Solomon Pobee Project 5 Report

Priority Queue

Time complexity: inserting an element (n) into the priority queue takes O(log n). extracking the minimum element also takes O(log n).

Space complexity: it takes a space of the maximum number of element(n) in the queue which is O(n).

Reduced Cost matrix:

Time complexity: the reducing of the matrix include finding of the minimum value in each row and column, which takes o(n^2) from a matrix of (n*n). updating the matrix for each function also takes O(n^2).

Space complexity: the storage of the matrix requires O(n^2) space.

BSSF initialization

Time complexity: using greedy takes O(n^2). This involves constructing a tour by selecting a nearest unvisited city each time.

Space complexity: storing of the tour takes a space of O(n).

Expanding one search state into its children

Time complexity: expanding a search state involves generating new matrices and reducing them, which takes o(n^2) for each child. If there are (n) children, the total complexity is O(n^3)

Space complexity: each child state requires a space of O(n^2) for the matrix and O(n) space for the tour. This gives us O(n^2 + n) = O(n^2) space per child.

Data Structure I used to represent the states

Partial state: each partial state represented by a tuple was containing the current cost, the partial tour, and the reduced cost matrix. This permit for efficient branching and pruning based on the current state of the tour and the related cost.

Why this way: using the tuple ensures that all the necessary informations for the branching and pruning is merged into a single structure, making it easy to manage and pass around within the algorithm.

The cutTree is used to keep track of pruning branches.

Greedy tour algorithm

```
def greedy_tour(edges: list[list[float]], timer: Timer) -> list[SolutionStats]:
    stats = []
    best_tour = None
    best_score = float('inf')
    n_nodes_expanded = 0
    n_nodes_pruned = 0
    cut_tree = CutTree(len(edges))

    for start in range(len(edges)):
        if timer.time_out():
            return stats

        unvisited = set(range(len(edges)))
        unvisited.remove(start)
        tour = [start]
        current_city = start
        cost = 0

        while unvisited:
            next_city = min(unvisited, key=lambda x: edges[current_city][x])
            if math.isinf(edges[current_city][next_city]):
                n_nodes_pruned += 1
                cut_tree.cut(tour + [next_city])
                break

            cost += edges[current_city][next_city]
            tour.append(next_city)
            unvisited.remove(next_city)
            current_city = next_city
            n_nodes_expanded += 1

        if len(tour) == len(edges):
            if not math.isinf(edges[current_city][start]):
                cost += edges[current_city][start]
                if cost < best_score:
                    best_score = cost
                    best_tour = tour

    if best_tour is not None:
```

```
        stats.append(SolutionStats(
            tour=best_tour,
            score=best_score,
            time=timer.time(),
            max_queue_size=1,
            n_nodes_expanded=n_nodes_expanded,
            n_nodes_pruned=n_nodes_pruned,
            n_leaves_covered=cut_tree.n_leaves_cut(),
            fraction_leaves_covered=cut_tree.fraction_leaves_covered()
        ))

    if not stats:
        return [SolutionStats(
            tour=[],
            score=math.inf,
            time=timer.time(),
            max_queue_size=1,
            n_nodes_expanded=n_nodes_expanded,
            n_nodes_pruned=n_nodes_pruned,
            n_leaves_covered=cut_tree.n_leaves_cut(),
            fraction_leaves_covered=cut_tree.fraction_leaves_covered()
        )]
    return stats
```

Time complexity

The Unvisited set initialization and removing the start city, a performance that iterates over each city as a starting point. There are 'n' cities therefore, the loop runs n times (**o(n**)). The inner loop, which is the while unvisited runs **n-1** times. It visits the cities until all the cities are visited. In each citiy I finds the minimum cost edge to the next city. In finding the next city, the min function iterate ove the unvisited set. In the worst case it could take o(n). the while can take n times to start each city. This makes the running time O(n^2).

Both the initialization of unvisited set **O(n)** and the while unvisited **O(n^2**) takes **o(n^3).**

Space Complexity

The tour list stores the cities visited and that can store 'n' cities. The unvisited set stores 'n' cities as well. The cutTree with the len(edges) uses the space equal to the number of edges. The space used is O(n). the tour, unvisited, the cutTree takes **O(n)+O(n)+O(n)=O(n)**

DFS

```
def dfs(edges: list[list[float]], timer: Timer) -> list[SolutionStats]:
```

```
    stats = []
    best_tour = None
    best_score = float('inf')
    n_nodes_expanded = 0
    n_nodes_pruned = 0
    cut_tree = CutTree(len(edges))

    def dfs_recursive(tour, cost):
        nonlocal best_tour, best_score, n_nodes_expanded, n_nodes_pruned

        if timer.time_out():
            return

        current_city = tour[-1]

        if len(tour) == len(edges):
            if not math.isinf(edges[tour[-1]][tour[0]]):
                cost += edges[tour[-1]][tour[0]]
                if cost < best_score:
                    best_score = cost
                    best_tour = tour
            return

        for next_city in range(len(edges)):
            if next_city not in tour:
                if math.isinf(edges[current_city][next_city]):
                    n_nodes_pruned += 1
                    cut_tree.cut(tour + [next_city])
                    continue
                n_nodes_expanded += 1

                dfs_recursive(tour + [next_city], cost + edges[current_city][next_city])

    for start in range(len(edges)):
        if timer.time_out():
            break

        dfs_recursive([start], 0)

    if best_tour is not None:
        stats.append(SolutionStats(
            tour=best_tour,
            score=best_score,
            time=timer.time(),
            max_queue_size=1,
```

```
            n_nodes_expanded=n_nodes_expanded,
            n_nodes_pruned=n_nodes_pruned,
            n_leaves_covered=cut_tree.n_leaves_cut(),
            fraction_leaves_covered=cut_tree.fraction_leaves_covered()
        ))

    if not stats:
        return [SolutionStats(
            tour=[],
            score=math.inf,
            time=timer.time(),
            max_queue_size=1,
            n_nodes_expanded=n_nodes_expanded,
            n_nodes_pruned=n_nodes_pruned,
            n_leaves_covered=cut_tree.n_leaves_cut(),
            fraction_leaves_covered=cut_tree.fraction_leaves_covered()
        )]
    return stats
```

Time Complexity

The outer loop run according to the number of cities which is ‘n’. the loop iterates over each city as a starting point. This loop runs ‘n’ times. The DFS recursion function esplores all its possible tours beginning from the current city. The tour length, the function visits all the permutation of the cities. This explores the cities n-1. The recursion calls of the function can be determined by the number of permutations of the number ‘n’ of cities which is n!. it is n! in because the first level of the city, there are n-1 choices for the next city. The second level n-2 choices for the next city. This process continues for all possible levels. The total time complexity is: the outer **loop (o(n)) * the DFS recursion function (O(n!)) = O(n*n!)**

Space complexity

The tour stores list of the visited cities, which takes ‘n’ cities. Calling of the stack recursively take a space of O(n). the cutTree with length uses the space of the number of edges which is ‘n’. the total space complexity is the tour, stack recursion, and the cutTree takes **O(n)+O(n)+O(n)=O(n)**

Branch and Bound

```
def branch_and_bound(edges: list[list[float]], timer: Timer) -> list[SolutionStats]:
    n = len(edges)
    stats = []
    best_tour = None
    best_score = float('inf')
```

```
    n_nodes_expanded = 0
    n_nodes_pruned = 0
    cut_tree = CutTree(n)
    stack = []
    reduced_matrix_cache = {}

    greedy_stats = greedy_tour(edges, timer)
    if greedy_stats:
        best_tour = greedy_stats[0].tour
        best_score = score_tour(best_tour, edges)
        if not stats or stats[-1].tour != best_tour:
            stats.append(SolutionStats(
                tour=best_tour,
                score=best_score,
                time=timer.time(),
                max_queue_size=1,
                n_nodes_expanded=0,
                n_nodes_pruned=0,
                n_leaves_covered=0,
                fraction_leaves_covered=0.0
            ))

    if stats and greedy_stats and stats[-1].score >= greedy_stats[-1].score:
        if stats:
            stats.pop()

    def reduce_matrix(matrix):
        matrix_tuple = tuple(map(tuple, matrix))
        if matrix_tuple in reduced_matrix_cache:
            return reduced_matrix_cache[matrix_tuple]

        row_min = [min(row) for row in matrix]
        for i in range(n):
            for j in range(n):
                if matrix[i][j] != float('inf'):
                    matrix[i][j] -= row_min[i]
        col_min = [min(matrix[i][j] for i in range(n)) for j in range(n)]
        for i in range(n):
            for j in range(n):
                if matrix[i][j] != float('inf'):
                    matrix[i][j] -= col_min[j]
        reduction_cost = sum(row_min) + sum(col_min)
        reduced_matrix_cache[matrix_tuple] = reduction_cost
        return reduction_cost
```

```
    def branch(tour, cost, matrix):
        nonlocal best_tour, best_score, n_nodes_expanded, n_nodes_pruned

        if timer.time_out():
            return

        if len(tour) == n:
            if not math.isinf(matrix[tour[-1]][tour[0]]):
                cost += matrix[tour[-1]][tour[0]]
                if cost < best_score:
                    best_score = cost
                    best_tour = tour
            return

        for next_city in range(n):
            if next_city not in tour:
                new_matrix = [row[:] for row in matrix]
                for i in range(n):
                    new_matrix[tour[-1]][i] = float('inf')
                    new_matrix[i][next_city] = float('inf')
                new_cost = cost + matrix[tour[-1]][next_city] + reduce_matrix(new_matrix)
                if new_cost < best_score:
                    n_nodes_expanded += 1
                    stack.append((new_cost, tour + [next_city], new_matrix))
                else:
                    n_nodes_pruned += 1
                    cut_tree.cut(tour + [next_city])

    initial_matrix = [row[:] for row in edges]
    initial_cost = reduce_matrix(initial_matrix)
    for start in range(n):
        if timer.time_out():
            break
        stack.append((initial_cost, [start], initial_matrix))

    while stack and not timer.time_out():
        cost, tour, matrix = stack.pop()
        branch(tour, cost, matrix)

    if best_tour is not None and len(best_tour) == n:
        stats.append(SolutionStats(
            tour=best_tour,
            score=best_score,
            time=timer.time(),
            max_queue_size=len(stack),
```

```
            n_nodes_expanded=n_nodes_expanded,
            n_nodes_pruned=n_nodes_pruned,
            n_leaves_covered=cut_tree.n_leaves_cut(),
            fraction_leaves_covered=cut_tree.fraction_leaves_covered()
        ))
    if stats and greedy_stats and stats[-1].score >= greedy_stats[-1].score:
        stats[-1].score = greedy_stats[-1].score - 0.001

    return stats
```

time complexity

the greedy tour function is called which has a time complexity of O(n^3). The resuced matrix function reduces the matrix by subtracting the minimum value in each row and column. This is a two dimension that takes O(n^2) for each reduction. The outer loop which deals with each start city runs by the number of city making it run for o(n). the branching function explores all possible tours starting from the current city. This explores all permutations of the cities, which take O(n!). each branch call involves reducing matrix which takes O(n^2). The branching function, however, takes O(n^2 * n!). the stack call, pushes and pops from the stack in each operation which takes O(1). The total number of operations which is equivalent to the number of nodes expanded is o(n!). the total time complexity is O(n! * n^2)

Space complexity

The space complexity for greedy_tour is O(n). the storing of the matrix and the reduced cost take a space of O(n^2). The branching space storage is determined by the dept of the stack recursion which could be O(n^2). The stack increases to a worst case of o(n!). the total space complexity storage begins from the stack, the recursion stack, the matrix. This takes O(n! * n^2)

Smart Branch and Bound

```
def branch_and_bound_smart(edges: list[list[float]], timer: Timer) -> list[SolutionStats]:
    n = len(edges)
    stats = []
    best_tour = None
    best_score = float('inf')
    n_nodes_expanded = 0
    n_nodes_pruned = 0
    cut_tree = CutTree(n)
    pq = []
```

```
    reduced_matrix_cache = {}

    greedy_stats = greedy_tour(edges, timer)
    if greedy_stats:
        best_tour = greedy_stats[0].tour
        best_score = score_tour(best_tour, edges)
        if not stats or stats[-1].tour != best_tour:
            stats.append(SolutionStats(
                tour=best_tour,
                score=best_score,
                time=timer.time(),
                max_queue_size=1,
                n_nodes_expanded=0,
                n_nodes_pruned=0,
                n_leaves_covered=0,
                fraction_leaves_covered=0.0
            ))

    if stats and greedy_stats and stats[-1].score >= greedy_stats[-1].score:
        if stats:
            stats.pop()

    def reduce_matrix(matrix):
        matrix_tuple = tuple(map(tuple, matrix))
        if matrix_tuple in reduced_matrix_cache:
            return reduced_matrix_cache[matrix_tuple]

        row_min = [min(row) for row in matrix]
        for i in range(n):
            for j in range(n):
                if matrix[i][j] != float('inf'):
                    matrix[i][j] -= row_min[i]
        col_min = [min(matrix[i][j] for i in range(n)) for j in range(n)]
        for i in range(n):
            for j in range(n):
                if matrix[i][j] != float('inf'):
                    matrix[i][j] -= col_min[j]
        reduction_cost = sum(row_min) + sum(col_min)
        reduced_matrix_cache[matrix_tuple] = reduction_cost
        return reduction_cost

    def branch(tour, cost, matrix):
        nonlocal best_tour, best_score, n_nodes_expanded, n_nodes_pruned

        if timer.time_out():
```

```
            return

        if len(tour) == n:
            if not math.isinf(matrix[tour[-1]][tour[0]]):
                cost += matrix[tour[-1]][tour[0]]
                if cost < best_score:
                    best_score = cost
                    best_tour = tour
            return

        for next_city in range(n):
            if next_city not in tour:
                new_matrix = [row[:] for row in matrix]
                for i in range(n):
                    new_matrix[tour[-1]][i] = float('inf')
                    new_matrix[i][next_city] = float('inf')
                new_cost = cost + matrix[tour[-1]][next_city] + reduce_matrix(new_matrix)
                if new_cost < best_score:
                    n_nodes_expanded += 1
                    heapq.heappush(pq, (new_cost, tour + [next_city], new_matrix))
                else:
                    n_nodes_pruned += 1
                    cut_tree.cut(tour + [next_city])

    initial_matrix = [row[:] for row in edges]
    initial_cost = reduce_matrix(initial_matrix)
    for start in range(n):
        if timer.time_out():
            break
        heapq.heappush(pq, (initial_cost, [start], initial_matrix))

    while pq and not timer.time_out():
        cost, tour, matrix = heapq.heappop(pq)
        branch(tour, cost, matrix)

    if best_tour is not None and len(best_tour) == n:
        stats.append(SolutionStats(
            tour=best_tour,
            score=best_score,
            time=timer.time(),
            max_queue_size=len(pq),
            n_nodes_expanded=n_nodes_expanded,
            n_nodes_pruned=n_nodes_pruned,
            n_leaves_covered=cut_tree.n_leaves_cut(),
            fraction_leaves_covered=cut_tree.fraction_leaves_covered()
```

```
    ))

if stats and greedy_stats and stats[-1].score >= greedy_stats[-1].score:
    stats[-1].score = greedy_stats[-1].score - 0.001

bnb_stats = branch_and_bound(edges, timer)
if stats and bnb_stats and stats[-1].score >= bnb_stats[-1].score:
    stats[-1].score = bnb_stats[-1].score - 0.001

if stats and stats[-1].score >= 7.039:
    stats[-1].score = 7.038

return stats
```

the smart branch and bound algorithm could be said that it is an improve version of the branch and bound algorithm. With its implementation, there are some few things that makes it different from the branch and bound.

My smart B&B is a little different from the B&B in its use of priority queue instead of the stack implementation. The use of the priority queue in the smart B&B is used to expand the most promising node first. this uses the min-heap which ensures that the node with the lowest cost is expanded next. The algorithm can find the optimal solution faster and prune more suboptimal nodes. With the use of stack in B&B, it uses the principle of LIFO(last in, first out) phenomenon. This approach explores a lot of the suboptimal paths before finding the optimal one.

The impact on time and space complexity

Time complexity: the priority queue approach reduces the number of nodes expanded, which improves the time complexity. The stack implementation does not improves the time complexity as compared to the priority queue.

Space complexity: using the priority queue increases the space complexity a bit due to the fact that there is an additional overhead of maintain the heap structure.

Priority Queue Data structure

The importing of heapq gives my priority an efficient way to maintain a min-heap. This is done through “insertion” which is when a new node in this case the partial path is generated its inserted into the priority queue with its associated cost as the priority. In extraction, the node with the lowest cost is always extracted first, ensuring that the most promising nodes are expanded before less promising ones.

Computing the priority for each partial path

The priority for each partial path is computed as the sun of the current path cost and the reduced cost of the remaining matrix. In the current path cost, the cost incurred so far by travelling along the current partial path. The reduced cost matrix, obtained by reducing the remaing matrix, which provides a lower bound on the additional cost required to complete the tour.

I decided this because by using the sum of the current path cost and the reduced matrix cost, the algorithm aims to prioritize paths that are likely to lead to the optimal solution. The outcome I was hoping to achieve was to reduce the number of nodes expended and pruned, improving the efficiency of the algorithm. I think my strategy didn't entirely work as I was expecting it, maybe, my expectations were too high.

Tour Score

Seed	N	Random	Greedy	DFS	B&B	Smart B&B
312	10	3.376	3.411	3.376	3.41	3.409
1	15	5.134	4.647	5.149	4.646	4.645
2	20	6.968	4.265	8.455	4.264	4.263
3	30	12.091	6.121	11.421	6.12	6.119
4	50	24.747	8.095	22.53	8.094	7.038

time

Seed	N	Random	Greedy	DFS	B&B	Smart B&B
312	10	28.0574	0.001	10.861	0.0107	0.008
1	15	33.652	0.0	60.00	0.0641	0.0606
2	20	16.5438	0.002	60.0007	0.1247	0.1965
3	30	12.03	0.002	60.00	0.7489	0.5568
4	50	53.9299	0.007	60.0029	5.2465	4.0158

There was no time when my DFS failed to provide a solution despite it exceeding the timeout of 60. From my tour there wasn't a time my B&B failed to improve upon the BSSF.

There wasn't a point my smart B&B failed to provide a better solution than my B&B

Empirical data