Longest Path Problem

Ahnaf Faisal, 1505005
Raihanul Alam, 1505010
Mahim Mahbub, 1505022
Zahin Wahab, 1505031
Bishal Basak Papan, 1505043

Department of Computer Science and Engineering,
Bangladesh University of Engineering and Technology

December 8, 2020

Outline:

1. Recap
2. Definition
3. Algorithms we discussed
4. Recap of Dynamic Programming
5. Simulated Annealing
6. Ant Colony
7. Dataset Description
8. Performance Analysis
9. References

Where we left off

- We have already talked about the **Longest Path Problem** in our previous presentations.

Where we left off

- We have already talked about the **Longest Path Problem** in our previous presentations.
- We showed exact exponential algorithms, approximation algorithms (for a specific class of graphs) and meta-heuristics for this problem.

Where we left off

- We have already talked about the **Longest Path Problem** in our previous presentations.
- We showed exact exponential algorithms, approximation algorithms (for a specific class of graphs) and meta-heuristics for this problem.
- Today rather than proposing algorithms, we will implement proposed algorithms from previous weeks.

Where we left off

- We have already talked about the **Longest Path Problem** in our previous presentations.
- We showed exact exponential algorithms, approximation algorithms (for a specific class of graphs) and meta-heuristics for this problem.
- Today rather than proposing algorithms, we will implement proposed algorithms from previous weeks.
- But before we proceed any further, let's just shed some light to our previous discussion.

What is a Longest Path Problem?

Let us present the optimization version of this problem.

What is a Longest Path Problem?

Let us present the optimization version of this problem.

Optimization Version

Given a weighted graph G, find a simple path in this graph which has the maximum weight.

Exact Exponential Algorithms

- We have proved that Naive algorithm is super exponential with time complexity $O(2^n n^n)$.
- So we focused on **improving** the time complexity and designed a dynamic programming algorithm whose time complexity is exponential but better than naive brute force.
- After extensive analysis, we have showed that if total number of sub-problems of size k is $k\binom{n}{k}$, then total running time of DP algorithm is: $O(\sum_{k=1}^{n} k^2\binom{n}{k})$.
- This dynamic programming approach (inspired from DP algorithm proposed by **Held and Karp** [1] to solve the Travelling Salesman Problem) is the best exact algorithm we could propose with our little knowledge. We have implemented it for this week from scratch.

Exact Exponential Algorithms

- We have proved that Naive algorithm is super exponential with time complexity $O(2^n n^n)$.
- So we focused on **improving** the time complexity and designed a dynamic programming algorithm whose time complexity is exponential but better than naive brute force.
- After extensive analysis, we have showed that if total number of sub-problems of size k is $k\binom{n}{k}$, then total running time of DP algorithm is: $O(\sum_{k=1}^{n} k^2\binom{n}{k})$.
- This dynamic programming approach (inspired from DP algorithm proposed by **Held and Karp** [1] to solve the Travelling Salesman Problem) is the best exact algorithm we could propose with our little knowledge. We have implemented it for this week from scratch.

Exact Exponential Algorithms

- We have proved that Naive algorithm is super exponential with time complexity $O(2^n n^n)$.
- So we focused on **improving** the time complexity and designed a dynamic programming algorithm whose time complexity is exponential but better than naive brute force.
- After extensive analysis, we have showed that if total number of sub-problems of size k is $k\binom{n}{k}$, then total running time of DP algorithm is: $O(\sum_{k=1}^{n} k^2 \binom{n}{k})$.
- This dynamic programming approach (inspired from DP algorithm proposed by **Held and Karp** [1] to solve the Travelling Salesman Problem) is the best exact algorithm we could propose with our little knowledge. We have implemented it for this week from scratch.

Exact Exponential Algorithms

- We have proved that Naive algorithm is super exponential with time complexity $O(2^n n^n)$.
- So we focused on **improving** the time complexity and designed a dynamic programming algorithm whose time complexity is exponential but better than naive brute force.
- After extensive analysis, we have showed that if total number of sub-problems of size k is $k\binom{n}{k}$, then total running time of DP algorithm is: $O(\sum_{k=1}^{n} k^2\binom{n}{k})$.
- This dynamic programming approach (inspired from DP algorithm proposed by **Held and Karp** [1] to solve the Travelling Salesman Problem) is the best exact algorithm we could propose with our little knowledge. We have implemented it for this week from scratch.

Approximate Algorithms

- No polynomial time algorithm is available for longest path problem unless $P = NP$.
- Exact solution will always need exponential time.
- To achieve good performance, we have to relax accuracy of the solution.
- This is where **Approximate Algorithms** come.
- We will see two types of **Approximate Algorithms**: **Approximation Algorithms** and **Meta-heuristics**.
- **Approximation algorithms** guarantee a bound for the solution, whereas accuracy of solution generated by Meta-heuristics depends on the initial solution set and hyper-parameters.

Approximate Algorithms

- No polynomial time algorithm is available for longest path problem unless $P = NP$.
- Exact solution will always need exponential time.
- To achieve good performance, we have to relax accuracy of the solution.
- This is where **Approximate Algorithms** come.
- We will see two types of **Approximate Algorithms**: **Approximation Algorithms** and **Meta-heuristics**.
- **Approximation algorithms** guarantee a bound for the solution, whereas accuracy of solution generated by Meta-heuristics depends on the initial solution set and hyper-parameters.

Approximate Algorithms

- No polynomial time algorithm is available for longest path problem unless $P = NP$.
- Exact solution will always need exponential time.
- To achieve good performance, we have to relax accuracy of the solution.
- This is where **Approximate Algorithms** come.
- We will see two types of **Approximate Algorithms**: **Approximation Algorithms** and **Meta-heuristics**.
- **Approximation algorithms** guarantee a bound for the solution, whereas accuracy of solution generated by Meta-heuristics depends on the initial solution set and hyper-parameters.

Approximate Algorithms

- No polynomial time algorithm is available for longest path problem unless $P = NP$.
- Exact solution will always need exponential time.
- To achieve good performance, we have to relax accuracy of the solution.
- This is where **Approximate Algorithms** come.
- We will see two types of **Approximate Algorithms**: **Approximation Algorithms** and **Meta-heuristics**.
- **Approximation algorithms** guarantee a bound for the solution, whereas accuracy of solution generated by Meta-heuristics depends on the initial solution set and hyper-parameters.

Approximate Algorithms

- No polynomial time algorithm is available for longest path problem unless $P = NP$.
- Exact solution will always need exponential time.
- To achieve good performance, we have to relax accuracy of the solution.
- This is where **Approximate Algorithms** come.
- We will see two types of **Approximate Algorithms**: **Approximation Algorithms** and **Meta-heuristics**.
- **Approximation algorithms** guarantee a bound for the solution, whereas accuracy of solution generated by Meta-heuristics depends on the initial solution set and hyper-parameters.

Approximate Algorithms

- No polynomial time algorithm is available for longest path problem unless $P = NP$.
- Exact solution will always need exponential time.
- To achieve good performance, we have to relax accuracy of the solution.
- This is where **Approximate Algorithms** come.
- We will see two types of **Approximate Algorithms**: **Approximation Algorithms** and **Meta-heuristics**.
- **Approximation algorithms** guarantee a bound for the solution, whereas accuracy of solution generated by Meta-heuristics depends on the initial solution set and hyper-parameters.

Approximation Algorithms

- We have proven the following theorems.

Theorem

If the longest-path problem has a polynomial-time algorithm that achieves a **constant factor approximation**, then it has a **PTAS**.

Theorem

There is **no PTAS** for the Longest Path problem; **unless P = NP**

Approximation Algorithms

- We have proven the following theorems.

Theorem

If the longest-path problem has a polynomial-time algorithm that achieves a **constant factor approximation**, then it has a **PTAS**.

Theorem

There is **no PTAS** for the Longest Path problem; **unless P = NP**

- These lead to the following corollary.

Corollary

There does not exist a **constant factor approximation algorithm** for the longest path problem, **unless P = NP**

Approximation Algorithms

- We have proven the following theorems.

Theorem

If the longest-path problem has a polynomial-time algorithm that achieves a **constant factor approximation**, then it has a **PTAS**.

Theorem

There is **no PTAS** for the Longest Path problem; **unless P = NP**

- These lead to the following corollary.

Corollary

There does not exist a **constant factor approximation algorithm** for the longest path problem, **unless P = NP**

- All the proofs presented so far are from **Karger et al. (1997)** [2].

Meta-Heuristics

- As we do not have an approximation algorithm for longest path problem, we have to move on to **Meta-Heuristics**.
- We proposed four **Meta-heuristic algorithms** in our last presentation namely **ant colony, genetic algorithms, simulated annealing** and **particle swarm optimization**.
- For this final week, we will implement two of these four aforementioned algorithms: **ant colony** and **simulated annealing**.

Meta-Heuristics

- As we do not have an approximation algorithm for longest path problem, we have to move on to **Meta-Heuristics**.
- We proposed four **Meta-heuristic algorithms** in our last presentation namely **ant colony, genetic algorithms, simulated annealing** and **particle swarm optimization**.
- For this final week, we will implement two of these four aforementioned algorithms: **ant colony** and **simulated annealing**.

Meta-Heuristics

- As we do not have an approximation algorithm for longest path problem, we have to move on to **Meta-Heuristics**.
- We proposed four **Meta-heuristic algorithms** in our last presentation namely **ant colony, genetic algorithms, simulated annealing** and **particle swarm optimization**.
- For this final week, we will implement two of these four aforementioned algorithms: **ant colony** and **simulated annealing**.

How will we compare?

- Meta-heuristics, as per our discussions, does not guarantee an optimal solution.

How will we compare?

- Meta-heuristics, as per our discussions, does not guarantee an optimal solution.
- So we will use the solutions from our Dynamic Programming approach (exact algorithms) to compare accuracy of solutions generated by meta-heuristcs.

How will we compare?

- Meta-heuristics, as per our discussions, does not guarantee an optimal solution.
- So we will use the solutions from our Dynamic Programming approach (exact algorithms) to compare accuracy of solutions generated by meta-heuristcs.
- But the datasets with large number of vertices could not be solved with DP approach as DP approach needs exponential time.

Dynamic Programming Approach - Recap

As discussed in earlier weeks, the recursion was set up similarly as the classic **Held-Karp** algorithm to solve for Travelling Salesman Problem [1].

Recurrence Relationship

$$OPT[c_i, S] = \max_{\forall c_j \in S - \{c_i\}} \{0, w(c_i, c_j) + OPT[c_j, S - \{c_i\}]\}$$
$$where\ c_j \in S - \{c_i\}$$

Code Snippet - Dynamic Programming (Generating subsets)

```
def generate_subsets_above_len(maxsize, least_len=0, max_len=2, exact_length=2):
    x = maxsize + 1 # len(s)
    s = np.arange(0, x)
    list_set = []
    for i in range(1, 1 << x):
        list_temp = [s[j] for j in range(x) if (i & (1 << j))]
        if (len(list_temp) == exact_length):
            list_set.append(list_temp)

    return list_set
```

Code Snippet - Dynamic Programming (Base Case)

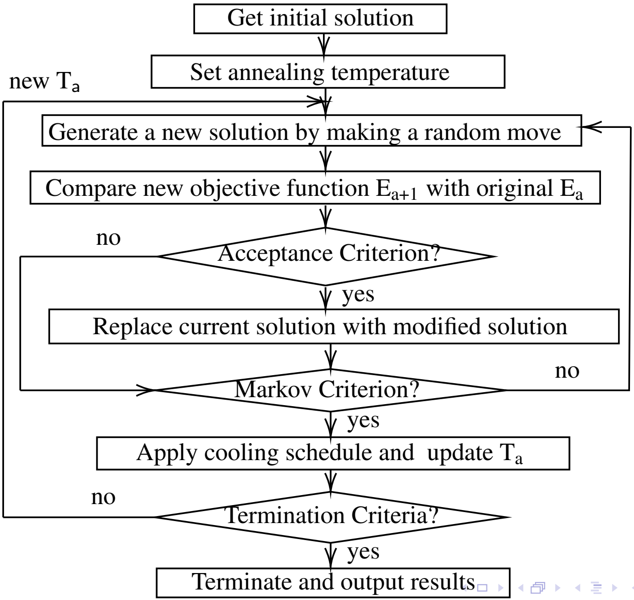
```
def compute_LP_DP(distance_matrix, NULL_VERTEX=-1, INFINITY=10000):
    source = 0 ## Assume source at 0 always
    OPT = {} ## Double dictionary key: pair of <source, {subset}> ||| value: (cost, kept_vertex)
    for outer_vertex in range(len(distance_matrix)): ## Single edge initialization . [Base Case]
        for inner_vertex in range(len(distance_matrix)):
            if inner_vertex == outer_vertex: ## Ignore for same
                continue
            cost = distance_matrix[outer_vertex][inner_vertex]
            list_1 = []
            if cost > 0: ## Only now CHOOSE this EDGE.
                list_1.append(inner_vertex)
                OPT[outer_vertex, tuple({inner_vertex})] = (cost, list_1)
            else: ## DO NOT choose this edge.
                list_1.append(NULL_VERTEX)
                OPT[outer_vertex, tuple({inner_vertex})] = (0, list_1)
```

Code Snippet - Dynamic Programming (Recursion)

```
for num_subset_length in range(2, len( distance_matrix )): ## Multiple edge condition. [Assume > 2 vertices !!]
    subsets_all = generate_subsets_above_len (maxsize=len(distance_matrix)−1, exact_length=num_subset_length) ##
        generate subset here
    for src_vertex in range(len( distance_matrix )):
        for subset in subsets_all :
            if src_vertex in subset: ## loop over all subsets for THIS length
                continue
            max_cost = −INFINITY
            best_neighbor = NULL_VERTEX, best_list = []
            for neighbor in subset: ## loop over THIS subset for EACH NEIGHBOR
                subset_without_inner_vertex = tuple( filter (lambda x: x!=neighbor, subset))
                current_cost_neighbor = distance_matrix[ src_vertex ][ neighbor] + OPT[neighbor,
                        subset_without_inner_vertex ][0] ## first−element is cost , second is
                if current_cost_neighbor > max_cost: ## update for THIS neighbor
                    max_cost = current_cost_neighbor
                    best_neighbor = neighbor
                    best_list = OPT[neighbor, subset_without_inner_vertex ][1] ## list of vertices

            update_list_flag = True
            if max_cost < 0: ## DON'T take anyone from THIS subset.
                max_cost = 0
                update_list_flag = False
            new_list = best_list .copy() ## keep a copy
            if update_list_flag == True:
                new_list . insert (0, best_neighbor ) ## add to head
            OPT[src_vertex, tuple(subset)] = (max_cost, new_list ) ## for now first−elemnt cost, second is
                neighbor/vertex_taken
return OPT ## Finally, Return
```

Simulated Annealing



Simulated Annealing

- At first we set the initial temperature, the final temperature and the maximum iteration number. We set the temperature range such that the value of $e^{\frac{del}{temp}}$ is not too high or not too low.
- Then, we select a random path(P) from source to destination
- At each temperature, we select a random path(S) and check the difference of the lengths of P and S. We update P on the basis of a random probability value(x) from 0 to 1 and the difference between x and $e^{\frac{del}{temp}}$.

Simulated Annealing

- At first we set the initial temperature, the final temperature and the maximum iteration number. We set the temperature range such that the value of $e^{\frac{del}{temp}}$ is not too high or not too low.
- Then, we select a random path(P) from source to destination
- At each temperature, we select a random path(S) and check the difference of the lengths of P and S. We update P on the basis of a random probability value(x) from 0 to 1 and the difference between x and $e^{\frac{del}{temp}}$.

Simulated Annealing

- At first we set the initial temperature, the final temperature and the maximum iteration number. We set the temperature range such that the value of $e^{\frac{del}{temp}}$ is not too high or not too low.
- Then, we select a random path(P) from source to destination
- At each temperature, we select a random path(S) and check the difference of the lengths of P and S. We update P on the basis of a random probability value(x) from 0 to 1 and the difference between x and $e^{\frac{del}{temp}}$.

SA Implementation

- We have implemented SA for solving LP using python language.
- We have used some libraries like, **networkx** and **numpy**.
- SA running time depends on the difference between maximum and minimum temperature and the maximum iteration count in each step.

SA Implementation

- We have implemented SA for solving LP using python language.
- We have used some libraries like, **networkx** and **numpy**.
- SA running time depends on the difference between maximum and minimum temperature and the maximum iteration count in each step.

SA Implementation

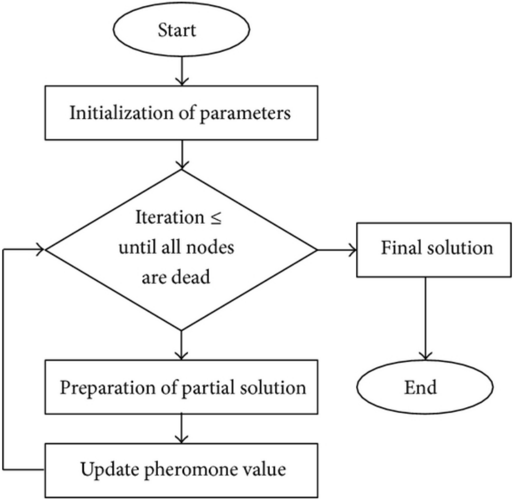
- We have implemented SA for solving LP using python language.
- We have used some libraries like, **networkx** and **numpy**.
- SA running time depends on the difference between maximum and minimum temperature and the maximum iteration count in each step.

Code Snippet - Simulated Annealing

```
def Anneal(G, source, destination, temp_init, temp_final, itermax):
    print("initial state:")
    P, length_p = select_random_path(G, source, destination)
    print("Path:", P)
    print("Length:", length_p)
    temp = temp_init
    best_path_lens = []
    best_path = P
    best_len = length_p
    while temp >= temp_final:
        iteration = 0
        while iteration < itermax:
            S, length_s = select_random_path(G, source, destination)
            if length_s > best_len:
                best_path = S

            delta = length_s - length_p
            if delta > 0:
                P = S
                length_p = length_s
            else:
                x = random.random()
                prob = math.exp(delta / temp)
                print(x)
                print(prob)
                if x < prob:
                    P = S
                    length_p = length_s
```

Ant Colony Flow Chart



Ant Colony

- We use probabilistic approach to build possible paths for each ant.

Ant Colony

- We use probabilistic approach to build possible paths for each ant.
- We use equation of the format $pr = ph * \alpha + weight * \beta$ where α and β are constants given as parameters.

Ant Colony

- We use probabilistic approach to build possible paths for each ant.
- We use equation of the format $pr = ph * \alpha + weight * \beta$ where α and β are constants given as parameters.
- For each iteration we find the local longest path and update the global longest path.

Ant Colony

- We use probabilistic approach to build possible paths for each ant.
- We use equation of the format $pr = ph * \alpha + weight * \beta$ where α and β are constants given as parameters.
- For each iteration we find the local longest path and update the global longest path.
- At the end of the iterations, the global longest path will be the solution our algorithm gives.

Code Snippet - Ant Colony Optimization

```
def build_path(self, probs):
    path = [self.init_vert]
    possible = probs.loc[1]
    possible = possible[~possible.index.isin(path + [self.N])]
    while not possible.empty:
        v = possible.sample(weights = 'weight').index.values[0]
        path.append(v)
        possible = probs.loc[v]
        possible = possible[~possible.index.isin(path + [self.N])]

    while self.N not in probs.loc[path[-1]].index:
        path.pop()
    path.append(self.N)
    path = list(zip(path, path[1:]))
    return path

def run(self, **params):
    solutions = []
    stats = []
    try:
        if 'seed' in params:
            np.random.seed(params['seed'])
        V, E = self.V, self.E
        ph = E.where(E.isnull(), 1)
        fitness = E
        gbest = (-float('inf'), [])
        glow=(float('inf'), [])
        for i in range(params['max_iter']):
            lbest = (-float('inf'), [])
            repeated_edges = defaultdict(lambda: -1)
```

Ant Colony Optimization

```
        probs = ph ** params['alpha'] + fitness ** params['beta']
        ants = []
        for k in range(params['ants']):
          path = self.build_path(probs)
          cost = E[E.index.isin(path)].sum()['weight']
          ants.append((path, cost))
          if cost > lbest[0]:
            lbest = (cost, list(path))
          if cost > gbest[0]:
            gbest = (cost, list(path))
          if cost < glow[0]:
            glow = (cost, list(path))
          for edge in path:
            repeated_edges[edge] += 1
        ants = pd.DataFrame(ants, columns = ['path', 'cost'])
        stats.append({
          'best': ants['cost'].max(),
          'worst': ants['cost'].min(),
          'mean': ants['cost'].mean(),
          'std': ants['cost'].std(),
          'size': ants['path'].apply(len).mean() + 1,
          'rep': sum(list(repeated_edges.values())),
          'lbest': lbest[1]
        })
        in_local_best = ph.index.isin(lbest[1])
        in_global_best = ph.index.isin(gbest[1])
        ph *= 1 - params['evap']
        ph[in_local_best] += params['Q'] * lbest[0]
        ph[in_global_best] += params['Q'] * gbest[0]
    except KeyboardInterrupt:
```

Dataset Description

- For comparison, we will use the datasets for the TSP problem from **Gerhard Reinelt (1991)** [3].

Dataset Description

- For comparison, we will use the datasets for the TSP problem from **Gerhard Reinelt (1991)** [3].
- All the graphs are **Complete** Graphs.

Dataset Description

- For comparison, we will use the datasets for the TSP problem from **Gerhard Reinelt (1991)** [3].
- All the graphs are **Complete** Graphs.
- Additionally, we will split the ha_30 dataset (30 nodes) into datasets with smaller number of nodes and perform further analysis.

Dataset Description

- For comparison, we will use the datasets for the TSP problem from **Gerhard Reinelt (1991)** [3].
- All the graphs are **Complete** Graphs.
- Additionally, we will split the ha_30 dataset (30 nodes) into datasets with smaller number of nodes and perform further analysis.
- Graphs with number of nodes > 20 will not be analyzed with exact algorithm (Dynamic Programming).
- Only the meta-heuristics will be used to perform analyses for those datasets ($|V| > 20$).

Dataset Details - Smaller Graphs

Details for dataset with $|V| < 20$. Dynamic Programming, Ant Colony, and Simulated Annealing are experimented with these datasets.

Dataset Name	Number of Vertices
co04	4
grid04	4
five	5
sh07	7
sp11	11
uk12	12
jb13	13
lau15	15
gr17	17

Dataset Details - Larger Graphs

Details for dataset with $|V| > 20$. Only Ant Colony and Simulated Annealing are experimented with these datasets.

Dataset Name	Number of Vertices
wg22	22
fri26	26
ha30	30
dantzig42	42
att48	48
kn57	57
wg59	59
sgb128	128
usca312	312

Dataset Details - Breakdown of ha30 Dataset

- Additionally, we have broken down the dataset for **ha30** previously described into smaller number of vertices for comparison with Dynamic Programming approach.
- Number of nodes varied from 5 $->$ 20 as shown below.

# Nodes	9	10	14	15	18	19	20

Performance Analysis on Various Datasets

- When exact algorithm DP is used, no destination vertex is necessary.

Performance Analysis on Various Datasets

- When exact algorithm DP is used, no destination vertex is necessary.
- However, meta-heuristics require a fixed destination vertex.

Performance Analysis on Various Datasets

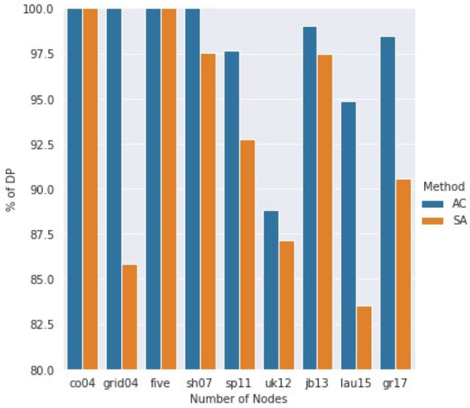
- When exact algorithm DP is used, no destination vertex is necessary.
- However, meta-heuristics require a fixed destination vertex.
- The destination vertex of exact algorithm DP is used in the meta-heuristics.

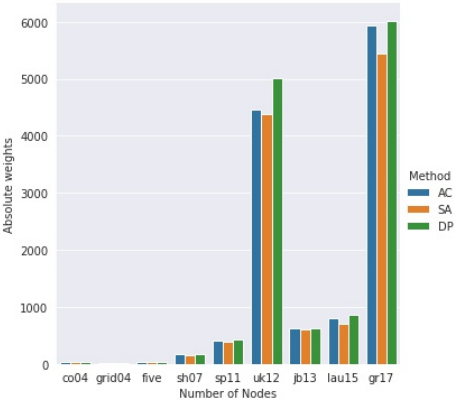
Comparison table

Table: Comparison between DP, Ant Colony and SA

Dataset	vertex	Source	Destn	DP	AC	SA
co04	4	0	2	26	26	26
grid04	4	0	1	14.12	14.12	14.12
five	5	0	1	26	26	26
sh07	7	0	2	163.4	163.4	159.4
sp11	11	0	1	427	417	396
uk12	12	0	4	5020	4458	4374
jb13	13	0	5	629	623	613
lau15	15	0	9	855	811	714
gr17	17	0	16	6021	5928	5454

Comparison for smaller datasets



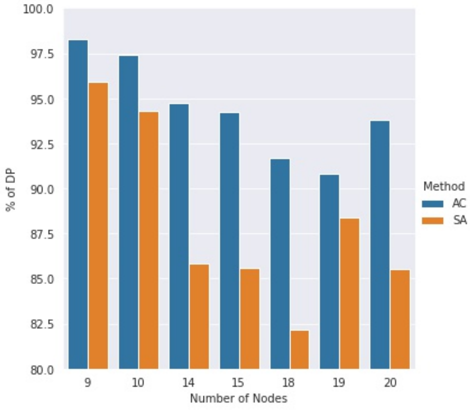


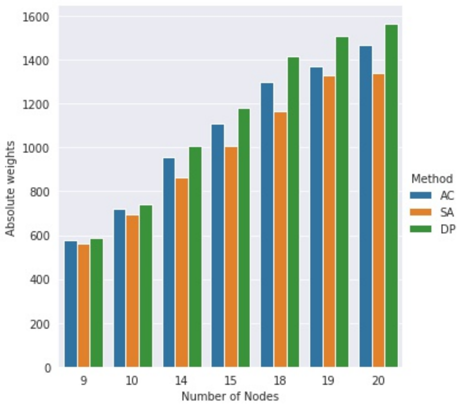
Comparison table

Table: Comparison between DP, Ant Colony and SA

Dataset	vertex	Source	Destn	DP	AC	SA
dataset4	9	0	5	585	575	561
dataset5	10	0	1	739	720	697
dataset9	14	0	10	1008	955	865
dataset10	15	0	10	1179	1111	1009
dataset13	18	0	17	1415	1297	1163
dataset14	19	0	17	1506	1368	1331
dataset15	20	0	2	1566	1469	1339

Comparison for datasets made from *ha30*



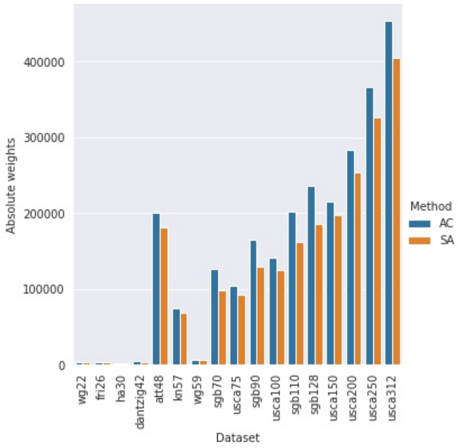


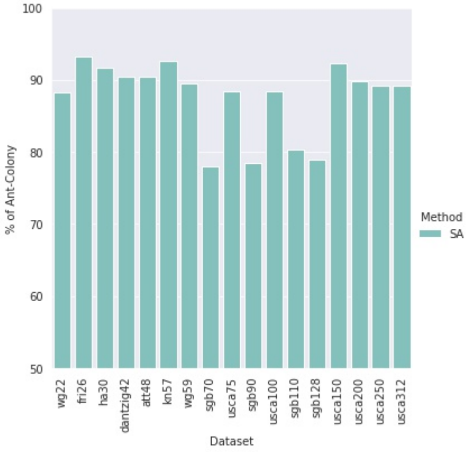
Comparison table

Table: Comparison between SA and Ant Colony

Dataset	vertex	Source	Destn	SA	AC
wg22	22	0	21	2782	3149
fri26	26	0	25	3029	3247
ha30	30	0	29	1975	2152
dantzig42	42	0	41	3464	3829
att48	48	0	47	180980	200125
kn57	57	0	56	68413	73914
wg59	59	0	58	5821	6502
sgb128	128	0	127	185921	235740
usca312	312	0	311	404788	453791

Comparison for larger datasets



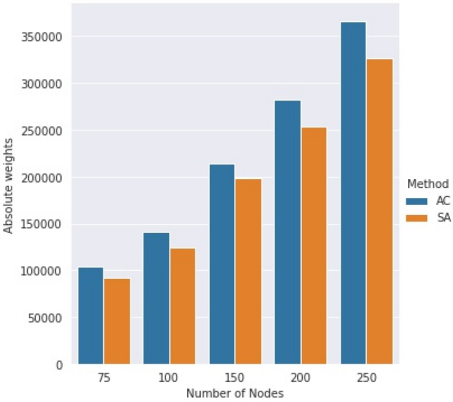


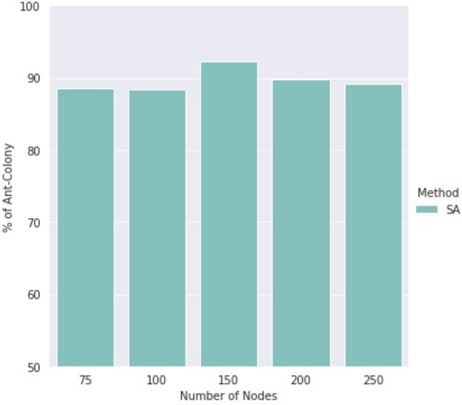
Comparison table

Table: Comparison between SA and Ant Colony

Dataset	vertex	Source	Destn	SA	AC
usca75	75	0	74	92315	104343
usca100	100	0	99	124836	141115
usca150	150	0	149	197946	214554
usca200	200	0	199	253989	282680
usca250	250	0	249	326964	366459

Comparison for larger datasets made from **usca**





References I

Michael Held and Richard M. Karp.
A dynamic programming approach to sequencing problems.
Journal of the Society for Industrial and Applied Mathematics, 10(1):196–210, 1962.

D. Karger, R. Motwani, and G. D. S. Ramkumar.
On approximating the longest path in a graph.
Algorithmica, 18(1):82–98, May 1997.

Gerhard Reinelt.
Tsplib—a traveling salesman problem library.
ORSA Journal on Computing, 3(4):376–384, 1991.