Exploration of the Branch and Bound Algorithm

Through the Traveling Salesman Problem

Katherine Wilsdon

Idaho State University

Reduced Cost Matrix Algorithm

Reduced Cost Matrix

The reducedCostMatrix() function reduces each row and column to contain a zero. The minimum cost of each row is determined. There are two nested for loops that reduce the row by the minimum cost. Similarly, the minimum cost of each column is determined. If the minimum cost in the column is not zero, the column is reduced by the minimum cost.

Time Complexity

Finding the minimum cost for each row and column is $O(n^2)$ because the argmin() function in numpy most likely has two for loops. The algorithm contains two sets of two nested for loops, which is $O(n^2)$.

Space Complexity

The space complexity is $O(n^2)$ because the cost matrix is a 2D array.

```
function reducedCostMatrix(cost_matrix, ncities)
   lower_bound = 0
   reduced_cost_matrix = cost_matrix
   # Reduce each row to contain a zero
   Find the minimum index in each row of the cost_matrix
   for i from 0 to ncities
       lower_bound += minimum index in row i
       for j from 0 to ncities
           if not infinity
               reduce row index by minimum index in row i

   # Reduce each column that does not contain a zero to have a zero
   Find the minimum index in each row of the cost_matrix
   for i from 0 to ncities
       if the minimum index is not 0
           lower_bound += minimum index in column i
           for j from 0 to ncities
               if not infinity
                   reduce row index by minimum index in column i
   return lower_bound, reduced_cost_matrix
```

Figure 1: Reduced Cost Matrix Pseudocode.

```python
def reducedCostMatrix(self, cost_matrix, ncities):
    lower_bound = 0
    row_cost_matrix = np.copy(cost_matrix)
    # Get the minimum index in each row
    min_row_indices = np.argmin(cost_matrix, axis=1)

    # Reduce each row to contain a zero
    for i in range(ncities):
        lower_bound += cost_matrix[i][min_row_indices[i]]
        for j in range(ncities):
            if not np.isinf(cost_matrix[i][j]):
                row_cost_matrix[i][j] -= cost_matrix[i][min_row_indices[i]]

    # Get the minimum index in each column
    min_col_indices = np.argmin(row_cost_matrix, axis=0)
    reduced_cost_matrix = np.copy(row_cost_matrix)
    # Reduce each column that does not contain a zero to have a zero
    for i in range(ncities):
        if row_cost_matrix[min_col_indices[i]][i] != 0:
            lower_bound += row_cost_matrix[min_col_indices[i]][i]
            for j in range(ncities):
                if not np.isinf(row_cost_matrix[j][i]):
                    reduced_cost_matrix[j][i] -= row_cost_matrix[min_col_indices[i]][i]
    return lower_bound, reduced_cost_matrix
```

Figure 2: Reduced Cost Matrix Algorithm.

Partial Path Search Algorithm

Stack Data Structure and Finding More Solutions Early

A python list was used to represent a stack. The pop function removes the item at the end of the list. The append function adds the item at the end of the list. A stack, first in first out (FIFO), was chosen because it uses a depth first approach where it prioritizes full solutions instead of prioritizing based on the lower bound as in a priority queue. Using a stack allows in finding solutions earlier than a breadth first search, which prunes less and processes every layer before reaching the final layer. I did not try to implement a priority queue.

States Data Structure

Each state that was put into the stack was a tuple consisting of the lower bound, the reduced cost matrix, and the partial solution. The reduced cost matrix was a numpy array and the partial solution was a list.

Partial Path Search

While the stack is not empty, remove a state from the stack. States that have a higher lower bound than the BSSF lower bound are pruned. The variable i is set to the last node in the parent's partial solution. For each unvisited city, a child state is created. The path cost at (i, j) of the parent cost matrix is added to the child's lower bound. The ith row and the jth column are set to infinity as well as the back edge (j, i). The minimum index in each row of the child cost matrix is found. If the minimum index cost does not equal zero, add the cost to the child's lower bound and reduce every index in the row by the minimum cost. The same approach is done for the columns to obtain a zero in every row and column of the unvisited nodes. If a full solution is reached, add the cost from the last node to node 0 to the child's lower bound. If the child's lower bound is less than the BSSF lower bound, the BSSF is updated. Else if the child's lower bound is less than the BSSF lower bound, the state is added to the stack. Else, the state is pruned.

Time Complexity

Within the while loop, the first for loop is used for visiting all the unvisited nodes. Within this for loop, there are multiple single for loops for setting rows and columns to infinity. Within the initial for loop, there are also two sets of two nested for loops for adjusting each row and column to contain a zero. The combination of these for loops has a time complexity of $O(n^3)$. The stack is O(1) because the only operations used are push and pop. The while loop that runs until the stack is empty has an exponential runtime, $O(b^n)$, where b is the number of nodes put on the stack and n is the number of cities. The combined time complexity is $O(n^3b^n)$.

Space Complexity

The space complexity is $O(n^2b^n)$ because one reduced cost matrix, $O(n^2)$, is stored for every state created and putting each b state on the stack for n cities is $O(b^n)$. The partial solution and the stack are $O(n)$ because it is a 1D list.

```
function partialPathSearch(stack, ncities, best_sln)
    bssf_lower_bound = best_sln[0]
    bssf_sln = best_sln[1]
    while stack not empty:
        parent_lower_bound, parent_cost_matrix, parent_partial_sln = stack.pop()
        if parent_lower_bound >= bssf_lower_bound
            prune state
        i = last node in parent_partial_sln

        for j from 0 to ncities
            if i != j and j not in parent_partial_sln
                child_lower_bound = parent_lower_bound
                child_cost_matrix = parent_cost_matrix
                child_partial_sln = parent_partial_sln
                Add j to child_partial_sln

                child_lower_bound += path cost at i, j of parent_cost_matrix

                for k from 0 to ncities
                    Set row i equal to infinity
                for k from 0 to ncities
                    Set column j equal to infinity
                Set back edge (j,i) to infinity

                Find the minimum index in each row of the child_cost_matrix
                for k from 0 to ncities
                    if the minimum index is not 0
                        child_lower_bound += minimum index in row k
                        for l from 0 to ncities
                            if not infinity
                                reduce row index by minimum index in row k

                Find the minimum index in each column of the child_cost_matrix
                for k from 0 to ncities
                    if the minimum index is not 0
                        lower_bound += minimum index in column k
                        for l from 0 to ncities
                            if not infinity
                                reduce row index by minimum index in column k

                if length(child_partial_sln) == ncities
                    child_lower_bound += cost from the last node to node 0
                    if child_lower_bound < bssf_lower_bound
                        update bssf_lower_bound and bssf_sln
                    else if child_lower_bound < bssf_lower_bound
                        stack.push(child_lower_bound, child_cost_matrix, child_partial_sln)
                    else
                        prune state
    return bssf_lower_bound, bssf_sln
```

Figure 3: Partial Path Search Pseudocode.

```
def partialPathSearch(self, stack, ncities, bssf, start_time, time_allowance):
    num_of_solutions = 0
    pruned_states = 0
    total_states = 0
    max_stack_size = 0
    best_sln = bssf

    while len(stack) != 0 and time.time() - start_time < time_allowance:
        if len(stack) > max_stack_size:
            max_stack_size = len(stack)
        active_state = stack.pop()
        parent_cost_matrix = np.copy(active_state[1])
        parent_partial_sln = copy.deepcopy(active_state[2])
        parent_lower_bound = int(active_state[0])
        """Prune the parent state if its lower bound is greater than best solution's lower bound"""
        if parent_lower_bound >= best_sln[0]:
            pruned_states += 1
            continue
        # Set i to be the last node in the partial solution
        i = parent_partial_sln[-1]
        breaker = False
```

Figure 4: Partial Path Search Algorithm (Part 1).

```
        # Loop through for each possible state of unvisited cities (j=column)
        for j in range(len(parent_cost_matrix)):
            if time.time() - start_time < time_allowance and j != i and j not in parent_partial_sln:
                child_lower_bound = int(active_state[0])
                child_cost_matrix = np.copy(active_state[1])
                child_partial_sln = copy.deepcopy(active_state[2])
                child_partial_sln.append(j)

                """Add the path cost to the lower bound"""
                if child_cost_matrix[i][j] == float("inf"):
                    continue
                else:
                    child_lower_bound += parent_cost_matrix[i][j]

                """Set to infinity"""
                # Set row i equal to infinity
                for k in range(len(child_cost_matrix)):
                    child_cost_matrix[i][k] = float("inf")
                # Set column j equal to infinity
                for k in range(len(child_cost_matrix)):
                    child_cost_matrix[k][j] = float("inf")
                # Set back edge (j,i) to infinity
                child_cost_matrix[j][i] = float("inf")
```

Figure 5: Partial Path Search Algorithm (Part 2).

```python
"""If the cost does not equal zero, reduce matrix to contain zeros in every row and column"""
if parent_cost_matrix[i][j] != 0:
    """Set each row to zero"""
    # Get the minimum index in each row
    min_row_indices = np.argmin(child_cost_matrix, axis=1)
    # For each row in min_row_indices (k=row)
    for k in range(len(min_row_indices)):
        if child_cost_matrix[k][min_row_indices[k]] != 0 and k != i:
            # if the row minimum is infinity and not in the partial solution, not a viable state
            if child_cost_matrix[k][min_row_indices[k]] == float("inf"):
                is_row_n_partial_sln = False
                # Determine if infinity row has already been visited (in the partial solution)
                for visited_node in parent_partial_sln:
                    if visited_node == k:
                        is_row_n_partial_sln = True
                        break
                # a row of infinities was introduced making the node unreachable
                if not is_row_n_partial_sln:
                    breaker = True
                    break
            # The minimum in a row is greater than 0
            else:
                # Reduce the row to contain a zero
                child_lower_bound += child_cost_matrix[k][min_row_indices[k]]
                # For each column in the provided row of min_row_indices (l=column), update the value
                for l in range(len(child_cost_matrix)):
                    if not np.isinf(child_cost_matrix[k][l]) and l != min_row_indices[k]:
                        child_cost_matrix[k][l] -= child_cost_matrix[k][min_row_indices[k]]
                child_cost_matrix[k][min_row_indices[k]] -= child_cost_matrix[k][min_row_indices[k]]
    # break out of second for loop to continue on to another state
    if breaker:
        breaker = False
        continue
```

Figure 6: Partial Path Search Algorithm (Part 3).

```python
"""Set each column to zero"""
# Get the minimum index in each column
min_col_indices = np.argmin(child_cost_matrix, axis=0)
# For each column in min_col_indices (k=col)
for k in range(len(min_col_indices)):
    if child_cost_matrix[min_col_indices[k]][k] != 0 and k != j:
        # if the column minimum is infinity and not in the partial solution, not a viable state
        if child_cost_matrix[min_col_indices[k]][k] == float("inf"):
            is_col_in_partial_sln = False
            len_of_child_partial_sln = len(child_partial_sln)
            # Determine if infinity column has already been visited (in the partial solution)
            for h in range(1, len_of_child_partial_sln):
                if child_partial_sln[h] == k:
                    is_col_in_partial_sln = True
                    break
            # a row of infinities was introduced making the node unreachable
            if not is_col_in_partial_sln:
                breaker = True
                break
        # the minimum in a column is greater than 0
        else:
            # Reduce the row to contain a zero
            child_lower_bound += child_cost_matrix[min_col_indices[k]][k]
            # For each column in the provided row of min_row_indices (l=row), update the value
            for l in range(len(child_cost_matrix)):
                if not np.isinf(child_cost_matrix[l][k]) and l != min_col_indices[k]:
                    child_cost_matrix[l][k] -= child_cost_matrix[min_col_indices[k]][k]
            child_cost_matrix[min_col_indices[k]][k] -= child_cost_matrix[min_col_indices[k]][k]
# break out of second for loop to continue on to another state
if breaker:
    breaker = False
    continue
```

Figure 7: Partial Path Search Algorithm (Part 4).

```
                """When in the reduced cost matrix form"""
                total_states += 1
                # If all cities have been visited
                if len(child_partial_sln) == ncities:
                    # Add the cost from the last node to node 0
                    child_lower_bound += child_cost_matrix[child_partial_sln[-1]][0]
                    # if the lower bound is better than the best solution's lower bound, update the best solution
                    if child_lower_bound < best_sln[0]:
                        best_sln = (child_lower_bound, child_partial_sln)
                        num_of_solutions += 1
                # If the partial solution is less than the best solution's lower bound, add to the stack
                elif child_lower_bound < best_sln[0]:
                    stack.append((child_lower_bound, child_cost_matrix, child_partial_sln))
                # Else prune the state
                else:
                    pruned_states += 1
                breaker = False
    return best_sln, num_of_solutions, pruned_states, total_states, max_stack_size
```

Figure 8: Partial Path Search Algorithm (Part 5).

Branch and Bound Algorithm

Initial BSSF

The initial best so far solution (BSSF) is obtained from the defaultRandomTour() function. This returns the first solution found but it is not the optimal solution. A better approach is to use a greedy algorithm to obtain the BSSF but due to time restrictions, it was not implemented in the project.

Branch and Bound

A 2D cost matrix is created and populated by calling the costTo( ) function. The lower bound and the reduced cost matrix are returned from the reducedCostMatrix( ) function. The initial state is pushed onto the stack. The BSSF lower bound and the BSSF solution are returned from the partialPathSearch( ). The variables are added to results and results is returned.

Time Complexity

Populating the cost matrix is $O(n^2)$ because of the two for loops. The reducedCostMatrix( ) function is $O(n^2)$ as previously stated in the Time Complexity section of the Reduced Cost Matrix Algorithm. The partialPathSearch( ) function is $O(n^3b^n)$ as stated in the Time Complexity section of the Partial Path Search Algorithm. The summation of time complexity is $O(n^2 + n^2 + n^3b^n) = O(n^3b^n)$. The default random tour has a time complexity of $O(nb^n)$ because it contains one for loop in a while loop.

```
function branchAndBound()
    Create cost_matrix 2D array
    ncities = length(cities)
    stack = []

    bssf_lower_bound, bssf_sln = defaultRandomTour()
    best_sln = (bssf_lower_bound, bssf_sln)

    for i from 0 to ncities
        for j from 0 to ncities
            populate cost matrix

    lower_bound, reduced_cost_matrix = reducedCostMatrix(cost_matrix, ncities)

    partial_sln = [0]
    stack.push(lower_bound, reduced_cost_matrix, partial_sln)

    bssf_lower_bound, p2 = partialPathSearch(stack, ncities, best_sln)

    add variables to results
    return results
```

Figure 9: Brand and Bound Pseudocode.

```python
def branchAndBound(self, time_allowance=60.0):
    results = {}
    cities = self._scenario.getCities()
    ncities = len(cities)
    foundTour = False
    cost_matrix = np.zeros((ncities, ncities), dtype=float)
    stack = []

    results_default = self.defaultRandomTour(60.0)
    partial_sln = []
    # Append index of city to partial solution
    for i in range(len(results_default['soln'].route)):
        partial_sln.append(results_default['soln'].route[i]._index)
    best_sln = (results_default['cost'], partial_sln)

    start_time = time.time()
    """Populate the cost_matrix"""
    for i in range(ncities):
        for j in range(ncities):
            cost_matrix[i][j] = cities[i].costTo(cities[j])

    """Obtain the reduced cost matrix"""
    lower_bound, reduced_cost_matrix = self.reducedCostMatrix(cost_matrix, ncities)
    """Put starting node on the stack"""
    partial_sln = [0]
    stack.append((lower_bound, reduced_cost_matrix, partial_sln))
    """Get the best solution so far"""
    best_sln, num_of_solutions, pruned_states, total_states, max_stack_size = self.partialPathSearch(stack,
        ncities, best_sln, start_time, time_allowance)

    route = []
    """Build the route of cities"""
    for i in range(ncities):
        route.append(cities[best_sln[1][i]])
    bssf = TSPSolution(route)
    if bssf.cost < np.inf:
        # Found a valid route
        foundTour = True
    end_time = time.time()
    """Add statistics to results and return results"""
    if results_default['cost'] == bssf.cost:
        results['cost'] = 0
    else:
        results['cost'] = bssf.cost if foundTour else math.inf
    results['time'] = end_time - start_time
    results['count'] = num_of_solutions
    results['soln'] = bssf
    results['max'] = max_stack_size
    results['total'] = total_states
    results['pruned'] = pruned_states
    return results
```

Figure 10: Branch and Bound Algorithm.

Results

Around 70 – 80% of the total states are pruned because their lower bound was greater than the BSSF. When the number of cities is lower, the ratio of pruned / total states is closer to 70%. As the number of cities increases, the ratio of pruned / total states gradually increases to about 80%. With 18 cities at a seed of 748, the algorithm took ~50 seconds and was the second highest total number of states created at 390,107. 21 cities at a seed of 7 had the highest total number of states created at 429,391. If I had to guess, this problem was almost solved before timing out at 60 seconds because the total number of states created was the highest and the number of cities, 21, is a little higher than 18 cities (the second highest total number of states that just barely finished). City 18 and city 26 had the highest number of BSSF updates. This indicates that the initial BSSF obtained from the default random tour was probably a poor starting solution. 49 cities had the highest number of max queue states which makes sense because it has the greatest number of cities. It seems the max queue states is proportionate to the number of cities.

# Cities	Seed	Running Time (seconds)	Cost of Best Tour Found	Max Queue States	# of BSSF Updates	Total # of States Created	Total # of States Pruned
15	20	10.117	10534	67	60	94,848	73,448
16	902	20.032	7954	82	62	165,865	132,373
17	88	10.638	10980	92	66	95,967	77,729
18	748	50.385	10929	98	118	390,107	316,354
11	151	0.283	6822	34	19	3,721	2,558
14	44	2.502	10741	61	62	25,174	19,648
26	111	60.001	20604	228	117	308,116	257,699
21	7	60.001	11278	145	62	429,391	358,833
49	424	60.001	45134	890	57	177,995	143,603
38	361	60.001	32633	533	33	225,341	179,263

Figure 11: Results From 10 Different Problems Ranging Between 10 and 40 Cities.