Josiah Saunders
Section 001
TSP Project Report
Nov. 30, 2019

Time and Space Complexity

Let V be the list of cities and |V| be the number of cities. This section is separated into sub-sections. BranchAndBoundSolver and GreedySolver are classes that inherit from SolverBase. These classes implement their respective traveling salesperson algorithms in the run_algorithm method.

TSPSolver (TSPSolver.py)

branchAndBound

Initializes and runs a branch and bound solver **(see BranchAndBoundSolver analysis below).**

- Time: O(|V|! * |V|^2)
- Space: O(qsize * |V|^2)

greedy

initializes and runs a greedy solver **(see GreedySolver analysis below)**

- Time: O(|V|^3)
- Space: O(|V|)

BranchAndBoundSolver (branch_bound/solver.py)

__init__

- Time: Simply sets up an empty priority queue and updates a finite number of values —> O(1)
- Space: The queue created is initially empty (and is capped at MAX_NODES) —> O(1)

run_algorithm

This method contains the meat of the branch and bound algorithm. Throughout this analysis, let qsize be the size of the queue capped at MAX_NODES.

The method first invokes the greedy solver to get an initial BSSF which takes O(|V|^3) time and O(|V|) space (which is far less complex than the rest of the algorithm and can be ignored). The method then enqueues the initial state and loops through all states put onto the queue (or until a time limit cap is met). Each state, worst case, is enqueued and dequeued from the queue, which takes O(log(qsize)) time as an upper bound.

When dequeued, each state is either expanded or considered for a solution (i.e. try and close the route). The state expansion has a more significant time complexity and happens much more frequently than closing a route, so only state expansion will be considered. Expanding a state invokes BranchNode.generate_children(), which takes O(|V|^2) time and space for each state generated. Each state generated is then placed onto the queue (so long as the queue size doesn't exceed MAX_NODES). To generate the key for each state, _get_node_key is invoked which takes up constant time and space. The depth of the tree is at most |V| levels deep. A given state, B, will generate O(|V| - B.get_depth()) other child states. This gives, worst case, a total of O(|V|!) states.

So, the following are the worst case time and space complexities (in practice, however, the algorithm performs much better than this due to pruning):

- Time: With O(|V|!) states, |V|^2 cost to generate each state, and O(log(qsize)) cost to enqueue and dequeue a state —> O(|V|! * |V|^2)
- Space: With at most qsize states stored at a given point in time and each state holding an RCM of size O(|V|^2) —> O(qsize * |V|^2)

_get_node_key

- Simply divides two already stored numbers —> O(1) time and space

---

RCM (branch_bound/rcm.py)

__init__

- Time: A |V|x|V| matrix is initialized, which requires |V|x|V| loops —> O(|V|^2)
- Space: A |V|x|V| matrix is allocated; the other allocations aren't significant —> O(|V|^2)

do_selection

- Time: Runs a single O(|V|) loop to update the row and columns selected. All other calls take constant time —> O(|V|)
- Space: No significant allocations, simply updates the matrix already cached —> O(1)

do_reduction

- Time: Performs a O(|V|) loop through each row. At each row, it performs two additional O(|V|) loops to find the min of the row and subtract the min from each value in the row. The method then performs the same operation for the columns —> O(|V|^2)
- Space: No significant allocations, simply updates the matrix already cached —> O(1)

BranchNode (branch_bound/bnode.py)

__init__ and __lt__

- Time: Both of these methods just assign and compare a finite amount of variables —> O(1)
- Space: Constant space complexity, significant variables are passed by reference —> O(1)

compute_path

- Time: Recurses upwards through the branch node tree. All other calls take constant time. Since the tree will be at most |V| levels deep —> O(|V|)
- Space: Builds a route array which stores at most |V| cities —> O(|V|)

generate_child_nodes (Expanding one SearchState into others)

- Time: Runs a |V| loop. Each loop, worst case, will copy an RCM (which is assumed to take O(|V|^2) for a |V|x|V| RCM). Each loop will then run a selection and a reduction on the RCM, which takes O(|V|) and O(|V|^2) respectively. All other calls take constant time —> O(|V|^3)
- Space: With |V| loops and a O(|V|^2) RCM allocation at each loop —> O(|V|^3)

GreedySolver (greedy/solver.py)

__init__

Just calls super().__init__(), constant time and space complexity

run_algorithm

- Time: Loops through each city and makes a call to _greedy_solve (the greedy solution doesn't work every time), each taking O(|V|^2) time. All other calls are constant —> O(|V|^3)
- Space: Calls _greedy_solve, which takes up O(|V|) space. But the memory is not stored after the call —> O(|V|)

_greedy_solve

- Time: Recurses through at most |V| cities and at worst case makes a call to _get_next_city which costs O(|V|). All other calls take constant time —> O(|V|^2)
- Space: Maintains a set of visited cities and a route of the city-visit-order. Each takes up at most |V| nodes —> O(|V|)

_get_next_city

- Time: Loops through all neighboring cities from the source city argument. Each loop does a constant amount of work —> O(|V|)
- Space: No significant allocations are made, just uses cached variables —> O(1)

SolverBase (solver_base.py)

This class simply abstracts the logic behind running the different algorithms. The solver classes implement the run_algorithm method (so the complexity will vary). All other methods in this class are just nice interface methods and require constant time and space. For these reasons, the complexity of this class will be ignored.

State Data Structure Used

In this solution, states are represented using the BranchNode class. BranchNodes have the following class members:

- **_depth**: Represents how many levels deep in the tree this node is (i.e. how many cities this partial path has traversed).
- **_rcm**: A |V|*|V| matrix which represents the reduced cost matrix for the current path. This class also holds the total cost that was produced by reducing the RCM down into its current state.
- **_children**: An array of child nodes which represent the possible cities that can be traversed to next for this BranchNode's current route. This array is initially empty, but is populated when generate_child_nodes is called.
- **_city_index**: The index of the city associated with this branch node.
- **_city**: A reference to the associated city of this branch node.
- **_parent:** A reference to the parent node of this current node (used to compute the route of a given BranchNode by recursively traveling up the tree through each node's parent).

Priority Queue Data Structure Used

This solution uses the PriorityQueue class provided in 'Lib/queue.py' (https://docs.python.org/3/library/queue.html). This priority queue is implemented as a BinaryHeap (using the heapq module). Here is how it works:

- **PriorityQueue.put((Float key, BranchNode node))**:
  - This method is equivalent to inserting an element into a binary heap. Upon inserting, heapq.heappush is called. This method will append the node into the tree (represented through an array) and logically swap the node with its parent while the node's associated key is less than the parent's associated key. This makes the time to insert logarithmic.
  - The object passed is a tuple, with its first value as the key for the given branch node. In the case where two keys are the same, the BranchNode implements the __lt__ function to break ties between nodes.
- **PriorityQueue.get():**
  - This method is equivalent to removing an element from a binary heap. Upon removing, heapq.heappop is called. This method returns the first element in the heap array (a tuple) and moves the last heap array element to the 0th index. Let the root element at index 0 be E. The method then repeatedly swaps E with its child with a smaller key (so long as it has a child with a key smaller than its own key). This makes the time to remove logarithmic.
  - The method returns the next tuple based on priority (via the key). Let this tuple be X. X[0] is the key of the tuple and X[1] is the branch node inserted.
- The other methods used, **empty(), full(), and qsize()**, are all constant time operations to check if qsize() is equal to 0, to check if qsize() is equal to capacity, and to return the number of elements in the queue respectively.

Initial BSSF Approach

To calculate the initial BSSF, this branch and bound TSP solution uses a greedy approach (outlined in the GreedySolver class). This greedy approach picks a random start node. From that node, the algorithm recursively travels through the cities (choosing only cities that haven't been traversed before). When it has a choice to pick between cities, it chooses the city which adds the smallest travel cost. As soon as all cities are traversed, the algorithm attempts to close the route and connect back to the start city.

This solution doesn't find a route every time. Because the cities can be asymmetric, the greedy solver may fail to close a route. For this reason, the solver tries every city as a starting city and runs the greedy algorithm for that city. As soon as a route is found, the solver finishes running and returns a (greedy) solution. Though this doesn't guarantee a solution, it works most of the time and gives a good upper bound for the most optimal TSP solution.

Mechanisms Tried

I tried several different mechanisms to find solutions early. I primarily focussed on creating keys for the priority queue that would allow promising states to be expanded early (so as to update the BSSF many times). Initially, I just used the cost of a state (based on the RCM) as the key for that state in the priority queue. This worked, but wasn't good for finding many solutions. This strategy made it so states were expanded in a breadth-like fashion. States with a lower cost (states closer to the root state) were being expanded first. This made it so very few, if any, solutions were found early.

I then factored in the depth of the tree to combat this. I used the cost of a state divided by the depth of the state (how complete the path is) as the priority key. This made it so states with a lower cost and a higher depth were expanded first. This strategy worked really well—it made it so several intermediate solutions were found, as the most promising states were expanded first. This is the strategy I ended up using for this algorithm.

I also tried factoring in a DFS to calculate a rough cost of a path back to the start city from the current city. This strategy made things much worse because of the complexity overhead of DFS as well as the fact that a DFS isn't guaranteed to hit all remaining cities to be searched.

Empirical Data

Consider the table below. For an increasing number of cities, the time taken to run the algorithm drastically increases. Running the algorithm on city lengths ranging between 12-17 seemed to be doable for most cases. There were, however, several times in which the algorithm would take much longer to compute than the others with the same amount of cities (i.e. row 6 in the table). With more cities (30+), the most optimal solution was never found. This all likely occurs because several states are expanded but don't result in a *more* optimal solution. This stems from the fact that using a state's depth mixed with that state's cost as a priority key is only a *good* guess, **not** necessarily the *right guess*, for dequeueing states and finding optimal routes.

The total number of states created seemed to be pretty variable through different TSP problems—with 49 and 17 cities, 32,576 and 150,512 states were created (respectively). I believe this difference results because this algorithm doesn't choose a perfect path to branch each time (it's variable). This seems to be directly related to the number of states pruned: for 49 and 17 cities, 26,338 and 124,367 are pruned (respectively). It seems that for an X amount of states created, there are a roughly X, but always less than X, states pruned. I believe this happens because of the frequency of BSSF updates. When more solutions are found, more states can be pruned. States that aren't pruned are states which contain a solution when expanded (which rarely happens). So it makes sense that the two would be roughly equal.

	A	B	C	D	E	F	G	H
1	# cities	Seed	Running time (sec)	Cost of best tour found (*=optimal)	Max # of stored states at a given time	# of BSSF updates	Total # of states created	Total # of states pruned
2	15	20	3.79	9367*	80	7	12150	10079
3	16	902	4.94	8352*	81	8	13455	11328
4	15	17	18.96	10318*	82	10	63149	50016
5	12	447	0.29	9149*	45	4	1275	1008
6	17	574	55.85	8960*	98	10	150512	124367
7	13	798	6.47	7664*	55	4	25749	20127
8	14	96	2.57	8092*	66	8	9860	7451
9	30	664	60	13732	366	8	71940	59930
10	35	773	60	15463	1031	8	59742	40343
11	40	990	60	18817	1111	5	46634	35963
12	45	954	60	17852	892	11	37097	28601
13	49	513	60	20007	1175	3	32576	26338

Source Code

TSPSolver (TSPSolver.py)

```
#!/usr/bin/python3

import itertools
import heapq
from TSPClasses import *
import numpy as np
import time

from branch_bound.solver import BranchAndBoundSolver
from greedy.solver import GreedySolver

from which_pyqt import PYQT_VER
if PYQT_VER == 'PYQT5':
    from PyQt5.QtCore import QLineF, QPointF
elif PYQT_VER == 'PYQT4':
    from PyQt4.QtCore import QLineF, QPointF
else:
    raise Exception('Unsupported Version of PyQt: {}'.format(PYQT_VER))


class TSPSolver:
    def __init__(self, gui_view):
        self._scenario = None

    def setupWithScenario(self, scenario):
        self._scenario = scenario

    ''' <summary>
        This is the entry point for the default solver
        which just finds a valid random tour.  Note this could be used to find your
        initial BSSF.
        </summary>
        <returns>results dictionary for GUI that contains three ints: cost of solution,
        time spent to find solution, number of permutations tried during search, the
        solution found, and three null values for fields not used for this
        algorithm</returns>
    '''

    def defaultRandomTour(self, time_allowance=60.0):
        results = {}
        cities = self._scenario.getCities()
        ncities = len(cities)
        foundTour = False
        count = 0
        bssf = None
        start_time = time.time()
        while not foundTour and time.time() - start_time < time_allowance:
            # create a random permutation
            perm = np.random.permutation(ncities)
            route = []
            # Now build the route using the random permutation
            for i in range(ncities):
                route.append(cities[perm[i]])
            bssf = TSPSolution(route)
            count += 1
            if bssf.cost < np.inf:
```

```
            # Found a valid route
            foundTour = True
    end_time = time.time()
    results['cost'] = bssf.cost if foundTour else math.inf
    results['time'] = end_time - start_time
    results['count'] = count
    results['soln'] = bssf
    results['max'] = None
    results['total'] = None
    results['pruned'] = None
    return results

''' <summary>
    This is the entry point for the greedy solver, which you must implement for
    the group project (but it is probably a good idea to just do it for the branch-and
    bound project as a way to get your feet wet).  Note this could be used to find your
    initial BSSF.
    </summary>
    <returns>results dictionary for GUI that contains three ints: cost of best solution,
    time spent to find best solution, total number of solutions found, the best
    solution found, and three null values for fields not used for this
    algorithm</returns>
'''

def greedy(self, time_allowance=60.0):
    # see the GreedySolver class
    # time:     O(|V|^3)
    # space:    O(|V|)
    solver = GreedySolver(self, time_allowance)
    solver.solve()
    return solver.get_results()

''' <summary>
    This is the entry point for the branch-and-bound algorithm that you will implement
    </summary>
    <returns>results dictionary for GUI that contains three ints: cost of best solution,
    time spent to find best solution, total number solutions found during search (does
    not include the initial BSSF), the best solution found, and three more ints:
    max queue size, total number of states created, and number of pruned states.</returns>
'''

def branchAndBound(self, time_allowance=60.0):
    # see the BranchAndBoundSolver class
    # time:     O(|V|! * |V|^2)
    # space:    O(queue_size * |V|^2)
    MAX_NODES = 100000
    solver = BranchAndBoundSolver(self, MAX_NODES, time_allowance)
    solver.solve()
    return solver.get_results()

''' <summary>
    This is the entry point for the algorithm you'll write for your group project.
    </summary>
    <returns>results dictionary for GUI that contains three ints: cost of best solution,
    time spent to find best solution, total number of solutions found during search, the
    best solution found.  You may use the other three field however you like.
    algorithm</returns>
'''

def fancy(self, time_allowance=60.0):
    pass
```

BranchAndBoundSolver (branch_bound/solver.py)

```
from branch_bound.rcm import RCM
from branch_bound.bnode import BranchNode
from queue import PriorityQueue
from copy import deepcopy
from solver_base import SolverBase
from greedy.solver import GreedySolver
from random import randrange
import math


# NOTE: for all code below, let |V| be the number of cities
# NOTE: also let MAX_NODES be the max number of nodes in the priority queue
class BranchAndBoundSolver(SolverBase):

    def __init__(self, tsp_solver, max_nodes, max_time):
        '''
        initializes the base solver as well as creates a priority queue
        with max nodes
        \ntime:     initialize an empty queue --> O(1)
        \nspace:    the queue is initialially empty, but is capped at max nodes --> O(1)
        '''

        super().__init__(tsp_solver, max_time)

        # cap the number of nodes allowed in the priority queue to MAX_NODES
        # NOTE: the priority queue is implemented as a binary tree
        # with O(log(MAX_NODES)) as the upper bound insertion and removal time
        self._node_queue = PriorityQueue(max_nodes)
        self.set_max_concurrent_nodes(0)
        self.try_update_max_concurrent_nodes(self._node_queue.qsize())

    def run_algorithm(self):
        '''
        runs the branch and bound algorithm for the TSP by creating an pruning
        routes as it goes, in an effort to find the most optimal solution
        \ntime:     consider to following:
                    1) the initial greedy solver takes O(|V|^3) time to compute;
                    2) each branch node has (|V| - node.get_depth()) sub-problems
                       (branch nodes), each costing O(|V|^2) to produce;
                    3) each branch node requires and upper bound of O(log(queue_size)
                       to be queued/dequeued;
                    4) computing a path from a branch node only happens if children aren't
                       generated, thus this cost can be ignored;
                    5) the tree generated is, worst case, n levels deep;
                    `total complexity --> O(|V|! * |V|^2)` capped at max_time
        \nspace:    with at most queue_size in the queue, each node takes O(|V|^2)
                    space --> O(queue_size * |V|^2)
        '''

        # O(|V|^3) time and O(|V|) space call to greedily compute the initial bssf
        greedy = GreedySolver(self.get_tsp_solver(), self.get_max_time())
        greedy.solve()
        self.set_bssf(greedy.get_bssf())
        if self.exceeded_max_time():
            return

        # pick the initial city and initialize the root RCM based on that city
        start_index = randrange(self.get_city_count())
        start_city = self.get_city_at(start_index)
        start_rcm = RCM(self.get_scenario())
```

```
        start_rcm.do_reduction()

        # construct the root branch node from the start city with its associated RCM
        # (the first state created for this branch and bound problem)
        root = BranchNode(start_rcm, start_city, start_index, None)
        self.increment_total()
        if root.get_cost() < self.get_best_cost():
            self._node_queue.put((self._get_node_key(root), root))
            self.try_update_max_concurrent_nodes(self._node_queue.qsize())

        # loop while there are still paths to process and theres still time
        while not self._node_queue.empty() and not self.exceeded_max_time():

            # binary heap removal: upper bound O(log(MAX_NODES)) to remove from the queue
            current = self._node_queue.get()[1]

            # prune paths that exceed the bssf as we go
            if current.get_cost() >= self.get_best_cost():
                self.increment_pruned()
                continue

            # if we've traversed all cities, try to connect the route to its start node
            # (indicating that we're considering a solution)
            if current.get_depth() == self.get_city_count():

                # O(1) check to see if the this route fails
                loop_cost = current.get_city().costTo(start_city)
                if loop_cost == math.inf or loop_cost >= self.get_best_cost():
                    continue

                # the current route is more optimal, choose it
                route = current.compute_path()      # O(|V|) time and space
                self.set_bssf_from_route(route)     # assumed constant time and space
                self.increment_solution_count()     # O(1)
                print('sol (t: {0:.3f})'.format(self.get_clamped_time()))
                continue

            # continue branching: create nodes for the currently selected node (popped
            # off of the priority queue) --> O(|V|^2) time and space for each child generated
            current.generate_child_nodes(self.get_cities())
            self.increment_total(current.get_child_count())

            # loop through all children nodes --> O(|V|)
            for child in current.get_children():
                # try to push the child onto the queue --> upper bound O(log(MAX_NODES))
                if not self._node_queue.full() and child.get_cost() < self.get_best_cost():
                    self._node_queue.put((self._get_node_key(child), child))
                else:
                    self.increment_pruned()

            # some nodes were inserted into the queue, try toupdate the max nodes
            # stored at a given time --> O(1)
            self.try_update_max_concurrent_nodes(self._node_queue.qsize())

        # count the remaining nodes as pruned
        self.increment_pruned(self._node_queue.qsize())
        print('fin (t: {0:.3f})'.format(self.get_clamped_time()))

    def _get_node_key(self, bnode):
        '''
        compute the key for a given node by factoring in the cost of a node
        (equivalent to the node's RCM cost) as well as the depth of the node.
        nodes with a lower cost and a higher depth (meaning that more of a
        path has been discovered) will have higher priority, and will thus
```

```
be dequeued first
\ntime:     constant --> O(1)
\nspace:    constant --> O(1)
'''
return bnode.get_cost() / bnode.get_depth()
```

RCM (branch_bound/rcm.py)

```
from typing import List
from TSPClasses import City
import numpy
import math


# NOTE: for all code below, let |V| be the number of cities
class RCM:

    def __init__(self, scenario):
        '''
        initializes the RCM based on city-to-city costs
        \ntime:     allocating and initializing the matrix --> O(|V|^2)
        \nspace:    allocating an |V|*|V| matrix --> O(|V|^2)
        '''

        super().__init__()

        cities = scenario.getCities()
        city_len = len(cities)

        # allocate the matrix, requiring O(|V|^2) space and assumed
        # to take O(|V|^2) time
        self._cost = 0.0
        self._length = city_len
        self._values = numpy.empty((self._length, self._length))

        # O(|V|^2) loop to set all inital city-to-city costs
        for row in range(self._length):
            for col in range(self._length):
                self._values[row, col] = cities[row].costTo(cities[col])

    def do_selection(self, row_index, col_index):
        '''
        performs a selection on the specified row and columns (i.e. marks
        cityX --> cityY as visited and increments the cost by taking that route)
        \ntime:     Loop through all values in a row and column --> O(|V|)
        \nspace:    no significant allocations --> O(1)
        '''

        # constant time increment the cost of the rcm based on the city chosen
        self._cost += self._values[row_index, col_index]

        # O(|V|) loop to update all values in the row/column to infinity, and
        # thereby mark cityX --> cityY as visited
        self._values[col_index, row_index] = math.inf
        for i in range(self._length):
            self._values[i, col_index] = math.inf
            self._values[row_index, i] = math.inf

    def do_reduction(self):
        '''
        row and column reduction to increment the cost of this RCM
        \ntime:     two (|V|^2) loops to reduce --> O(|V|^2)
        \nspace:    just using memory alread allocated --> O(1)
        '''

        # O(|V|) time loop to reduce all rows, O(|V|) time at each loop
        for row in range(self._length):
```

```
            # compute the min value in the row, O(|V|) time (don't
            # reduce if no min was found)
            min_val = math.inf
            for col in range(self._length):
                if self._values[row, col] < min_val:
                    min_val = self._values[row, col]
            if min_val == math.inf:
                continue

            # O(|V|) loop to decrement all values in the row by the min
            self._cost += min_val
            for col in range(self._length):
                self._values[row, col] -= min_val

        # O(|V|) time loop to reduce all columns, O(|V|) time at each loop
        for col in range(self._length):

            # compute the min value in the column, O(|V|) time (don't
            # reduce if no min was found)
            min_val = math.inf
            for row in range(self._length):
                if self._values[row, col] < min_val:
                    min_val = self._values[row, col]
            if min_val == math.inf:
                continue

            # O(|V|) loop to decrement all values in the column by the min
            self._cost += min_val
            for row in range(self._length):
                self._values[row, col] -= min_val

    ####### REGION: HELPER METHODS #######
    # all these methods are assumed to require consant time and space

    def get_cost(self) -> float:
        return self._cost

    def get_row_count(self) -> int:
        return self._length

    def get_col_count(self) -> int:
        return self._length

    def get_value_at(self, row, col) -> float:
        return self._values[row, col]

    def print(self, label="RCM"):
        print('{}: {}'.format(label, self.get_cost()))
        print(self._values)
        print('')

    ####### END REGION: HELPER METHODS #######
```

BranchNode (branch_bound/bnode.py)

```
from branch_bound.rcm import RCM
from TSPClasses import City
from copy import deepcopy
from typing import List
import math


# NOTE: for all code below, let |V| be the number of cities
class BranchNode:

    def __init__(self, rcm: RCM, city: City, city_index: int, parent):
        '''
        intializes this branch node based on the arguments passed
        \ntime:     O(1)
        \nspace:    assigned class variables by reference --> O(1)
        '''

        super().__init__()

        # use the parent node to obtain the depth of this node
        self._depth = 1 if parent == None else parent.get_depth() + 1
        self._rcm = rcm
        self._children = []
        self._city_index = city_index
        self._city = city
        self._parent = parent

    def __lt__(self, other):
        '''
        compares two nodes; used to break ties in the priority queue
        \ntime:     O(1)
        \nspace:    O(1)
        '''
        return self.get_cost() < other.get_cost()

    def compute_path(self) -> List[City]:
        '''
        recursively travel up the tree to compute the route from the given node
        \ntime:     with at most |V| cities --> O(|V|)
        \nspace:    allocates a route array --> O(|V|)
        '''

        # base case, reached the end of the route
        if self.get_parent() == None:
            return [self.get_city()]

        # recurse to the next parent node at most O(|V|) times
        parent = self.get_parent()
        path = parent.compute_path()
        path.append(self.get_city())
        return path

    def generate_child_nodes(self, cities: List[City]):
        '''
        for the city associated with this node, generate child nodes
        based on the cities this node's city has a valid path to
        \ntime:     O(|V|) loop with O(|V|^2) at each loop --> O(|V|^3)
        \nspace:    generate O(|V|) children costing O(|V|^2) each --> O(|V|^3)
        '''
```

```
        # based on this nodes current city (and the row in the RCM associated
        # with that city), loop through all neighboring cities --> O(|V|) time
        from_idx = self.get_city_index()
        for to_idx in range(self.get_rcm().get_col_count()):

            # ensure that there is a valid path to that city
            cost = self.get_rcm().get_value_at(from_idx, to_idx)
            if cost < math.inf:

                # O(|V|^2) space and assumed O(|V|^2) time to copy
                rcm = deepcopy(self.get_rcm())
                rcm.do_selection(from_idx, to_idx)  # O(|V|) time
                rcm.do_reduction()                  # O(|V|^2) time

                # create and add the child node with the values allocated --> O(1)
                child = BranchNode(rcm, cities[to_idx], to_idx, self)
                self._children.append(child)

    ####### REGION: HELPER METHODS #######
    # all these methods are assumed to require consant time and space

    def get_depth(self) -> int:
        return self._depth

    def get_city_index(self) -> int:
        return self._city_index

    def get_rcm(self) -> RCM:
        return self._rcm

    def get_parent(self):
        return self._parent

    def get_city(self) -> City:
        return self._city

    def get_cost(self) -> float:
        return self.get_rcm().get_cost()

    def get_children(self):
        return self._children

    def get_child_count(self) -> int:
        return len(self.get_children())

    ####### END REGION: HELPER METHODS #######
```

GreedySolver (greedy/solver.py)

```
from solver_base import SolverBase
from copy import copy
from random import randrange
import math


# NOTE: for all code below, let |V| be the number of cities
class GreedySolver(SolverBase):

    def __init__(self, tsp_solver, max_time):
        '''
        initialize this solver using all the defaults provided in solver base
        \ntime:     constant
        \nspace:    constant
        '''
        super().__init__(tsp_solver, max_time)

    def run_algorithm(self):
        '''
        repeatedly tries to greedily solve the TSP by calling 'self._greedy_solve'
        \ntime:     O(|V|) loop, with O(|V|^2) at each loop --> O(|V|^3)
        \nspace:    for the visted set and route array --> O(|V|)
        '''

        # pick a random city to start at
        start_index = randrange(self.get_city_count())

        # O(|V|) loop through all the cities and try to run the greedy algorithm
        for i in self.get_city_range():

            if self.exceeded_max_time():
                return

            # prepare for a new greedy traversal, O(|V|) space at most
            original = (start_index + i) % self.get_city_count()
            current = original
            visited = {current}
            route = [self.get_city_at(current)]

            # O(|V|^2) cost to solve greedily starting at the original node
            sol = self._greedy_solve(original, current, visited, route)
            if sol == None:
                continue

            # a greedy solution was found, update bssf
            self.set_bssf_from_route(sol)
            self.increment_solution_count()
            return

    def _greedy_solve(self, original, current, visited, route):
        '''
        a recursive definition to greedily attempt to find optimal routes
        \ntime:     recurses at most O(|V|) times, each costing O(|V|) --> O(|V|^2)
        \nspace:    visted set and route array take up at most O(2|V|) --> O(|V|)
        '''

        # base case: if all cities have been visited, try and close this route: O(1) time
        # (fails if there is no route back to the original city)
        if len(visited) == self.get_city_count():
            original_city = self.get_city_at(original)
```

```
            cost_to_original = self.get_city_at(current).costTo(original_city)
            return None if cost_to_original == math.inf else route

        # O(|V|) time to get the next city (see below)
        target = self._get_next_city(current, visited)
        if target == None:
            return None     # base case: failed to complete route
        else:
            # append the target node found and continue traversing; recurses
            # at most O(|V|) times (as a solution must obtain every city)
            visited.add(target)                     # assumed constant time, at most O(|V|) space
            route.append(self.get_city_at(target))  # assumed constant time, at most O(|V|) space
            return self._greedy_solve(original, target, visited, route)

    def _get_next_city(self, source, visited):
        '''
        gets the least expensive, unvisted neighboring city to the source node
        \ntime:     loops through all neighboring cities --> O(|V|)
        \nspace:    no significant allocations --> O(1)
        '''

        min_cost = math.inf
        min_index = None

        # O(|V|) loop to find the least expensive neighboring city
        for i in self.get_city_range():
            if i in visited:
                continue

            # constant time update for the cheapest
            # neighboring city not already visted
            target = self.get_city_at(i)
            cost_to_city = self.get_city_at(source).costTo(target)
            if cost_to_city < min_cost:
                min_cost = cost_to_city
                min_index = i

        # return the index of the neighboring least
        # expensive city to traverse to
        return min_index
```

SolverBase (solver_base.py)

```
from TSPClasses import City
from TSPClasses import Scenario
from TSPClasses import TSPSolution
from typing import List
from abc import abstractmethod
import time
import math


# a base class for all TSP solvers that helps abstract the logic
# for creating, returning, and updating solutions
class SolverBase:

    def __init__(self, tsp_solver, max_time):
        '''
        initialize references to helper variables
        \ntime:     constant
        \nspace:    constant
        '''
        super().__init__()

        # cache values that are used throughout
        # the different solvers
        self._results = {}
        self._tsp_solver = tsp_solver
        self._scenario = tsp_solver._scenario
        self._cities = self.get_scenario().getCities()
        self._city_count = len(self._cities)
        self._max_time = max_time
        self._start_time = None
        self._end_time = None
        self._intermediate_cnt = 0
        self._total = 0
        self._pruned = 0

        self.set_bssf(None)
        self.set_max_concurrent_nodes(None)

    def solve(self):
        '''
        runs and solves the TSP using some algorithm (based on whichever solver
        implements this class); thus, the time and space complexity will vary
        '''
        # time and run the algorithm
        self._start_time = time.time()
        self.run_algorithm()

        # update the results dictionary
        bssf = self.get_bssf()
        self._results['cost'] = bssf.cost if bssf != None else math.inf
        self._results['time'] = self.get_clamped_time()
        self._results['count'] = self._intermediate_cnt
        self._results['soln'] = bssf
        self._results['max'] = self.get_max_concurrent_nodes()
        self._results['total'] = self._total
        self._results['pruned'] = self._pruned

    @abstractmethod
    def run_algorithm(self):
        '''
```

```
        child classes will implement this method based on different ways of
        solving the TSP (time and space complexity will vary)
        '''
        pass

    ####### REGION: HELPER METHODS #######
    # all methods in this region are assumed to have constant time and space
    # complexity and will not be counted toward the total complexity of the
    # algorithms (these methods are self explanatory and just make the code
    # more readable)

    def get_max_time(self):
        return self._max_time

    def get_results(self):
        return self._results

    def get_tsp_solver(self):
        return self._tsp_solver

    def get_scenario(self) -> Scenario:
        return self._scenario

    def get_cities(self) -> List[City]:
        return self._cities

    def get_city_count(self) -> int:
        return self._city_count

    def get_city_at(self, index) -> City:
        return self.get_cities()[index]

    def get_city_range(self):
        return range(self.get_city_count())

    def set_bssf_from_route(self, route):
        self.set_bssf(TSPSolution(route))

    def get_best_cost(self) -> float:
        return math.inf if self.get_bssf() == None else self.get_bssf().cost

    def get_bssf(self) -> TSPSolution:
        return self._bssf

    def set_bssf(self, value):
        self._bssf = value

    def get_max_concurrent_nodes(self):
        return self._max

    def set_max_concurrent_nodes(self, value):
        self._max = value

    def increment_total(self, amount=1):
        self._total += amount

    def increment_pruned(self, amount=1):
        self._pruned += amount

    def increment_solution_count(self, amount=1):
        self._intermediate_cnt += amount

    def get_total_time(self):
        return time.time() - self._start_time
```

```
def get_clamped_time(self):
    return min(self.get_max_time(), self.get_total_time())

def exceeded_max_time(self):
    return self.get_total_time() > self.get_max_time()

def try_update_max_concurrent_nodes(self, new_value):
    updated_val = max(self.get_max_concurrent_nodes(), new_value)
    self.set_max_concurrent_nodes(updated_val)

####### END REGION: HELPER METHODS #######
```