Advanced Algorithms Project

TSP Solver

Team Members

Tho Vo

Omar Mohammed

Ahmed Hassan

TABLE OF CONTENTS

Introduction:

Algorithms:

1. Brute Force
    a. Definition:
    The **Brute Force** is the most basic and general approach of algorithms, to solve most kinds of problems. It enumerates all plausible solutions for a problem, then check which one is the optimal solution.
    Of course, Brute Force is computationally too expensive, and that is why it can't be used to solve problem of big input size. However, it should alywas provide exact solutions.

    The brute force method used here first enumerates all solutions, in an optimized way, where cyclic solutions are treated and discarded. The a linear search is used over each solution to explore the best candidate out of all the found solution space.

    The time complexity for the algorithm implemented is O((n-1)!) for the enumeration part, and $O(n^2)$ for the linear search part.
    b. Pseudocode

```
Bruteforce(AdjMatrix):
        nodes = list(AdjMatrix)
        all_paths = enumerate_all(nodes)
        best_cost = 0
        for each path in all_paths:
                current_cost = calculate_cost(path)
                if current_cost > best_cost:
                        best_cost = current cost
                        best_path  = path
        return best_path, best_cost
```

2. Branch and Bound
    a. Definition:
    Branch and Bound is a slightly smarter version of the Brute Force algorithm. It uses the tree pruning approach to cut down the time spent in searching the tree of solutions for the best one. It does that by keeping a current best solution until it finds a better one, and it uses this current best solution to

prune branches that exceed the current best before calculating the cost of the whole branch. Branch and Bound also generates exact solutions.
The implemented algorithm uses only pruning on the current branch it is calculating the cost for, so in some conditions when the input size is small, has small weights range or when it is sorted in a way that the best solution is at the rightmost part of the tree, it can take slightly more time than the brute force algorithm.
In it's average case it should cut down the running time running time considerably
The time complexity in the worst case , however, is the same as that of the Brute Force, which is (n-1)!

b. Pseudocode

```
BranchNBound(cities):
        nodes = list(cities)
        all_paths = enumerate_all(nodes)
        best_cost = inf
        for each path in all_paths:
                current_cost = 0
                for each edge in path:
                        current_cost += calculate_cost(edge)
                        if current_cost > best_cost:
                                break /* prune current branch */
                if first_path:
                        best_cost = current_cost
                        best_path = path
                else if current_cost < best_cost
                        best_cost = current_cost
                        best_path = path
        return best_path, best_cost
```

3. 2-Opt

a. Definition

In optimization, **2-opt** is a simple local search algorithm first proposed by Croes in 1958 for solving the traveling salesman problem. The main idea behind it is to take a route that crosses over itself and reorder it so that it does not.[1]

The idea of 2-opt is not too hard to understand:

1. We use a initial tour, take from the other algorithm
2. Choose two breakpoints inside the tour, one is called *i* and the other is called *k*
3. At this breakpoint, we divide the tour in 3 new paths
    a. The first path from the beginning to i-1, keep the same
    b. The reversed path from i to k
    c. The third path keep the same from k+1

[1] http://en.wikipedia.org/wiki/2-opt

4. Consider the cost for the new path, if it is better than the initial, keep the new path
5. Redo from step 2 until we can't find any new better path

The time complexity of 2-opt for n cities of TSP is O(n^2) because each node can be exchanged with n-1 other nodes and there are n nodes in total, there are n(n-1)/2 different exchanges.

b. Pseudocode

```
define new_route(path,i,k):
        new_path.append(path[0,i-1])
        new_path.append(reverse(path[i,k]))
        new_path.append(path[k+1,end])
        return new_path
define 2-opt():
        while still_can_swap:
               best_cost = calculate_cost(path)
               i=0
               k=i+1
               while i < number_of_nodes_can_swap:
                  while k < number_of_nodes_can_swap:
                     new_path = new_route(path,i,k)
                     new_cost = calculate_cost(new_path)
                     if new_cost < best_cost:
                         best_cost = new_cost
```

4. Minimum Spanning Tree

a. Definition:

The **Minimum Spanning Tree** algorithm can be used to find approximate solutions for the TSP problem, by constructing a tree walk after finding the minimum spanning tree of the edges of the problem graph.

The algorithm uses Kruskal's[2] approach to find one minimum spanning tree for a given list of sorted edges by their weights.
The Kruskal algorithm, uses a data structure called "**union-find**"[3], which helps generate the MST by assigning nodes to groups and subgroups.
After obtaining the MST, we proceed with doing a "**pre-order**"[4] walk, that will generate a path, which is might not be optimal, but it will be at most "2 times the optimal solution"

One drawback with this approach is that it can't guarantee finding a path with the generated MST, if there is an increased level of sparsity. And to solve

[2] http://en.wikipedia.org/wiki/Kruskal%27s_algorithm
[3] http://en.wikipedia.org/wiki/Disjoint-set_data_structure
[4] http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/AproxAlgor/TSP/tsp.htm

this, we need to generate all the minimum spanning trees, which is a costly problem in itself

b. Pseudocode

```
MinSpanTree(cities):
        edges = extract_edges(cities)
        edges = sort_with_distance(edges)
        min_span_tree = Kruskal(edges)
        mst_copy = min_span_tree
        while mst_copy != None:
                path = add_to_path(min_span_tree[0])
                next_edge = find_next_edge()
                if next_edge != null:
                        path = add_to_path(min_span_tree[0])
                else:
                        backtrack()
        cost = calculate_cost(path)
        return path, cost
```

5. Greedy

a. Definition

A **greedy algorithm** is an algorithm that follows **the problem solving heuristic** of making the locally optimal choice at each stage with the hope of finding a global optimum. In many problems, a greedy strategy does not in general produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that approximate a global optimal solution in a reasonable time.

For example, a greedy strategy for the traveling salesman problem (which is of a high computational complexity) is the following heuristic: "At each stage visit an unvisited city nearest to the current city". This heuristic need not find a best solution but terminates in a reasonable number of steps; finding an optimal solution typically requires unreasonably many steps. In mathematical optimization, greedy algorithms solve combinatorial problems having the properties of matroids.[5]

From the idea of this algorithm, when I applied this one in the project, the step to achieve a Local Optimal is very simple.

1. We start from a random city in the path
2. Choose the **nearest** city to move next
3. Add this city in the path and mark to not come back (Greedy will never reconsider its choice in the past or in the future, only the current time)

[5] http://en.wikipedia.org/wiki/Greedy_algorithm

4. Start from the city that we have added in the path
5. Do again step 2 until all cities is visited

Usually, when we use **Greedy,** we won't reach the Global Optimal because the start city is randomly and because **Greedy** won't reconsider its previous choice. It is somehow missing the good path.

Pros: Always have an answer, it usually is a Local Optimal but sometimes it can reach the Global Optimal with a little fortune. The best advantage of **Greedy** is it finished in a reasonable time so we can satisfy with its good approximations to the optimum.

Cons: The start city is very important, it means in some specific cases, **Greedy** may provide the unique worst answer.

The complexity time of **Greedy** is O(n) and for **Greedy** version that try all the cities to find a best solution will be O(n^2)

b. Pseudocode

Normal Greedy:

```
start_city = random(cities)
current_city = start_city
path = [start_city]
while still_have_city_unvisited:
      next_city = find_the_nearest_city(current_city)
      path.append(next_city)
```

Greedy Exhaustive: Try to find all the answers based on selecting a new start city until no city remains. The best solution will base on the cost of the tour from the start city.

```
define Greedy(cities,start_city = 0)
define Greedy_Exhaustive(cities):
       while still_have_city_not_a_start_city:
           current_cost = Greedy(cities,start_city)
           if(current_cost < better_cost):
                 better_cost = current_cost
```

6. Randomized

a. Definition

The pretty simple algorithm: Randomized means at every step, we just choose randomly a next city to visit without caring about the cost until we find the solution.

Usually, if we want to have a good answer, we should try to run the algorithm thousand times. It varies from 100 to 10000 times, depend on how large the number of cities is. Although the program runs thousand times, the result is

somehow always worst. It is usually two times of the cost compare to other algorithm like **Greedy.**

Pros: We have a quick answer in a linear time.

Cons: The answer never reaches the Global Optimal. Actually, the result bases on how lucky the program is.

The time complexity is O(n)

b. Pseudocode

```
while still_have_city_unvisited:
        next_city = choose_random()
        path.append(next_city)
```

7. Evolutionary

a. Definition

Evolutionary Algorithms belong to the Evolutionary Computation field of study concerned with computational methods inspired by the process and mechanisms of biological evolution. The process of evolution by means of natural selection (descent with modification) was proposed by Darwin to account for the variety of life and its suitability (adaptive fit) for its environment. The mechanisms of evolution describe how evolution actually takes place through the modification and propagation of genetic material (proteins). Evolutionary Algorithms are concerned with investigating computational systems that resemble simplified versions of the processes and mechanisms of evolution toward achieving the effects of these processes and mechanisms, namely the development of adaptive systems. Additional subject areas that fall within the realm of Evolutionary Computation are algorithms that seek to exploit the properties from the related fields of Population Genetics, Population Ecology, Co-evolutionary Biology, and Developmental Biology[6].

In this algorithm, we will be creating the simplest evolutionary algorithm, a serial hill climber. The algorithm is known as a hill climber because one can imagine the space of all entities that are to be searched as lying on a surface, such that similar entities are near one another. The height of the surface at any one point indicates the quality, or fitness of the entity at that location. This is known as a fitness landscape. A hill climber gradually moves between nearby entities and always moves 'upward': it moves from an entity with lower fitness to a nearby entity with higher fitness[7].

b. Pseudocode

[6] "Clever Algorithms: Nature-Inspired Programming Recipes", By Jason Brownlee PhD

[7] Ludobots free online course on Evolutionary Robotics, http://www.reddit.com/r/ludobots/wiki/core01

```
Create a random parent
Evaluate the parent fitness
For number of generations:
        1. Create a child from the parent - through mutation -.
        2. Measure the fitness of the child.
        3. If the child fitness is better than the parent:
                - Consider this child as the new parent.
                - Otherwise, return to step
```

8. Genetic

a. Definition

The Genetic Algorithm is inspired by population genetics (including heredity and gene frequencies), and evolution at the population level, as well as the Mendelian understanding of the structure (such as chromosomes, genes, alleles) and mechanisms (such as recombination and mutation).
The objective of the Genetic Algorithm is to maximize the payoff of candidate solutions in the population against a cost function from the problem domain.
The strategy for the Genetic Algorithm is to repeatedly employ surrogates for the recombination and mutation genetic mechanisms on the population of candidate solutions, where the cost function (also known as objective or fitness function) applied to a decoded representation of a candidate governs the probabilistic contributions a given candidate solution can make to the subsequent generation of candidate solutions.
In this case, the possible genetic operators have been studies carefully. We have chosen an approach which includes generating the new population of solutions from performing different types of mutation processes[8].
One of the well known problems of the genetic algorithms is that they are subject to falling into a local optima. This happens when we follow the traditional scheme of selecting the best individuals in order to reproduce to build the new generation.
One way to solve this is actually to shuffle the current population, then divide it into clusters. From each cluster, select the best individual. The selected best individuals from all the clusters are now chosen for the reproduction.
This ensures that the sample selected actually represent the current population, and can escape the local optima.
This approach has shown its effectiveness in the ability of the code to converge to almost near the optimal point for the chosen datasets from the TSP LIB[9], as will be shown in the subsequent analysis.

b. Pseudocode

[8] http://studentdavestutorials.weebly.com/traveling-santa-claus-genetic-algorithm-solutions.html
[9] http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/

```
Create a random parent population
Evaluate this population fitness
Get the best performance individual for this population

For number of generations:
           a.  Create a new generation from the parent population - through
               different types of mutation -.
           b.  Measure the fitness of the new generation.
           c.  Get the best performance individual for this population
           d.  If the child best performer fitness is better than the parent
               best performer fitness:
                 i.    Consider this child generation as the new population.
                ii.    Otherwise, return to step
```

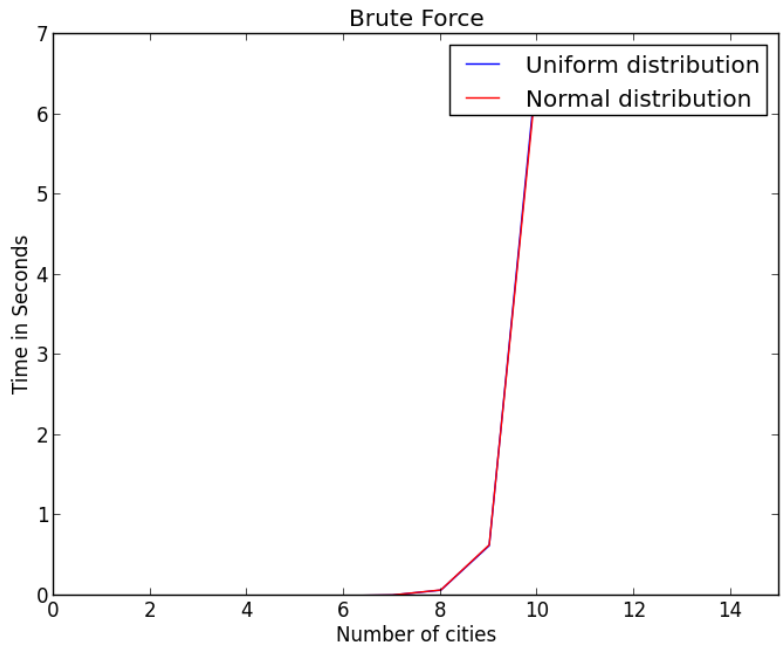
Results and Analysis:

Individual Analysis:

1. **Brute Force**

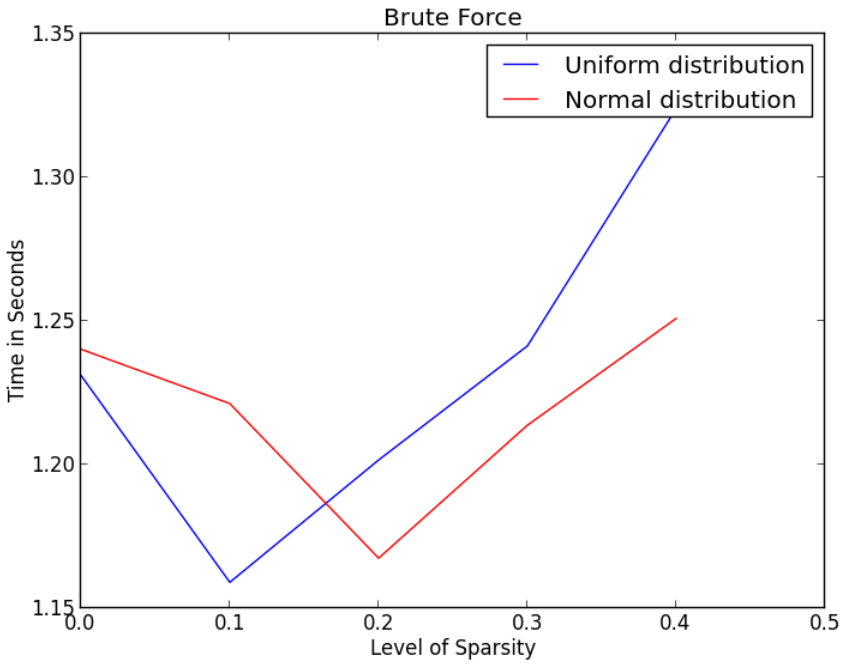
Results and analysis

a. **Analysis of Size vs Time, with respect to distribution**



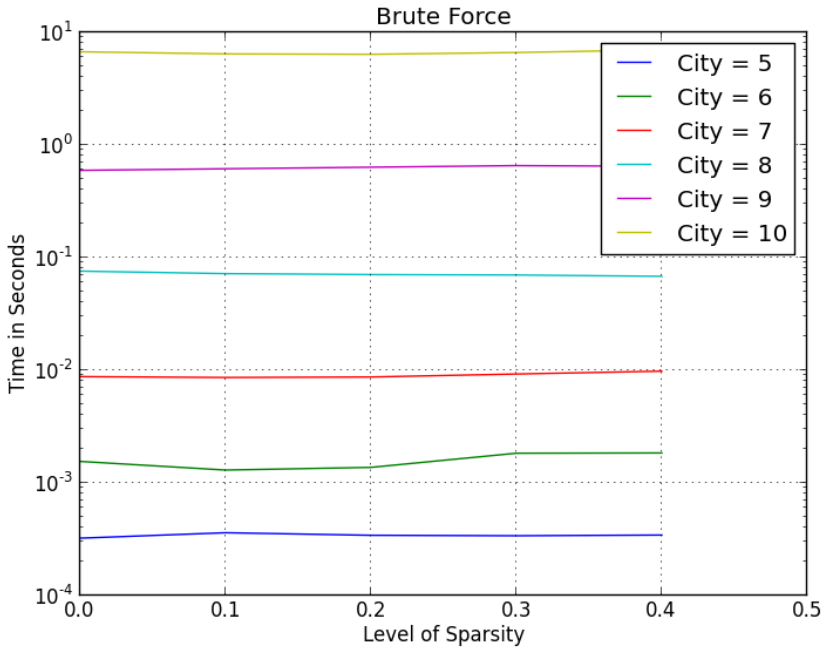
It is apparent that the brute force algorithm, even with small sizes, is tending to n! time. The distribution has almost no effect on the time, and this is understandable, as the algorithm always enumerates and calculates all the possible paths whatever the weights distribution given

b. **Analysis of Sparsity vs Time, with respect to distribution**



When there problem is a complete one (no sparsity), the algorithm almost runs for the same time for both uniform and normal distributions. But, when the sparsity starts increasing, it lowers the running time of the algorithm considerably. For the uniform distribution it seems to lower much faster, but almost at the mean of the sparsity level used, the uniform distribution start taking more time than the normal.

c. **Analysis of Sparsity vs Time, with respect to problem size**

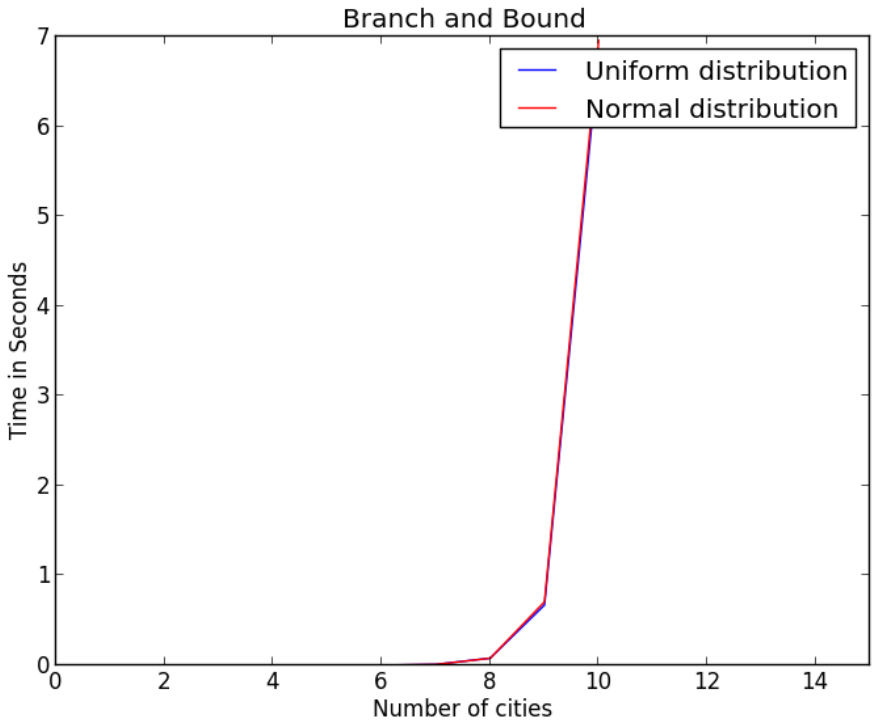


It is clear that the level of sparsity has almost no effect of the time when the size of the problem is to be taken into consideration.

2. Branch and Bound

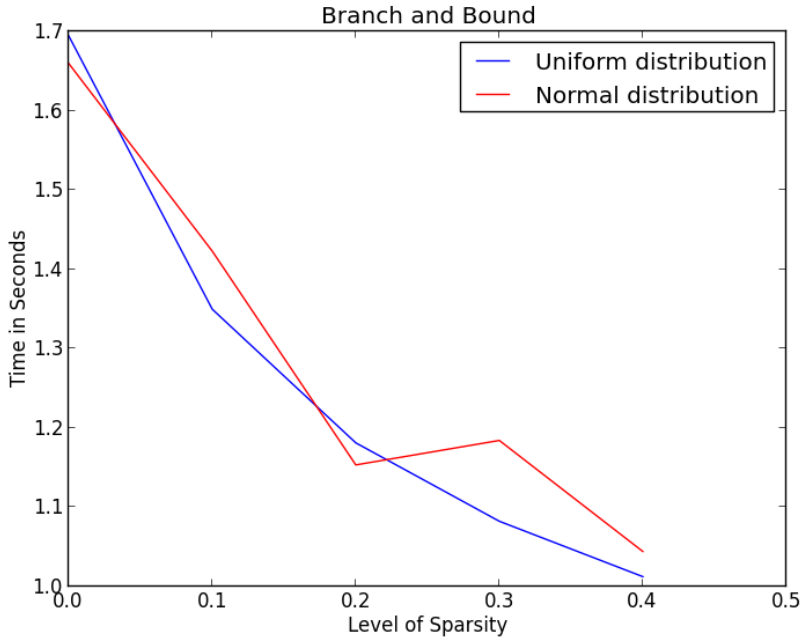
Results and analysis

a. **Analysis of Size vs Time, with respect to distribution**



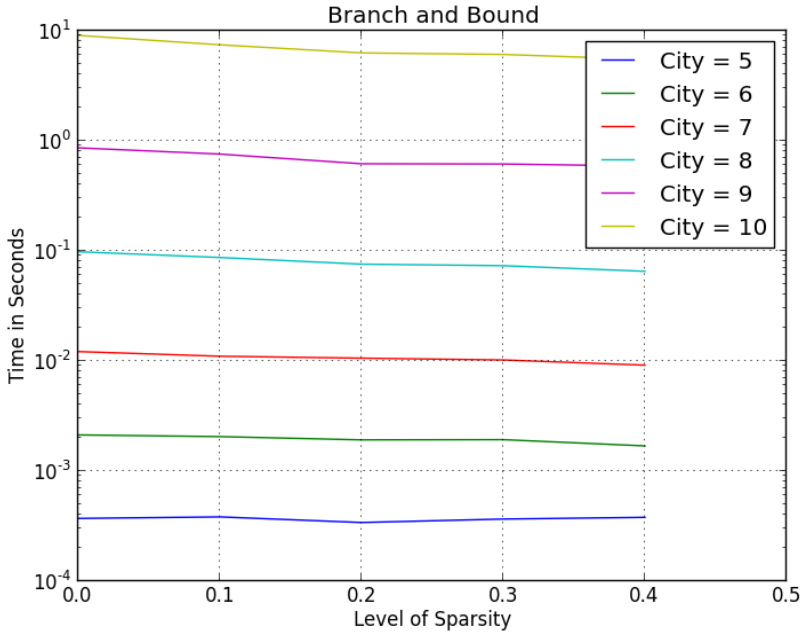
The branch and bound has the same behaviour as the brute force, just as expected, which is also O(n!).

b. **Analysis of Sparsity vs Time, with respect to distribution**



It is clear from the graph that the time of computation for the branch and bound drastically decreases as the level of sparsity increases, and this is expected as well, because the less edges you have, the more pruning you can do

c. Analysis of Sparsity vs Time, with respect to size

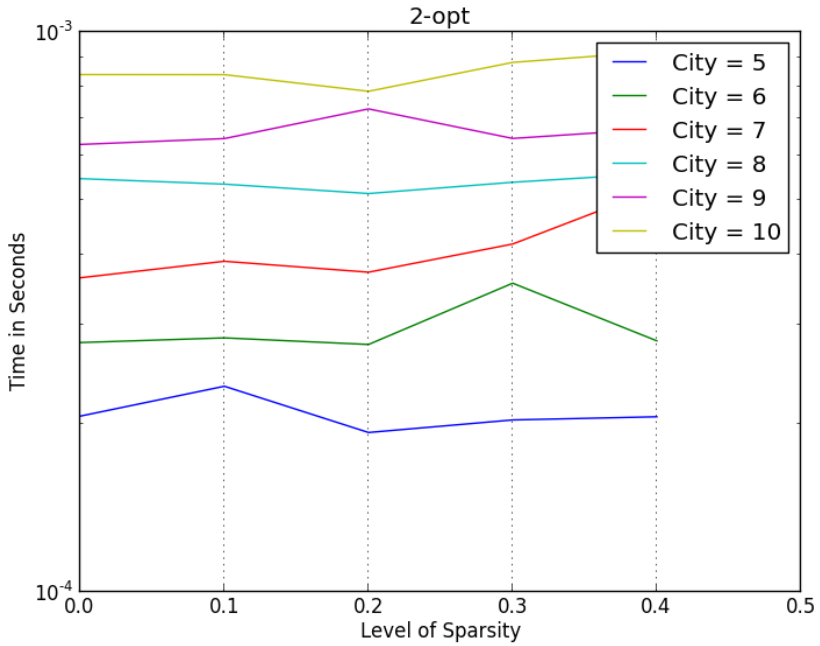


For the different sizes as well, the time for the branch and bound is experiencing a noticeable decrease, but it seems to increase as the number of cities increase

3. 2-Opt

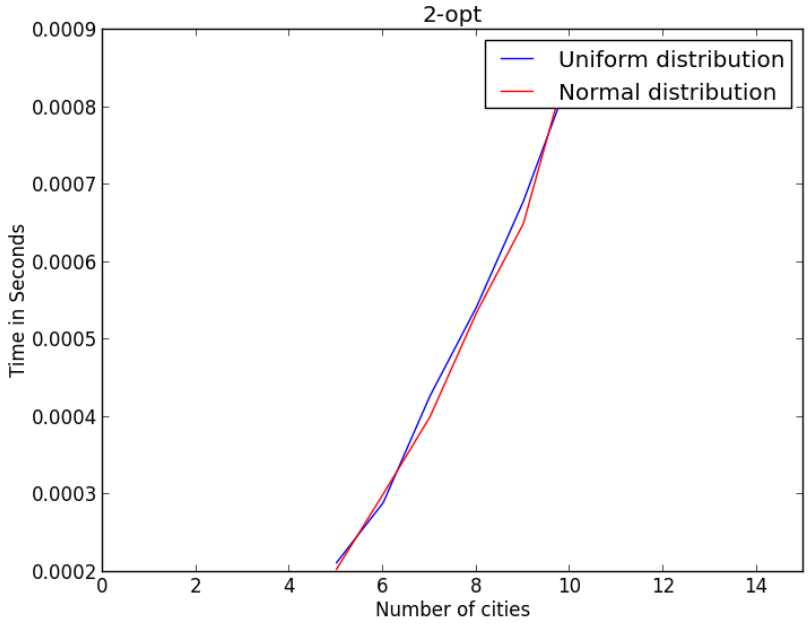
Results and analysis

1) **Sparsity VS Time, with respect to the number of cities.**



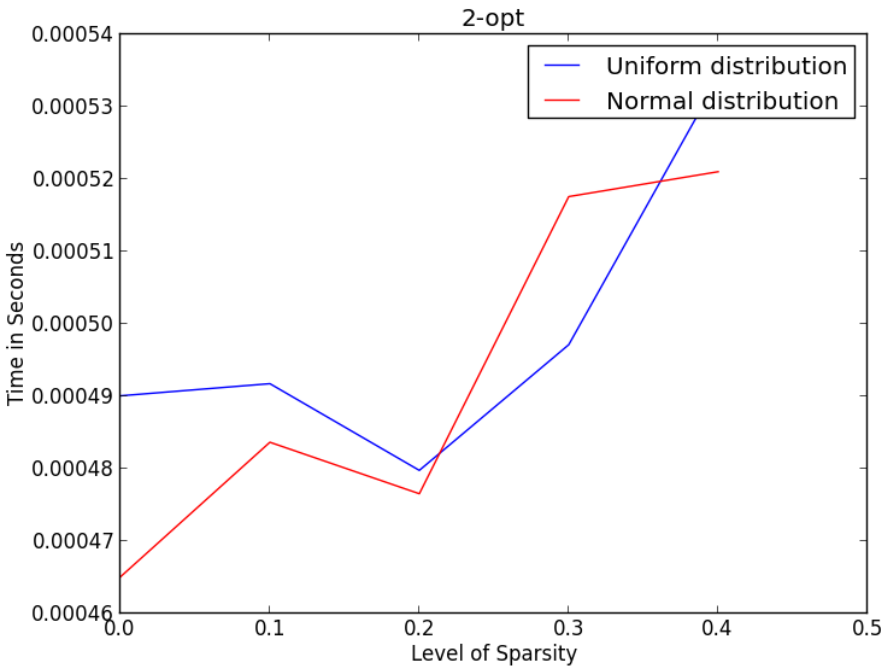
We can see that the number of cities and level of sparsity have a close relation. When we increase the number of cities and the level of sparsity, the time increase either.

2) **Size VS Time, with respect to the distribution**



There is not too much different between normal distribution and uniform distribution just because the 2-opt just do the swap between each node on the path to find a best solution so actually the distribution not affects too much.

3) Sparsity VS Time, with respect to the distribution

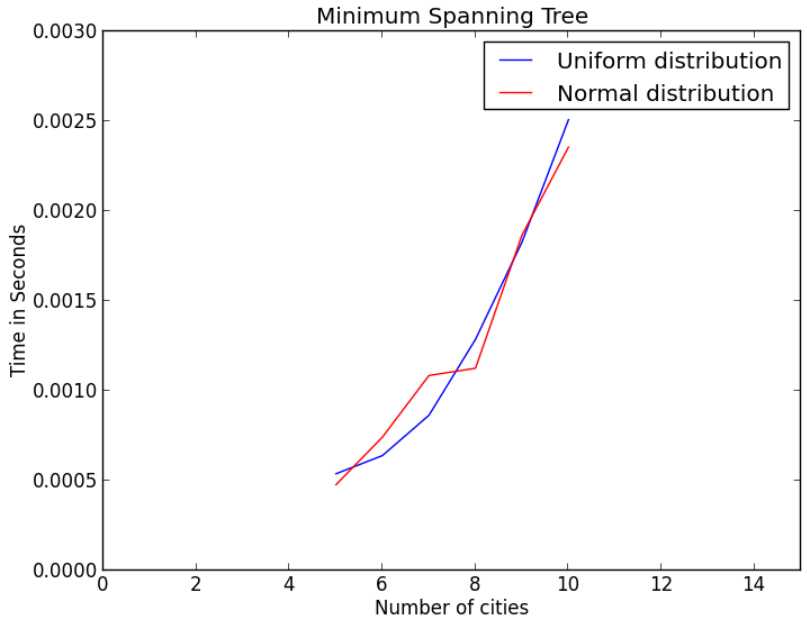


When we add the sparsity here, the difference in time between two type of distribution is still not too much large. Just because we choose the INFINITY to describe the lost connection between two cities, 2-opt just takes this number to compare to get the result so it still not affect too much to the time to finish.

4. Minimum Spanning Tree

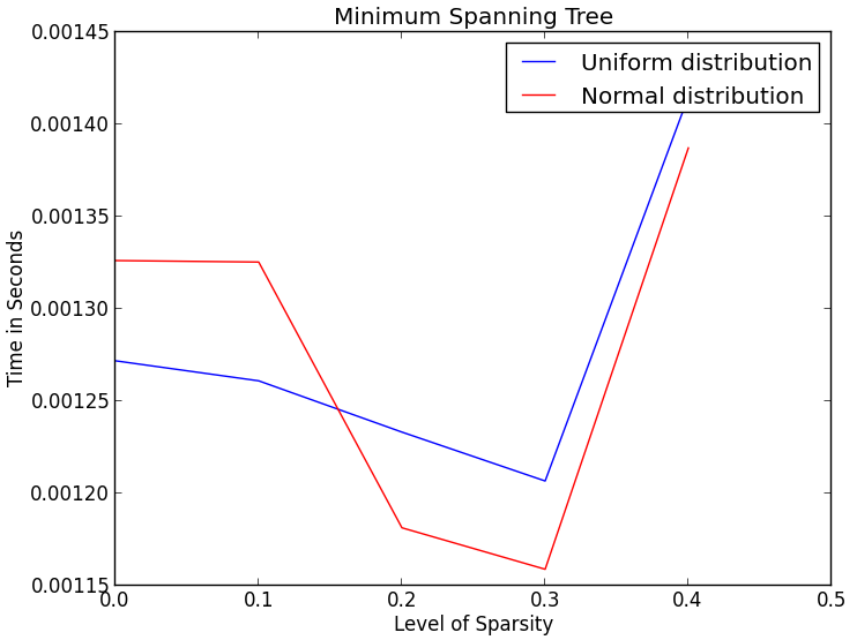
Results and analysis

a. Analysis of Size vs Time, with respect of distribution



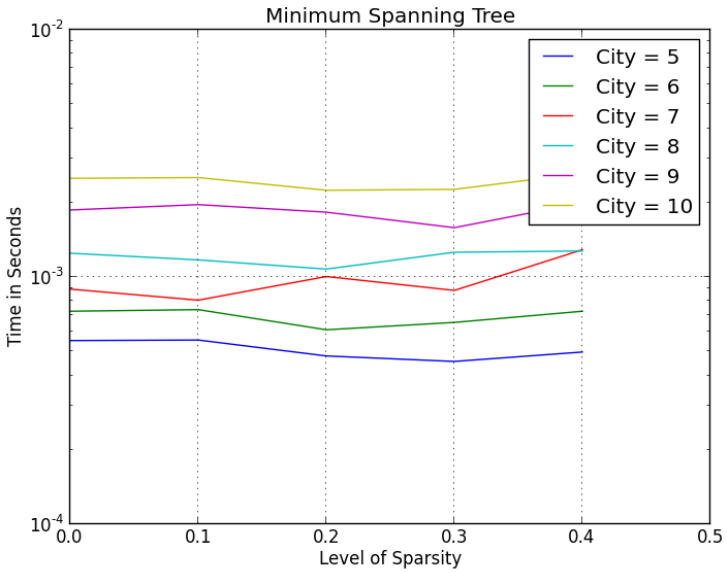
The MST experiences n^2 evolution in time, which is very clear in the case of uniform distribution, while the normal distribution is showing a better time evolution.

b. Analysis of Sparsity vs Time, with respect of distribution



The MST algorithms is having better running with increased sparseness, although it is generating paths that have invalid edges sometimes.

c. Analysis of Sparsity vs Time, with respect of size



The level of sparseness seems to have random effect on the running time of the MST, which is applicable for all the cities.

d. **Because the MST, can generate invalid paths with sparse input, we run over some TSPLIB files that are complete graphs.**

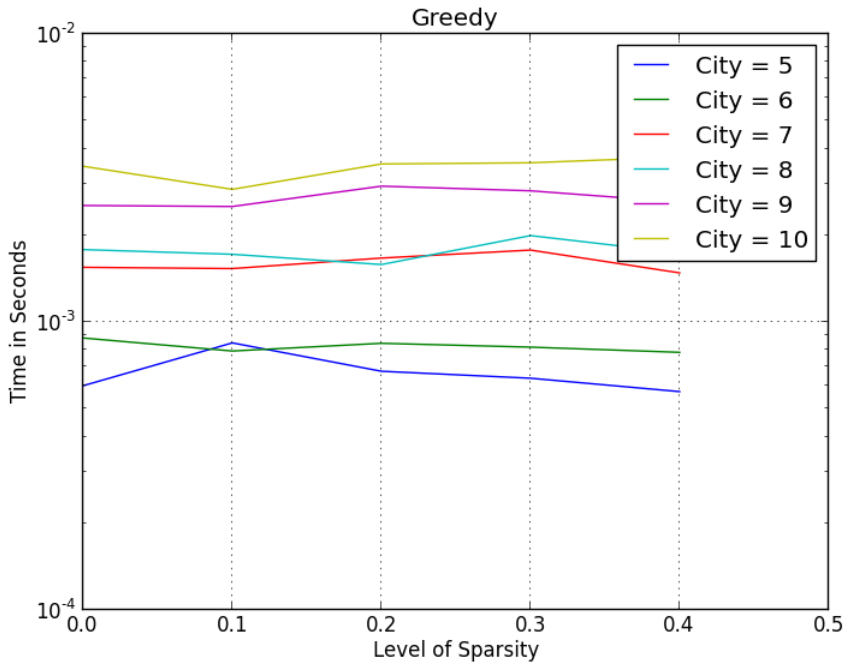
TSPLIB name	Cost	Time (s)	Optimality
gr17.xml	3587	0.009423	0.5812
gr21.xml	5936	0.005727	0.4560
gr24.xml	2275	0.014809	0.5591
gr48.xml	9526	0.063683	0.5297

It is clear that the times are very good, but the cost generated is always very bad compared to other heuristic algorithms

5. Greedy

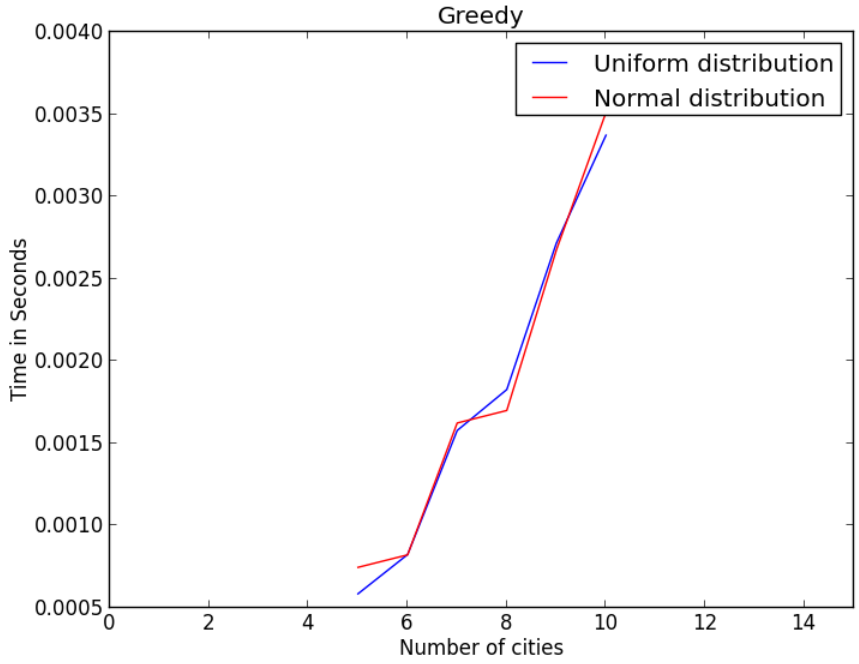
Results and Analysis

1. **Sparsity VS Time, with respect to the number of cities.**



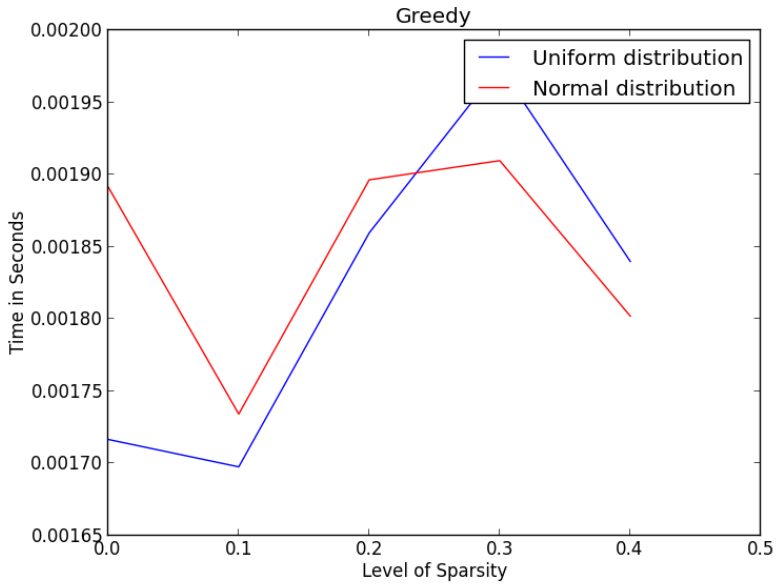
Actually, we don't choose a normal Greedy, we are running the Exhaustive Greedy, in this algorithm, it will choose the best result it has but it doesn't affect too much the time to run the algorithm because at each step, the algorithm just choose the nearest city and in some case the cost to the last city may be INFINITY, it will lead to worst case of Greedy

2. **Size VS Time, with respect to the distribution**



There is not too much different between normal distribution and uniform distribution just because the Greedy just choose the nearest city to visit.

3. **Sparsity VS Time, with respect to the distribution**

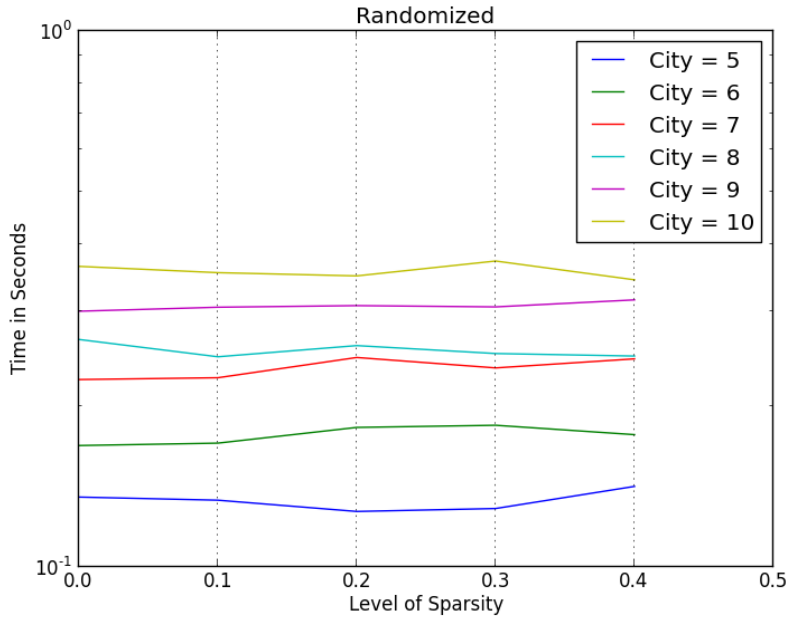


When we add the sparsity here, the difference in time between two type of distribution is still not too much large. Just because we choose the INFINITY to describe the lost connection between two cities, and we choose the Exhaustive Greedy, it just run until finished choosing all the city as start city.

6. Randomized

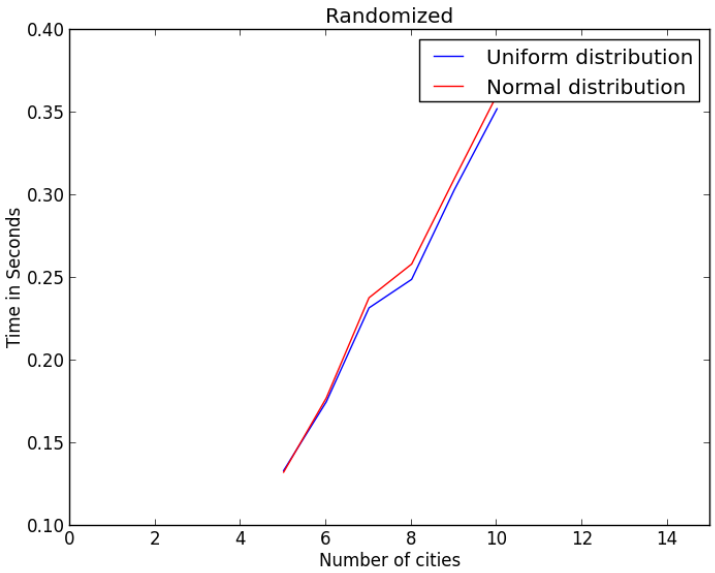
Results and Analysis

1. **Sparsity VS Time, with respect to the number of cities.**



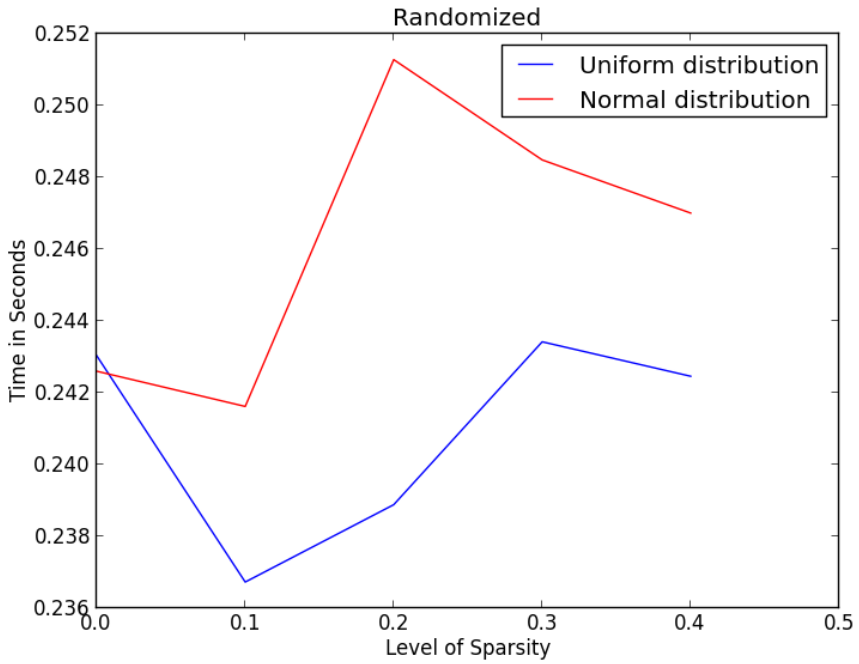
In fact, we don't run a Naive Randomized, we are running about 10000 times the Randomized. It will choose the best result it has after thousand of running time and it still can add the INFINITY to the cost in some path but it will usually ignore this worst and still keep the good result. The running time will not be affected by the sparsity, only the cost.

2. **Size VS Time, with respect to the distribution**



There is not too much different between normal distribution and uniform distribution just because the Randomized just choose a random next city to visit.

3. Sparsity VS Time, with respect to the distribution

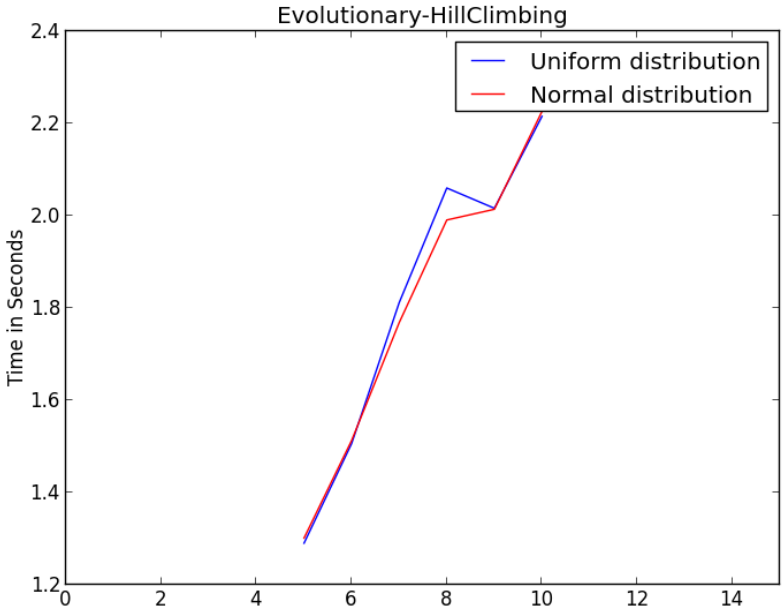


When we add the sparsity here, the difference in time between two type of distribution is very large, just because at each step, Randomized will choose a random city, in Normal distribution, the INFINITY will concentrate at some specific point, and the machine take more time when it need to compare with the INFINITY

7. Evolutionary

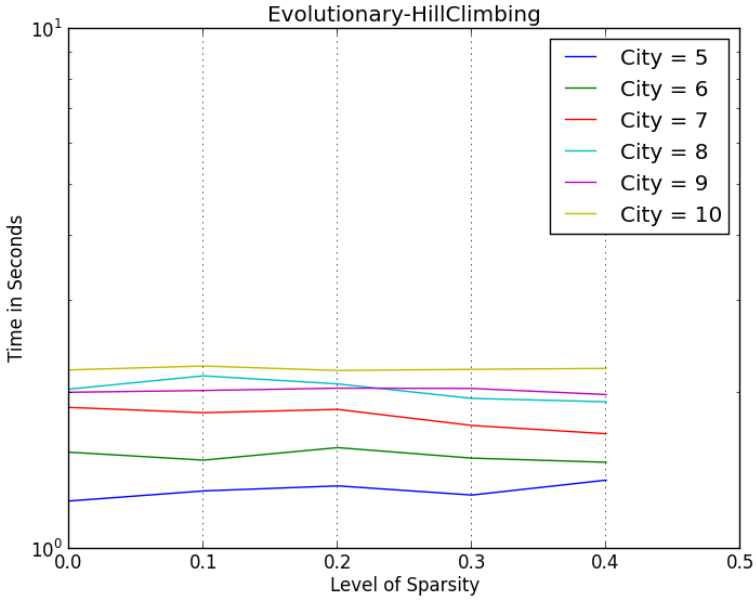
a. Results and Analysis:

i. **Size VS Time, with respect to the distribution**



Based on what we can see from this graph, we can see that the distribution didn't actually affect the running time of the algorithm, even with increasing number of cities.

ii. **Sparsity VS Time, with respect to the number of cities.**

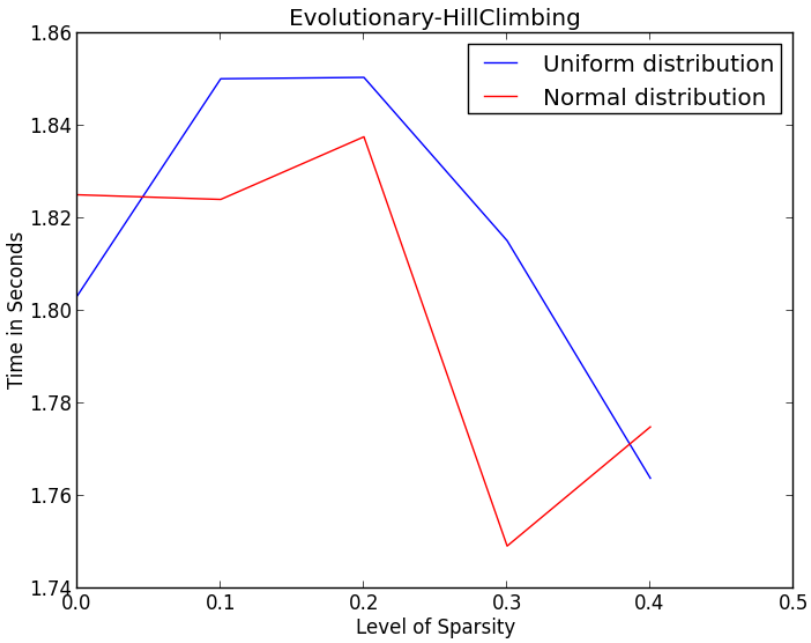


We can notice here some change in the running time with respect to the sparsity, yet we believe this change is pretty small, and we can't infer from the graph a real relation between the sparsity level and the running time.

An interesting fact is that for number of cities = 9,10, we can see that

the graph is almost straight horizontal line. This may suggest that as the number of cities increase, the effect of the sparsity disappears. More experiments to test this trend should be made.

iii. Sparsity VS Time, with respect to the distribution

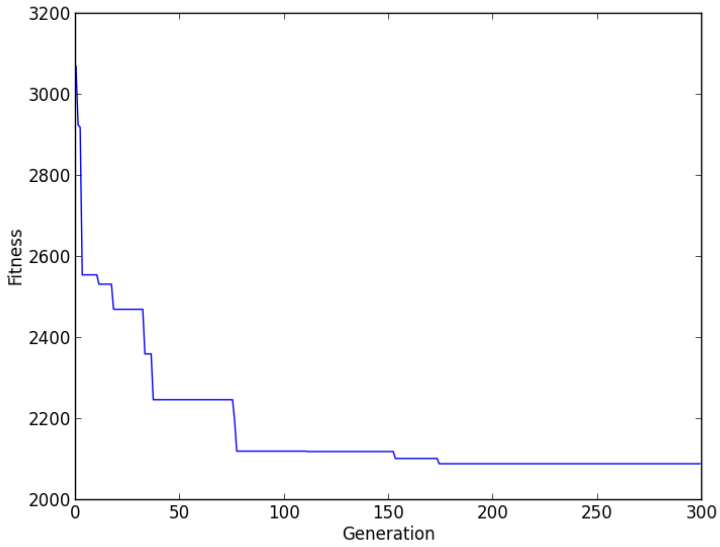


In this graph, the uniform distribution shows better running time than the normal distribution for most of the sparsity levels we selected.
A further statistical analysis should be made, to test if this difference is significant enough to make this conclusion valid.

8. Genetic

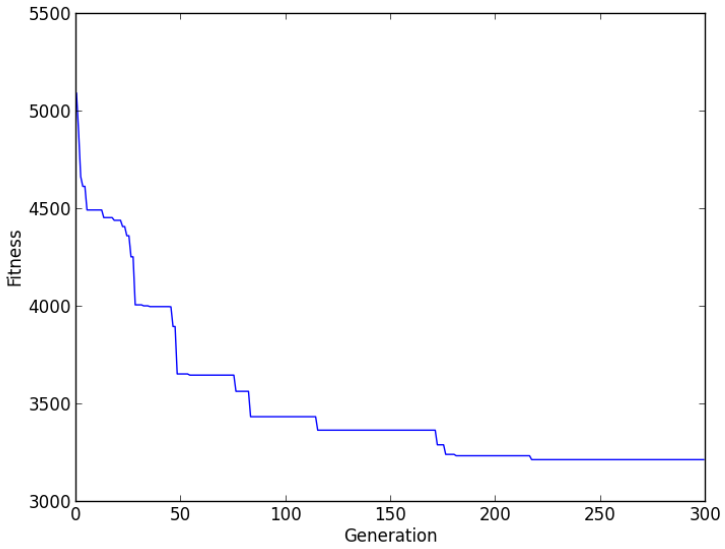
a. Results and Analysis: Unlike other algorithms, the GA take considerable time. Analysis on the convergence of the algorithm with respect to the number of generation has been performed for 4 datasets from the TSP LIB.

i. **gr17**



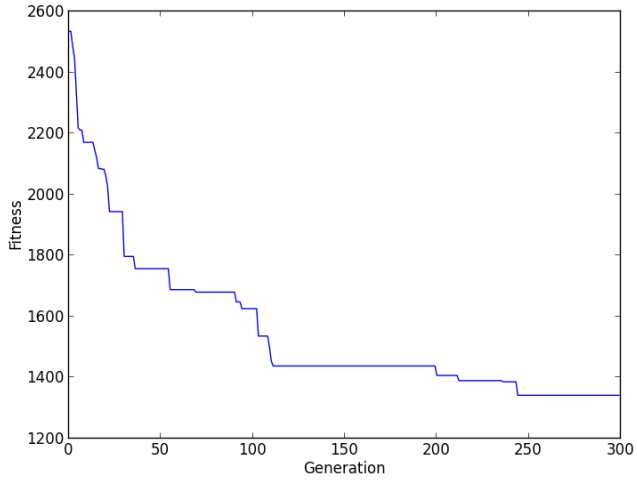
In this case, the GA reaches its max convergence in about 175 generations.
The optimal cost for this dataset is '2085'. The achieved cost by our GA is 2090.0

ii. **gr21**



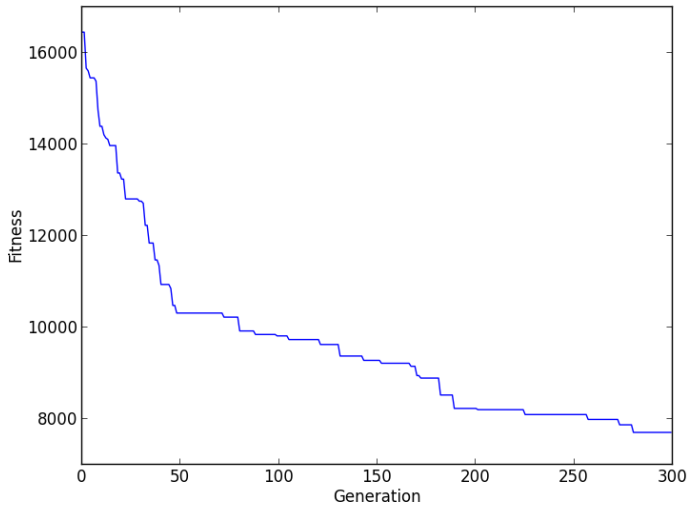
In this case, the GA reaches its max convergence in about 225 generations.
The optimal cost for this dataset is '2707'. The achieved cost by our GA is 3217.0

iii. gr24



In this case, the GA reaches its max convergence in about 250 generations.
The optimal cost for this dataset is '1272'. The achieved cost by our GA is 1342.0

iv. gr48



In this case, the GA reaches its max convergence in about 275 generations. It's possible that with more generations, more convergence could have happened.
The optimal cost for this dataset is '5046'. The achieved cost by our GA is 7708.0

b. General notes:
   i. It can be noted here that as the number of cities increase, there is a need for more generations in order to achieve better results.

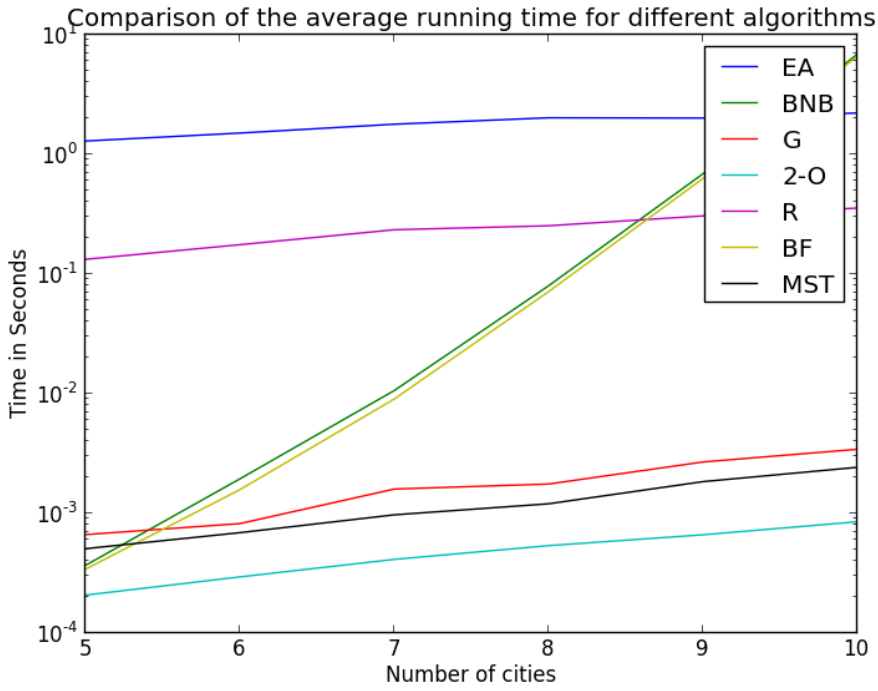
ii. Further studies should be performed on:

1. The relation between the population size and the time needed to reach a near optimal solution, with respect to the number of the cities.
2. The relation between the number of generations and the time needed to reach a near optimal solution, with respect to the number of the cities.

Combined Analysis

In this section, we will explore some comparative analysis between all algorithms in different aspects:

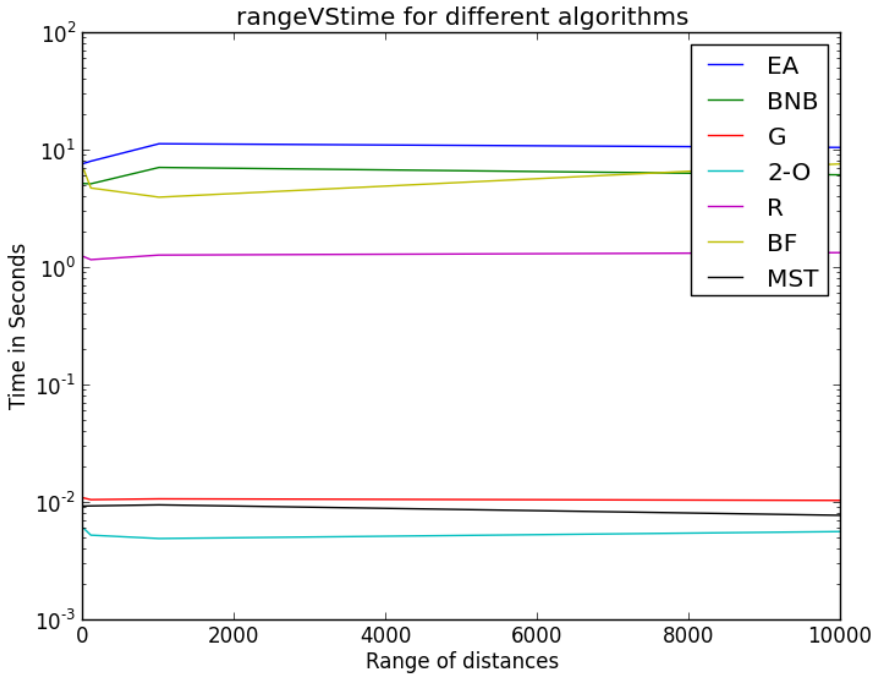
- Size VS Time



It can be shown from this graph that:

- 'Brute force' and 'Branch and Bound' algorithms achieves the highest growth rates.
- The 'Greedy', '2-opt', 'Minimum Spanning Tree' and the 'Random' achieves slight growth rate, as the number of the cities increased.
- The 'Evolutionary Algorithm' has a very slow growth rate. However, this should be related to the time it takes to reach near optimal solution. This fact suggests that the growth rate of this algorithm will increase if we take this into account. Further experiments to study this more closely is needed in order to determine the general trend of growth for this algorithm.

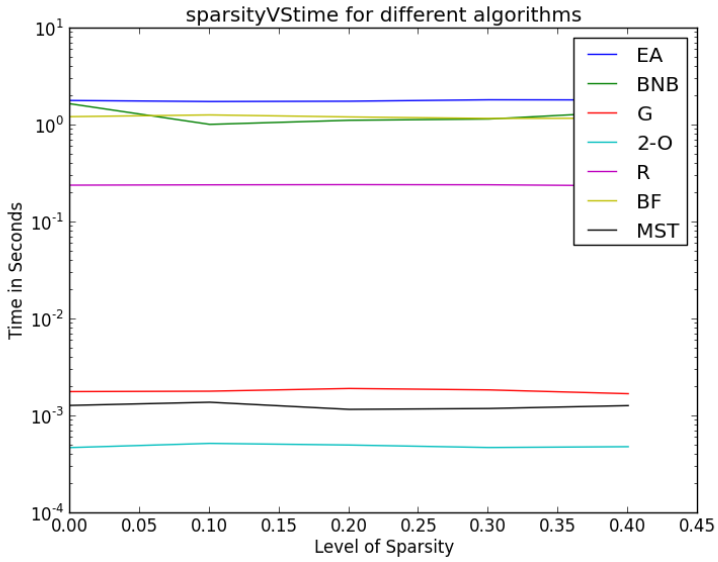
- **Range VS Time**



It can be shown from this graph that:

- **Almost all the algorithms doesn't get really affected by the range.**
- **Slight change in the performance of the 'Evolutionary Algorithm', 'Brute Force' and 'Branch and Bound' happened at the beginning, but after that the performance gets stable. No reason to suggest that there's a trend here.**

- **Sparsity VS Time**



It can be shown from this graph that:

- **Almost all the algorithms doesn't get really affected by the the sparsity level - very slight changes -.**
- **Slight change in the performance of the 'Minimum Spanning Tree', 'Brute Force' and 'Branch and Bound' happened. No decisive conclusion can be made from current analysis. We believe that there's no obvious trend here.**

The Experimental Setup

In this section, we will go through the different logistics needed to complete this project:

- **Random Data Generator**
  - **In order to make testing, we needed to design a random data generator, that can be configured with different parameters to generate different types of datasets.**
  - **The parameters available are:**
    - number of cities
    - range of distances
    - sparsity
    - distribution
  - One of the issues noted was that at some levels of sparsity, problem happens in our datasets. After careful debugging, we found there is a similarity between the generated datasets and the famous 'percolation' problem. The problem was that, as the level of sparsity increases, the possibility of having at least one valid path in the graph decreases, till it disappears at one point.
  - To counter for this problem, the level of sparsity has been set not to get over 0.4. Also, an algorithm - to check if there's at least one valid path in the graph -

has been deployed as a checker. If the generated graph doesn't meet this criteria, the generator are forced to try again to generate a new dataset.

- Dataset configurations generator
    - To automate the generation of the datasets with different configuration, we created a script to automate the use of the 'random generator' to generate different datasets with different configurations.
    - The configurations selected are:
        - number of cities = [5,6,7,8,9,10]
        - range of distances = [10,100,1000,10000]
        - sparsity = [0.0, 0.1,0.2,0.3,0.4] (0 is no sparsity, 1 is max sparsity)
        - distribution = ["uniform","normal"]
- Github
    - In order to enhance the productivity and the collaborativity between the team member, a github repository was initiated.
    - All codes have been communicated with that repository.
    - You can find it at this address https://github.com/thovo/aa_practice_project
- XML parser
    - In order to parse the data from the TSP LIB, a special parser needed to be implemented in order to convert the XML datasets from the XML format into a list of lists.
    - This parser take as an input the path to a XML file
    - This parser returns a list of
        - The name of the data set as a string
        - The number of cities as an integer
        - The data set itself, as a list of lists
- The Experimenter
    - This script has been implemented in order to automate the process of testing all the algorithms on all the datasets (the generated datasets).
    - To make the results structured, a JSON file is generated for each algorithm, that contain a number of rows corresponding to the number of datasets we have.
    - Each row has the following attributes:
        - sparsity
        - cost
        - num_cities
        - symmetry
        - algorithm
        - time
        - weights_range
        - optimality
        - distribution

    - This made it easier for us to perform meaningful analysis on the algorithms and their different aspects.
- Graphing utility
    - A special script has been implemented in order to automate the generation of all the graphs used in this report.
    - The ‘pylab’ package have been deployed in order to make these graphs.