Amazon delivery truck scheduling

Jiaxin Li (jl69925)

December 1, 2019

1 Introduction

This C++ project aims to simulate the scheduling of delivery truck in different scenarios. Basically, the code tries to solve the TSP problem - minimizing the total distance that the truck has to travel.
There are two classes (Address, Route) implemented in this code, and each of them has member functions within. The greedy algorithm and the Kernighan-Lin algorithm are applied while constructing the delivery route. The code is tested on both single truck and multiple trucks scenarios.

2 Class Address

Class Address describes the location, that is, the x and y coordinates of each delivery address.
1. Values of x and y are passed into the Address through the constructor.
2. The member function distance() calculates the Euclidean distance between two addresses. To call the function, use point "." operator. E.g. Address a.distance(Address b).
3. Variables x and y are stored privately in this class. Their value can be got outside the class from function the_x() and the_y() where the values are returned.

```
class Address{
private:
  double x,y;
public:
  Address(double x,double y)                // constructor, passing in value of x and y
    : x(x),y(y) {};
  double distance(Address temp){            // calculating distance between two Address points
    double m=temp.x;
    double n=temp.y;
    return sqrt((m-x)*(m-x)+(n-y)*(n-y));
  };
  double the_x(){return x;};                // return the value of x
  double the_y(){return y;};                // return the value of y
};
```

Figure 1: Class Address.

3 Class Route

Class Route collects a list of addresses and stores them into the vector list. Each route stands for a way to travel through the delivery route.
1. Each route starts from (0,0) and ends at (0,0) again. Thus each route is initialized with start and end point as Address(0,0).
2. The function add_address() adds new address point to the address list. Call function insert(address_list.end()-1,a) to add the point to the end of the list but keep the (0,0) still as the end point.
3. The length() function calculates the length to travel through the address list in order. For each point, call distance function to calculate the distance between the point and the point next to it in the list. Add the distance together to get the length of the route.
4. The size() function returns the size of the address list.
5. The as_string() function prints out the addresses in the route one by one. To get the coordinates of x and y for each address, call the_x() or the_y().

```
class Route{
private:
  vector <Address> address_list;                  // the list of addresses
public:
  Route()                                         // constructor of an address list
    :address_list(2,Address(0,0)){};

  void add_address(Address a){                    // insert new address into the list
    address_list.insert(address_list.end()-1,a);
  }

  double length(){                                // calculate the length to travel the list in order
    double len=0.;
    for(int i=0;i<address_list.size()-1;i++){
      Address p=address_list.at(i);
      Address q=address_list.at(i+1);
      double dist_temp=p.distance(q);
      len+=dist_temp;
    }
    return len;
  };

  int size(){                                     // return the size of the list
    return address_list.size();
  };

  void as_string(){                               // print the list as string
    for(int i=0;i<address_list.size();i++){
      Address a = address_list.at(i);
      cout<<"("<<a.the_x()<<","<<a.the_y()<<") ";
    }
    cout<<endl;
  };
```

Figure 2: Class Route.

4 Greedy Route

The greedy_route() function returns a new Route constructed from greedy algorithm. For each time, a new point that is closest to the previous one is added to the list. Note that the return type should be Route so that the route can call the function length() or size(). And the greedy algorithm does not promise the optimal route.

To construct the new route, a new list greedy_list should be initialized with start and end point (0,0). Points from original list will be added to this list. Copy list gets all the address points from the original list address_list, and erases the start and end points.
Now starts from the begin point of the greedy_list, we loop through the copy_list to find the address that is closest to the current point, add the point into the greedy_list and erase it from the copy_list. Keep adding points in this way until the copy_list is empty, which means all of the points are added to the new list.

```
Route greedy_route(){
  Route greedy;                                          // create a new Route, greedy
  vector<Address> greedy_list=greedy.address_list;       // the greedy_list, used to construct new list
  vector<Address> copy_list=address_list;                // copy the original list
  copy_list.erase(copy_list.begin());                    // erase the begin and end point
  copy_list.erase(copy_list.end());

  Address we_are_here=greedy_list.at(0);                 // the current point
  while(!copy_list.empty()){                             // loop until the copy_list is empty
    int target=0;
    double dist_temp=copy_list.at(0).distance(we_are_here);
    for(int i=1;i<copy_list.size();i++){
      if(copy_list.at(i).distance(we_are_here)<dist_temp){  // find the point closest to current point
        target=i;                                        // keep its index
      }
    }
    we_are_here=copy_list.at(target);                    // update current point
    greedy_list.insert(greedy_list.end()-1,we_are_here);  // insert the point into the new list
    copy_list.erase(copy_list.begin()+target);           // erase the point from copy list
  }
  greedy.address_list=greedy_list;                       // update the list in Route greedy
  return greedy;                                         // return the new Route
};
```

Figure 3: Greedy Route.

5 Reverse Route

The reverse_route() function returns a new Route constructed from Kernighan-Lin algorithm. Different from the greedy route above, this algorithm aims to solve the crossed route by reversing it. Once a crossed path is find in the original route, it is not hard to tell that reversing the crossed path will give a shorter length.
To find the crossed path, try to reverse all the sublists from the original list. The temp_list here is used to store these attempts. If the length becomes shorter, then update the reverse_list.
The time complexity of the algorithm is O(n^3), since it uses O(n^2) to find each sublist, and for each list, the reverse of the list takes O(n).

6 Test Results

The greedy() and reverse_route() functions are tested on the cases below.
1. The original shows the original route and the distance to travel all the addresses in the list in order.
2. The greedy algorithm reconstruct the list, and gets a better result in distance.
3. The reverse algorithm solve the crossed path in the original route, which also yields a shorter route.

```
Route reverse_route(){
  Route reverse;                                          // create a new Route, reverse
  reverse.address_list=address_list;                      // initialize the reverse route
  for(int i=1;i<address_list.size()-2;i++){               // for each sublist
    for(int j=i+1;j<address_list.size()-1;j++){
      vector<Address> temp_list=address_list;             // create a temp_list to store reverse list
      int left=i;
      int right=j;
      while(left<=j){                                     // reverse the sublist
        temp_list.at(left)=address_list.at(right);
        left++;
        right--;
      }
      Route temp;                                         // create a new Route, temp
      temp.address_list=temp_list;
      if(temp.length()<reverse.length()){                 // if the temp_list is shorter, update reverse
        reverse.address_list=temp_list;
      }
    }
  }
  return reverse;                                         // return the Route reverse
};
```

Figure 4: Reverse Route.

4. The reverse algorithm is also tested on greedy route constructed in step2. Here we try to keep reversing the crossed path from previous route until the distance became stable. From the results, we can tell that the final route shows the best performance in the tests above.

```
Single Truck!
Original!
39.9275    (0,0) (0,5) (2,3) (5,0) (1,1) (3,0) (4,4) (2,0) (5,5) (0,0)
Greedy!
30.0351    (0,0) (2,0) (4,4) (5,5) (2,3) (1,1) (3,0) (5,0) (0,5) (0,0)
Original Reverse!
31.7985    (0,0) (0,5) (2,3) (5,0) (1,1) (3,0) (4,4) (5,5) (2,0) (0,0)
Greedy Reverse!
25.615   (0,0) (2,0) (5,0) (3,0) (1,1) (2,3) (5,5) (4,4) (0,5) (0,0)
25.1224   (0,0) (2,0) (5,0) (3,0) (1,1) (2,3) (4,4) (5,5) (0,5) (0,0)
25.0095   (0,0) (2,0) (3,0) (5,0) (1,1) (2,3) (4,4) (5,5) (0,5) (0,0)
24.3716   (0,0) (2,0) (3,0) (5,0) (0,5) (5,5) (4,4) (2,3) (1,1) (0,0)
22.016   (0,0) (2,0) (3,0) (5,0) (4,4) (5,5) (0,5) (2,3) (1,1) (0,0)
22.016   (0,0) (1,1) (2,3) (0,5) (5,5) (4,4) (5,0) (3,0) (2,0) (0,0)
```

Figure 5: Results for Single Truck.

7 Multiple Trucks

To simulate scenarios like multiple trucks or a single truck on different days, two routes should be initialized here. To shorten the sum of distance of the two routes, try to switch the sublists of the two routes from i to j. If shorter combinations found, then update the route.

Note that the Route here are passed into the function as reference. This ensures that the route changed can be get outside the function. If we don't want to change the original

route passed in, and we want to return two new routes from the function, then a struct with two Routes within can be used here.

```
void switch_route(Route &path1, Route &path2){        // passed as reference
  vector<Address> rem1=path1.address_list;            // remember the original lists
  vector<Address> rem2=path2.address_list;

  for(int i=1;i<path1.address_list.size()-2;i++){     // switch sublists from i to j
    for(int j=i;j<path1.address_list.size()-1;j++){
      vector<Address> one=rem1;
      vector<Address> two=rem2;
      for(int k=i;k<=j;k++){
        one.at(k)=path2.address_list.at(k);
        two.at(k)=path1.address_list.at(k);
      }

      Route temp_one;
      Route temp_two;
      temp_one.address_list=one;
      temp_two.address_list=two;
      temp_one.as_string();
      temp_two.as_string();
      if(temp_one.length()+temp_two.length()<path1.length()+path2.length()){
        path1.address_list=one;                       // if shorter,then update
        path2.address_list=two;
      }
    }
  }
};
```

Figure 6: Switch Route.

The code is tested in the case below. From the results we can tell that the switch_route function shorten the sum distance successfully.

```
Multiple Truck!
19.6251    (0,0) (2,0) (3,2) (2,3) (0,2) (0,0)
(0,0) (3,1) (2,1) (1,2) (1,3) (0,0)
17.153    (0,0) (2,0) (2,1) (1,2) (0,2) (0,0)
(0,0) (3,1) (3,2) (2,3) (1,3) (0,0)
```

Figure 7: Results for Multiple Trucks.

8 Other Scenarios

To implement the scenario when there are addresses cannot be switched, Class Address should be updated with both location and the route it belongs to. When trying to switch the routes, check firstly if the sublists contain the addresses that cannot be switched. When the ratio of prime to non-prime addresses becomes larger, the distance of the two routes will be longer.

In the scenario where every day a random number of new deliveries is added to the list, the size of the two routes can be different. Then when switching the sublists, we need to check the i and j index first to ensure that the points between i and j are valid.