r-2012-11-22

František Hajnovič
ferohajnovic@gmail.com

December 29, 2012

Contents

1 Adjustment of Gavoille's algorithm for graphs with $r(n)$ separator [GPPR04] for time-dependent scenario

1.1 Original algorithm

In [GPPR04], an algorithm answering distance queries in graphs is presented. The algorithm preprocesses the input graph, producing the so-called distance labels for each vertex. After preprocessing, a distance query between a pair of vertices u and v can be answered much quicklier then with e.g. Dijkstra's algorithm.

The algorithm takes advantage of recursive separators in the graph. The *size of the separator* and *how quickly we can find it* are two factors that influence the efficiency of the algorithm:

- Size of the separator influences the **resulting size of the distance labels**. If we define

$$\mathbf{R}(\mathbf{n}) = \sum_{i=0}^{\log_{3/2} n} r(n(2/3)^i)$$

  where $r(n)$ is the size of the recursive separator, the resulting total size of the preprocessed distance labels is

$$\mathcal{O}(nR(n)\log n + n\log^2 n)$$

  Note, that $R(n) = \mathcal{O}(r(n))$ for any $r(n) \geq n^{\epsilon}$.

  Also, the **time it takes to answer the query** is influenced by the size of the separator and it is

$$\mathcal{O}(R(n))$$

- The time ($t(n)$) it takes us to find the separator, on the other hand, influences the **preprocessing time** of

$$\mathcal{O}((\log n)(t(n) + r(n)(m + n \log n)))$$

E.g., in planar graphs, we can find a recursive separator of size $\mathcal{O}(\sqrt{n})$ ([Eri]) in linear time ($\mathcal{O}(n)$). That leads to the preprocessing time of $\mathcal{O}(n^{3/2} \log^2 n)$ (see **??** for details).

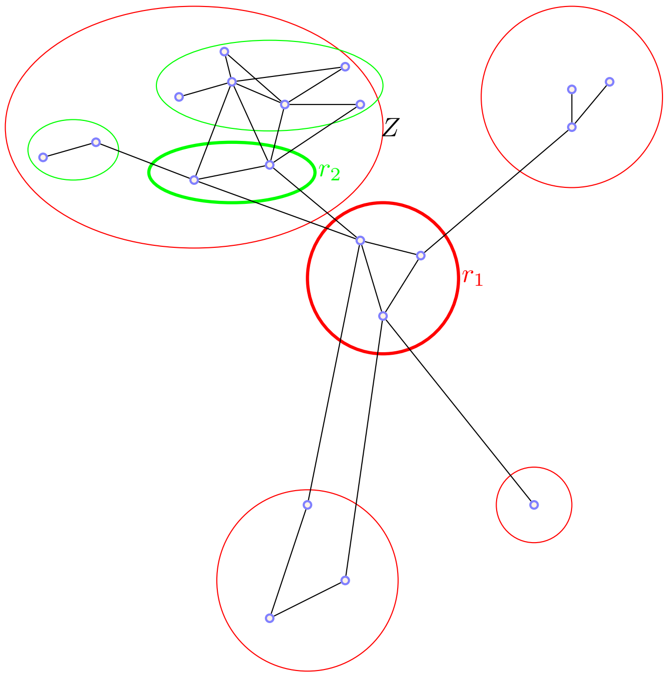

Figure 1: Separator splits graph into components. They in turn have again separators of respective size that further split the graph.

We will now briefly describe how the Gavoille's algorithm works (more details can be found in [GPPR04]).

1. A separator S is found and the graph is split into corresponding components
2. Each node in the component gets the list of distances to all the nodes from S
3. Each node in S also gets the list of distances to all other nodes from S
4. A node in a component gets the identifier of the component
5. We proceed recursively in components

The article does not specify *how* we perform point *2)* and *3)*. We will assume that we use Dijkstra's algorithm, as explained in *preprocessing of Gavoille's algorithm, r-12-11-15.*

After the preprocessing, answering a distance query between a pair of vertices u and v is simple. If u and v are from different components on the same level of recursion, the shortest path connecting them must pass through the separator whose removal created the components. We thus consider all of its vertices and find the distance as

$$d(u, v) = min_{s \in S}\{d(u, s) + d(s, v)\}$$

In case u and v are from the same component C, their corresponding shortest path may still pass through the separator whose removal created C, but it may also be completely within the component C (in which case we proceed recursively). Thus we take the minimum out of these two values. Since we may eventually consider series of decreasing separators (each at most 2/3 the size of the previous), we arrive at query time of $\mathcal{O}(R(n))$. For more details, please refer to [GPPR04], section *2.2*.

1.2 Adjustments

The algorithm is not designed for oriented or weighted graphs. The adjustments are, however, trivial:

- **Edge weights**: There is basically no adjustment necessary, the only difference will be in the size of the resulting labels. If we can bound the edge weights by a constant W, the total size of the preprocessed distance labels will be

$$\mathcal{O}(nR(n)\log W + n\log^2 n)$$

- **Oriented edges**: Oriented edges simply mean, that we will grow two shortest path trees (instead of just one) - one following the directions of the arcs and the other one going against them. We will thus have the distance labels of twice the original size, but it is necessary since we need to know the distance *to* as well as *from* the given separator vertex to component vertices.

In what follows, we will talk about **timetable graphs** and therefore we here provide a short definition. In timetable graph G_T for a given timetable T_G, we have nodes in the form $[v, t]$ where v is a vertex from the underlying graph G (we will call these vertices **cities**) and t is a departure [arrival] time of some elementary connection from T_G beginning [ending] in v. No other nodes exist in G_T. We create an oriented edge from $[x, p]$ to $[y, q]$ for each elementary connection (x, y, p, q) from the timetable T_G. See *r-12-11-15* for more details and illustrations.

We will now be interested, if this algorithm can be adjusted to work effectively for timetable graphs. Of course, as timetable graphs are also graphs, so we can simply take them as an input to the original algorithm. Mind that this way we will be querying for *shortest paths* in the timetable graphs - and any path between a pair of vertices in a timetable graph is the shortest one (as the vertices have timestamps and the length of the path is just their difference). This is not a problem, as looking for a shortest path in a timetable graph can be easily transformed into solving the earliest arrival problem (see picture 2).

The problem is, however, that even if the underlying graph of the timetable has small separator that can be effectively found, there is generally no guarantee that the same could be said about the graph of the timetable. We therefore try to adjust the algorithm.

Suppose we have a graph G for which we can find a separator of size $r(n)$ in time $t(n)$ and a timetable T without overtaking and express lines on top of this graph. In case there is overtaking in the timetable, we will first pre-process it to remove the overtaken connections. This can be done in time $\mathcal{O}(|T|)$ (supposing we have the timetable appropriately sorted) as follows:

```
1 for each oriented edge (x,y) in graph G:
2   for all elementary connections (x, y, p, q) (in an ascending order with respect to p):
3     t = q
4     destroy all already seen elementary connections (x, y, p', q') such that q' > t
```

Listing 1: Removing overtaken elementary connections

The row number 4 (destroying overtaken connections) will be executed at most $\mathcal{O}(|T|)$ times.

We will now adjust the Gavoille's algorithm to solve EAP on timetable T. The idea stems from the original algorithm: We first find the separator in the underlying graph that splits the timetable graph into components (based on the components in the underlying graph). For each *vertex* in the component, we would like to know the earliest arrival to each *city* from the separator and, consequently, the earliest arrival from each *vertex* in the separator to each *city* from the components. Having these kind of labels, we would be able to solve EAP across the separator the same way we solved shortest paths across the separator - by considering all cities of the separator as a transfer station and taking the one that minimizes the total time. We must also have EAP values for each separator vertex to each separator city - for the case when the query is for two cities from within the separator.

To obtain the mentioned EAP values we may grow shortest path trees in the timetable graph. This, however, is unnecessarily costly (see picture 3). A better approach is to create a time-dependent graph representing the timetable T. The cost of the arc in such a graph is a function that for a given moment

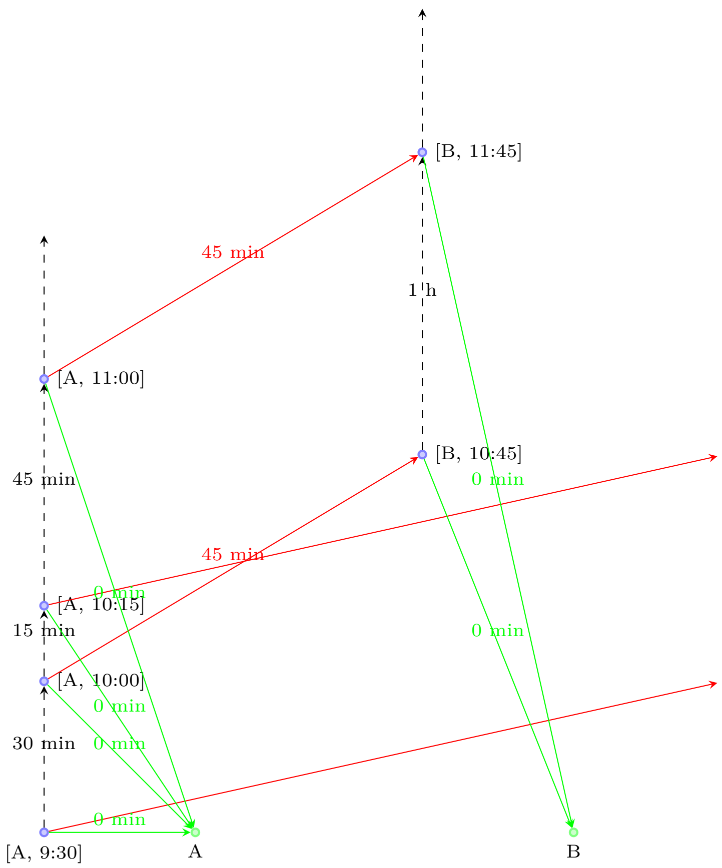

Figure 2: Effective looking for shortest paths in timetable graphs can be easily transformed into effective solving of EAP. We will simply look for shortest paths leading to respective newly added vertices (marked by green color) - this way we the earliest arriving connection to the given city.

outputs the traversal time of the arc. Growing shortest path trees in this graph is less costly, as we do not return again to a city that has already been settled. We still get the necessary EAP values, though.

The **preprocessing** of the algorithm will proceed recursively into components. Initially, the component $C = G$ (i.e. the whole underlying graph):

- First we find a $r(n)$ separator S in time $t(n)$ in C
- For each vertex $[v, t]$ where $v \in S$, we grow a backward time dependent shortest-path tree. We grow the tree until we settle all cities in C. For each settled city u and the time q at which it was settled, there is a node $[u, q]$ in the timetable graph. For this node we set the EAP label to city v to value $(t - q)$, but only if this label is still undefined or is greater then $(t - q)$. At the end of this procedure, we have the EAP values from each vertex of C to each city in S.
- We do the same thing, except that we grow *forward* (or simply normal) time dependent shortest-path trees. At the end, we will have the EAP values from each vertex of S to each city in C. This time, however, we store the EAP values with the vertices of S instead of the vertices corresponding to settled cities at a given time (this is due to the way we answer queries).

Note that we might end up with some vertices for which the required EAP value to the separator will be unset (e.g., if we destroy the edge $([B, 10:00], [A, 11:00])$ in figure 3, the EAP value from $[C, 9:00]$ to city A will be unset). We may solve this with a top-to-bottom sweep of the timetable graph during preprocessing, filling out unset values from the last previously seen set value. Another approach would be to solve this at the query time, but it might increase the time complexity of the query whereas the preprocessing time does not change with the inclusion of the sweeping.

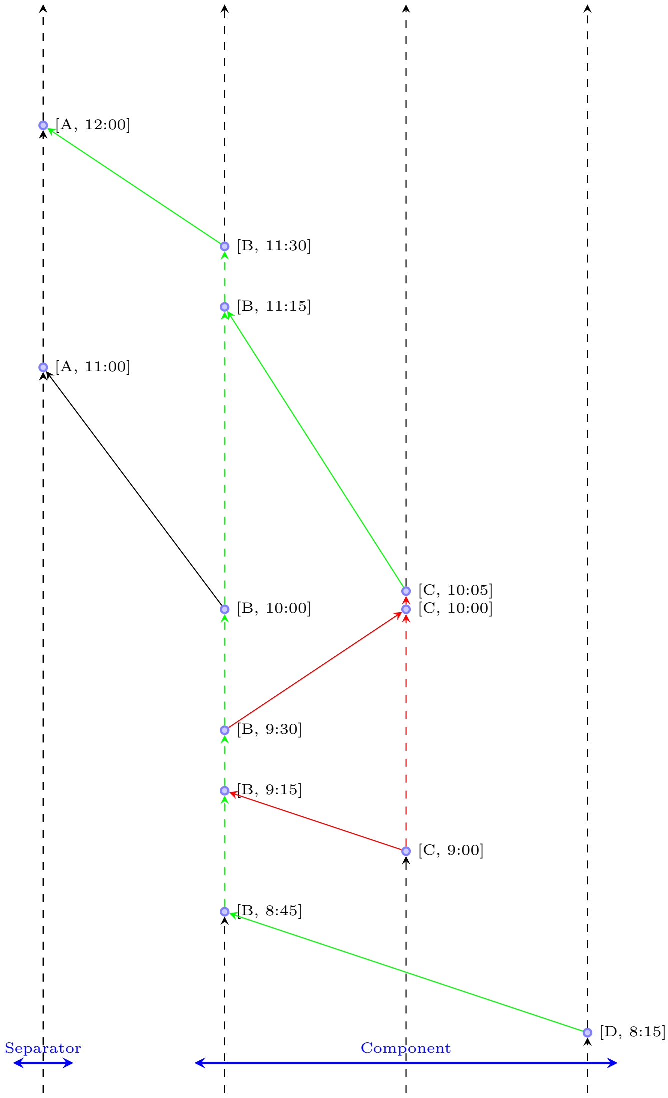

Figure 3: In this example, we grow backward shortest path tree from the vertex $[A, 12:00]$ (belonging to the separator) in order to set EAP values for the settled vertices. We explore bigger part of the graph (marked by red) then it is necessary (marked by green). Time-dependent approach will not continue the growth of the tree back into the cities which have already been visited.

```
function preprocess(C):
  #find the separator
  S = r(n)-separator(C)

  #set EAP values
  for [v, t] such that v ∈ S:
    grow backward time dependent shortest-path tree starting from v in time t:
      for each settled city u in time q:
        if EAP_{[u,q]}[v] > (t - q):
          EAP_{[u,q]}[v] = (t - q)
    grow time dependent shortest-path tree starting from v in time t:
      for each settled city u in time q:
        if EAP_{[v,t]}[u] > (q - t):
          EAP_{[v,t]}[u] = (q - t)

  #sweep to solve unset EAP values
  for each u such that u ∈ C:
```

```
    for each v ∈ S:
      lastEAP[v] = ∞
    for   each [u, t] in decreasing order by t:
      for each v ∈ S:
        if EAP_[u,t][v] == ∞:
          EAP_[u,t][v] = lastEAP[v]
        else:
          lastEAP[v] = EAP_[u,t][v]

  #set component id
  for each v in C:
    if v is in S:
      C_v  .= "S"
    else:
      C_v  .= component number of component containing v

  #proceed recursively
  for each component C':
    preprocess(C')

function solve_unset(G):
  for each v in G
    for each [v, t]

function main():
  #initialize labels
  for each [v, t], u:
     EAP_[v,t][u] = ∞
  for each v:
    C_v = ""

  #run preprocessing
  preprocess(G)
```

Listing 2: Preprocessing of the adjusted Gavoille's algorithm

1.3 Complexity

What is the time complexity of the **preprocessing**? First of all, we assume the following to be satisfied *before* preprocessing:

- For each arc of the underlying graph, the corresponding elementary connections are sorted by departure time. Define $height(T) = max_{v \in G}\{s \mid s = |\{(v,u,p,q) \in T\}|\}$, we can achieve the required order in time $\mathcal{O}(n\ h \log h)$ where $h = height(T)$.
- We have no overtaking connections (can be removed in $\mathcal{O}(|T|)$, which is $\mathcal{O}(n\ h)$)
- We have a time-dependent representation of the timetable T, with *constant-time* computation of the edge cost functions. This can be achieved in time $\mathcal{O}(n\ h)$, creating a data structure of size $\mathcal{O}(n\ h)$.

The last point may raise some questions - how to build the edge cost function in such a way, that the traversal time at any given moment could be retrieved in constant time?. As the cost function of an individual arc is simply a piece-wise linear function, the task at hand is to be able to output the specific piece of the function (or its time interval) for a given time in constant time (see picture 4).

One solution, though not in a constant time, is a binary search with logarithmic time complexity with respect to the number of intervals (interpolation points) which is h at most, thus $\mathcal{O}(\log h)$. For such lookups, all we need is an array of beginnings and respective endings of the intervals, easily computable in $\mathcal{O}(n\ h)$. The next approach will work only in cases when we have a certain granularity in times specified in the timetable - e.g. a granularity in minutes would mean that all the times are expressed up to the detail of minutes, but not further. In such case, we can simply precompute the traversal time for each moment (e.g. second). However, the preprocessing time (as well as space) would be $\mathcal{O}(n\ U)$ where U is the time range of the timetable.

Finally, we may vertically (according to time) divide the timetable into h equally large "buckets" $H_1, H_2, ...H_h$ - for each H_i we store the intervals of the time-dependent function that are within or overlap with H_i. This would take us $\mathcal{O}(n\ h)$ time and space in preprocessing. The expected number of intervals covered by each H_i is constant and thus in average case we could retrieve the interval I_j for a specified t in constant time.

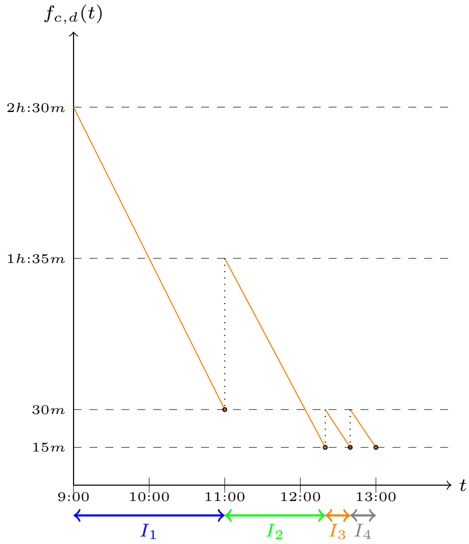

Figure 4: The problem behind constant cost function on an arc (from C to D in this case) - for each t we have to be able to determine in constant time its interval.

So the pre-preprocessing takes $\mathcal{O}(n\ h \log h)$ time. The time complexity of the preprocessing itself can be computed similarly as in *preprocessing of Gavoille's algorithm, r-12-11-15*. We thus explain it here only briefly. Here is a summary of costly actions in one iteration of function *preprocess* with component of size n as an input:

1. A separator of size $r(n)$ is found in $t(n)$ time
   - $\mathcal{O}(t(n))$
2. We grow the shortest path trees using Dijkstra's algorithm. We do so for each vertex of the current separator, thus it takes $\mathcal{O}(h \cdot (m + n \log n) r(n))$
   - $\mathcal{O}(h \cdot (m + n \log n) r(n))$
3. We apply the sweeping algorithm to solve unset EAP vales. This costs $\mathcal{O}(h \cdot n \cdot r(n))$
   - $\mathcal{O}(h \cdot n \cdot r(n))$

The time for one iteration of the function is thus $\mathcal{O}(t(n) + h \cdot (m + n \log n) r(n))$. Same way as in **??** we will get to the total upper bound on preprocessing

$$\mathcal{O}((\log n)(t(n) + r(n) \cdot h \cdot (m + n \log n)))$$

Which for planar graphs equals to

$$\mathcal{O}(h\ n^{3/2} \log^2 n)$$

Now we would like to look at the **size of the labels** that have been created in the preprocessing phase. What do the labels look like? We may look at the label $L(x)$ for vertex x from the perspective of how it was created during the algorithm. In each iteration where x was in a component, $L(x)$ gets the component identifier and EAP values to separator cities. There may be up to $\mathcal{O}(\log n)$ such iterations, each time with decreasing component size. At the end x is a trivial component of size 1 or it belongs to a separator, in which case it lists the EAP values to each city.

$$L(x) = \underbrace{\overbrace{[x \in \mathit{component}]}^{\text{[comp. id, EAPs to S]}} [x \in \mathit{component}] \cdots [x \in \mathit{component}]}_{\mathcal{O}(\log n)} \overbrace{[x \in \mathit{separator}]}^{\text{EAPs to all}}$$

We would like to make an upper bound on the total size of the precomputed labels. First, we ignore the separator part of the label and assume that each vertex "survived" (was part of the component) to the very last iteration. If we denote $l(n)$ the size of the label acquired on a graph with n vertices, we get

$$l(n) \leq l(2n/3) + \mathcal{O}(r(n) \log U + \log n)$$

where U is the time range of the timetable, $\log U$ necessary space to write down earliest arrival and $\log n$ necessary space to mark the component identifier. The recursive definition can be solved to

$$l(n) = \mathcal{O}(R(n) \log U + \log^2 n)$$

And as we have at most $\mathcal{O}(h \cdot n)$ vertices, the total size of the "component" parts of the labels is at most $\mathcal{O}(h \cdot n \cdot R(n) \log U + h \cdot n \cdot \log^2 n)$

In a case where vertex belongs to the separator, it lists the EAP values to all the cities in the current graph, which requires $\mathcal{O}(n \log U)$ bits of memory, i.e. $\mathcal{O}(h \cdot r(n) \cdot n \log U)$ bits for the given iteration. In the subsequent iteration, the components are of total size at most n, but their separators are smaller - at most $r(\frac{2}{3}n)$. Since a vertex can belong to a separator only once, we compute the total size of the "separator" parts of the labels as $\mathcal{O}(h \cdot n \cdot \log U(r(n) + r(\frac{2}{3}n) + r(\left(\frac{2}{3}\right)^2 n) + ...)) = \mathcal{O}(h \cdot n \cdot R(n) \log U)$. We see that this is bound by the "component" part of the labels resulting in the total $\mathcal{O}(h \cdot n \cdot R(n) \log U + h \cdot n \cdot \log^2 n)$ bits of memory consumed by preprocessed labels.

As for the **query time** - i.e. computing earliest arrival between from x at time t to y from EAP labels of x and y - there is not any new idea compared to the original algorithm so the query time can be estimated as $\mathcal{O}(R(n))$, with analogical explanation as in 1.1.

Finally, the answers of the queries are exact, i.e. **stretch** $= 1$, which completes the parameters of this distance oracle method:

- Preprocessing time: $\mathcal{O}((\log n)(t(n) + r(n) \cdot h \cdot (m + n \log n)))$
- Size: $\mathcal{O}(h \cdot n \cdot R(n) \log U + h \cdot n \cdot \log^2 n)$, is usually $\mathcal{O}(h \cdot n \cdot R(n) \log U)$
- Query time: $\mathcal{O}(R(n))$
- Stretch: 1

2 Open points

- Hierarchy of express lines $\rightarrow$ what properties can be propagated in time-expansion?
- Instant cost function - more formal and details

3 To do

- United airlines extract data
- Road network of SVK - process data
- Continue the diagnostic program
- Properties propagation in simple timetables
- Machine learning

References

[Eri] Jeff Erickson. Graph separators.

[GPPR04] Cyril Gavoille, David Peleg, Stéphane Pérennes, and Ran Raz. Distance labeling in graphs. *Journal of Algorithms*, 53(1):85 – 112, 2004. ISSN 0196-6774. URL `http://www.sciencedirect.com/science/article/pii/S0196677404000884`.