PROBLEM 7 REPORT

Fundamentals of optimization

Mai Việt Bảo   Nguyễn Nam Khánh
Lê Duy Anh   Võ Tá Quang Nhật

SoICT

January 2024

Table of Contents

Problem Statement

CBUS

There are n passengers $1, 2, \ldots, n$. The passenger i want to travel from point i to point $i + n (i = 1, 2, \ldots, n)$. There is a bus located at point 0 and has k places for transporting the passengers (it means at any time, there are at most k passengers on the bus).

Problem Statement

CBUS

There are n passengers $1, 2, \ldots, n$. The passenger i want to travel from point i to point $i + n (i = 1, 2, \ldots, n)$. There is a bus located at point 0 and has k places for transporting the passengers (it means at any time, there are at most k passengers on the bus). You are given the distance matrix c in which $c(i, j)$ is the traveling distance from point i to point j $(i, j = 0, 1, \ldots, 2n)$.
Compute the shortest route for the bus, serving n passengers and coming back to point 0.

Table of Contents

Backtracking - Idea

A constructive exact algorithm.

Backtracking - Idea

A constructive exact algorithm.

Constructs configurations step-by-step.

Tries all feasible configurations to see which one has the best objective value.

Backtracking - Implementation

Initialization

```
class Solver:
    def __init__(self, N, K, distance_matrix):
        self.N = N
        self.K = K
        self.Num_Nodes = 2 * N + 1
        self.Distance_Matrix = distance_matrix

        self.visited = [False for _ in range(self.Num_Nodes)]
        self.demands = [1 if 1 <= i <= N else -1 for i in range(self.
    Num_Nodes)]
        self.ans = list()
        self.best_dist = float('inf')
        self.time = time.time()

        self.current_node = 0
        self.capacity = 0
```

Backtracking - Implementation

Recursive function, loop, and condition checking

```
    def solve_backtrack(self, curr_route=(), curr_dist=0):
        curr_route = [node for node in curr_route]

        for next_node in range(1, self.Num_Nodes):

            if self.visited[next_node] or self.capacity + self.demands[
    next_node] > self.K:
                continue

            if next_node > self.N and not self.visited[next_node - self.N]:
                continue
```

Backtracking - Implementation

Backtracking and solution check

```
            curr_route.append(next_node)
            curr_dist += self.Distance_Matrix[self.current_node][next_node]
            self.visited[next_node] = True
            self.capacity += self.demands[next_node]

            if len(curr_route) == self.Num_Nodes - 1:
                total_dist = curr_dist + self.Distance_Matrix[self.
    current_node][0]
                if total_dist < self.best_dist:
                    self.best_dist = total_dist
                    self.ans = [node for node in curr_route]
            else:
                last_node = self.current_node
                self.current_node = next_node
                self.solve_backtrack(tuple(curr_route), curr_dist)
                self.current_node = last_node

            curr_route.remove(next_node)
            curr_dist -= self.Distance_Matrix[self.current_node][next_node]
            self.visited[next_node] = False
            self.capacity -= self.demands[next_node]
```

Backtracking - Performance

This algorithm is exact, meaning it can find the optimal solution.
But as you can see, this brute-forcing technique is computationally intensive.
Running time for this approach lies within a polynomial factor of O(n!), making this technique impractical even for N=10 or 20 cities.

```
5
5 4 3 9 10 1 2 8 6 7

Best distance found: 37
Time taken: 0.1947925090789795
```

Figure: Result for a N=5 problem

Branch and Bound - Idea

Branch - breaking down the problem into tiny subproblems that are easier to solve.

Branch and Bound - Idea

Branch - breaking down the problem into tiny subproblems that are easier to solve.

Bound - using a bounding function to eliminate extensive searches into subproblems that cannot surpass the best solution so far.

Branch and Bound - Idea

Branch - breaking down the problem into tiny subproblems that are easier to solve.

Bound - using a bounding function to eliminate extensive searches into subproblems that cannot surpass the best solution so far.

The lower bound function is:

$$g(N) = d + \text{emin} \times (2N - l)$$

where:

d: the current accumulated path length
emin $=$ the shortest edge to any of the unvisited cities
$2N - l$: number of unvisited cities, where l is the length of current route.

Branch and Bound - Implementation

Find the smallest edge to remaining cities

```
    def solve_bnb(self, curr_route=(), curr_dist=0):
        curr_route = [node for node in curr_route]

        # Find the smallest edge to unvisited nodes for BnB
        unvisited_nodes = [tup[0] for tup in list(enumerate(self.visited)) if tup[1] is False]
        reduced_mat = [[self.Distance_Matrix[row][node] for node in unvisited_nodes] for row in unvisited_nodes]
        reduced_mat = [reduced_mat[row][:row] + reduced_mat[row][row+1:] for row in range(len(unvisited_nodes))]
        smallest_edge = min([min(row) for row in reduced_mat])
```

Branch and Bound - Implementation

Bounding function preventing unnecessary deepening

```
            curr_route.append(next_node)
            curr_dist += self.Distance_Matrix[self.current_node][next_node]
            self.visited[next_node] = True
            self.capacity += self.demands[next_node]

            if len(curr_route) == self.Num_Nodes - 1:
                total_dist = curr_dist + self.Distance_Matrix[self.current_node][0]
                if total_dist < self.best_dist:
                    self.best_dist = total_dist
                    self.ans = [node for node in curr_route]
            else:  # Check the value from bounding with best answer
                if curr_dist + (self.Num_Nodes - 1 - len(curr_route)) * smallest_edge < self.best_dist:
                    last_node = self.current_node
                    self.current_node = next_node
                    self.solve_bnb(tuple(curr_route), curr_dist)
                    self.current_node = last_node

            curr_route.remove(next_node)
            curr_dist -= self.Distance_Matrix[self.current_node][next_node]
            self.visited[next_node] = False
            self.capacity -= self.demands[next_node]
```

Branch and Bound - Performance

These exact algorithms can solve to optimality, but only suitable for a very very small number of cities.

```
5
5 4 3 9 10 1 2 8 6 7

Best distance found: 37
Time taken: 0.1947925090789795
```

Figure: Backtracking, N=5

```
5
5 4 3 9 10 1 2 8 6 7

Best distance found: 37
Time taken: 0.04803180694580078
```

Figure: BnB, N=5

Table of Contents

CP - Idea

A mathematical exact algorithm.

CP - Idea

A mathematical exact algorithm.

This problem is classified as Capacitated Vehicle Routing Problem with Pickup and Delivery (CVRPPD).

We turn the problem into a series of linear constraints:

CP - Idea

- Decision variables: $x_{ij} \in \{0,1\}\ \forall i,j \in \{0,...,2N\}$
- Objective function: $\Sigma_i \Sigma_j c_{ij} x_{ij}$ min
- Flow control: $\Sigma_i x_{ij} = \Sigma_j x_{ij} = 1\ \forall i,j \in \{0,...,2N\}$
- Subtour elimination (DFJ):

$$\sum_{i \in Q} \sum_{j \neq i, j \in Q} x_{ij} \leq |Q| - 1\ \forall Q \subset \{0,...,2N\}, |Q| \geq 2$$

- Capacity constraint:

$$\sum_{i=1}^{n} \sum_{j=2}^{n} q_j x_{ij} \leq K,\ q_j = \begin{cases} 1 & \text{if } j \leq N \\ -1 & \text{otherwise} \end{cases}$$

- Pickup - drop-off order constraint: $T_i < T_{i+N}$, T_i is the accumulated path length from 0 to i in the current solution.

CP Implementation

Initializing Index manager for indexing cities and Routing model for solving

```
    def solve_cp(self):
        pickups_deliveries = [(i, i + self.N) for i in range(1, self.N + 1)
    ]
        # Create the routing index manager.
        manager = pywrapcp.RoutingIndexManager(len(self.distance_matrix),
    1, 0)

        # Create Routing Model.
        routing = pywrapcp.RoutingModel(manager)
```

CP Implementation

Function calculating distance between 2 cities

```
        def distance_callback(from_index, to_index):
            """Returns the distance between the two nodes."""
            from_node = manager.IndexToNode(from_index)
            to_node = manager.IndexToNode(to_index)
            return self.distance_matrix[from_node][to_node]

        transit_callback_index = routing.RegisterTransitCallback(
    distance_callback)
        routing.SetArcCostEvaluatorOfAllVehicles(transit_callback_index)
```

CP Implementation

Pickup - drop-off order constraint

```
        # Add Distance constraint.
        routing.AddDimension(
            transit_callback_index,
            0,  # no slack
            1000,  # vehicle maximum travel distance
            True,  # start cumul to zero
            "Distance",
        )
        distance_dimension = routing.GetDimensionOrDie("Distance")

        # Define Transportation Requests.
        for request in pickups_deliveries:
            pickup_index = manager.NodeToIndex(request[0])
            delivery_index = manager.NodeToIndex(request[1])
            routing.AddPickupAndDelivery(pickup_index, delivery_index)
            # passenger must be picked up and dropped off by the same vehicle
            routing.solver().Add(
                routing.VehicleVar(pickup_index) == routing.VehicleVar(delivery_index)
            )
            # passenger must be picked up before dropped off
            routing.solver().Add(
                distance_dimension.CumulVar(pickup_index) <= distance_dimension.CumulVar(delivery_index)
            )
```

CP Implementation

Bus capacity constraint

```
        # Add Capacity constraint.
        def demand_callback(from_index):
            """Returns the demand of the node."""
            from_node = manager.IndexToNode(from_index)
            return 1 if from_node <= self.N else -1

        demand_callback_index = routing.RegisterUnaryTransitCallback(
    demand_callback)
        routing.AddDimension(
            demand_callback_index,
            0,
            self.K,  # vehicle maximum capacities
            True,
            "Capacity",
        )
```

CP Implementation

Solve and save the result

```
        # Setting first solution heuristic.
        search_parameters = pywrapcp.DefaultRoutingSearchParameters()
        search_parameters.first_solution_strategy = routing_enums_pb2.
    FirstSolutionStrategy.AUTOMATIC

        # Solve the problem.
        solution = routing.SolveWithParameters(search_parameters)

        # Add solution to Solver object's property.
        if solution:
            index = routing.Start(0)
            self.ans = [manager.IndexToNode(index)]
            while not routing.IsEnd(index):
                index = solution.Value(routing.NextVar(index))
                self.ans.append(manager.IndexToNode(index))
            self.ans = self.ans[1:-1]
            self.best_dist = solution.ObjectiveValue()
```

CP - Performance

The solver works great till N=100 (or 200 cities). From that point onward, it takes extremely long, eventually quitting and returning ‘inf’.

```
10
6 4 10 1 5 15 14 11 8 20 3 9 7 19 2 18 13 12 16 17

Best distance found: 41
Time taken: 0.035009145736694336
```

Figure: result for N=10 (21 cities)

```
Best distance found: 122
Time taken: 18.97257089614868
```

Figure: result for N=100 (201 cities)

Table of Contents

Nearest Neighbor - Idea

A constructive heuristic algorithm.

Nearest Neighbor - Idea

A constructive heuristic algorithm.

Builds the solution step-by-step.

But instead of considering all feasible solutions, it will only create one solution and always (naively and greedily) choose the nearest unvisited city for the next move.

Nearest Neighbor - Implementation

```python
    def solve_greedy(self):
        for _ in range(self.Num_Nodes - 1):
            min_dist = float('inf')
            best_node = -1

            for node in range(1, self.Num_Nodes):
                if self.visited[node] or self.capacity + self.demands[node] > self.K:
                    continue

                if node > self.N and not self.visited[node - self.N]:
                    continue

                if self.Distance_Matrix[self.current_node][node] < min_dist:
                    min_dist = self.Distance_Matrix[self.current_node][node]
                    best_node = node

            self.ans.append(best_node)
            self.visited[best_node] = True
            self.capacity += self.demands[best_node]
            self.best_dist += self.Distance_Matrix[self.current_node][best_node]

            self.current_node = best_node

        self.best_dist += self.Distance_Matrix[self.current_node][0]
```

Nearest Neighbor - Performance

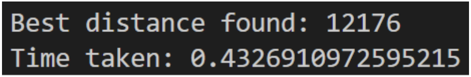


Figure: NN's solutions for a problem with size up to N = 1000, blazing fast

Nearest Neighbor - Performance

Comparing solution of this to the optimal one by Backtracking, it runs fast, but is nowhere near optimality.

```
5
5 4 3 9 10 1 2 8 6 7

Best distance found: 37
Time taken: 0.1947925090789795
```

Figure: Backtracking result with N = 5

```
5
1 2 6 7 5 10 3 4 8 9

Best distance found: 49
Time taken: 0.0
```

Figure: NN result with N = 5

It could be use as a pretty good starting point for these subsequent Local Search algorithms.

Table of Contents

Local search heuristic algorithm

A heuristic method for solving computationally hard optimization problems.

Local search heuristic algorithm

A heuristic method for solving computationally hard optimization problems.

Procedure of a Local search algorithm:

- Find a feasible initial solution.
- Apply changes to the existing solution(s) to get a candidate space.
- Repeatedly move from solution to solution and altered existing ones till a terminating condition is met, e.g. time, iteration, or objective value limit.

Hill climbing - Idea

A local search heuristic algorithm.

Hill climbing - Idea

A local search heuristic algorithm.

It starts with an arbitrary state, then tries to find a better state from the previous state's neighborhood.

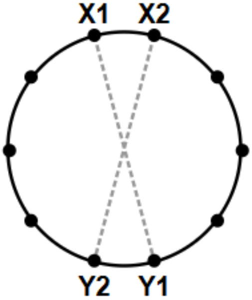
If none of the neighbors is better, the algorithm is terminated.

Neighborhood generation with k-opt

To find neighbors of a solution, we use the k-opt technique.

k-opt intentionally removes k edges from the current solution and adds k new ones, creating a new and (in most cases) better solution.

2-opt - Idea



Replace edges when $d(X1, Y1) + d(X2, Y2) < d(X1, X2) + d(Y1, Y2)$

where $X1Y1$ and $X2Y2$ are newly added edges, $X1X2$ and $Y1Y2$ are removed edges.

2-opt - Implementation

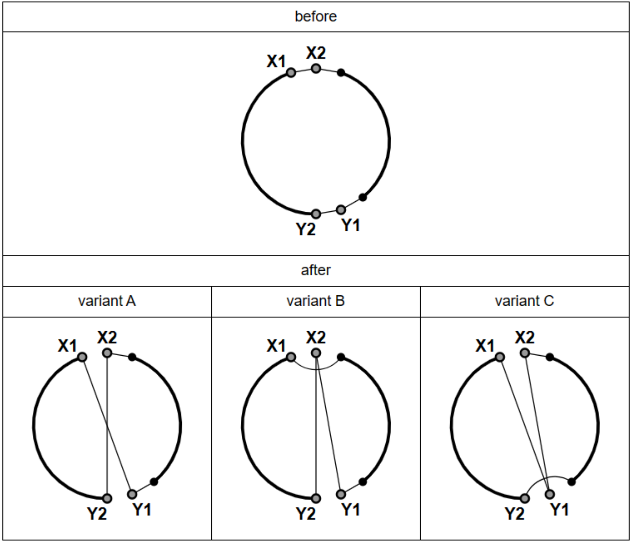
```
def get_neighborhood(self, route, opt='2', time_limit=None):
    neighbor_list = list()
    route = [0] + route
    num_nodes = len(route)

    for i in range(num_nodes - 3):
        x1 = route[i]
        x2 = route[(i + 1) % num_nodes]

        if i == 0:
            j_max = num_nodes - 2
        else:
            j_max = num_nodes - 1

        for j in range(i + 2, j_max):
            y1 = route[j]
            y2 = route[(j + 1) % num_nodes]

            if self.Distance_Matrix[x1][x2] + self.Distance_Matrix[y1][y2] > self.Distance_Matrix[x1][y1] + self.Distance_Matrix[x2][y2]:
                neighbor = self.reverse_segment(route, i, j)
                neighbor = neighbor[neighbor.index(0):] + neighbor[:neighbor.index(0)]
                neighbor = neighbor[1:]
                if self.validate_route(neighbor):
                    neighbor_list.append(neighbor)
```

2.5-opt - Idea

For 3-opt, every 3-edge manipulation results in $2^3 - 1 = 7$ new neighbors $\Longrightarrow$ computationally intensive!

We consider 2.5-opt, which is 2-opt and 2 simple cases of 3-opt.

2.5-opt - Idea


before
X1
X2
Y2
Y1
after
variant A
variant B
variant C
X1
X2
Y2
Y1
X1
X2
Y2
Y1
X1
X2
Y2
Y1

2.5-opt - Implementation

```
if opt == '2.5':
    # Look forward, shift x2 to after y1
    x3 = route[(i + 2) % num_nodes]
    if x3 != y1:
        if self.Distance_Matrix[x1][x2] + self.Distance_Matrix[x2][x3] + self.Distance_Matrix[y1][y2] > self.Distance_Matrix[x1][x3] + self.Distance_Matrix[y1][x2] + self.Distance_Matrix[x2][y2]:
            neighbor = self.node_shift(route, (i + 1) % num_nodes, j)
            neighbor = neighbor[neighbor.index(0):] + neighbor[:neighbor.index(0)]
            neighbor = neighbor[1:]
            if self.validate_route(neighbor):
                neighbor_list.append(neighbor)

    # Look backward, shift y1 to after x1
    y0 = route[(num_nodes + j - 1) % num_nodes]
    if y0 != x2:
        if self.Distance_Matrix[y0][y1] + self.Distance_Matrix[y1][y2] + self.Distance_Matrix[x1][x2] > self.Distance_Matrix[y0][y2] + self.Distance_Matrix[x1][y1] + self.Distance_Matrix[y1][x2]:
            neighbor = self.node_shift(route, j, i)
            neighbor = neighbor[neighbor.index(0):] + neighbor[:neighbor.index(0)]
            neighbor = neighbor[1:]
            if self.validate_route(neighbor):
                neighbor_list.append(neighbor)
```

Hill climbing - Implementation

```python
    def solve_hill_climbing(self, opt='2'):
        """Solve the problem using Local Hill Climbing algorithm."""
        self.time = time.time()
        self.get_first_solution()
        neighborhood = self.get_neighborhood(self.ans, opt=opt)

        while neighborhood:
            neighbor = neighborhood.pop(0)
            neighbor_dist = self.get_distance(neighbor)
            if neighbor_dist < self.best_dist:
                self.ans = neighbor[:]
                self.best_dist = neighbor_dist
                neighborhood = self.get_neighborhood(self.ans, opt=opt)
            else:
                break
```

Hill climbing - Performance

We set Hill climbing with both 2-opt and 2.5-opt against NN in 5 randomly generated N=100 problems.

Hill climbing - Performance

		Nearest Neighbor	Hill climbing 2-opt	Hill climbing 2.5-opt
Problem 25	best distance	750	750	664
	time	0.004066467	0.010215282	0.088514566
Problem 26	best distance	748	748	704
	time	0.004149675	0.010880709	0.047587872
Problem 27	best distance	782	782	706
	time	0.00402379	0.010456085	0.116880894
Problem 28	best distance	726	726	683
	time	0.004202366	0.010346413	0.070104837
Problem 29	best distance	621	621	605
	time	0.004003763	0.012079	0.039534807

Figure: Hill climbing 2-opt and 2.5-opt

Hill climbing - Performance

Key takeaways:

- Neighborhood searching method is key!
- 2-opt (in our case) was not able to find better neighbors.
- 2.5-opt gives way better result, with increased computing time in return.

For subsequent local search algorithms, we will use 2.5-opt.

Beam search - Idea

A local search heuristic algorithm.

Beam search - Idea

A local search heuristic algorithm.

It keeps track of and expand a set number of most promising neighbors in an iteration. That number is β - beam width.

Beam search - Implementation

Expand on all neighbors

```python
    def solve_beam_search(self, iterations=100, opt='2', beam_width=5):
        """Solve the problem using Local Beam Search algorithm."""
        self.time = time.time()
        self.get_first_solution()
        neighborhood = self.get_neighborhood(self.ans, opt=opt)
        neighbor_index = 0
        candidate_list = list()

        while iterations > 0:
            if neighbor_index < len(neighborhood):
                neighbor = neighborhood[neighbor_index]
                candidates = self.get_neighborhood(neighbor, opt=opt)
                candidate_list += candidates
                neighbor_index += 1
```

Beam search - Implementation

Sort and retain β candidates for next iteration

```
            else:
                # Remove duplicates in candidate_list
                candidate_list = [tuple(x) for x in candidate_list]
                candidate_list = list(set(candidate_list))
                candidate_list = [list(x) for x in candidate_list]
                candidate_list = list(filter(lambda x: x not in neighborhood, candidate_list))

                # Sort candidate_list by total distance and take the top candidates
                candidate_list = sorted(candidate_list, key=lambda x: self.get_distance(x))
                if len(candidate_list) == 0:
                    break
                if self.get_distance(candidate_list[0]) < self.best_dist:
                    self.ans = candidate_list[0][:]
                    self.best_dist = self.get_distance(self.ans)
                neighborhood = candidate_list[:min(len(candidate_list), beam_width)]

                candidate_list = list()
                neighbor_index = 0
                iterations -= 1
```

Beam search - Performance

We tested Beam search with $\beta = 5$, $\beta = 10$, each in 100 iterations, and gauged their improvement over Hill climbing.

Beam search - Performance

		Hill climbing 2.5-opt	Beam search 2.5-opt iter 100 width 5	Beam search 2.5-opt iter 100 width 10
Problem 25	best distance	664	657	657
	time	0.088514566	1.411660671	19.76223016
Problem 26	best distance	704	662	625
	time	0.047587872	11.55299807	24.7884059
Problem 27	best distance	706	676	647
	time	0.116880894	0.682408094	17.58623958
Problem 28	best distance	683	577	654
	time	0.070104837	10.00823689	29.06178284
Problem 29	best distance	605	605	550
	time	0.039534807	16.06260228	37.56542706

Figure: Beam search with $\beta = 5$ and 10

Beam search - Performance

Key takeaways:

- Beam search immediately gives better results than Hill climbing.
- Beam search takes a lot more computing power - this is to be expected.
- Increasing beam width improves the result in most cases, but the increased computing time is brutal.

Table of Contents

Local search-based metaheuristic algorithm

A higher-level heuristic method designed to assist or guide simple Local search heuristic algorithms toward better solutions.

Tabu search - Idea

A local search-based metaheuristic algorithm.

Tabu search - Idea

A local search-based metaheuristic algorithm.

Like Hill climbing, it expands the best neighbor, but enhances the performance by relaxing some rules:

- Accept "best in the neighborhood but worse than the best ever" neighbors.
- Keep track of a tabu list of previously best neighbors of predetermined size to avoid revisiting.

Tabu search - Implementation

Repeatedly check with and update the tabu list

```python
 def solve_tabu_search(self, iterations=100, opt='2', tabu_list_size=100):
     """Solve the problem using Tabu Search algorithm."""
     self.time = time.time()
     self.get_first_solution()

     current_ans = self.ans[:]
     tabu_list = list()

     while iterations > 0:
         neighborhood = self.get_neighborhood(current_ans, opt=opt)
         neighborhood = [neighbor for neighbor in neighborhood if neighbor not in tabu_list]

         if len(neighborhood) == 0:
             # No non-tabu neighbors found, terminate the search
             break

         best_neighbor = neighborhood[0]
         best_neighbor_dist = self.get_distance(best_neighbor)

         current_ans = best_neighbor
         tabu_list.append(best_neighbor[:])

         iterations -= 1
```

Tabu search - Performance

We tested Tabu search with tabu list size of 50 and 100, each in 100 iterations, and compared them with Hill climbing.

Tabu search - Performance

		Hill climbing 2.5-opt	Tabu search 2.5-opt iter 100 size 50	Tabu search 2.5-opt iter 100 size 100
Problem 25	best distance	664	664	664
	time	0.088514566	0.098850012	0.094198227
Problem 26	best distance	704	651	651
	time	0.047587872	0.204253674	0.207988977
Problem 27	best distance	706	706	706
	time	0.116880894	0.107273817	0.116250992
Problem 28	best distance	683	631	631
	time	0.070104837	0.672608852	0.630461693
Problem 29	best distance	605	588	588
	time	0.039534807	4.625675201	4.663621902

Figure: Tabu search with tabu list sized 50 and 100

Tabu search - Performance

Key takeaways:

- Tabu search, thanks to the relaxed rules, finds better solutions than Hill climbing. All while keeping the increase in computing time pretty negligible.
- Increasing the tabu list size did not yield better results, at least in our test cases with 100 iterations.

Tabu Beam - Idea

A local search-based metaheuristic algorithm.

Tabu Beam - Idea

A local search-based metaheuristic algorithm.

This time, tabu list is used to enhance Beam Search. Tabu list helps eliminate recurring solutions in the beam, e.g. the solution where 2 pairs of edge are repetitively added/removed.

Tabu Beam - Implementation

Expand all kept neighbors like normal Beam search

```
    def solve_tabu_beam(self, iterations=100, opt='2', tabu_list_size=100,
   beam_width=5):
        """Solve the problem using Tabu-Beam search hybrid/fusion(?)."""
        self.time = time.time()
        self.get_first_solution()

        tabu_list = list()
        neighborhood = self.get_neighborhood(self.ans, opt=opt)
        neighbor_index = 0
        candidate_list = list()

        while iterations > 0:
            if neighbor_index < len(neighborhood):
                neighbor = neighborhood[neighbor_index]
                candidates = self.get_neighborhood(neighbor, opt=opt)
                candidate_list += candidates
                neighbor_index += 1
```

Tabu Beam - Implementation

Filter out tabu-ed neighbors and update tabu list

```
            else:
                # Remove duplicates in candidate_list
                candidate_list = [tuple(x) for x in candidate_list]
                candidate_list = list(set(candidate_list))
                candidate_list = [list(x) for x in candidate_list]
                candidate_list = list(filter(lambda x: x not in neighborhood and x not in tabu_list, candidate_list))
                candidate_list = sorted(candidate_list, key=lambda x: self.get_distance(x))

                if len(candidate_list) == 0:
                    break

                best_neighbor = candidate_list[0]
                best_neighbor_dist = self.get_distance(best_neighbor)

                tabu_list.append(best_neighbor[:])
                if len(tabu_list) > tabu_list_size:
                    tabu_list.pop(0)
```

Tabu Beam - Performance

We tested Tabu Beam hybrid search with $\beta = 5$ and $\beta = 10$, each kept a tabu list sized 50 and ran in 100 iterations.

We measured their improvement over simple Tabu search with similar parameters.

Tabu Beam - Performance

		Hill climbing 2.5-opt	Beam search 2.5-opt iter 100 width 5	Tabu Beam 2.5-opt iter 100 size 50 width 5
Problem 25	best distance	664	657	657
	time	0.088514566	1.411660671	1.494745493
Problem 26	best distance	704	662	625
	time	0.047587872	11.55299807	7.293416977
Problem 27	best distance	706	676	676
	time	0.116880894	0.682408094	0.674106836
Problem 28	best distance	683	577	577
	time	0.070104837	10.00823689	2.64825201
Problem 29	best distance	605	605	605
	time	0.039534807	16.06260228	2.259266853

Figure: Tabu Beam search with $\beta = 5$

Tabu Beam - Performance

With beam width of 5, Tabu Beam does not seem to give better results than simple Beam search in most cases.

Tabu Beam reaches terminating condition much faster compared to Beam search.

Tabu Beam - Performance

		Hill climbing 2.5-opt	Beam search 2.5-opt iter 100 width 10	Tabu Beam 2.5-opt iter 100 size 50 width 10
Problem 25	best distance	664	657	655
	time	0.088514566	19.76223016	28.39361954
Problem 26	best distance	704	625	599
	time	0.047587872	24.7884059	25.89411712
Problem 27	best distance	706	647	634
	time	0.116880894	17.58623958	12.32661223
Problem 28	best distance	683	654	641
	time	0.070104837	29.06178284	25.70584631
Problem 29	best distance	605	550	580
	time	0.039534807	37.56542706	41.34595585

Figure: Tabu Beam search with $\beta = 10$

Tabu Beam - Performance

With beam width of 10, Tabu Beam can discover better result than simple Beam search, at the cost of even higher computing time.

Key takeaway: Tabu heuristic can greatly improve Beam search, and beam width is the decisive factor for the algorithm.

Table of Contents

Simulated Annealing - Idea

Simulated annealing is a better version of hill climbing. It's not always trying to find a better state than the previous one. Sometimes, a worse state is acceptable with a probability decreasing by time.

Simulated Annealing - Idea

In this algorithms, instead of using k-opt technique as hill climbing algorithms above (which we can do that), we use random walk. Each new neighbor is the result of one of the mutating functions below with equal probability:

- inverse(route): invert the order between 2 cities in the route
- swap(route): swap two arbitrary cities
- insert(route): select random point j in the route and insert it to the i^{th} position
- swap_routes(route): select a subroute $[a : b + 1]$ and insert in at another position

Simulated Annealing - Idea

The probability decreases proportionally to the decrease of time, also the less different the new state is, the more likely it will be accepted.

Those are the intuitive reasons behind how the acceptance probability is:

$$p = e^{\frac{-\Delta}{T}}$$

Simulated Annealing Implementation

```python
def Simulated_Annealing(n, k, distance_matrix):
    temperature = 5000
    alpha = 0.99
    time_limit = 180

    shortest_route = generate_initial_solution(n)
    t = time.time()

    while time.time() - t < time_limit:
        candidate_route = get_neighbor(n, k, shortest_route)

        delta = distance(candidate_route, distance_matrix) - distance(
    shortest_route, distance_matrix)
        if delta < 0: # better
            shortest_route = candidate_route
        else:
            p = math.exp(-delta / temperature)
            r = random.uniform(0, 1)
            if r < p:
                shortest_route = candidate_route

        temperature *= alpha


    return shortest_route
```

Simulated Annealing - Performance

```
100
[97, 24, 53, 99, 22, 47, 57, 60, 82, 46, 10, 54, 79, 59, 146, 61, 1, 32, 93, 3
7, 161, 193, 160, 28, 122, 17, 36, 101, 96, 41, 128, 77, 33, 12, 8, 71, 85, 23
, 38, 19, 185, 72, 197, 4, 27, 117, 154, 108, 177, 45, 145, 133, 34, 74, 30, 6
, 81, 11, 119, 130, 62, 39, 98, 43, 3, 13, 65, 50, 134, 92, 113, 153, 157, 179
, 198, 5, 44, 181, 58, 14, 123, 103, 42, 159, 147, 40, 95, 144, 182, 88, 83, 1
38, 73, 199, 69, 29, 192, 183, 56, 111, 150, 66, 16, 166, 63, 51, 163, 89, 91,
 191, 142, 35, 31, 15, 52, 172, 26, 135, 94, 80, 180, 75, 84, 21, 121, 173, 13
1, 194, 143, 189, 49, 105, 112, 174, 152, 25, 70, 55, 110, 136, 18, 76, 7, 184
, 176, 126, 165, 132, 155, 129, 125, 107, 2, 64, 87, 86, 115, 78, 186, 124, 13
7, 156, 196, 114, 149, 151, 175, 162, 139, 90, 169, 116, 190, 158, 171, 140, 1
70, 20, 188, 141, 9, 195, 106, 100, 67, 164, 109, 104, 68, 120, 178, 48, 167,
187, 102, 200, 148, 118, 127, 168]
Best result found:  133
```

Figure: result for $n = 100$, time limit is 180s

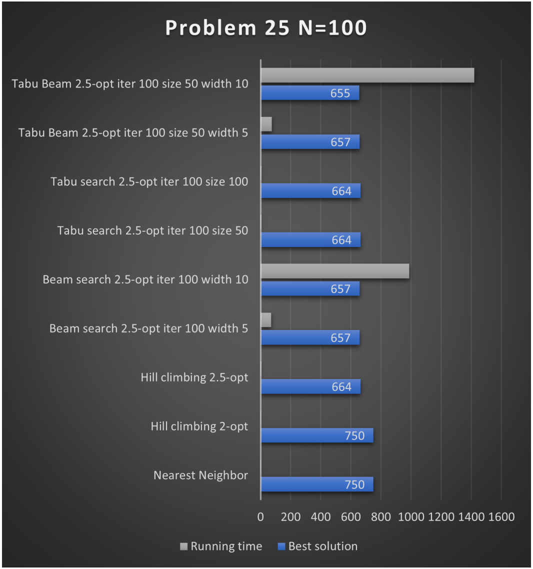
Simulated Annealing - Performance

The algorithm performs better than Hill climbing since it produces routes that tend to the global optimum.

But there are some disadvantages:

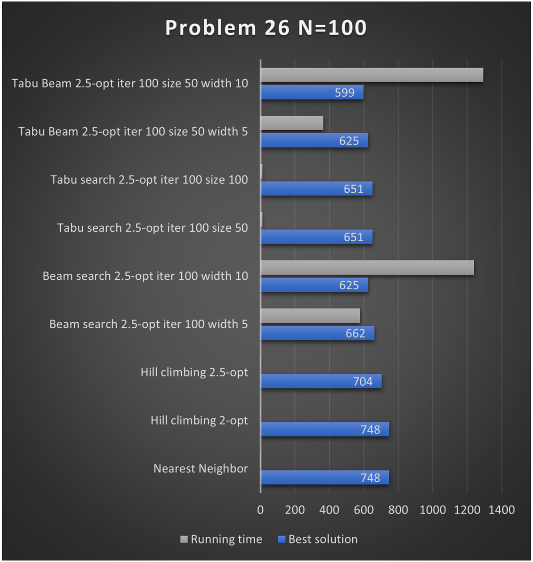
- The initial temperature and the decreasing speed of it need to be adjusted manually.
- Potential increase in computational time

Result



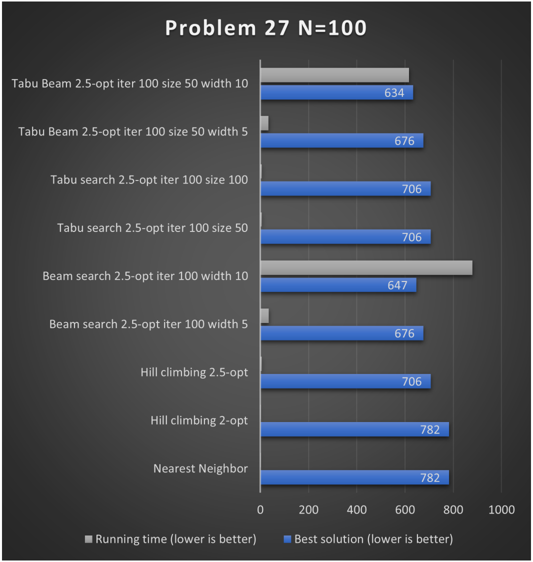
Problem 25 N=100
Tabu Beam 2.5-opt iter 100 size 50 width 10
655
Tabu Beam 2.5-opt iter 100 size 50 width 5
657
Tabu search 2.5-opt iter 100 size 100
664
Tabu search 2.5-opt iter 100 size 50
664
Beam search 2.5-opt iter 100 width 10
657
Beam search 2.5-opt iter 100 width 5
657
Hill climbing 2.5-opt
664
Hill climbing 2-opt
750
Nearest Neighbor
750
0 200 400 600 800 1000 1200 1400 1600
Running time
Best solution

Result

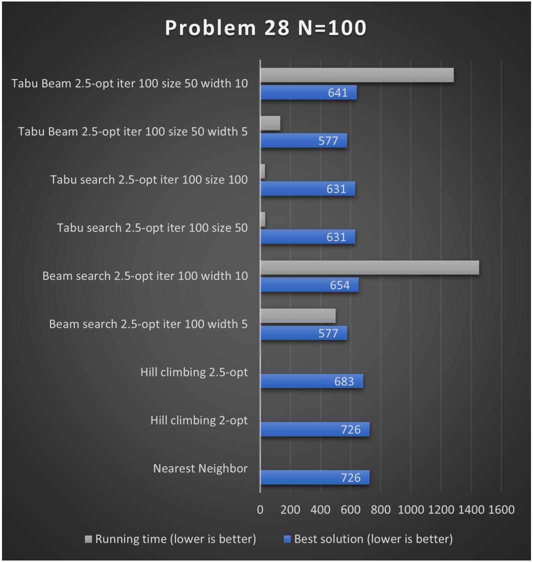


Problem 26 N=100
Tabu Beam 2.5-opt iter 100 size 50 width 10
Tabu Beam 2.5-opt iter 100 size 50 width 5
Tabu search 2.5-opt iter 100 size 100
Tabu search 2.5-opt iter 100 size 50
Beam search 2.5-opt iter 100 width 10
Beam search 2.5-opt iter 100 width 5
Hill climbing 2.5-opt
Hill climbing 2-opt
Nearest Neighbor
599
625
651
651
625
662
704
748
748
0 200 400 600 800 1000 1200 1400
Running time
Best solution

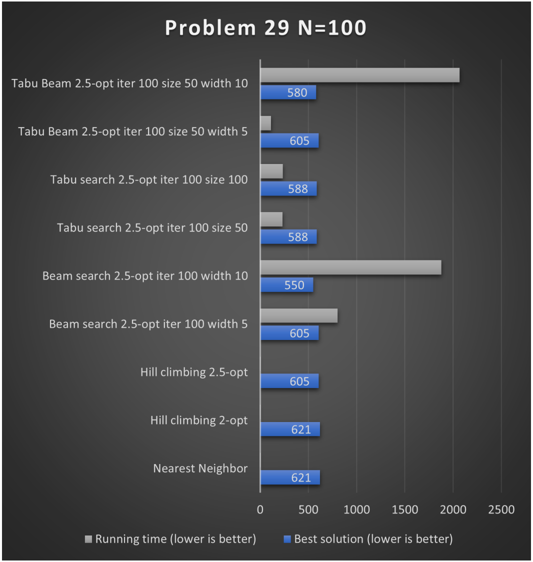
Result


Problem 27 N=100
Tabu Beam 2.5-opt iter 100 size 50 width 10
Tabu Beam 2.5-opt iter 100 size 50 width 5
Tabu search 2.5-opt iter 100 size 100
Tabu search 2.5-opt iter 100 size 50
Beam search 2.5-opt iter 100 width 10
Beam search 2.5-opt iter 100 width 5
Hill climbing 2.5-opt
Hill climbing 2-opt
Nearest Neighbor
634
676
706
706
647
676
706
782
782
0
200
400
600
800
1000
Running time (lower is better)
Best solution (lower is better)

Result


Problem 28 N=100
Tabu Beam 2.5-opt iter 100 size 50 width 10
641
Tabu Beam 2.5-opt iter 100 size 50 width 5
577
Tabu search 2.5-opt iter 100 size 100
631
Tabu search 2.5-opt iter 100 size 50
631
Beam search 2.5-opt iter 100 width 10
654
Beam search 2.5-opt iter 100 width 5
577
Hill climbing 2.5-opt
683
Hill climbing 2-opt
726
Nearest Neighbor
726
0 200 400 600 800 1000 1200 1400 1600
Running time (lower is better)
Best solution (lower is better)

Result


Problem 29 N=100
Tabu Beam 2.5-opt iter 100 size 50 width 10
580
Tabu Beam 2.5-opt iter 100 size 50 width 5
605
Tabu search 2.5-opt iter 100 size 100
588
Tabu search 2.5-opt iter 100 size 50
588
Beam search 2.5-opt iter 100 width 10
550
Beam search 2.5-opt iter 100 width 5
605
Hill climbing 2.5-opt
605
Hill climbing 2-opt
621
Nearest Neighbor
621
0
500
1000
1500
2000
2500
Running time (lower is better)
Best solution (lower is better)

Result

In terms of solution optimality:

$$\text{Greedy} < \text{Heuristic} < \text{Metaheuristic} < \text{OR-Tools} \leq \text{BnB}$$

Result

In order to compare algorithms in terms of input size and computational time, we had ran more test and had the result as below:

	N=5	N=10	N=100	N=500
Branch and Bound	0.05	too long	too long	too long
OR-Tools	0.01	0.03	19	too long
Greedy	0	0	0.005	0.1
Hill Climbing	0	0	0.27	930

Conclusion

This problem can be solved by various optimization techniques, from exact algorithms to heuristics. But there is a trade-off between accuracy and speed.

Conclusion

This problem can be solved by various optimization techniques, from exact algorithms to heuristics. But there is a trade-off between accuracy and speed.

Exact algorithms only perform well with small N. Heuristic algorithms can only find the good local optima in an acceptable computing time. With metaheuristic algorithms, the solution could tend to the global optimal but also nothing is guaranteed.

Conclusion

This problem can be solved by various optimization techniques, from exact algorithms to heuristics. But there is a trade-off between accuracy and speed.

Exact algorithms only perform well with small N. Heuristic algorithms can only find the good local optima in an acceptable computing time. With metaheuristic algorithms, the solution could tend to the global optimal but also nothing is guaranteed.

The project has given us a lot of precious insights and knowledge about Operations Research, about algorithms and professional tools. We had also studied valuable techniques that can be applied to our future problems. We have expanded our knowledge thanks to this mini-project.

Conclusion

THANKS FOR YOUR TIME