tsp_solver.py

```
#!/usr/local/bin/python3

import copy
import time
from heapq import heappop, heappush, heapify
import numpy as np
from tsp_classes import TSPSolution


class TSPSolver:
    def __init__(self):
        self._scenario = None
        self._bssf = None
        self._count = 0
        self._current = None
        self._range = None

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

    def default_random_tour(self, start_time, time_allowance=60.0):
        '''
        <summary>
            This is the entry point for the default solver
            which just finds a valid random tour
        </summary>
        <returns>
            results array for GUI that contains three ints:
            cost of solution,
            time spent to find solution,
            number of solutions found during search (not counting initial BSSF
estimate)
        </returns>
        '''

        results = {}
        start_time = time.time()
        cities = self._scenario.get_cities()
        ncities = len(cities)
        found_tour = False
        count = 0

        while not found_tour:
            if time.time() > (start_time + time_allowance):
                results['cost'] = np.inf
                results['soln'] = None
                results['count'] = 0
                results['time'] = time.time()-start_time
                return

            # create a random permutation
            perm = np.random.permutation(ncities)
            to_visit = np.ones(ncities)
```

```
            # perm = np.arange(ncities)

            # print('\n\nNEW PERMUTATION: {}'.format(perm))
            failed = False

            for i in range(ncities):
                # print('In city {}'.format(i))
                src = perm[i]
                dest = perm[(i+1) % ncities]

                if self._scenario.edge_exists[src, dest]:
                    to_visit[dest] = False
                    continue
                elif i+1 == ncities:
                    # print('CAN\'T GET BACK TO START!')
                    failed = True  # can't get back to start so try a new
permutation
                    break

                else:  # try to swap the destination with a reachable one
                    # print('EDGE {}-->{} DOESN\'T EXIST'.format(src,dest))
                    reachable = np.nonzero(
                        self._scenario.edge_exists[src, :] * to_visit
                    )[0]

                    # print(reachable)

                    if np.sum(reachable > 0) == 0:
                        # print('DEAD END')
                        failed = True  # can't get back to start so try a new
permutation
                        break

                    swapind = reachable[np.random.randint(
                        np.sum(reachable > 0)
                    )]

                    # print('BEFORE: {}'.format(perm))
                    perm_loc_of_swapind = np.nonzero(perm == swapind)
                    perm[(i+1) % ncities] = perm[perm_loc_of_swapind]
                    perm[perm_loc_of_swapind] = dest
                    to_visit[swapind] = False
                    # to_visit[swapind] = False
                    # print('AFTER: {}'.format(perm))
                    # print('REACHABLE: {}, picked {}'.format(reachable,swapind))

            if failed:
                # print('Trying a new permutation')
                continue  # try a new permutation

            route = []

            # Now build the route using the random permutation
            for i in range(ncities):
                route.append(cities[perm[i]])

            bssf = TSPSolution(route)
```

```
            # bssf_cost = bssf.cost()
            # count++;
            count += 1

            # if costOfBssf() < float('inf'):
            if bssf.cost_of_route() < np.inf:
                # Found a valid route
                found_tour = True

        #} while (costOfBssf() == double.PositiveInfinity); // until a valid route
is found
        # timer.Stop();
        # costOfBssf().ToString(); // load results array
        results['cost'] = bssf.cost_of_route()
        results['time'] = time.time() - start_time
        results['count'] = count
        results['soln'] = bssf

       # return results;
        return results

    def greedy(self, start_time, time_allowance=60.0):
        cities = self._scenario.get_cities()
        length = len(cities)
        bssf = self.default_random_tour(cities)['soln']
        current_time = time.time() - start_time
        count = 0

        for city in cities:
            if current_time >= time_allowance:
                break

            remaining_cities = copy.copy(cities)
            remaining_cities.remove(city)
            path = [city]
            src = city

            while remaining_cities and current_time < time_allowance:
                closest_city = min(remaining_cities, key=src.cost_to)

                if src.cost_to(closest_city) == np.inf:
                    break

                remaining_cities.remove(closest_city)
                path.append(closest_city)
                src = closest_city
                current_time = time.time() - start_time

            if len(path) == length:
                solution = TSPSolution(path)
                count += 1

                if solution.cost_of_route() < bssf.cost_of_route():
                    bssf = solution

            current_time = time.time() - start_time
```

```python
        return {
            'cost': bssf.cost_of_route(),
            'time': time.time() - start_time,
            'count': count,
            'soln': bssf
        }

    def branch_and_bound(self, start_time, time_allowance=60.0):
        current_time = time.time() - start_time

        # set helpful length variables
        cities = np.array(self._scenario.get_cities())
        self._range = range(len(cities))
        length = len(cities)

        # init containers
        self._bssf = self.default_random_tour(cities)['soln']
        bssf_updates = 0
        pq = []

        # initialize elements to start from the first node
        cost = 0
        count = 0
        current_lower_bound = 0
        cur_index = 0
        pruned = 0
        created = 0
        path = [cur_index]
        rcm = self._init_matrix(cities)

        # get reduced cost matrix
        rcm, current_lower_bound = self._get_rcm(
            rcm,
            current_lower_bound
        )

        heappush(pq, (cost, (path, rcm, cur_index, current_lower_bound)))
        max_states = 1

        while current_time < time_allowance and pq:
            current_time = time.time() - start_time
            current = heappop(pq)
            cur_path = current[1][0]
            cur_rcm = current[1][1]
            cur_index = current[1][2]
            cur_lb = current[1][3]
            cur_cost = current[0]

            if len(cur_path) == length + 1:
                # found solution
                count += 1

                if cur_cost < self._bssf.cost_of_route():
                    # solution is new bssf
                    cur_path.pop()
                    city = []
```

```
                for x in cur_path:
                    city.append(cities[x])

                new_sol = TSPSolution(city)
                self._bssf = new_sol
                bssf_updates += 1

        for i in self._range:
            if i != cur_index:
                # use updated cost from rcm
                cost_to = cur_rcm[cur_index][i]

                child_rcm = self._select_path(
                    cur_rcm,
                    cur_index,
                    i
                )

                child_rcm, child_lb = self._get_rcm(
                    child_rcm,
                    cur_lb
                )

                total_cost = cost_to + child_lb

                if total_cost < self._bssf.cost_of_route():
                    # path could lead to next bssf
                    child_path = cur_path.copy()
                    child_path.append(i)

                    heappush(
                        pq,
                        (total_cost, (child_path, child_rcm, i, total_cost))
                    )

                    if len(pq) > max_states:
                        max_states = len(pq)

                    created += 1
                else:
                    pruned += 1

    print(max_states)
    print(bssf_updates)
    print(created)
    print(pruned)

    return {
        'cost': self._bssf.cost_of_route(),
        'time': time.time() - start_time,
        'count': count,
        'soln': self._bssf
    }

def _select_path(self, rcm, src_index, dest_index, stop_backtrack=True):
    new_matrix = np.copy(rcm)
```

```
        for i in self._range:
            # set row to inf
            new_matrix[src_index][i] = np.inf

            # set col to inf
            new_matrix[i][dest_index] = np.inf

        # can't backtrack
        if stop_backtrack:
            new_matrix[dest_index][src_index] = np.inf

        return new_matrix

    # O(n(n - 1)!)
    def fancy(self, start_time, time_allowance=60.0):
        cities = self._scenario.get_cities()
        length = len(cities)
        bssf = self.default_random_tour(cities)['soln']
        count = 0

        self._range = range(length)
        matrix = self._init_matrix(cities)  # O(n^2)
        rcm, _ = self._get_rcm(matrix)  # O(n^2)

        # O(n^3)
        solution_matrix, covered_rows, covered_cols, n_lines = self._cross_zeros(
            rcm
        )

        current_time = time.time() - start_time

        # O(n^4)
        while n_lines < length and current_time < time_allowance:
            uncovered_min = np.min(solution_matrix)

            for i in self._range:
                for j in self._range:
                    if i not in covered_rows:
                        rcm[i][j] -= uncovered_min

                    if j in covered_cols:
                        rcm[i][j] += uncovered_min

            # O(n^3)
            solution_matrix, covered_rows, covered_cols, n_lines =
self._cross_zeros(
                rcm
            )

            current_time = time.time() - start_time

        path = [0]
        minimum = 0
        solution = None

        # O(n(n - 1)!)
        while not solution and current_time < time_allowance:
```

```
        # O(n(n - 1)!)
        if self._found_tour(rcm, 0, minimum, path, 1):
            solution = list(map(lambda x: cities[x], path))
        else:
            # O(n^2)
            minimum = self._find_next(rcm, minimum)
            current_time = time.time() - start_time

    if solution:
        bssf = TSPSolution(solution)
        count += 1

    return {
        'cost': bssf.cost_of_route(),
        'time': time.time() - start_time,
        'count': count,
        'soln': bssf
    }

# O(n^2), nested for loops
def _init_matrix(self, cities):
    length = len(cities)
    matrix = np.full((length, length), np.inf)

    for i in range(length):
        for j in range(length):
            if i != j:
                # set cost from each city to each other city
                matrix[i][j] = cities[i].cost_to(cities[j])

    return matrix

# O(n^2), nested for loops
def _get_rcm(self, matrix, lower_bound=0):
    # O(n^2)
    for row in range(matrix.shape[0]):
        mini = min(matrix[row, :])

        if mini != np.inf:
            lower_bound += mini
            matrix[row, :] -= mini

    # O(n^2)
    for col in range(matrix.shape[1]):
        mini = min(matrix[:, col])

        if mini != np.inf:
            lower_bound += mini
            matrix[:, col] -= mini

    return matrix, lower_bound

# This returns:
#  a solution matrix with the min number of lines used to cross out the zeros
#  A list of the indices of the covered rows
#  A list of the indices of the covered col
# O(n^3) because it crosses out up to n-lines within O(n^2) loops.
```

```
    def _cross_zeros(self, rcm):
        solution_matrix = rcm.copy()
        covered_rows = []
        covered_cols = []

        while np.isin(0, solution_matrix):
            # Checking rows
            for i in self._range:
                zero_count = 0
                zero_index = 0

                for j in self._range:
                    if solution_matrix[i][j] == 0:
                        zero_count += 1
                        zero_index = j

                if zero_count == 1:
                    solution_matrix[:, zero_index] = np.inf
                    covered_cols.append(zero_index)

            # Checking cols
            for j in self._range:
                zero_count = 0
                zero_index = 0

                for i in self._range:
                    if solution_matrix[i][j] == 0:
                        zero_count += 1
                        zero_index = i

                if zero_count == 1:
                    solution_matrix[zero_index, :] = np.inf
                    covered_rows.append(zero_index)

        n_lines = len(covered_cols) + len(covered_rows)
        return solution_matrix, covered_rows, covered_cols, n_lines

    # Worst case O(n((n - 1)!)), but optimized because it cuts after
    # finding a solution path.
    def _found_tour(self, rcm, row, minimum, path, length):
        sorted_row = list(rcm[row, :])
        sorted_row = enumerate(sorted_row)
        sorted_row = list(map(lambda t: (t[1], t[0]), sorted_row))
        heapify(sorted_row)

        while sorted_row:
            j = heappop(sorted_row)

            if j[0] <= minimum:
                if j[1] == 0 and length == len(self._range):
                    return True
                elif j[1] not in path:
                    path.append(j[1])
                    length += 1

                    if not self._found_tour(rcm, j[1], minimum, path, length):
                        del path[-1]
```

```
                    length -= 1
                else:
                    return True

        return False

    # O(n^2), nested for loops
    def _find_next(self, rcm, minimum):
        low = np.inf

        for i in self._range:
            for j in self._range:
                if low > rcm[i][j] and rcm[i][j] > minimum:
                    low = rcm[i][j]

        return low
```