**Raptor Algorithm:**

**Problem definition:**

**Attribute Definition:**

**Algorithm Summary:**

The algorithm works in rounds. Round k computes the fastest way of getting to every stop with at most k 1 transfers (i. e., by taking at most k trips). Note that some stops may not reachable at all because not enough transfers for reach to this station.. To explain the algorithm, we bound the number of rounds by K (which can be dynamically extended during the algorithm, if necessary). More precisely, the algorithm associates with each stop p a multilabel (⌧0(p),⌧1(p),...,⌧K(p)), where ⌧i(p) represents the earliest known arrival time at p with up to i trips. All values in all labels are initialized to 1. We then set ⌧0(ps) = ⌧. We maintain the following invariant: at the beginning of round k (for k 1), the first k entries in ⌧(p) (from ⌧0(p) to ⌧k1(p)) are correct, i.e., entry ⌧i(p) represents the earliest arrival time at p using at most i trips. The remaining entries are set to 1. The goal of round k is to compute ⌧k(p) for all p. It does so in three stages.

The first stage of round k sets ⌧k(p) = ⌧k1(p) for all stops p; this sets an upper bound on the earliest arrival time at p with at most k trips.

The second stage then processes each route in the timetable exactly once. Consider a route r, and let T (r) = (t0, t1, . . . , t|T (r)|1) be the sequence of trips that follow route r, from earliest to latest. When processing route r, we consider journeys where the last (k-th) trip taken is in route r. Recall that ⌧ch(pi) is the minimum change time at pi required for changing trips. Let et(r,pi) be the earliest trip in route r that one can catch at stop pi, i.e., the earliest trip t such that ⌧dep(t,pi) ⌧k1(pi)+⌧ch(pi). Note that (1) this trip may not exist, in which case et(r,pi) is undefined, and (2) in the first round we do not need to add the minimum change time ⌧ch(pi). To process the route, we visit its stops in order until we find a stop pi such that et(r,pi) is defined. This is when we can “hop on” the route. Let the corresponding trip t be the current trip for k. We keep traversing the route. For each subsequent stop pj, we can update ⌧k(pj) using this trip. To reconstruct the journey, we set a parent pointer to the stop 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 ⌧k1(pi) + ⌧ch(pi) < ⌧dep(t, pi) and update t by recomputing et(r,pi). Again, we do not need to consider the minimum change time ⌧ch(pi) in the first round.

Finally, the third stage of round k considers foot-paths. For each foot-path (pi,pj) 2 F it sets ⌧k (pj ) = min{⌧k (pj ), ⌧k (pi ) + `(pi , pj )}. Note that since F is transitive, we always find the fastest walking path, if one exists. The algorithm can be stopped after round k, if no label ⌧k(p) was improved.

**Solution:**

**Initialize:**

**Solve:**

**Build again result:**

**Algorithm Complexity:**

The worst-case running time of our algorithm can be bounded as follows. In every round, we scan each route at most once. If is the number of stops along , then we look at stops in total to process the route. For each stop, we must find the earliest trip et(r,·). If we keep the list of trips serving r sorted by time, while traversing r we can find all et(r, ·) values with a single sweep over this list, since et(r,·) can only decrease. In total, RAPTOR takes time, where K is the number of rounds. Note that the running time per round is potentially sublinear in the size of the input: The work per route is linear in the number of trips and the size of the route, but most of the departure/arrival times associated with individual trips are not considered. Constant access to the stops along routes and the arrival and departure times of specific trips can be achieved by a few arrays (see the appendix for details). In contrast, a similar analysis for the route-based model reveals that using Dijkstra algorithms are slower by at least a logarithmic factor, due to the priority queues.