Contraction Hierarchies briefly explained

Stefan Funke

January 12, 2017

Abstract

We give a brief explanation of Contraction Hierachies (CH) as introduced by Geisberger et al. in 2008.

1 Outline of the Method

Problem Description

We are given a directed, weighted graph $G(V, E, c)$ with vertex set V, edge set E and a cost function $c : E \to \mathbb{R}^+$. The goal is to *preprocess* G such that subsequent *shortest path queries* specified by a source node s and a target node t can be answered very quickly (but still provably optimal).

To put this into perspective: computing the shortest path from some s to some t in the OSM road network of Germany[1] takes a few *seconds* for a decent implementation of Dijkstra's algorithm. We are aiming for a preprocessing scheme which allows for queries to be answered in around 1 *milisecond*, that is, a speed-up of a factor more than 1000!

Rough Idea

Preprocessing Phase

In the preprocessing phase we *augment* G by additional edges E' – shortcuts representing shortest paths in the original graph G – and furthermore assign each node $v \in V$ a natural number, the $level(v)$.

The result of the preprocessing phase is the function $level : V \to \mathbb{N}$ and a new graph $G^*(V, E^*, c^*)$ where $E^* = E \cup E'$ and $c^*(e) := c(e)$ if $e \in E$, otherwise $c^*(e)$ is set to the length of the respective shortest path which e represents.

In practice E^* is hardly twice the size of E, that is, the space consumption of the result of the preprocessing phase is of the same order as the original graph G.

Query Phase

To answer a query we execute two slightly modified Dijkstra runs on G^*, one starting in s, one in t. Amongst all nodes settled from both Dijkstra runs the one where the added distances from s and to t are minimal determines the shortest path from s to t. The query is highly efficient since the modified Dijkstra runs can discard most of the nodes and edges of G^* visiting only a miniscule fraction of the graph G^*.

[1] $|V| \approx 20$Mio and $|E| \approx 40$Mio.

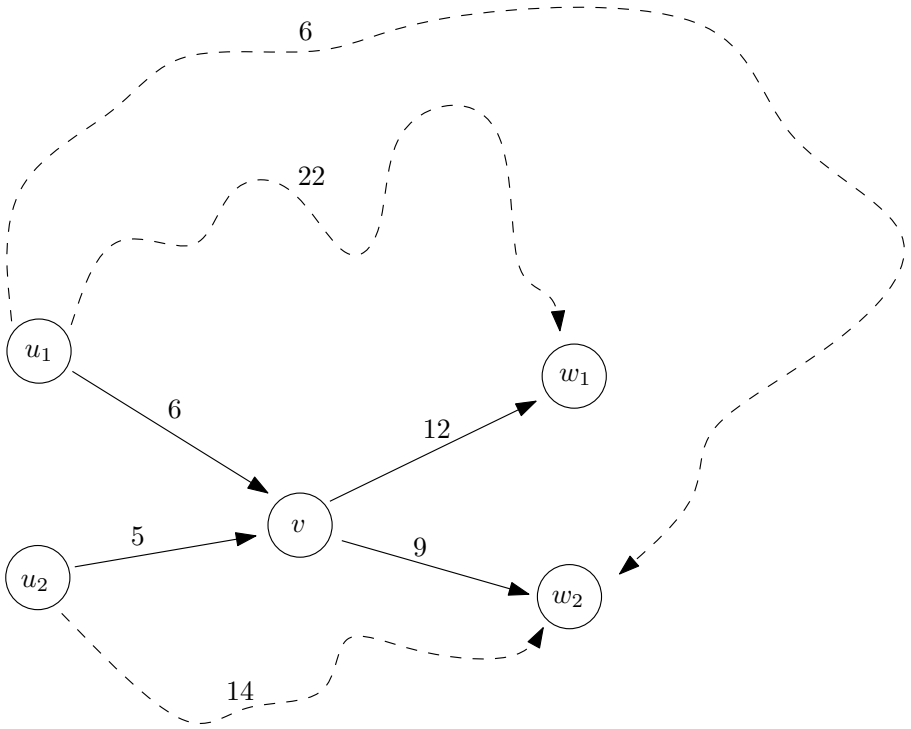

Figure 1: When contracting node v we have to create shortcuts (u_1, w_1) with cost 18 (since the shortest path from u_1 to w_1 avoiding v has cost 22) and (u_2, w_1) with cost 17 (since there is no other path from u_2 to w_1 than u_2vw_1). Shortcut (u_1, w_2) is not necessary since we can get from u_1 to w_2 on a path avoiding v cheaper than u_1vw_2, the shortcut (u_2, w_2) is not necessary since even after removal of v we can get from u_2 to w_2 at the same cost as u_2vw_2.

2 Preprocessing

Node Contraction of a node v

The crucial operation of the preprocessing phase is a so-called *node contraction*. Here the goal is to remove a node v from the graph G *without affecting shortest path distances between the other nodes*. To that end, it might be necessary to add shortcuts between neighbors of v.

Consider the situation in Figure 1 where node v is to be contracted. If the shortest path distance from some s to some t is affected by the removal of v it must be the case that every shortest path from s to t contains as subpath some u_ivw_j. If we add a short cut (u_i, w_j) with cost $c(u_i, v) + c(v, w_j)$ for each u_i, v, w_j for which u_ivw_j is the the only shortest path from u_i to w_j, we can ensure that shortest path distances between any pair of nodes s, t are preserved.

In practice we can implement the contraction of node v by starting a Dijkstra run from each u_i until all w_j are settled. A shortcut from u_i to w_j of cost $c(u_i, v) + c(v, w_j)$ is created if u_ivw_j is the only shortest path from u_i to w_j. The required effort for that are as many Dijkstra runs as there are neighbors u_i. If the u_i, v, and w_j are all 'nearby', the individual Dijkstra runs are fast since they can be aborted once all w_j are settled.

The preprocessing algorithm

With node contraction as a basic building block, the preprocessing works as follows:

```
counter=1;
while |V|>1 do
```

```
  select some node v from V and contract
  level(v):=counter++;
od
return level() and original graph with all created shortcuts added
```

The order in which the nodes from v are contracted has great influence on the number of added shortcuts as well as the query times. We will discuss that later.

3 Query

To answer a query from s to t we proceed as follows:

1. in G^* run Dijkstra starting in s; when pulling a node v from the priority queue, only consider edges (v, w) with $level(w) > level(v)$. This yields paths and distances $d_s(v)$ from s to some nodes $v \in V$.

2. in G^* run a 'reverse' Dijkstra[2] starting in t; when pulling a node v from the priority queue, only consider edges (w, v) with $level(w) > level(v)$. This yields paths and distances $d_t(v)$ from some nodes v to t.

3. amongst the nodes settled from both Dijkstras, the node v^* with minimal $d_s(v^*) + d_t(v^*)$ determines the result of the query

The resulting path might contain shortcut edges which of course can be recursively unpacked (each shortcut replaces two other edges by construction).

4 Correctness

It is not clear why the path returned by the query routine is indeed a shortest path from s to t, we will argue in the following that this is really the case. To that end assume for simplicity that shortest paths are unique (can easily be enforced by symbolic perturbation) – one can also extend the argument for the case of ambiguous shortest paths in several ways.

Consider a shortest path $abcdefgh$ (black edges) from a to h as depicted in Figure 2. Here, the vertical position of a node denotes its *level*, that is, the nodes of this path were contracted in the order $b < g < a < f < h < d < c < e$. Our query routine would not find the path $abcdefgh$ since it cannot be decomposed into two parts, one with increasing levels of the nodes (to be explored from s) and one with decreasing levels of the nodes (to be explored from t). Fortunately, we can argue that the preprocessing phase has added shortcuts such that the path $abcdefgh$ has a representation (using shortcuts) which is a sequence of upward edges followed by a sequence of downward edges.

Consider the time when node b was contracted. At that time a and c were neighbors of b and (uniqueness of shortest paths!) removing b would have altered the shortest path distance from a to c, hence the shortcut (a, c) (with added costs of the edges (a, b) and (b, c)) must have been added. For the same reason, shortcuts (f, h), (c, e) and (e, h) must have been added.

Now let v^* be the node in the path $abcdefgh$ with highest level — in our example $v^* = e$. We claim that there exists a representation of the shortest path consisting of upward edges only from a to v^*. Assume otherwise, then consider the 'last' downward edge (u, v) on some path π from a to v^* and let w be the the successor of v on this path. We have $level(u) > level(v) < level(w)$. But then, at the time when v was contracted, the shortcut (u, w) must have been created, hence there must be also a shortest path from a to v^* exactly the same as π but without the downward edge (u, v) (but possibly a downward edge $((u, w))$. But then we apply the same argument to get rid of this downward edge as well (reducing the number of edges on the path). At some point we must end up with a path from a to v^* consisting only of upward edges (at latest when only the

[2]Reverse Dijkstra considers edges in the opposite direction.

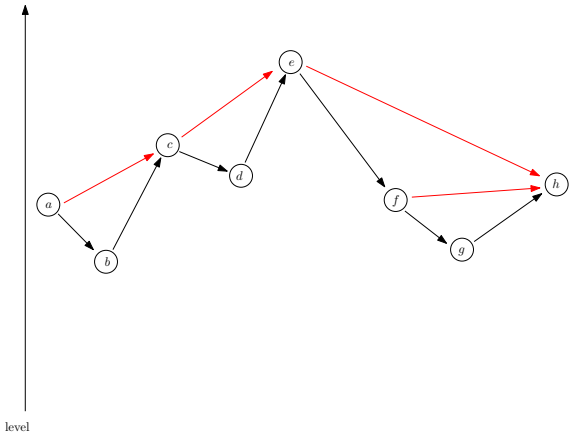

Figure 2: Correctness of the Query routine.

edge (a, v^*) is still there). The same argument applies for the downward path from v^* to t. So our preprocessing routine has ensured that a representation of the shortest path from s to t will be found.

5 Efficiency

5.1 Contraction Order

The efficiency of the whole approach relies on the hope that in the two Dijkstra searches in G^* only few nodes/edges have to be considered. First of all, it seems reasonable to choose a contraction order where only few shortcut edges are added. Intuitively, we are happy to contract dead-end nodes since their removal does not induce any shortcuts at all, but refrain from contracting nodes corresponding to important crossings of the road network. If we have, for example, an important 5-way crossing the contraction of the respective node might induce 25 shortcuts. So a natural strategy to pick the next node to contract in the preprocessing phase is as follows:

- compute for every node v what would happen if v was contracted, i.e. how many edges would be deleted, how many shortcuts created
- actually contract the node where the difference between # shortcuts created - # edges deleted is minimal

Unfortunately, applying only this criterion leads to some undesirable effects. To that end consider the small portion of a larger graph in Figure 3.

Applying the edge difference strategy would contract the nodes from the dead-end road one-by-one, not introducing any shortcuts, but also not speeding-up the query at all (when for example quering with the source as the end of the dead-end). Hence it seems reasonable not to contract immediately a neighbor of a recently contracted node. In fact, the original paper by Geisberger et al. proposes several strategies to implement this idea.

Recent implementations of the Contraction Hierarchy approach go a bit further and do the following:

- construct an independent set $I \subset V$
- compute the edge differences for all nodes in I
- contract all nodes from I with 'low' edge difference

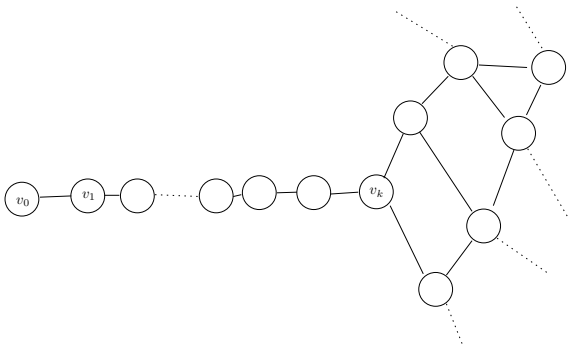

Figure 3: Bad contraction order.

- assign all contracted nodes the same level counter
- increase the counter and repeat

Note that here many nodes might be assigned the same level – still, neighboring nodes are never assigned the same level. The independent set construction enforces a more uniform contraction process, still, taking the edge difference into account leads to a small number of shortcuts added. The query algorithm as well as the argumentation for correcntess does not change.

5.2 Query Optimization

There are some improvements to the query process possible one of which we sketch in the following.

Probably in contrast to our intuition, the shortest path tree grown in the upwards graph rooted in s does not only contain paths that are shortest paths in the original graph (when replacing the shortcuts by the respective shortest paths in the original graph — Exercise: Give an example!). If we knew at the time when a node v is pulled from the priority queue that its distance value from s is not a shortest path distance in the original graph, we could simply discard v from consideration – and in particular do not relax any of its outgoing edges. While testing this exactly seems too expensive, there is the strategy of *stall-on-demand* which provides a simple yet effective test, which discards such nodes in many cases (look it up in the original paper). With this strategy, query times are again drastically reduced (by a factor of around 10).

6 Summary

In practice, the preprocessing sketched above takes few minutes on the road network of Germany roughly doubling the number of edges, the queries can be answered within few millisecond (touching around 1000 nodes only in the two Dijkstras). The resulting paths contain shortcuts, of course, but these can easily be unpacked to obtain the complete path in the original graph.