temperature such that it ends at 0 (always choosing the better solution) when the maximum number of iterations is reached.

3.2 Evolutionary Algorithm

- POP = InitialGuess(POP_TOT)
- For i < TOTALITERATIONS:
  - POP_SEL = ChooseParents(POP)
  - CHILD = PerturbPopulation(POP_SEL)
  - POP = SelectPop(CHILD, POP)

  Soltuion = Min_Value(Pop)

InitialGuess – Initial population of solutions generated randomly
TOTALITERATIONS – Number of generations
ChooseParents – Epsilon-greedy choose best solutions in population based on some amount of noise
PerturbPopulation – Finds neighbor solutions based on choosen parents. The amount of variation to original solution is based on an amount of noise. Variation is implemented as the number of switches between neighboring cities in a solution. Higher noise means more neighboring cities will be switched.
SelectPop – Choose the best solutions from the original population and mutated children. The resulting number of solutions is equal to the starting population size.
Min_Value – From a given population, returns the member that has the best solution value.

In this assignment, Population size was 10. Number of children produced each round was 5. Noise was decreased from 1 to 0 based on current iteration number. For this algorithm, the total number of iterations was choosen as 1,000. This is ten times less than the other two approaches in order to have a similar number of total generated solutions.

3.3 Monte Carlo Tree Search

The implementation used in this assignment is not optimized. As a result, it has slow run times but is able to find solutions.
Main Algorithm

- InitializeTree()
- For i < TOTALITERATIONS:
  - Parent = PickParent(Tree)
  - Node = PickChild(Parent)
  - Leaf_Value = DescendTree(Node)
  - BackPropogateValue(Leaf_Value)

DescendTree

- If Node has Children
  - Node = PickChild(Node)

- DescendTree(Node)

- Else: PlayOut()

InitializeTree – Initializes tree structure. Nodes are initialized with best value to promote exploration.
TOTALITERATIONS – Number of solutions to try
PickParent – Probabilistically chooses the first node of the tree based on value.
PickChild – Probabilitically choose a child from a node. If the node does not exist, create it.
DescendTree – Subfunction that continues to descend the tree in a probablisitic manner based on node value
PlayOut – Determines value of a node on its first visit. The remaining cities are choosen at random to create a valid path. Returns the value of the full path.
BackPropogateValue – Value of the current node is pushed to the parent. The parent adds the new value to its current value based on the total number of visits. For example, if a parent has been visited once before, then $(Old\_Value * Number\_Of\_Visits + New\_Child\_Value)/(Number\_Of\_Visits + 1)$. This is effectively an average value of the node based on all explored solutions. A node with a good value is more likely to lead to a good solution than a node with a worse value.

3.4 Results

References

[1] Nina Totenberg. This supreme court case could radically reshape politics, Oct 2017.