Combinatorial Optimization examples: Graph Coloring and TSP

Combinatorial Optimization

Optimization problems often search for the best optimal solution of a set of variables to find the answer. These seem to divide on two categories: solutions that are found in *real-valued* variables and those encoded in *discrete variables.* In the latter, we find those problems called as **Combinatorial Optimization**.

In CO problems, *Papadimitriou* and *Steiglitz* describe it as looking for an object from a finite set that satisfies given conditions. This *object* is typically an integer number, subset, permutation or graph structure.

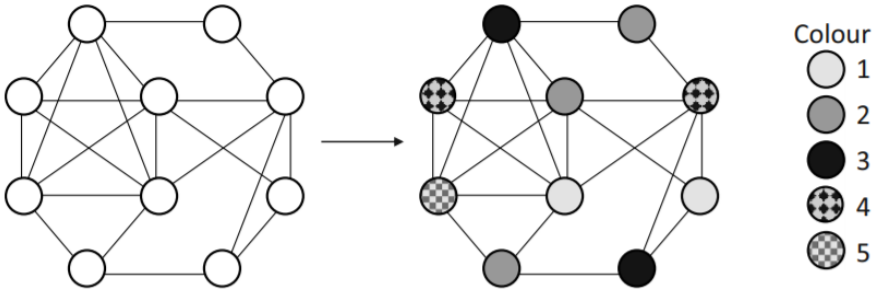
A Combinatorial Optimization or a Constraint Satisfaction problem can be defined by*:*

- A set of variables $X = \{X_1 \dots X_n\}$;
- A set of domains of values $D = \{D_{X1} \dots D_{Xn}\}$, one for each variable X_i. The product of all the domains is called the ***search space***;
- A set of constraints imposed on the variables.

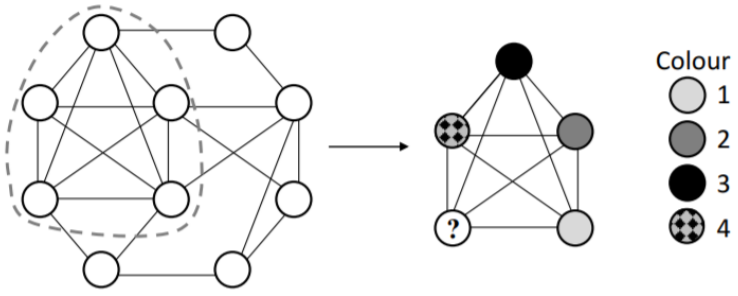
Examples

Graph Coloring Problem

The graph colouring problem is one of the most famous problems in the field of graph theory. It asks, given any graph, how might we go about assigning "colours" to all of its vertices so that (a) no vertices joined by an edge are given the same colour, and (b) the number of different colours used is minimised?



But what if we wanted to colour this graph using fewer than five colours? When we remove everything from outside the selected area, we are left with a subgraph containing just five vertices.



Importantly, we can see that every pair of vertices in this subgraph has an edge between them. If we were to have only four colours available to us, as indicated in the figure we would be unable to properly colour this subgraph, since its five vertices all need to be assigned to a different colour in this instance.

This allows us to conclude that the solution in the first figure is actually optimal, since there is no solution available that uses fewer than five colours.

A popular puzzle can be observed on the videogame "*Professor Layton and the Azran Legacy"* based on this topic. In the puzzle, the game asks the player to do the following:

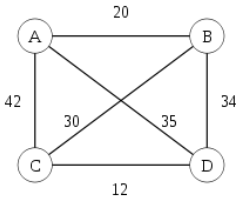
"*You've decided to sew your friend a garish flower from a number of red, blue, yellow and purple petals. You want it to be a little different, so you've decided that no two petals of the same colour should touch at any point. That's your only rule. Touch a petal to choose its colour and create a lovely floral gift for your friend."*

Puzzle #13 already has eight predefined colored petals and you must alternate from a set of four different colors: red, blue, yellow, purple. This makes sure there's only one optimal answer to the puzzle as opposed to starting with no predefined colored petals.

Traveling Salesman problem

The Traveling Salesman Problem (TSP) is the problem of finding a least-cost sequence in which to visit a set of cities, starting and ending at the same city, and in such a way that each city is visited exactly once.

Since its seminal formulation as a mathematical programming problem in the 1950's the problem has been at the core of most of the developments in the area of Combinatorial Optimization.



The distance between each city is given and is assumed to be the same in both directions. The objective is to minimize the total distance to be travelled.

References

- Gendreau, M., & Potvin, J. Y. (2005). Metaheuristics in combinatorial optimization. *Annals of Operations Research*, *140*(1), 189-213.
- Johnson, D. S. (1982). Combinatorial Optimization: Algorithms and Complexity. By Christos H. Papadimitriou and Kenneth Steiglitz.
- Guerri, A. (2007). *Structural analysis of combinatorial optimization problem characteristics and their resolution using hybrid approaches* (Doctoral dissertation, alma).
- Lewis, R. (2015). *A guide to graph colouring* (Vol. 7). Berlin: Springer.
- Diaby, M. (2006). The traveling salesman problem: a linear programming formulation.

Assignment Problem

Description of problem

The assignment problem is a combinatorial optimization problem that consists of choosing an optimal assignment of *n* agents and *n* tasks; however, when an agent is assigned to a task, it comes with a cost. The objective would be to find a solution where you need to assign a unique agent to a unique task in a way that the total cost is minimized.

A mathematical model would be as it follows: let r_{ij} be the cost of agent m_i for task j_j. The elements of the matrix are said to be independent as it would not collide with any other elements in rows or columns. There, a set of *n* elements of the matrix must be chosen (r_{ij}) such that the cost of these is minimized.

Method to solve (Algorithm):

1. Locate the smallest element in the matrix
2. Replace all elements by rows and columns of the matrix by the index of the smallest element with a predefined high number in a way that does not interfere with the elements of the matrix.
3. Repeat steps 1 and 2 until the smallest number of the matrix is equal to the predefined high number.

Method to solve (Pseudocode):

```
read csv file
high number = 100,000

while True
    find minimum number
    find index of minimum number

    if minimum number is equal to predefined high number then
        break loop cycle

    replace columns and rows of indexed minimum number with high number

    add minimum number to total cost collection
```

Experiments

Normally, these solutions can be done quickly in certain conditions where n size of agents and tasks are low. A solution was implemented in different scenarios where *n* = 4, 10, 25, 50, 200 and the elements of the matrix where randomly generated from 1, 2, 3…1000 in integers that is going to be compared at the end.

Using Python and Numpy, four different instances of each different scenario were generated using Numpy, returning random integers from the “discrete uniform” distribution. Using the

pseudocode, a solution was implemented in Python in such a way that can be used for any n size of agents and tasks.

```
matrix = np.loadtxt('inst3_200.csv', dtype=int)
high_number = 100000

while True:
    # Finds the minimum number and the index in the matrix.
    min_number = matrix.min()
    min_index = np.unravel_index(matrix.argmin(), matrix.shape)

    # If there's no minimum number left in the matrix, the answer has been found
and the loop ends.
    if min_number == high_number:
        break

    # Replaces the rows and columns when a minimum number has been found.
    matrix[:, min_index[1]] = high_number
    matrix[min_index[0], :] = high_number
    assignment_cost += min_number
```

This process was done for each instance, where time was calculated for each one, as well as the total cost, and size. Then the results were saved to a comma-separated value file and plotted in matplotlib reading the statistics recollected from the previous results.

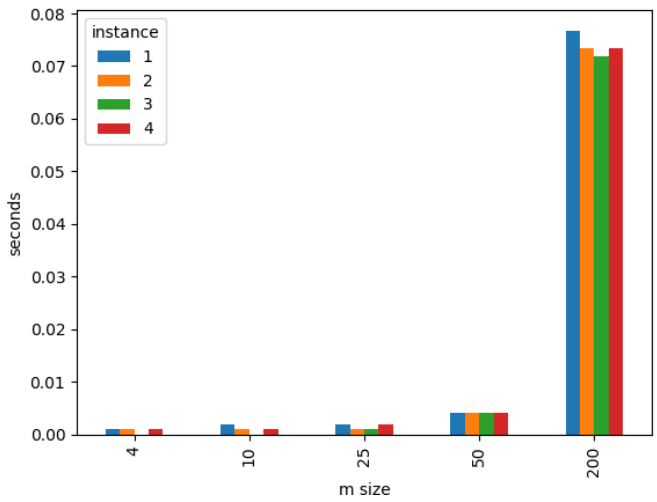


Fig 1. Total seconds it took to compile for each instance in Python

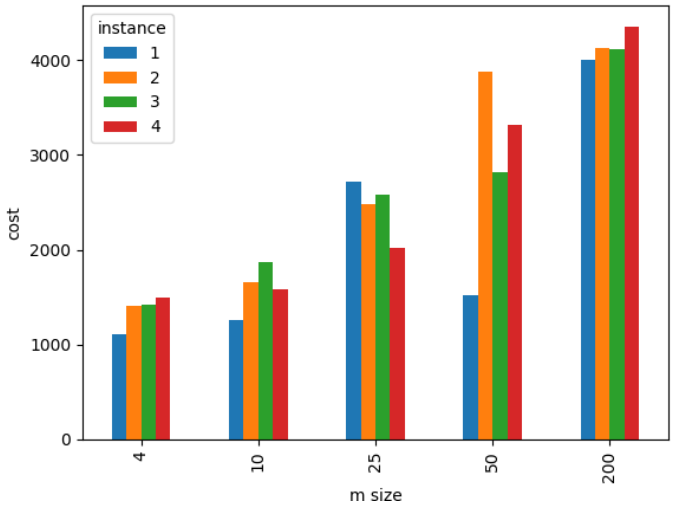


Fig 2. Total cost of each instance and size

According to the results of figure one, all instances where *m* size was of 4, 10, and 25 took less than 0.001 seconds to compute, however, instances where *m* size was of 50, it took between 0.003-0.004 seconds.

The total computing time grows exponentially when *m* size is of 200, as the total elements in the matrix reach 40,000 numbers, compared to when *m* is size 50 and total elements are of 2,500. These results vary between 0.07-0.09 seconds.

Compared to figured two, these results do not grow compared to the previous chart. Here, the total cost could be higher than the next *m* size, as seen in instance 4 where *m* equals 25, compared to instance 1 where *m* equals 50. Basically, as the *m* size grows, the total cost is expected to grow as well, but not always, as it can be assumed that a better solution can be found when this happens.

Conclusions

Based on the results, we can conclude that, the higher the *n* value of the assignment problem, the longer it will take to compute, as seen on figure one. The results are consistent enough to show this take.

However, in figure two, we do not see this happening. The total cost tends to be higher as *m* grows, but not always. Based on these results, we can conclude this method, while not the best, can be optimal for quick results compared to other algorithms.

References

- Munkres, J. (1957). Algorithms for the assignment and transportation problems. *Journal of the society for industrial and applied mathematics*, *5*(1), 32-38.
- Numpy and Scipy Documentation — Numpy and Scipy documentation. (2019). Docs.scipy.org., from https://docs.scipy.org/doc/

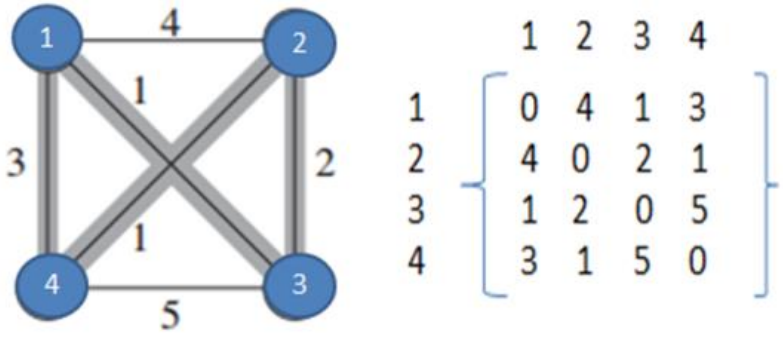
Traveling Salesman Problems solving methods

Description

The Traveling Salesman Problem (TSP) is a combinatorial optimization problem in operations research in where, given a set of nodes of cities, find a route where the total kilometers, times or costs are at a minimum. In a small number of cities, the optimal route can be found; however, it increases enormously when the number of cites increases.

The problem owes it name to the problem of determining the optimal route of a traveler who must visit n places and must return to his place of departure at the end of the journey.

The problem is generally presented in a set of nodes, represented by matrix of distances, or a list of cities and its coordinates:



City/City	1	2	3	4	5	6
1	0	10	100	50	33	66
2	10	0	22	86	952	3
3	100	22	0	6	86	2
4	50	86	6	0	5	4
5	33	952	86	5	0	9
6	66	3	2	4	9	0

Heuristics Pseudo-code

Given a list of cities and its coordinates, we must apply the Nearest Neighbor and the Cheapest Insertion heuristic methods, along with the route and cost, and if possible, compare them with the optimal solutions available, starting from the first city.

Nearest Neighbor algorithm

1. Initialize all vertices as unvisited.
2. Select an arbitrary vertex, set it as the current vertex u. Mark u as visited
3. Find out the shortest edge connecting the current vertex u and an unvisited vertex V.
4. Set V as the current vertex u. Mark V as visited
5. If all the vertices in the domain are visited, then end. Else, go to step 3.

Cheapest insertion algorithm

1. Start with a sub-graph consisting of node i only.
2. Find node r such that C_{ir} is minimal and form sub-tour $i - r - r$
3. Find (i, j) in sub-tour and r not, such that $C_{ir} + C_{rj} - C_{ij}$ is minimal. Insert r between i and j.
4. Repeat until all cities are visited

Nearest Neighbor Pseudo-code

```
high number = 10,000
starting_city = 0
route = [starting_city]
for count in distances table.length:
    for city in route:
        replace distances_table city index with high number
    min_number = find minimum number in distances_table[count]
    min_number_index = find index of minimum number in distances_table[count]
    append mind_number_index to route
```

Cheapest Insertion Pseudo-code

```
high number = 10,000
i_city = starting_city
route = [i_city]

j_city = index of minimum number in distances_table[i_city]
replace distances_table[i_city][j_city] with high number
route = [i_city, j_city]
j_city = index of minimum number in distances_table[i_city]
replace distances_table[i_city][j_city] with high number
insert j_city between index - and 1 of list

k_cities = list of all cities that are not on route

for count in k_cities.length:
    distances = []
    for city in k_cities:
        formula_distances = []
        for count2 in route.length - 1:
            i_city = route[count2]
            j_city = route[count2 + 1]
            formula_result = distances_table[i_city][k_city] + distances_table[k_city][
j_city] - distances_table[i_city][j_city]
            append formula_result to formula_array
        append forumla distances to distances
    min_cost_index = find minimum cost index in all of distances list
    city_to_add = city found at min_cost_index in list
    insert city_to_add to route list with the index of min_cost_index
    k_cities = list of all cities that are not on route

append starting_city to route
```

Experimental Tests

Tests were processed from a total of 9 different instances found at http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/. The name of each instance indicates the number of cities and includes the nodes and coordinates in a .tsp file. Some files have optimal solutions available: **a280.tsp, att48.tsp, berlin52.tsp, ch130.tsp, ch150.tsp**.

The results show the different processing time for each method and the total time to run the script in python. The script was run for all available files and compared to optimal solution if available.

PC Information

- Intel Core i5 CPU 2.30 GHz
- 8 GB RAM
- Windows 10
- GPU Nvidia GeForce 940 MX

Computational Time in seconds

Filename	Total Time in seconds	Cheapest Insertion Seconds	Nearest Neighbour Seconds
a280	2.6867	2.5427	0.0099
ali535	16.6876	16.2386	0.0379
att48	0.0403	0.011	0
att532	14.8229	14.3835	0.0419
berlin52	0.0428	0.016	0
bier127	0.2295	0.1865	0.003
ch130	0.2536	0.1925	0.002
ch150	0.388	0.3032	0.007
kroB200	0.8258	0.751	0.006

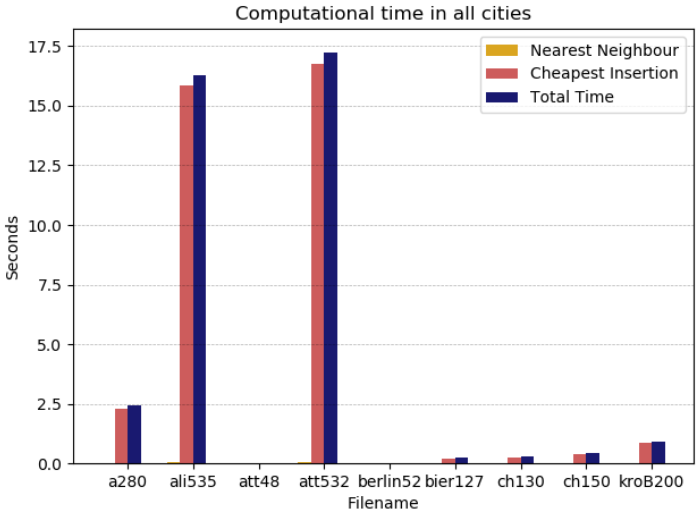
The total time represents the processing time reading the file and representing it to a matrix while also adding up the other two methods processing time.

Cost and Gap

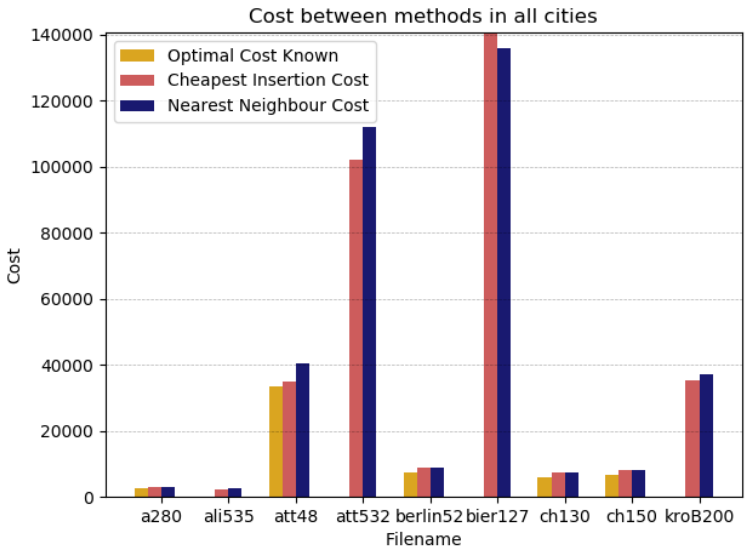
Filename	Optimal Cost (if known)	CI Method Cost	NN Method Cost	CI Gap	NN Gap	Minimum Known Gap
a280	2586.76965	3073.2687	3148.10993	18.8072043 %	21.7004358 %	2.4352%
ali535	0	2369.72531	2671.06834	0	0	12.7163%
att48	33523.7085	34817.9529	40526.4211	3.86068392	20.8888362 %	16.3951%
att532	0	102201.612	112099.446	0	0	9.6846%
berlin52	7544.3659	9014.89316	8980.91828	19.4917277	19.0413932 %	0.3783%
bier127	0	140690.935	135751.778	0	0	3.6383%
ch130	6110.86095	7279.2076	7575.28629	19.1191823 %	23.9643048 %	4.0674%
ch150	6532.28093	7991.22094	8194.61433	22.3343121 %	25.4479777 %	2.5452%
kroB200	0	35421.7008	36981.5901	0	0	4.4037%

If an optimal solution was available, the script would compare it to the optimal solution and to the cheapest one found (represented in the minimum known gap). Otherwise it just compares to minimum known gap.

Analysis



Computational time for the nearest neighbor method was found to be almost non-existent as the highest found was 0.04 seconds on the 532 cities and even lesser on the others. However, implementing the cheapest insertion method took a big hit on the computational time for most of the processing of the cities. Going from 2 seconds in a 280 node connected cities to 15.5 seconds in a 535 node connected cities. Total time is found by reading the files and implementing all methods.



The optimal cost known for some cities is found to be closer to the other two methods. The cheapest insertion method was found to be better in all cites except for two: bier127 and berlin52.

Conclusions

It was found to the nearest neighbor was a fast method to get results in a set of cities, but second in finding costs in most cities.

The cheapest insertion was found to be a higher cost on the computational time but generally found to be finding lower costs in cities than the nearest neighbor method.

References

- Mijwil, Maad (2016) Traveling Salesman Problem Mathematical Description. Journal
- Matai, R., Singh, S.P., & Mittal, M.L. (2010). Traveling salesman problem: an overview of applications, formulations, and solution approaches. Traveling salesman problem, theory and applications, 1.
- Georg, S. (2008). MP-TESTDATA-The TSPLIB symmetric traveling salesman problem instances.

Local Search

Local Search Description

Local search algorithms are frequently used to tackle the TSP problem. They iteratively improve the current solution by searching for a better one in its predefined neighborhood. The algorithm stops when there is no better solution in the given neighborhood or if a certain number of iterations has been reached.

The 2-opt is a successful algorithm to solve larger TSP instances. It's a local search algorithm where you select two boundaries and reverse the nodes between these. The result is a new solution.

Pseudocode

First Improvement

```
counter = 0
max_counter = 200
while loop:
    city 1 = random city in list
    city 2 = random city after city 1 in list
    new_solution = reverse function between city 1 and city 2
    new_cost = calculate_cost(new_solution)
    if new_cost < old_cost:
        print("a new solution was found")
        counter = 0
    else:
       counter = counter + 1
    if counter == max_counter = :
        break loop
```

Experimental Tests

The experimental tests were divided into two parts; the first part focuses on using the 2-opt method in a full neighborhood. Two tests were performed with a time limit of 120 seconds and 30 minutes (1800 seconds).

Tests were processed from a total of 9 different instances found at http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/. The name of each instance indicates the number of cities and includes the nodes and coordinates in a .tsp file.

PC Information

- Intel Core i5 CPU 2.30 GHz
- 8 GB RAM
- Windows 10
- GPU Nvidia GeForce 940 MX

Reverse method computational time

Full Neighborhood reverse method (120 seconds limit)

A time limit of 90 seconds was applied for the full neighborhood reverse method.

Filename	Best cost found between the two methods NN and CI	Local Search computational time for full neighborhood	Best cost found at local search	Gap between both costs
a280	3073.2687	197.499 seconds	2918.52514	5.3021 %
ali535	2369.72531	343.6829 seconds	2340.82978	1.2344 %
att48	34817.9529	0.3736 seconds	34153.1938	1.9464 %
att532	102201.612	332.8576 seconds	101665.671	0.5272 %
berlin52	8980.91828	1.7569 seconds	7841.38779	14.5323 %
bier127	135751.778	63.8724 seconds	120473.77	12.6816 %
ch130	7279.2076	33.0957 seconds	6706.09753	8.5461 %
ch150	7991.22094	72.5554 seconds	7317.20891	9.2113 %
kroB200	35421.7008	180.0757 seconds	33324.7555	6.2925 %

Full Neighborhood reverse method (30 minutes limit)

A time limit of 30 minutes (1800 seconds) was applied for the full neighborhood reverse method.

Filename	Best cost found between the two methods NN and CI	Local Search computational time for full neighborhood	Best cost found at local search	Gap between both costs
a280	3073.2687	745.069 seconds	2817.29206	9.0859 %
ali535	2369.72531	2015.5911 seconds	2307.46773	2.6981 %
att48	34817.9529	0.6508 seconds	34153.1938	1.9464 %
att532	102201.612	1934.7394 seconds	100581.177	1.6111 %
berlin52	8980.91828	2.4214 seconds	7841.38779	14.5323 %
bier127	135751.778	65.4044 seconds	120473.77	12.6816 %
ch130	7279.2076	33.5339 seconds	6706.09753	8.5461 %
ch150	7991.22094	90.048 seconds	7317.20891	9.2113 %
kroB200	35421.7008	374.7446 seconds	33136.9236	6.895 %

First Improvement and sequence reverse method (50 tries)

A counter of 50 tries was applied to the first improvement sequencing method.

Filename	Best cost found between the two methods NN and CI	Local Search computational time for sequence method	Best cost found at local search	Gap between both costs
a280	3073.2687	0.0439 seconds	0	0 %
ali535	2369.72531	0.0519 seconds	0	0 %
att48	34817.9529	0.006 seconds	0	0 %
att532	102201.612	0.0618 seconds	0	0 %
berlin52	8980.91828	0.0259 seconds	8725.25207	2.8468 %
bier127	135751.778	0.0718 seconds	130817.931	3.6345 %
ch130	7279.2076	0.014 seconds	0	0 %
ch150	7991.22094	0.0289 seconds	0	0 %
kroB200	35421.7008	0.0269 seconds	0	0 %

First Improvement and sequence reverse method (100 tries)

A counter of 100 tries was applied to the first improvement sequencing method.

Filename	Best cost found between the two methods NN and CI	Local Search computational time for sequence method	Best cost found at local search	Gap between both costs
a280	3073.2687	0.0678 seconds	0	0 %
ali535	2369.72531	0.2204 seconds	0	0 %
att48	34817.9529	0.0189 seconds	0	0 %
att532	102201.612	0.1227 seconds	0	0 %
berlin52	8980.91828	0.1157 seconds	8206.24418	8.6258 %
bier127	135751.778	0.0808 seconds	135403.321	0.2567 %
ch130	7279.2076	0.0339 seconds	0	0 %
ch150	7991.22094	0.0359 seconds	0	0 %
kroB200	35421.7008	0.0588 seconds	0	0 %

First Improvement and sequence reverse method (200 tries)

A counter of 200 tries was applied to the first improvement sequencing method.

Filename	Best cost found between the two methods NN and CI	Local Search computational time for sequence method	Best cost found at local search	Gap between both costs
a280	3073.2687	0.2097 seconds	3056.28702	0.5526 %
ali535	2369.72531	0.2483 seconds	2366.88326	0.1199 %
att48	34817.9529	0.0521 seconds	34310.3545	1.4579 %
att532	102201.612	0.4258 seconds	102114.983	0.0848 %
berlin52	8980.91828	0.0419 seconds	8414.78008	6.3038 %
bier127	135751.778	0.1835 seconds	132275.429	2.5608 %
ch130	7279.2076	0.1077 seconds	7263.49837	0.2158 %
ch150	7991.22094	0.0684 seconds	7988.07834	0.0393 %
kroB200	35421.7008	0.0598 seconds	0	0 %

The full neighborhood method results can be seen as consistent and will always find the same solution as opposed to the first improvement method, where chooses randomly and there is no consistent results and will almost always produce different results in each process.

Analysis

In the full neighborhood method, there were no notable differences between the two tests (120 seconds vs 30 minutes) for the last five instances. In the two instances where it had above 500 cities, it only found a minimal increase of 1% in the gap. The only notable difference was found in the a280.tsp instance where it increased to 4%.

For the first improvement method, the first two tests (50 and 100 tries) didn't find much improvement on the found costs, it did however, on the 200 tries with not much significant improvement but in a fast-computational time.

Conclusions

A full neighborhood method is effective when trying to find cost improvements in city nodes below 200 cities, as the maximum computational time it took was around 3 minutes (kroB200). However, it takes a computational time hit above where it doesn't find much improvement.

The first improvement method is found to be ineffective when trying to find significant improvement but found to be a really fast computational method as it found a 8% increase in the berlin52.tsp problem in just under 0.12 seconds.

References

- Mijwil, Maad (2016) Traveling Salesman Problem Mathematical Description. Journal
- Matai, R., Singh, S.P., & Mittal, M.L. (2010). Traveling salesman problem: an overview of applications, formulations, and solution approaches. Traveling salesman problem, theory and applications, 1.
- Georg, S. (2008). MP-TESTDATA-The TSPLIB symmetric traveling salesman problem instances.
- Mersmann, O., Bischl, B., Bossek, J., Trautmann, H., Wagner, M., & Neumann, F. (2012, January). Local search and the traveling salesman problem: A feature-based characterization of problem hardness. In International Conference on Learning and Intelligent Optimization (pp. 115-129). Springer, Berlin, Heidelberg.

Greedy Randomized Adaptive Search Procedure (GRASP)

GRASP is an iterative randomized method in which each iteration provides a solution to the problem at hand. The final solution over all GRASP iterations is kept as the final result. There are two phases within each GRASP iteration: the first makes an initial solution via a randomized greedy function; the second applies a local search procedure to the constructed solution in hope of finding an improvement.

Traveling Salesman Problem

The Traveling Salesman Problem (TSP) is a combinatorial optimization problem in operations research in where, given a set of nodes of cities, find a route where the total kilometers, times or costs are at a minimum. In a small number of cities, the optimal route can be found; however, it increases enormously when the number of cites increases.

Because of its simplicity, the nearest neighbor heuristic is one of the first algorithms that comes to mind in attempting to solve the traveling salesman problem (TSP), in which a salesman has to plan a tour of cities that is of minimal length. In this heuristic, the salesman starts at some city and then visits the city nearest to the starting city, and so on, only taking care not to visit a city twice. At the end all cities are visited, and the salesman returns to the starting city.

Local Search

Local search algorithms are frequently used to tackle the TSP problem. They iteratively improve the current solution by searching for a better one in its predefined neighborhood. The algorithm stops when there is no better solution in the given neighborhood or if a certain number of iterations has been reached.

The 2-opt is a successful algorithm to solve larger TSP instances. It's a local search algorithm where you select two boundaries and reverse the nodes between these. The result is a new solution

Pseudocode

```
1.  procedure grasp()
2.      InputInstance();
3.      for GRASP stopping criterion not satisfied →
4.          ConstructGreedyRandomizedSolution(Solution);
5.          LocalSearch(Solution);
6.          UpdateSolution(Solution,BestSolutionFound);
7.      end for;
8.      return(BestSolutionFound)
9.  end grasp;
```

The GRASP iterations end when some termination criterion, such as maximum number of iterations have occurred (as done in this project). Line 4 is the GRASP construction phase (Nearest Neighbor Algorithm TSP), while line 4 is the local search phase (reverse 2opt first improvement). If an improved solution is found, the initial solution is updated in line 6.

Greedy Randomized Solution

An initial solution is created using the nearest neighbor heuristic algorithm. A key element in this phase is the restricted candidate list (RCL) which consists of adding a random element from a set of best elements and be added to the partial solution. The choice of these best elements is made using a greedy evaluation that assures the RCL contains the best candidates. This greedy evaluation is $c_{min} - \alpha(c_{max} - c_{max})$ where c (in this project) indicates the distances list of said city and α indicates the candidate resolution. The probabilistic component of a GRASP is characterized by randomly choosing one of the best candidates in the list, but not necessarily the top candidate.

Local Search on solution

The 2-opt method is local search algorithm for solving TSP problems and doing local search on solutions. This method is found in line 5 *(LocalSearch(Solution))*. Another key element found in this phase is the added randomness in 2-opt as it chooses from a random point in a solution and reverses it, it iterates until it finds a better solution and it stops when it does not.

Updating the solution and stopping criterion

If solution in previous line is found to be better than initial solution, then it is to be updated. This iterative process is to be repeated a set number of times.

Experimental Tests

Tests were processed from a total of 4 different instances found at http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/. The name of each instance indicates the number of cities and includes the nodes and coordinates in a .tsp file. Each of these files include an optimal route and these include: **att48, berlin52, ch130, ch150**.

The results show the different processing time for each method and the best cost found to run the script in python. The script was run for all files.

Some key variables include:

- α = 0.4
- GRASP stopping criterion = 350
- first improvement stopping criterion = 300

PC Information

- Intel Core i5 CPU 2.30 GHz
- 8 GB RAM
- Windows 10
- GPU Nvidia GeForce 940 MX

Results

When compared to the optimal problems.

Filename	Time	Optimal Cost	Best found cost	Gap
att48	68.7043	33523.7085	34146.4951	1.85774968
berlin52	78.1427	7544.3659	7853.03989	4.09145034
ch130	573.5108	6110.86095	7806.45933	27.7472912
ch150	704.9091	6532.28093	8874.07316	35.8495332

Other Results

With local search iterations as 50.

Filename	Time	Best cost found	Alpha	GRASP Iterations
a280	3.9176	10609.2964	0.3	10
ali535	29.0108	11002.1444	0.3	10
att532	21.0308	482717.576	0.3	10
berlin52	0.5143	10579.151	0.3	10
bier127	2.7448	227156.58	0.3	10
ch130	1.1948	14796.5039	0.3	10
ch150	2.024	16387.6838	0.3	10
kroB200	3.6632	88729.8854	0.3	10

Filename	Time	Best cost found	Alpha	GRASP Iterations
a280	0.6134	11707.8913	0.3	2
ali535	6.6931	11304.2439	0.3	2
att532	7.5957	451618.457	0.3	2
berlin52	0.1326	11518.164	0.3	2
bier127	0.4883	233848.408	0.3	2
ch130	0.3531	16248.5733	0.3	2
ch150	0.3391	16520.4036	0.3	2
kroB200	0.4618	117637.599	0.3	2

With higher grasp iterations, there is a better improvement on the best cost found as seen in every case that it got improved. However, this comes with a little higher computational time.

With α = 0.6

Filename	Time	Best cost found	Alpha	GRASP Iterations
a280	3.2284	11602.2259	0.6	2
ali535	7.4751	13435.6059	0.6	2
att532	8.4516	515344.597	0.6	2
berlin52	0.1087	10807.7387	0.6	2
bier127	0.5017	286662.605	0.6	2
ch130	0.5286	15938.9498	0.6	2
ch150	0.4917	21071.6878	0.6	2
kroB200	1.501	88442.0334	0.6	2

Filename	Time	Best cost found	Alpha	GRASP Iterations
a280	11.9381	11285.7299	0.6	10
ali535	51.3411	12292.9896	0.6	10
att532	39.0895	488553.388	0.6	10
berlin52	0.641	10353.0964	0.6	10
bier127	3.4548	222057.592	0.6	10
ch130	2.907	15513.7982	0.6	10
ch150	3.3765	17815.0943	0.6	10
kroB200	6.1209	98127.6945	0.6	10

Improving α made a slightly worse as expected, since it makes for even more randomness in the candidate selection.

With α = 1

Filename	Time	Best cost found	Alpha	GRASP Iterations
a280	2.8005	11898.9924	1	2
ali535	8.5481	15432.3136	1	2
att532	9.9205	519865.811	1	2
berlin52	0.1098	11808.7163	1	2
bier127	0.6533	272750.886	1	2
ch130	0.6582	16267.0557	1	2
ch150	0.8916	19833.4332	1	2
kroB200	1.8132	109558.662	1	2

Filename	Time	Best cost found	Alpha	GRASP Iterations
a280	14.965	10973.9351	1	10
ali535	44.8483	12682.2293	1	10
att532	43.6607	558335.179	1	10
berlin52	0.5202	12438.188	1	10
bier127	3.3999	257138.634	1	10
ch130	4.469	14515.6771	1	10
ch150	4.496	16678.9841	1	10
kroB200	8.4659	97728.0491	1	10

Analysis

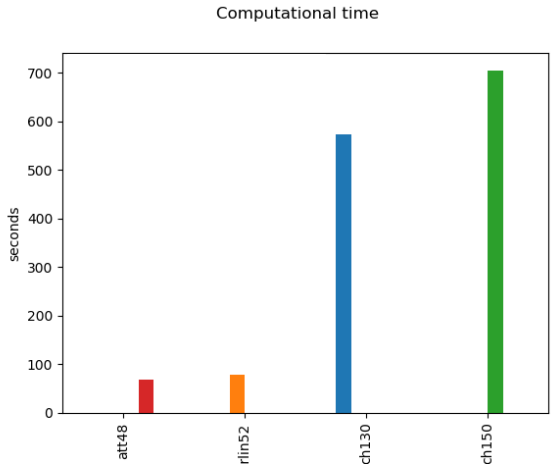


Fig 1. Computational time

TSP files found to be at 50 cities took a computational time of around 70 seconds with minimal difference with the best cost found and the optimal cost. However, this increases with the number of cities as it takes a large quantity of time to compute procedure for the last two cities in the TSPLIB.

On the last two TSP found problems, a difference can be noticed in the best cost found and optimal cost section with very large differences.

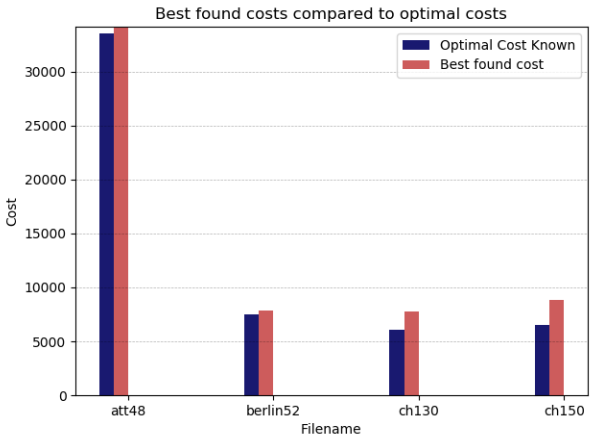


Fig 2. Costs compared to optimal

In the first two files, the difference is minimal, as perhaps increasing the iteration numbers could give us a close or same result as the optimal one. This does not happen with 'chxx' files as they take a great computational time and the differences are found to be large.

Conclusion

This procedure is found to be efficient for low quantity of cities and somewhat effective for a large quantity of cities found in TSP problems. Further tests are required.

References

- Georg, S. (2008). MP-TESTDATA-The TSPLIB symmetric traveling salesman problem instances.
- Mersmann, O., Bischl, B., Bossek, J., Trautmann, H., Wagner, M., & Neumann, F. (2012, January). Local search and the traveling salesman problem: A feature-based characterization of problem hardness. In International Conference on Learning and Intelligent Optimization (pp. 115-129). Springer, Berlin, Heidelberg.
- Feo, T. A., & Resende, M. G. (1995). Greedy randomized adaptive search procedures. Journal of global optimization, 6(2), 109-133.
- Glover, F. W., & Kochenberger, G. A. (Eds.). (2006). Handbook of metaheuristics (Vol. 57). Springer Science & Business Media.