Minimum spanning tree re-optimization

Giovanni Righini

University of Milan

UNIVERSITÀ DEGLI STUDI DI MILANO

MST re-optimization

Given

- a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$,
- a cost function $c : \mathcal{E} \mapsto \Re$,
- a minimum spanning tree $\mathcal{T}^*$ of $\mathcal{G}$,

find $n = |\mathcal{V}|$ alternative minimum spanning trees, one for each graph that is obtained by deleting a vertex of $\mathcal{G}$.

Naive procedure: re-compute as many minimum spanning trees as the number n of vertices of $\mathcal{G}$, each one spanning $\mathcal{V}$ with the exception of a vertex $\overline{p} \in V$.

MST re-optimization

Main idea: compute all the alternative edges that are needed to rebuild a minimum cost spanning tree when a vertex is deleted from $\mathcal{G}$.

Main result: This goal is achieved with the same worst-case computational complexity required by the computation of a single minimum spanning tree of $\mathcal{G}$.

Input:

- a weighted graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, with $n = |\mathcal{V}|$ vertices and $m = |\mathcal{E}|$ edges,
- a minimum spanning tree $\mathcal{T}^*$,
- a sorted list of all edges $\mathcal{E}$ (by non-decreasing cost).

Output:

- an optimal set of alternative edges.

Overview: p-components

When a vertex $p \in \mathcal{V}$ is deleted, $\mathcal{T}^*$ is disconnected into a forest $\mathcal{F}_p^*$ made of as many connected components as the degree of p in $\mathcal{T}^*$.

In the remainder they are called *p-components*, to mean that they arise when p is deleted.

When vertex $p \in \mathcal{V}$ is deleted, *alternative edges* must be found to reconnect the p-components and they must be selected to produce a new minimum spanning tree $\mathcal{T}_p^*$.

Naive method

Naive method. For each deleted vertex $p \in \mathcal{V}$, recompute a minimum spanning tree $\mathcal{T}_p^*$ with Kruskal or Boruvka algorithm, after having restored the state of a suitable data-structure to represent the p-components of $\mathcal{F}_p^*$.

The vertices in each p-component can be identified in $O(n)$ and this is the complexity to set up the state of a Union-Find data-structure.

Then, the computation of an alternative minimum spanning tree with Kruskal algorithm requires $O(m + n \log d(p))$, where $d(p)$ indicates the degree of vertex $p \in V$.

Hence, repeating this naive procedure for all vertices would require $O(mn + n^2 \log n)$.

Search of alternative edges in parallel

All edges are scanned in non-decreasing order of their cost, as in Kruskal algorithm.

For each edge $[i, j] \notin T^*$ the algorithm searches the vertices p such that edge $[i, j]$ reconnects two different p-components of F_p^*.

For each vertex p and for each pair of p-components in F_p^*, the first edge that is found to reconnect them, certainly belongs to T_p^* owing to the edge ordering.

In this way all forests F_p^* can be populated in parallel with alternative edges for all $p \in V$ until all alternative minimum spanning trees T_p^* are produced.

Accelerating the search

This search, if performed as described above, would be very inefficient.

Idea for accelerating the search:

Consider the minimum spanning tree T^* and an edge $[i,j] \notin T^*$.

Adding edge $[i,j]$ to T^* defines a unique cycle $C(i,j)$.

The vertices along $C(i,j)$, except i and j, are those for which $[i,j]$ is a candidate alternative edge: if a vertex $p \in V$ along $C(i,j)$ is deleted from T^*, then $[i,j]$ reconnects two different p-components.

On the contrary, for all vertices $p \in V$ not belonging to $C(i,j)$, $[i,j]$ cannot be a candidate alternative edge, because both i and j belong to the same p-component.

To efficiently scan $C(i,j)$, the algorithm skips in $O(1)$ all vertices p along $C(i,j)$ for which an alternative edge reconnecting the p-components of i and j has already been found.

Accelerating the search

When edge $[i', j']$ is considered, it can be an alternative edge for vertices a, b, c, d and e. Later on, when the more expensive edge $[i'', j'']$ is considered, it can be an alternative edge for vertices f, g, h and l, but there is no point in considering it for a, b, c, d and e again.

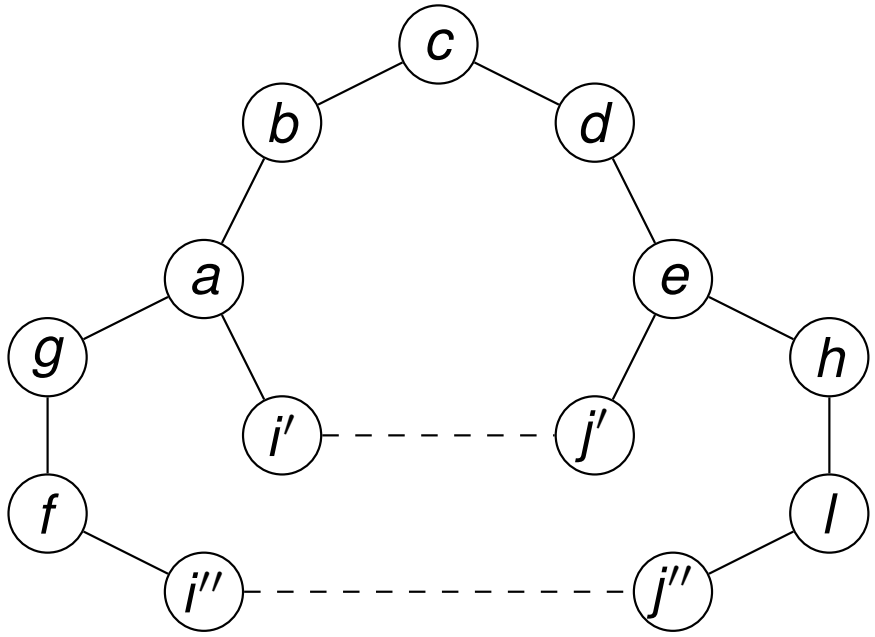


Figure: Cycles $C(i', j')$ and $C(i'', j'')$ partially overlap: since $[i', j']$ is cheaper than $[i'', j'']$, the vertices in $C(i', j')$ are skipped when $[i'', j'']$ is considered.

Computational complexity

With this idea: overall computational complexity $O(m + n \log n)$,

because

- all operations when a candidate alternative edge is discarded: $O(1)$ for each edge, $O(m)$ overall;
- all operations when a candidate alternative edge is inserted: $O(\log n)$ for each insertion, $O(n \log n)$ overall.

The oriented minimum spanning tree

The minimum spanning tree T^* is initially oriented from an arbitrarily selected root vertex $r \in V$.

Definition

The predecessor of any vertex $p \in V \setminus \{r\}$, indicated by Pred(p), *is the vertex adjacent to p along the path between p and r in T^*.*

Property

Since T^* is a spanning tree, there exists a unique path between any vertex $p \in V \setminus \{r\}$ and r; therefore the predecessor exists and is unique for each $p \in V \setminus \{r\}$.

Pred(r) is set to a null value to indicate that the root vertex r has no predecessor.

The depth of vertices

Definition

On the oriented tree T^* the depth of any vertex $p \in V$, indicated by $\text{Depth}(p)$*, is the number of edges between r and p.*

From Definition 2 and Property 1 the following property follows.

Property

For any two vertices u and v such that $v = \text{Pred}(u)$, it holds $\text{Depth}(u) = \text{Depth}(v) + 1$.

Up and Down

Consider a depth-first-search visit of T^* and let us define a *move* to happen every time an edge is traversed in any direction.

Definition

We define $\mathrm{Dn}(r) = 1$. *For all* $p \in V \backslash \{r\}$, $\mathrm{Dn}(p)$ *is the progressive number of the move that reaches* p *from* $\mathrm{Pred}(p)$. *For all* $p \in V$, $\mathrm{Up}(p)$ *is the progressive number of the move that reaches* $\mathrm{Pred}(p)$ *from* p.

The following properties hold.

Property

(i) The indices $\mathrm{Dn}(p)$ *and* $\mathrm{Up}(p)$ *have unique values in* T^*. *(ii) Their values span the interval* $[1, \ldots, 2n]$. *(iii) For any vertex* $v \in V$ *and for each vertex* $u \in V$ *in the oriented subtree rooted at* v *with* $u \neq v$,

$$\mathrm{Dn}(v) < \mathrm{Dn}(u) < \mathrm{Up}(u) < \mathrm{Up}(v).$$

Example: Up, Dn

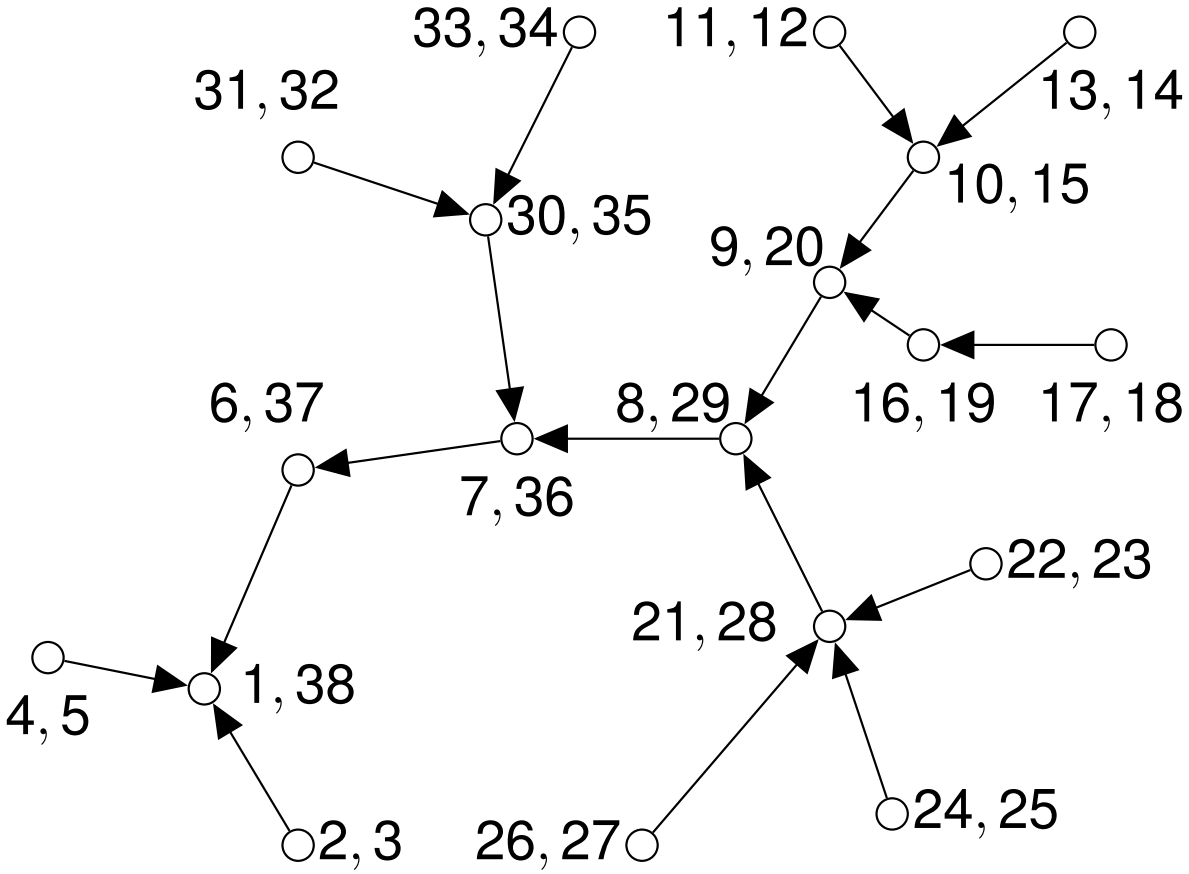


Figure: An oriented spanning tree with the values (Dn, Up) for each vertex.

The apex of a cycle

Consider an edge $[i, j] \notin T^*$. When $[i, j]$ is added to T^* a unique cycle, $C(i, j)$, is produced.

Definition

The apex of a cycle $C(i, j)$ is the minimum depth vertex along it.

Owing to the orientation of T^* and to Property 3, the following properties hold.

Property

(i) For any cycle $C(i, j)$, the apex exists and it is unique; (ii) if vertex p is the apex of $C(i, j)$, then $\mathrm{Dn}(p) \leq \min\{\mathrm{Dn}(i), \mathrm{Dn}(j)\}$ *and* $\mathrm{Up}(p) \geq \max\{\mathrm{Up}(i), \mathrm{Up}(j)\}$.

Identifying the apex

Let SubTree(p, i, j) be a boolean function that tests Property above, to check whether one of the endpoints of $[i, j]$ is the apex of $C(i, j)$ or not.

Property

Given any edge $[i, j] \notin T^*$*, if a vertex p verifies* SubTree(p, i, j) *and it also belongs to* $C(i, j)$*, then it is the apex of* $C(i, j)$*.*

Procedure Orient

Algorithm 1 Orient

1: **for** $p \in V$ **do**
2: $\quad \delta(p) \leftarrow \emptyset$
3: **for** $[i, j] \in T^*$ **do**
4: $\quad \delta(i) \leftarrow \delta(i) \cup \{j\}$
5: $\quad \delta(j) \leftarrow \delta(j) \cup \{i\}$
6: $r \leftarrow$ Select
7: Pred$(r) \leftarrow 0$
8: Depth$(r) \leftarrow 0$
9: $\alpha \leftarrow 0$
10: DFS(r)

$\delta(i)$ is the star of each generic vertex $i \in \mathcal{V}$.

Procedure *DFS*()

Algorithm 2 DFS(p)

1: $\alpha \leftarrow \alpha + 1$
2: Dn(p) $\leftarrow \alpha$
3: **for** $k \in \delta(p)$ **do**
4: **if** $k \neq$ Pred(p) **then**
5: Pred(k) $\leftarrow p$
6: Depth(k) $\leftarrow$ Depth(p) $+ 1$
7: DFS(k)
8: $\alpha \leftarrow \alpha + 1$
9: Up(p) $\leftarrow \alpha$

A counter α counts every move in either direction along the edges of T^*.

Complexity of Orient

The computation of the stars of all vertices (lines 1 to 5) takes $O(n)$, because T^* contains $n-1$ edges.

All the operations on lines 6-9 of Orient can be done in $O(1)$.

The time complexity taken by all instructions on lines 1, 2, 8 and 9 of DFS is $O(n)$, because α ranges from 1 to $2n$.

The total number of iterations of the loop on lines 3-7 of DFS is twice the number of edges of T^*, i.e. $2(n-1)$, and the body of the loop (lines 5 and 6) is executed in $O(1)$.

Therefore the time complexity for visiting T^* with DFS is $O(n)$.

Local subgraphs

Definition

For each vertex $p \in V$ a local subgraph $\mathcal{G}(p) = (\mathcal{V}(p), \mathcal{T}(p))$ is defined: it has $|\mathcal{V}(p)| = d(p)$ vertices, where $d(p)$ is the degree of p in T^*.

Vertices in $\mathcal{V}(p) \Leftrightarrow$ neighbors of p in T^*, p-components.

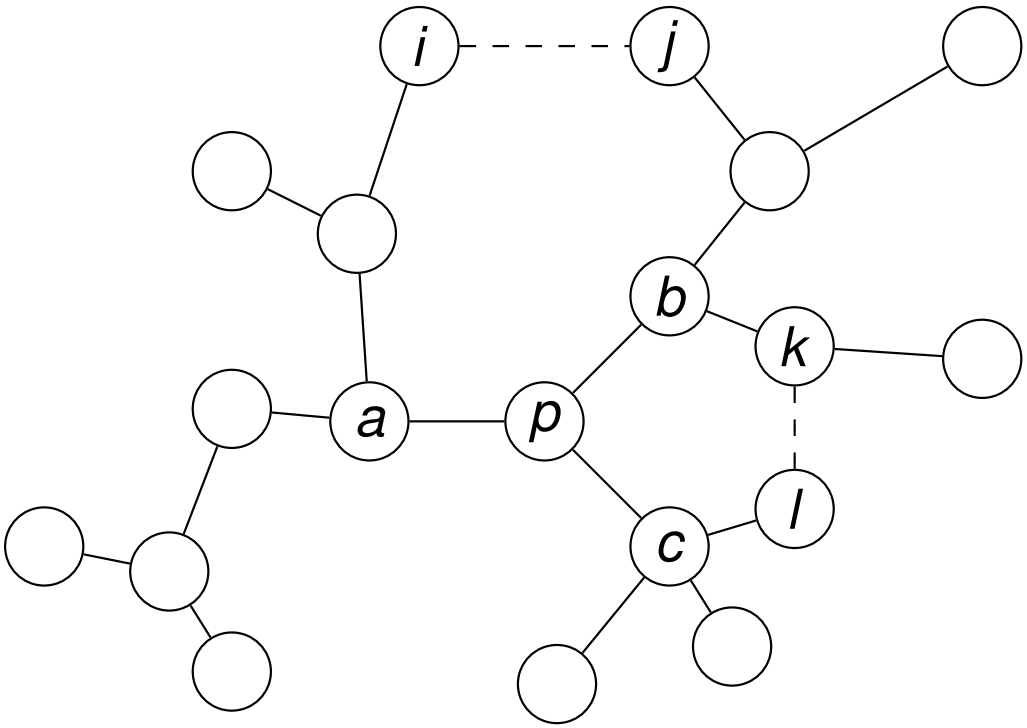
Edges in local subgraphs: *links* in the remainder.

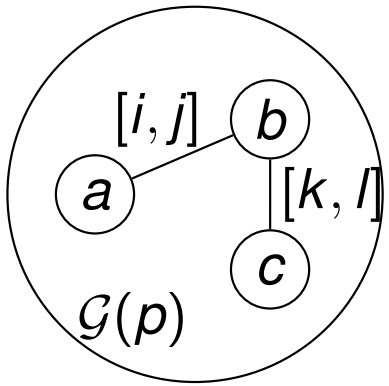
$\mathcal{T}(p)$ is a forest $\forall p \in V$; initially empty; eventually, a spanning tree of $\mathcal{G}(p)$.

Each link in $\mathcal{T}(p)$ is a record with two fields: the link itself between two vertices of $\mathcal{G}(p)$, and the alternative edge $[i, j]$ reconnecting the two corresponding p-components.

Local subgraphs: an example

When vertex p is deleted from the spanning tree, edges $[i, j]$ and $[k, l]$ are used to reconnect the three resulting p-components at minimum cost. Links $[a, b]$ and $[b, c]$ are inserted in $\mathcal{G}(p)$, forming a spanning tree $\mathcal{T}(p)$. Each link in $\mathcal{T}(p)$ has an associated alternative edge in G.





Local subgraphs: horizontal and vertical links

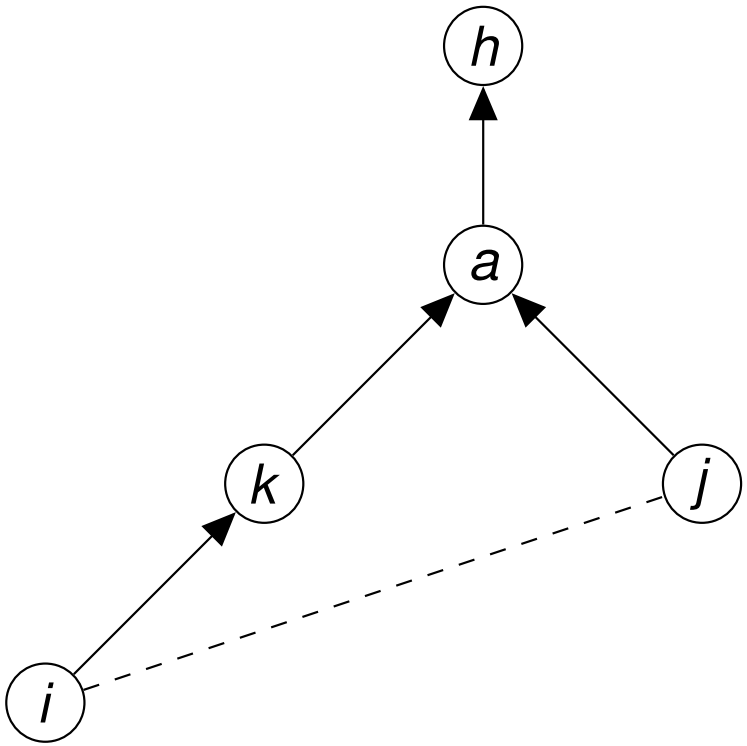
Definition

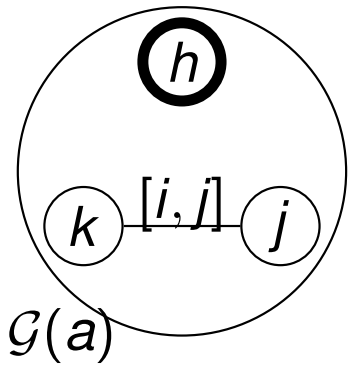
Links in $\mathcal{T}(p)$ *incident to the local vertex corresponding to* Pred(p) *are vertical links; links in* $\mathcal{T}(p)$ *not incident to the local vertex corresponding to* Pred(p) *are horizontal links.*

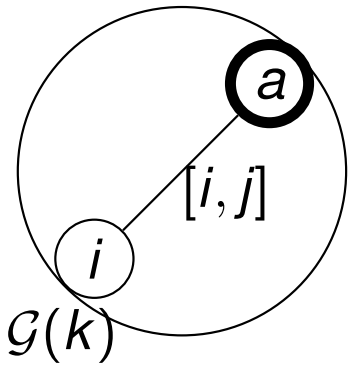
Remark. The local forest $\mathcal{T}(r)$ cannot include vertical links, since r has no predecessor.

Horizontal and vertical links: an example

When the candidate alternative edge $[i, j]$ is considered, a vertical link between the local vertices i and a is inserted in $\mathcal{G}(k)$, because k is a vertex between i and the apex; a horizontal link between the local vertices k and j is inserted in $\mathcal{G}(a)$, because a is the apex.





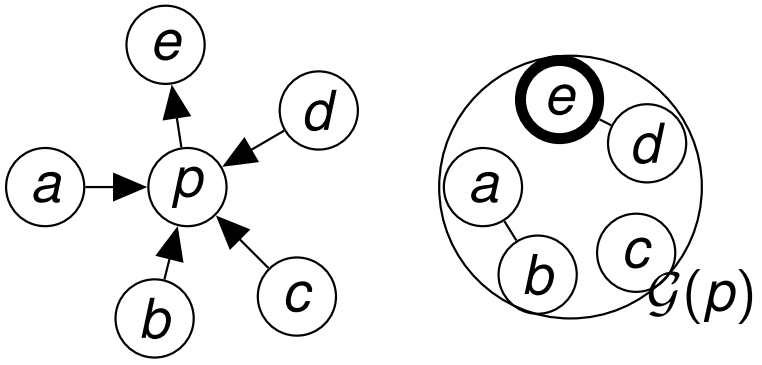


Local Union-Find data-structure

A Union-Find data-structure is kept for each vertex $p \in V$:

- an array of $d(p)$ linked lists. Each list $\mathcal{L}(p,k)$ is initialized with a single element $k \in \mathcal{V}(p)$, corresponding to a neighbor k of p in T^*.
- Card(p,k): cardinality of $\mathcal{L}(p,k)$;
- Head(p,k): head of the list to which k belongs.

Local Union-Find: an example



Vertex	Head	Card	$\mathcal{L}$
a	a	2	$\{a, b\}$
b	a	0	$\{\}$
c	c	1	$\{c\}$
d	d	2	$\{d, e\}$
e	d	0	$\{\}$

A vertex p with five neighbors (left), its local subgraph $\mathcal{G}(p)$ (center) and the corresponding state of the Union-Find data-structure (right).

Oriented arcs: predecessors in T^*.

$Pred(p)$: thick line.

Initialization of local subgraphs

Algorithm 3 Initialization

for $p \in V$ **do**
  $\mathcal{V}(p) \leftarrow \emptyset$
for $[i,j] \in T^*$ **do**
  $\mathcal{V}(i) \leftarrow \mathcal{V}(i) \cup \{j\}$
  $\mathcal{V}(j) \leftarrow \mathcal{V}(j) \cup \{i\}$
  $\mathcal{L}(i,j) \leftarrow \{j\}$
  $\mathrm{Card}(i,j) \leftarrow 1$
  $\mathrm{Head}(i,j) \leftarrow j$
  $\mathcal{L}(j,i) \leftarrow \{i\}$
  $\mathrm{Card}(j,i) \leftarrow 1$
  $\mathrm{Head}(j,i) \leftarrow i$
for $p \in V$ **do**
  $\mathcal{T}(p) \leftarrow \emptyset$
  $\mathrm{AltTreeCost}(p) \leftarrow 0$
$\mu \leftarrow n-2$

AltTreeCost(p): total cost of the alternative edges for vertex $p \in V$.

μ: number of missing alternative edges.

Complexity: $O(n)$.

Oriented paths and trees

Consider an edge $[i, j] \notin T^*$ and the corresponding cycle $C(i, j)$.

Definition

The oriented path $P(i, j)$ *goes from vertex* i *to* apex(i, j) *in* T^**; the oriented path* $P(j, i)$ *goes from vertex* j *to* apex(i, j) *in* T^**.*

One of the two paths may not exist, when i or j is the apex of $C(i, j)$.

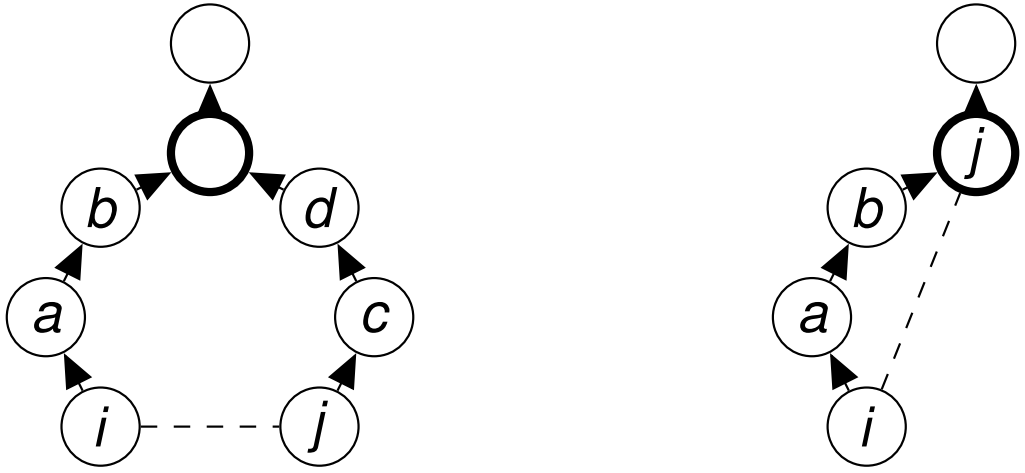


Figure: Left: edge $[i, j]$ corresponds to two oriented paths, one including a and b, the other including c and d. Right: edge $[i, j]$ corresponds to a single path, because one of its endpoints is the apex of $C(i, j)$. Apex: thick line. Oriented arcs: predecessors in T^*.

Internal vertices

Definition

The internal vertices of a non-empty oriented path are the vertices along it excluding its endpoints.

The apex(i, j) and the internal vertices of $P(i, j)$ and $P(j, i)$ are those for which edge $[i, j]$ can be used as an alternative edge.

In the local subgraph of apex(i, j) edge $[i, j]$ corresponds to a horizontal link.

In the local subgraphs of the internal vertices edge $[i, j]$ corresponds to a vertical link.

For this reason, the apex is processed separately from the internal vertices.

Root of an oriented path

Definition

The root of a non-empty oriented path $P(i,j)$ *or* $P(j,i)$ *is its vertex* t *with* $\mathrm{Depth}(t) = \mathrm{Depth}(\mathrm{apex}(i,j)) + 1$.

For each internal vertex p of an oriented path $P(i,j)$, edge $[i,j]$ reconnects two p-components, one of them containing i and the other one containing $\mathrm{Pred}(p)$ (and the same holds for $P(j,i)$ swapping i with j).

For each candidate alternative edge $[i,j]$ the algorithm scans $P(i,j)$ from i and $P(j,i)$ from j up to their roots.

Therefore all local forests of internal vertices of the two paths are considered to possibly insert a vertical link in each of them.

Oriented trees

Definition

Two oriented paths P' and P'' overlap if and only if they have at least an internal vertex in common.

When two (or more) oriented paths overlap, the algorithm merges them to form an *oriented tree* with the following properties.

Property

(i) Each vertex belongs to at most one oriented tree; (ii) each oriented tree has a unique root, that is the minimum depth root of its paths; (iii) there is at least one vertical link in the local forest of each vertex in an oriented tree.

Owing to this property, every time the algorithm scans a path and it detects that it (partially) overlaps with an existing oriented tree, the oriented tree is skipped in $O(1)$ and the scan resumes directly from its root.

Relevant paths

For each vertex $p \in V$ a variable Path(p) records the index of the first path that introduces a vertical link in $\mathcal{G}(p)$.

Definition

An oriented path is relevant if and only if it inserts a vertical link in at least one subgraph.

Property

Since the number of vertical links is bounded above by $n - 2$, this is also the maximum number of relevant paths.

Counting relevant paths

Every time an oriented path is scanned, a path counter π is increased by 1 and the path root is initialized at 0.

Every time a vertical link is inserted in some local forest $\mathcal{T}(p)$, the path root is updated to p.

At the end of the scan, if the path root is still equal to 0, then the path is discarded as non-relevant and the path counter π is decreased by 1.

No update to any data-structure is done, while the currently scanned path has not yet been recognized as relevant.

The Tree-Union-Find data-structure

A suitable data-structure, called *Tree-Union-Find*, is used to efficiently merge relevant oriented paths into oriented trees.

The Tree-Union-Find includes

- an array TList of linked lists,
- an array TCard,
- an array THead,
- an array TRoot.

Each component of such arrays corresponds to an oriented tree, i.e. a set of overlapping relevant oriented paths:

- TList(τ): set of paths merged into the tree τ;
- TCard(τ): their number;
- THead(τ): index of a representative path in TList(τ);
- TRoot(τ): root of tree τ, i.e. minimum depth vertex among the roots of the paths in TList(τ).

All arrays in the Tree-Union-Find are made by at most $n - 2$ items.

Initialization of a new path

Algorithm 4 InitPath

$\pi \leftarrow \pi + 1$
$\text{TList}(\pi) \leftarrow \{\pi\}$
$\text{TCard}(\pi) \leftarrow 1$
$\text{THead}(\pi) \leftarrow \pi$
$\text{TRoot}(\pi) \leftarrow 0$

Complexity: $O(1)$.

Deletion of non-relevant paths

Algorithm 5 PurgePath(π)

1: **if** TRoot(THead(π)) $= 0$ **then**
2: $\quad \pi \leftarrow \pi - 1$
3: $\quad$ TList(π) $\leftarrow \emptyset$

Complexity: $O(1)$.

Merging two subtrees

TreeMerge(π', π'') merges the subtrees containing paths π' and π''.

Algorithm 6 TreeMerge(π', π'').

if TCard(THead(π')) $>$ TCard(THead(π'')) **then**
  $\tau' \leftarrow$ THead(π')
  $\tau'' \leftarrow$ THead(π'')
else
  $\tau' \leftarrow$ THead(π'')
  $\tau'' \leftarrow$ THead(π')
if (Depth(TRoot(τ'')) $<$ Depth(TRoot(τ'))) **then**
  TRoot(τ') $\leftarrow$ TRoot(τ'')
for $\pi \in$ TList(τ'') **do**
  THead(π) $\leftarrow \tau'$
TCard(τ') $\leftarrow$ TCard(τ') + TCard(τ'')
TCard(τ'') $\leftarrow 0$
TList(τ')) $\leftarrow$ TList(τ')) $\cup$ TList(τ''))

The main algorithm

Input:

- a graph G,
- a cost function c,
- a minimum spanning tree T^*,
- a sorted list $\mathcal{S}$ of all edges $e \in E$.

Algorithm 7 Main($G, c, T^*, \mathcal{S}$)

Orient
Initialization
Search

Complexity of *Search*: $O(m + n \log n))$, to be proven.

Search

Algorithm 8 Search.

1: $\pi \leftarrow 0$
2: **repeat**
3: **repeat**
4: $[i, j] \leftarrow \text{Extract}(\mathcal{S})$
5: **until** $[i, j] \notin T^*$
6: $u \leftarrow i$
7: **if** $\neg\text{SubTree}(i, i, j)$ **then**
8: $\text{PathScan}(i, j, u)$
9: $v \leftarrow j$
10: **if** $\neg\text{SubTree}(j, i, j)$ **then**
11: $\text{PathScan}(j, i, v)$
12: **if** $\neg\text{SubTree}(i, i, j) \wedge \neg\text{SubTree}(j, i, j) \wedge (\text{Stop}(u) \vee \text{Stop}(v))$ **then**
13: $\text{ProcessApex}(u, v, i, j)$
14: **until** $\mu = 0$

Search, part I

At each iteration of the main loop a candidate alternative edge is considered and all the corresponding vertical and horizontal links are inserted.

Lines 3-5: the next candidate alternative edge $[i, j]$ is extracted from the sorted list $\mathcal{S}$.

Search, part II

Lines 6-11: paths $P(i,j)$ and $P(j,i)$ are scanned to possibly insert vertical links.
Indices $u \in P(i,j)$ and $v \in P(j,i)$: current vertices on the paths.
PathScan is called only if the corresponding path exists (lines 7 and 10).
SubTree(p,i,j) checks whether an endpoint of $[i,j]$ is the apex of $C(i,j)$ or not. So, empty paths are disregarded.

Two effects:

- to insert all possible vertical links in local forests;
- to indicate how the search along each path terminates. This is represented by the value of Stop(u) and Stop(v).

Stop(u) is true: the current vertex u is within the cycle $C(i,j)$; hence, when the loop is over, Pred(u) is apex(i,j).
Stop(u) is false: the path on the side of i has been merged with a pre-existing oriented tree, rooted at apex(i,j) or above; hence, Pred(u) is out of $C(i,j)$.

Search, part III

Lines 12-13: a horizontal link is possibly inserted in the local forest of apex(i, j) by ProcessApex.

This is done only if the apex is different from i and j (i.e. both oriented paths exist) and if at least one of the two current vertices u and v is within the cycle $C(i, j)$.

If both u and v have reached the apex, then both the apex(i, j)-components of i and j are already connected with that of Pred(apex(i, j)) and therefore no horizontal link must be inserted in $\mathcal{T}$(apex(i, j)).

Procedure PathScan and vertical links

Algorithm 9 PathScan(i, j, w).

```
1:  Stop(w) ← true
2:  InitPath
3:  while ¬SubTree(Pred(w), i, j) do
4:      p ← Pred(w)
5:      if (Head(p, w)) ≠ Head(p, Pred(p)) then
6:          /* Insert a vertical link */
7:      if Path(p) = 0 then
8:          Path(p) ← π
9:      else
10:         /* Skip Path(p) up to its root */
11: PurgePath(π)
```

Insert a vertical link

Algorithm 10 Insert a vertical link.

1: $\mathcal{T}(p) \leftarrow \mathcal{T}(p) \cup \{([w, \text{Pred}(p)], [i, j])\}$
2: $\text{AltTreeCost}(p) \leftarrow \text{AltTreeCost}(p) + c(i, j)$
3: $\mu \leftarrow \mu - 1$
4: $\text{Merge}(p, w, \text{Pred}(p))$
5: $\text{TRoot}(\text{THead}(\pi)) \leftarrow p$
6: $w \leftarrow p$

Skip Path(p) up to its root

Algorithm 11 Skip the current path up to its root.

1: **if** TRoot(THead(π)) $\neq 0$ **then**
2: TreeMerge(π, Path(p))
3: $w \leftarrow$ TRoot(THead(Path(p)))
4: **if** SubTree(w, i, j)) **then**
5: Stop(w) $\leftarrow$ false

Procedure ProcessApex and horizontal links

The procedure ProcessApex(u, v, i, j), shown in Algorithm 12 and 13, inserts a horizontal link in the local forest of apex(i, j) if and only if the apex(i, j)-components of i and j are not already connected in $\mathcal{T}$(apex(i, j)). If they are already connected, the procedure has no effect.

To check whether the horizontal link can be inserted, it is necessary to know the indices of the two vertices adjacent to the apex along $P(i, j)$ and $P(j, i)$. These correspond to u and v when Stop(u) and Stop(v) are true: if Stop(u) $\wedge$ Stop(v), then Pred(u) $=$ Pred(v) $=$ apex(i, j).

Hence, the apex is found as the predecessor of the current vertex for which Stop is true.

If ProcessApex is executed, at least one of Stop(u) and Stop(v) is guaranteed to be true, owing to the test on line 12 of Search.

Procedure ProcessApex and horizontal links

If one of the two current vertices, say u, has been moved to the apex or above (Stop(u) is false), then the p-component of i is already connected with the p-component of Pred(p) in $\mathcal{T}(p)$. Hence a test on the local Union-Find data-structure of vertex p must be done to check whether v and Pred(p) are connected or not. Hence, a variable u' is set to u if Stop(u) is true and to Pred(p) if Stop(u) is false (lines 5-8). The same is done for v' (lines 9-12).

If the test on the Union-Find data-structure succeeds (line 1), then a horizontal link is inserted; otherwise the procedure terminates with no effect.

Procedure *Find*

Before inserting the horizontal link, it is necessary to find the index of both vertices adjacent to the apex p along $P(i, j)$ and $P(j, i)$. They are not both available if Stop is false for one of the two current vertices. Therefore, u or v is reset to the position just below the apex, in case it is not (lines 2-3 and 4-5). This is done by a procedure Find, that exploits the values of Up and Dn of all vertices adjacent to p.

Assume Stop(u) be false. The execution of Find(p, i) implies a search among the vertices of $\mathcal{V}(p)$ to find the vertex which lies on the path between i and p. This requires to find the (unique) vertex $u \in \mathcal{V}(p)$ such that $(\text{Dn}(u) \leq \text{Dn}(i)) \wedge (\text{Up}(u) \geq \text{Up}(i))$.

Procedure ProcessApex(u, v, i, j), part I

Algorithm 12 ProcessApex(u, v, i, j), part I.

1: **if** (Stop(u)) **then**
2: $p \leftarrow$ Pred(u)
3: **else**
4: $p \leftarrow$ Pred(v)
5: **if** (Stop(u)) **then**
6: $u' \leftarrow u$
7: **else**
8: $u' \leftarrow$ Pred(p)
9: **if** (Stop(v)) **then**
10: $v' \leftarrow v$
11: **else**
12: $v' \leftarrow$ Pred(p)

Procedure ProcessApex(u, v, i, j), part II

Algorithm 13 ProcessApex(u, v, i, j), part II.

1: **if** (Head(p, u') $\neq$ Head(p, v')) **then**
2: **if** $\neg$Stop(u) **then**
3: $u \leftarrow$ Find(p, i)
4: **if** $\neg$Stop(v) **then**
5: $v \leftarrow$ Find(p, j)
6: $\mathcal{T}(p) \leftarrow \mathcal{T}(p) \cup \{[u, v], [i, j]\}$
7: AltTreeCost(p) $\leftarrow$ AltTreeCost(p) $+ c(i, j)$
8: $\mu \leftarrow \mu - 1$
9: Merge(p, u, v)

Complexity of Search (1)

The number of iterations of the outer loop of Search (lines 2-14) is $O(m)$.

Therefore the time taken by all constant time operations in Search (including SubTree that takes constant time), PathScan and ProcessApex (including InitPath and PurgePath), yield an overall $O(m)$ contribution to the time complexity.

The loop on lines 3-5 can be implemented so that it takes $O(1)$ time per iteration, i.e. $O(m)$ overall.

This requires to sort the edges of T^* with the same criterion used to sort the edges in $\mathcal{S}$ (non-decreasing cost plus some additional lexicographic criterion to break ties).

This pre-sorting step requires $O(n \log n)$ because T^* contains $n-1$ edges.

Hence, the pre-sorting step and the loop on lines 3-5 require $O(m + n \log n)$ time.

Complexity of Search (2)

The block of operations on line 6 of PathScan is executed $O(n)$ times, because a vertical link is inserted each time and the number of possible vertical links is $O(n)$.

Since all operations excepted Merge have $O(1)$ complexity, their overall contribution to the time complexity of the algorithm is $O(n)$.

The test on line 7 of PathScan can succeed at most n times. Therefore the total contribution to the time complexity of the $O(1)$ operation on line 8 is $O(n)$.

Complexity of Search (3)

The block of operations on line 10 of PathScan takes $O(1)$ with the exception of the contribution of TreeMerge.

The complexity analysis requires to distinguish the case in which π is non-relevant (TRoot(THead(π)) $= 0$) from the case in which π is relevant (TRoot(THead(π)) > 0).

An empty path can exist only twice for each candidate alternative edge, once for each side of $C(i, j)$. Therefore the execution of this block with non-relevant π can occur $O(m)$ times. In these cases TreeMerge is not executed and thus the total contribution is $O(m)$.

The number of relevant paths in the Tree-Union-Find data-structure is $O(n)$, because all roots are different. Therefore, a call to TreeMerge can occur at most $O(n)$ times. Hence, the number of times the block is executed with relevant π is $O(n)$ and therefore the total contribution of these iterations is $O(n)$ plus the contribution of TreeMerge.

Complexity of Search (4)

The number of times the loop on lines 3-10 of PathScan is executed is the sum of both types of iterations, i.e. with relevant π and non-relevant π. Therefore the tests on lines 5 and 7 are executed $O(m+n)$ times.

The procedure ProcessApex is called at most once for each edge, i.e. $O(m)$ times. Therefore all $O(1)$ operations on lines 1-12 contribute $O(m)$.

The test on line 1 can succeed $O(n)$ times, because the number of horizontal links that can be inserted is $O(n)$. Excluding the contribution of Merge and Find, the total contribution of the operations on lines 2-9 to the time complexity is $O(n)$.

Complexity of Find

The execution of Find(p, i) implies a search among the vertices of $\mathcal{V}(p)$ (i.e. the neighbors of p in T^*) to find the vertex which lies on the path between p and i on T^*, i.e. the (unique) vertex u satisfying $(\mathrm{Dn}(u) \leq \mathrm{Dn}(i)) \wedge (\mathrm{Up}(u) \geq \mathrm{Up}(i))$.

This task can be accomplished in $O(\log d(p))$ (which is not worse than $O(\log n)$) by binary search, because the vertices of $\mathcal{V}(p)$ are sorted by their values of Dn and Up after the execution of Orient. Since the number of calls to Find is $O(n)$, as observed above, the overall contribution of Find to the total time complexity is $O(n \log n)$.

Procedure Merge

Algorithm 14 Merge(p, u, v).

if $\text{Card}(p, \text{Head}(p, u)) < \text{Card}(p, \text{Head}(p, v))$ **then**
  $k' \leftarrow v$
  $k'' \leftarrow u$
else
  $k' \leftarrow u$
  $k'' \leftarrow v$
for $k \in \mathcal{L}(p, \text{Head}(p, k''))$ **do**
  $\text{Head}(p, k) \leftarrow \text{Head}(p, k')$
$\text{Card}(p, \text{Head}(p, k')) \leftarrow \text{Card}(p, \text{Head}(p, k')) + \text{Card}(p, \text{Head}(p, k''))$
$\text{Card}(p, \text{Head}(p, k'')) \leftarrow 0$
$\mathcal{L}(p, \text{Head}(p, k')) \leftarrow \mathcal{L}(p, \text{Head}(p, k')) \cup \mathcal{L}(p, \text{Head}(p, k''))$

Complexity of Merge and TreeMerge

At each call of Merge two lists in the local subgraph $\mathcal{G}(p)$ of some vertex $p \in V$ are merged. This can happen $O(n)$ times overall, because $O(n)$ alternative edges must be found.

For the well-known property of the Union-Find data-structure: for each local graph with $d(p)$ vertices, the time taken by the update operations is $O(d(p) \log d(p))$. Summing up these contributions over all vertices results in an $O(n \log n)$ contribution to the complexity of the whole algorithm, because $\sum_{p \in V} d(p) \log d(p) \leq \sum_{p \in V} d(p) \log n = \log n \sum_{p \in V} d(p) = 2|T^*| \log n = 2(n-1) \log n$.

The total time taken by TreeMerge to merge trees is $O(n \log n)$, because of the properties of Union-Find: every time two or more trees are merged, their lists are merged so that the shortest one is appended to the longest one. This guarantees that no representative is updated more than $\log n$ times since the number of oriented trees is $O(n)$.

Remark. To achieve this, it is necessary to accept that the root of an oriented tree does not necessarily belong to its representative path.

Conclusion

The asymptotic worst-case time complexity of an algorithm to pre-compute an optimal set of alternative edges, so that a minimum cost spanning tree can be immediately restored in a graph if a vertex is deleted is $O(m + n \log n)$ which is the same of a single run of Kruskal algorithm (or Prim algorithm implemented with Fibonacci heaps).