International Institute of Information Technology, Bangalore

Project Report

Dial A Ride

Author:
Arjun S Bharadwaj

Supervisor:
Dr. Muralidhara V. N.

A report submitted in fulfilment of the requirements of the project

in the course

Data Structures and Algorithms

November 2013

Chapter 1

Dial A Ride

1.1 Problem Statement

In public intracity transport system (like BMTC in Bangalore), the pick-up and drop-off lo- cations are determined by the service provider, not by the passenger. On the other hand, a taxi (like merucabs, easy cabs in Bangalore) is a small vechile with a driver, used by individual (or a small group of people) to travel between locations of their choice. The taxi service is the most comfortable, but the costliest mode of transport, where as the public/bus system is the cheapest, but least comfortable. Dial-a-ride is a form of public transport system which tries to achieve the comfort level of a taxi service at low cost. It is characterized by flexible routing and scheduling of small vehicles operating in shared-ride mode (there will be others who will share the vechile with you during the journey) between pick-up and drop-off locations as requested by passengers. There are many variants of this problem, we will consider one such (simple) variant for this project.
A typical passenger request in Dial a ride problem will look like,
(ElectronicCity, InternationalAirport, (10 : 40AM, 11 : 15AM))
meaning the passenger wants to go from Electronic City to Bangalore International Airport(BIA) and he would like to be picked up in Electronic city between 10:40 AM and 11:15 AM. Such requests are generally registered though telephone or Internet. It is not necessary that vechile will take the shortest route, it might deviate from the shortest route in order to pick some other passenger, but the passenger will be charged based on the shortest distance between the locations that he is travelling. The main objective of the project is to schedule the cabs, in such a way as to maximize the revenue. You are given n(100) main locations of the city and distances to from each location to some of its neighboring locations. You can compute the shortest distance from this data (you

may assume that every location is reachable from every other location). The amount of money that a passenger will pay to go from location A to B is proportional to the shortest distance from A to B. You can assume that the base rate is Rs.1 per KM.
You are given N(250) vehicles, each of capacity c(5). You can assume that the average speed of these vehicle is 2 minutes per KM. You are also given the location of these vehicles at midnight. You are given R(5000) passenger requests. A request is of the form A, B, t1, t2 implying that the passenger would like to picked up at location A between time t1 and t2, and he should be dropped at location B. For simplicity, you can assume that all the locations are integers between 1 and n and the time is in minutes between 0 and 1440, with midnight as reference point. 10 : 40AM will be noted as 640. You have to write a program to schedule the N vehicles in such a way as to maximize the revenue. You can decide to reject a passenger request (it may not be possible to meet all the requests).
The input will be given in the following format
nNcR
n * n matrix indicating the distances to nearest locations.
A sequence of N locations - indicating where the vehicle is at midnight.
A sequence of R requests.
Expected output of the program is schedule for each vehicle and the total revenue generated.

1.2 Algorithm Overview

A Greedy Event based Algorithm is employed to solve the problem of Dial A Ride. In the initial setup, the user requests are treated as events and are added to a Priority Queue. The elements in priority queue are later removed based on the timestamp of the arrival requests. For every request, the algorithm checks whether a free cab is available in the node to serve the request at that time interval. If a cab is found in the above search, it is allocated to satisfy the request and intermediate nodes that bare part of the shortest path computed from Dijkstra's algorithm are updated with the arrival time of the cab.
If a cab isn't available at the node, a search is made in the neighboring nodes. If a cab is found, it is checked whether it's already serving another request or not. If it's not serving another request, then the cab is routed to the pickup location and the intermediate nodes is updated with the arrival times of the cab.
However, if the cab is already serving and the current request is a sub path of the cab's path or if the cab's path is the sub path of the cab's path, then the cab is rerouted to pick up the customer and the version of the cab is incremented and the new route is

computed to serve all the requests and the corresponding nodes are updated.
The visit of each node, both intermediate and final node is added as an event in the priority queue. Whenever a destination node is reached an event will be fired and the passenger count is decreased in the corresponding cab. Whenever a pickup node is reached, the passenger count is increased.
This process is continued until the priority queue is not empty. The advantage with this algorithm is it takes the requests in real time and processes it.
An additional optimization to maximize the revenue is done on this algorithm by dropping off the requests that generate very low revenue. This optimization selectively adds the requests and maximizes the revenue.

1.3 Data Structures used in implementation

The following data structures are used in the implementation:

- Priority Queue
- Interval Tree - based on left leaning Red Black Tree
- Hash Map
- Lists - Array List
- Sets - Hash Set
- Adjacency List

1.4 Algorithm

```
public class CabScheduler {

  public void scheduleDialARide(DialARideModel model, int thresholdValue) {
    int successfulScheduling = 0;
    int rejectedReqs = 0;
    int revenue = 0;

    Queue<Event> eventQueue = new PriorityQueue<Event>();
    SortedMap<Integer, List<RideRequest>> rideRequests = model
        .getRideRequests();

    for (RideRequest rideRequest : rideRequests) {
      eventQueue.add(new Event(EventTypes.CUSTOMER_REQUEST));
    }

    computeAllNodesShortestPaths();

    while (!eventQueue.isEmpty()) {
      Event event = eventQueue.poll();

      switch (event.getEventType()) {
      case CUSTOMER_REQUEST:
        boolean found = findACab(model, event.getRideRequest(), eventQueue,
            (int) event.getTimeInstant());
        if (found) {
          successfulScheduling += 1;
          revenue += model.getCost()[src][dest];
        }
        break;
      case REACHED_INTERMEDIATE_NODE:
        if (event.getVersion() == event.getCab().getVersion()) {
          handleReachedIntermediateNode(model, event);
        }
        break;
      case REACHED_PICKUP_NODE:
        handleReachedPickupNode(model, event);
        break;
      case REACHED_PARTIAL_TARGET_NODE:
        if (event.getVersion() == event.getCab().getVersion()) {
          handleReachedTargetNode(model, event, true);
        }
        break;
      case REACHED_TARGET_NODE:
        if (event.getVersion() == event.getCab().getVersion()) {
          handleReachedTargetNode(model, event, false);
        }
```

```
        break;
      default:
        break;
      }
    }
    model.setSuccessfullyScheduledRequests(successfulScheduling);
    model.setMaxRevenue(revenue);
    model.setTotalDistance(totalTravelled);
  }

  private void handleReachedTargetNode(DialARideModel model, Event event,
      boolean isPartial) {
    Cab cab = event.getCab();
    Node node = model.getNodes().get(event.getNodeNumber());
    IntervalTree<Interval> cabsSet = node.getCabsSet();
    cabsSet
        .delete(new Interval(event.getTimeInstant(), event.getTimeInstant()
    ));

    if (!isPartial) {
      cab.setDroppingAnyone(false);
      cabsSet.add(new Interval(event.getTimeInstant(), END_OF_DAY, cab));
      cab.setPassengers(0);
    } else {
      cab.setPassengers(cab.getPassengers() - event.getDropCounts());
    }
  }

  private void handleReachedIntermediateNode(DialARideModel model, Event
    event) {
    Cab cab = event.getCab();
    Node node = model.getNodes().get(event.getNodeNumber());
    IntervalTree<Interval> cabsSet = node.getCabsSet();
    cabsSet
        .delete(new Interval(event.getTimeInstant(), event.getTimeInstant()
    ));
    cab.getPath().remove(0);
  }

  private void handleReachedPickupNode(DialARideModel model, Event event) {
    Cab cab = event.getCab();
    cab.setPickingAnyone(false);
    Node node = model.getNodes().get(event.getNodeNumber());
    IntervalTree<Interval> cabsSet = node.getCabsSet();
    cabsSet
        .delete(new Interval(event.getTimeInstant(), event.getTimeInstant()
    ));
  }
```

```
  private boolean findACab(DialARideModel model, RideRequest rideRequest,
      Queue<Event> eventQueue, int now) {
    boolean found = false;
    int source = rideRequest.getSource();
    double maxDistOfCab = (rideRequest.getTimeEnd() - rideRequest
        .getTimeStart()) / SchedulerConstants.MINUTES_PER_KM;
    List<Node> nodes = model.getNodes();
    nodeQueue.add(nodes.get(source));
    searchedNodes.add(source);

    while (!nodeQueue.isEmpty()) {
      Node node = nodeQueue.remove();
      IntervalTree<Interval> cabsSet = node.getCabsSet();
      Interval searchInterval = new Interval(rideRequest.getTimeStart(),
          rideRequest.getTimeEnd());
      Iterable<Interval> intervals = cabsSet.keys(searchInterval);

      for (Interval interval : intervals) {
        Cab cab = interval.getCab();
        boolean feasibility = isTheSchedulingFeasibleForTheCab(cab);

        if (feasibility) {
          if (!cab.isPickingAnyone()
              && (cab.getCapacity() > cab.getPassengers())) {
            found = true;
            double timeInstant = rideRequest.getTimeStart();
            if (cab.isDroppingAnyone()) {
              found = checkIfPathIsSubPathAndScheduleARide(cab, model);
              if (found) {
                cab.setPassengers(cab.getPassengers() + 1);
              }
            } else {
              if (node.getNodeNumber() != source) {
                timeInstant = scheduleAPickUpFromNeighboringNode(cab, model
  );
              }
              scheduleARide(cab, model, interval);
              cab.setPassengers(cab.getPassengers() + 1);
            }
          }
        }
      }
    }
    return found;
  }
}
```

Bibliography

[1] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 1990.

[2] Robert Sedgewick. Event driven simulation.

[3] Robert Sedgewick and Kevin Wayne. Algorithms, 4th edition.