**Raptor Algorithm:**

**Problem definition:**

We need to find the fasted path (the path with less time for traveling) from one station to one station go through some bus routes with some additional conditions: departure time, number of transfers.

**Introduction:**

This problem can be solved by using Raptor algorithm. Raptor algorithm is an algorithm based on paper named “Round-Based Public Transit Routing”. This paper is published by researchers Daniel Delling, Thomas Pajor and Renato F. Werneck on 2012.

Above problem can be known as computing all Pareto-optimal journeys in a dynamic public transit network for multiple criteria such as departure time and number of transfers.

Existing algorithms consider this as a graph problem, and solve it using variants of Dijkstra algorithm such as Layered Dijkstra (2004), multi-label-correcting algorithm (2008). Unfortunately, those algorithms lead to either high query times or suboptimal solutions by at least a logarithmic factor, due to the priority queues property of Dijkstra algorithm.

Starting from arrival time and number of transfers as criteria, this algorithm can be extended to handle more arbitrary additional criteria such as arrival time, ticket price, change time at each station or reliability of transfers.

**Attribute Definition:**

In below algorithm explanation, we use some notations:

* maximum bus transfers for this search. We can choose this dynamically when searching.
* is set of city’s stations.
* is set of city’s routes.
* is set of trips of all routes. And is set of trips of route . In our algorithm, we will sort those trips of each route based on departure time of each trip for searching optimization.
* start station.
* end station. Note that in our algorithm, there are many end stations.
* earliest arrival time that we can come to this station using at most k transfers. Our algorithm will optimize those set of values. Note that two points:

1. This time is real time base on departure time (ie: 5:30 PM, 8:15 AM)
2. The total transfers can be any from to . If value equals 0, it means we cannot reach this station on time by using any routes.

* earliest arrival time to this station. After loops finish,
* be the earliest trip in route r that one can catch at station
* be the departure time of trip t at . Note that, currently, our bus system, arrival time and departure time of trip t at are equals. That means no wait time at each station of each trip.

We also have some variables which means:

**Algorithm Summary:**

The algorithm works in rounds. Round k computes the fastest way of getting to every station with at most transfers (for example by taking at most k trips). Note that some stations may not reachable at all because not enough transfers for reach to this station. WLOG, we can assume the number of rounds is K. (which can be dynamically extended during the algorithm, if necessary).

More precisely, the algorithm associates with each station p an array of values (, , , …, ). All values in all labels are initialized to. We then initialize . , with is departure time.

We maintain the following invariant: at the beginning of round k (for ), the first k entries in (from to ) are correct (because we have optimized at last k loop). The remaining entries still set to. The goal of round k is to compute for all station p in system. It does so in two main stages.

The first stage of round k:

* we initialize value for by sets for all stations p; This sets an upper bound on earliest arrival time at p with at most k trips. We will optimize this value in this round on second stage.

The second stage of round k:

* We will process each route in system exactly once. Consider a route r, and let be the sequence of trips of route r in this day sorting base on departure time of trip. When processing route r, we consider journeys where the last trip taken is in route r. We will find value by using following formula: we navigate all sorted trips of this route and find the first trip t such that . This formula means: departure time at station p of trip t must be later than earliest arrival time at p. Note that this trip may not exist due to no trips exist anymore at that time. In that case, . Otherwise, we assign found trip to .
* To process the route, we visit its stations in order until we find a stations such that is defined. This is when we can improve . Let the corresponding trip t be the current trip we have found. We keep traversing the route of this trip. For each subsequent station , we can update using this trip.
* To reconstruct the journey, we set a parent pointer to the station at which t was boarded. Moreover, we may need to update the current trip for k: At each stop pi along r it may be possible to catch an earlier trip (because a quicker path to pi has been found in a previous round). Thus, we have to check if and update trip t by recomputing .

Base on above core algorithm, we have four improvements:

* At each round , traversing all routes in system that cannot be reached by previous round seems wasteful since there is no way to improve to any trips go to those route. For improving this algorithm, at round , we mark all stations that have updated. On next round k, we just list all routes that go through those stations.
* Base on above enhancement and when we traverse along those routes, we continue to update trip go through those stations. So we only need to traverse from very first updated station of this route so we can save an amount of time for traversal one route.
* For avoiding copying value (need one additional loop for each station), we can use for keep track of earliest possible time to get to station because equals to when ending round . So with this improvement, first stage of algorithm can be avoiding.
* Raptor algorithm computes all journeys to all stations in network but we are only interest in journeys with some target stations . So during round k, there is no need to mark stations whose arrival times are greater than (the best arrival time at ). So we can avoid many stations for next round.

We have an image for algorithm demonstration:

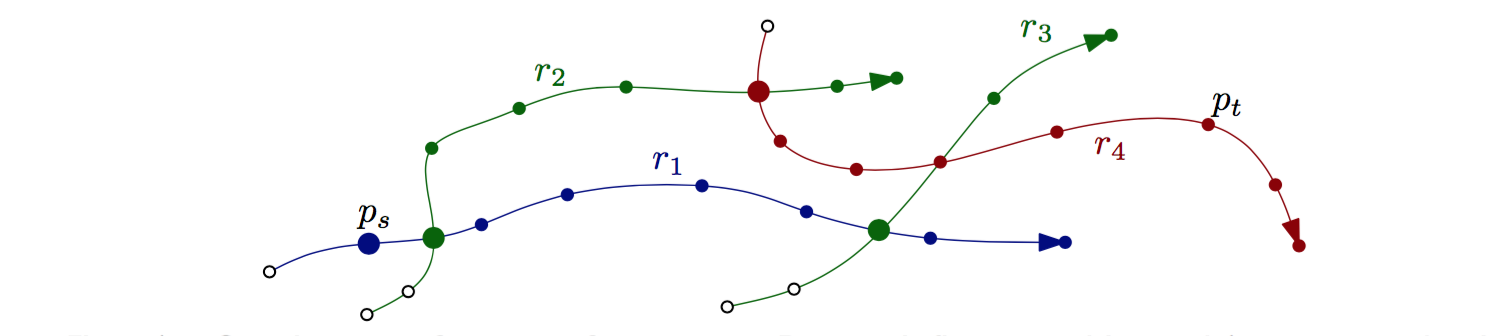


Figure: Scanning routes for a query from to . Route is first scanned in round 1, routes and in round 2, and finally, route in round 3. Scanning a route begins at the earliest marked stations (bold). Hollow stations are never visited.

**Solution:**

Initialize:

Solve:

Build again result:

**Algorithm Complexity:**

The worst-case running time of our algorithm can be bounded as follows:

In every round, algorithm scans each route at most once. If is the number of stations of route , then we look at stations in total to process the route. For each station , we must find the earliest trip. If we keep the list of trips serving r sorted by time, while traversing r we can find all values with a single sweep over this list, since can only decrease. In total, RAPTOR takes time for each round. Note that the running time per round is potentially sublinear in the size of the input.

So in total, algorithm complexity is: )