Implementation of Thorup's Linear Time Algorithm for Undirected Single-Source Shortest Paths with Positive Integer Weights

Nick Prühs

Department of Computer Science, CAU Kiel

September 30, 2009

Introduction

Introduction

Thorup's algorithm

Introduction

Thorup's algorithm

Implementation details

Introduction

Thorup's algorithm

Implementation details

Performance

Introduction

The Single-Source Shortest Paths Problem

- in:

Introduction

The Single-Source Shortest Paths Problem

- in:
  - undirected, connected graph $G = (V, E)$ with $|V| = n$ vertices and $|E| = m$ edges

Introduction

The Single-Source Shortest Paths Problem

- in:
  - undirected, connected graph $G = (V, E)$ with $|V| = n$ vertices and $|E| = m$ edges
  - positive edge weight function $w : V \times V \to \mathbb{N}$ with $\forall (u, v) \notin E : w(u, v) = \infty$

Introduction

The Single-Source Shortest Paths Problem

- in:
  - undirected, connected graph $G = (V, E)$ with $|V| = n$ vertices and $|E| = m$ edges
  - positive edge weight function $w : V \times V \to \mathbb{N}$ with $\forall (u, v) \notin E : w(u, v) = \infty$
  - distinguished source vertex $s \in V$

Introduction

The Single-Source Shortest Paths Problem

- in:
  - undirected, connected graph $G = (V, E)$ with $|V| = n$ vertices and $|E| = m$ edges
  - positive edge weight function $w : V \times V \to \mathbb{N}$ with $\forall (u, v) \notin E : w(u, v) = \infty$
  - distinguished source vertex $s \in V$
- out: $d(v) = dist(v, s)$ for all other vertices $v \in V \setminus \{s\}$

Introduction

Dijkstra's algorithm

- proposed by Edsger W. Dijkstra [Dij59]

Introduction

Dijkstra's algorithm

- proposed by Edsger W. Dijkstra [Dij59]
- additional definitions:
  - set of *visited* vertices $S \subseteq V$
  - *super distance* $D(v) \geq d(v)$ for every vertex $v \in V$, with

$$D(v) = \begin{cases} d(v), & v \in S \\ \min_{u \in S}\{d(u) + w(u, v)\}, & v \notin S \end{cases}$$

Introduction

Dijkstra's algorithm

- proposed by Edsger W. Dijkstra [Dij59]
- additional definitions:
  - set of *visited* vertices $S \subseteq V$
  - *super distance* $D(v) \geq d(v)$ for every vertex $v \in V$, with
    $$D(v) = \begin{cases} d(v), & v \in S \\ \min_{u \in S}\{d(u) + w(u, v)\}, & v \notin S \end{cases}$$
- initialization:
  - $S = \{s\}$
  - $D(s) = d(s) = 0$
  - $\forall v \neq s : D(v) = w(s, v)$

Introduction

Dijkstra's algorithm

- proposed by Edsger W. Dijkstra [Dij59]
- additional definitions:
  - set of *visited* vertices $S \subseteq V$
  - *super distance* $D(v) \geq d(v)$ for every vertex $v \in V$, with
    $$D(v) = \begin{cases} d(v), & v \in S \\ \min_{u \in S}\{d(u) + w(u, v)\}, & v \notin S \end{cases}$$
- initialization:
  - $S = \{s\}$
  - $D(s) = d(s) = 0$
  - $\forall v \neq s : D(v) = w(s, v)$
- algorithm:
  1. while $S \neq V$: *visit* the vertex $v \notin S$ minimizing $D(v)$

Introduction

Dijkstra's algorithm

- proposed by Edsger W. Dijkstra [Dij59]
- additional definitions:
  - set of *visited* vertices $S \subseteq V$
  - *super distance* $D(v) \geq d(v)$ for every vertex $v \in V$, with
$$D(v) = \begin{cases} d(v), & v \in S \\ \min_{u \in S}\{d(u) + w(u,v)\}, & v \notin S \end{cases}$$
- initialization:
  - $S = \{s\}$
  - $D(s) = d(s) = 0$
  - $\forall v \neq s : D(v) = w(s,v)$
- algorithm:
  1. while $S \neq V$: *visit* the vertex $v \notin S$ minimizing $D(v)$
     - 1.1 move v to S, because $D(v) = d(v)$
     - 1.2 for all $(u,v) \in E$, decrease $D(u)$ to $D(v) + w(u,v)$, if the latter is less

Introduction

The problem: The bottleneck of Dijkstra's algorithm

- running time results from two operations:

Introduction

The problem: The bottleneck of Dijkstra's algorithm

- running time results from two operations:
  - `deleteMin`: find a vertex $v \notin S$ minimizing $D(v)$ (exactly $n - 1$ times)

Introduction

The problem: The bottleneck of Dijkstra's algorithm

- running time results from two operations:
  - `deleteMin`: find a vertex $v \notin S$ minimizing $D(v)$ (exactly $n - 1$ times)
  - `decreaseKey`: decrease $D(u)$ (at most m times)

Introduction

The problem: The bottleneck of Dijkstra's algorithm

- running time results from two operations:
  - `deleteMin`: find a vertex $v \notin S$ minimizing $D(v)$ (exactly $n-1$ times)
  - `decreaseKey`: decrease $D(u)$ (at most m times)
- naive implementation:
  - `deleteMin` in $O(n)$
  - `decreaseKey` in $O(1)$
  - total running time is $O(n^2 + m)$

Introduction

The problem: The bottleneck of Dijkstra's algorithm

- running time results from two operations:
  - `deleteMin`: find a vertex $v \notin S$ minimizing $D(v)$ (exactly $n-1$ times)
  - `decreaseKey`: decrease $D(u)$ (at most m times)
- naive implementation:
  - `deleteMin` in $O(n)$
  - `decreaseKey` in $O(1)$
  - total running time is $O(n^2 + m)$
- implementation with Fibonacci heaps [FT84]:
  - `deleteMin` has amortized running time $O(\log n)$
  - `decreaseKey` has running time $O(1)$
  - total running time is $O(n \log n + m)$

Introduction

The problem: The bottleneck of Dijkstra's algorithm

- running time results from two operations:
  - `deleteMin`: find a vertex $v \notin S$ minimizing $D(v)$ (exactly $n-1$ times)
  - `decreaseKey`: decrease $D(u)$ (at most m times)
- naive implementation:
  - `deleteMin` in $O(n)$
  - `decreaseKey` in $O(1)$
  - total running time is $O(n^2+m)$
- implementation with Fibonacci heaps [FT84]:
  - `deleteMin` has amortized running time $O(\log n)$
  - `decreaseKey` has running time $O(1)$
  - total running time is $O(n \log n + m)$
- linear time Dijkstra requires linear time sorting

Introduction

The problem: The bottleneck of Dijkstra's algorithm

- running time results from two operations:
  - `deleteMin`: find a vertex $v \notin S$ minimizing $D(v)$ (exactly $n-1$ times)
  - `decreaseKey`: decrease $D(u)$ (at most m times)
- naive implementation:
  - `deleteMin` in $O(n)$
  - `decreaseKey` in $O(1)$
  - total running time is $O(n^2 + m)$
- implementation with Fibonacci heaps [FT84]:
  - `deleteMin` has amortized running time $O(\log n)$
  - `decreaseKey` has running time $O(1)$
  - total running time is $O(n \log n + m)$
- linear time Dijkstra requires linear time sorting
- sorting using comparisons only requires $\Omega(n \log n)$ comparisons

Introduction

The solution: Avoiding the sorting bottleneck

Introduction

The solution: Avoiding the sorting bottleneck

- Thorup [Tho99] does not visit the vertices in order of increasing distance from s

Introduction

The solution: Avoiding the sorting bottleneck

- Thorup [Tho99] does not visit the vertices in order of increasing distance from s
- identifies vertex pairs that can be visited in any order, using a hierarchical bucketing structure

Introduction

The solution: Avoiding the sorting bottleneck

- Thorup [Tho99] does not visit the vertices in order of increasing distance from s
- identifies vertex pairs that can be visited in any order, using a hierarchical bucketing structure
- requires several other data structures to be computed before

Thorup's algorithm

Overview

- Thorup's algorithm inherits...

Thorup's algorithm

Overview

- Thorup's algorithm inherits...
  - ...the definition of G, V, E, n, m, w, d, s, S and D

Thorup's algorithm

Overview

- Thorup's algorithm inherits...
  - ...the definition of G, V, E, n, m, w, d, s, S and D
  - ...the initialization of S and D

Thorup's algorithm

Overview

- Thorup's algorithm inherits...
  - ...the definition of G, V, E, n, m, w, d, s, S and D
  - ...the initialization of S and D
  - ...*visiting* a vertex $v \in V$, which might decrease $D(u)$ for some adjacent vertices $u \in V$ and moves v to S

Thorup's algorithm

Overview

- Thorup's algorithm inherits...
  - ...the definition of G, V, E, n, m, w, d, s, S and D
  - ...the initialization of S and D
  - ...*visiting* a vertex $v \in V$, which might decrease $D(u)$ for some adjacent vertices $u \in V$ and moves v to S
- but Thorup allows visiting a vertex $v \notin S$ not minimizing $D(v)$

Thorup's algorithm

Overview

- Thorup's algorithm inherits...
  - ...the definition of G, V, E, n, m, w, d, s, S and D
  - ...the initialization of S and D
  - ...*visiting* a vertex $v \in V$, which might decrease $D(u)$ for some adjacent vertices $u \in V$ and moves v to S
- but Thorup allows visiting a vertex $v \notin S$ not minimizing $D(v)$
- in order to identify the next vertex to be visited, the vertices are placed in *buckets*

Thorup's algorithm

Overview

- Thorup's algorithm inherits...
  - ...the definition of G, V, E, n, m, w, d, s, S and D
  - ...the initialization of S and D
  - ...*visiting* a vertex $v \in V$, which might decrease $D(u)$ for some adjacent vertices $u \in V$ and moves v to S
- but Thorup allows visiting a vertex $v \notin S$ not minimizing $D(v)$
- in order to identify the next vertex to be visited, the vertices are placed in *buckets*
- every bucket is associated with a component, a node of the component tree explained later

Thorup's algorithm

The main routine

The whole algorithm can be summarized in top-level pseudo-code as follows:

Thorup's algorithm

The main routine

The whole algorithm can be summarized in top-level pseudo-code as follows:

1. Construct an *msb*-minimum spanning tree $\mathscr{M}$ in $O(m)$.

Thorup's algorithm

The main routine

The whole algorithm can be summarized in top-level pseudo-code as follows:

1. Construct an *msb*-minimum spanning tree $\mathscr{M}$ in $O(m)$.
2. Construct the component tree $\mathscr{T}$ in $O(m)$.

Thorup's algorithm

The main routine

The whole algorithm can be summarized in top-level pseudo-code as follows:

1. Construct an *msb*-minimum spanning tree $\mathscr{M}$ in $O(m)$.
2. Construct the component tree $\mathscr{T}$ in $O(m)$.
3. Construct the unvisited data structure $\mathscr{U}$ in $O(n)$.

Thorup's algorithm

The main routine

The whole algorithm can be summarized in top-level pseudo-code as follows:

1. Construct an *msb*-minimum spanning tree $\mathscr{M}$ in $O(m)$.
2. Construct the component tree $\mathscr{T}$ in $O(m)$.
3. Construct the unvisited data structure $\mathscr{U}$ in $O(n)$.
4. Set $S = \{s\}$.
5. Set $D(s) = 0$.
6. For all $v \in V$ with $v \neq s$: Set $D(v) = w(s, v)$.

Thorup's algorithm

The main routine

The whole algorithm can be summarized in top-level pseudo-code as follows:

1. Construct an *msb*-minimum spanning tree $\mathscr{M}$ in $O(m)$.
2. Construct the component tree $\mathscr{T}$ in $O(m)$.
3. Construct the unvisited data structure $\mathscr{U}$ in $O(n)$.
4. Set $S = \{s\}$.
5. Set $D(s) = 0$.
6. For all $v \in V$ with $v \neq s$: Set $D(v) = w(s, v)$.
7. Visit the root of the component tree $\mathscr{T}$.
8. Return D.

Thorup's algorithm

Constructing a minimum spanning tree in linear time?

- construction of a minimum spanning tree is possible in linear time [FW90]

Thorup's algorithm

Constructing a minimum spanning tree in linear time?

- construction of a minimum spanning tree is possible in linear time [FW90]
- this requires a priority queue called *atomic heap*, which requires $n > 2^{12^{20}}$

Thorup's algorithm

Constructing a minimum spanning tree in linear time?

- construction of a minimum spanning tree is possible in linear time [FW90]
- this requires a priority queue called *atomic heap*, which requires $n > 2^{12^{20}}$
- find a different way for today's computers

Thorup's algorithm

The *msb*-minimum spanning tree $\mathscr{M}$

- let $msb(x) = \lfloor \log_2 x \rfloor$ denote the position of the most significant bit of $x \in \mathbb{N}$

Thorup's algorithm

The *msb*-minimum spanning tree $\mathscr{M}$

- let $msb(x) = \lfloor \log_2 x \rfloor$ denote the position of the most significant bit of $x \in \mathbb{N}$
- an *msb-minimum spanning tree* of a graph G is a spanning tree that is minimal in G where each weight x is replaced by $msb(x)$

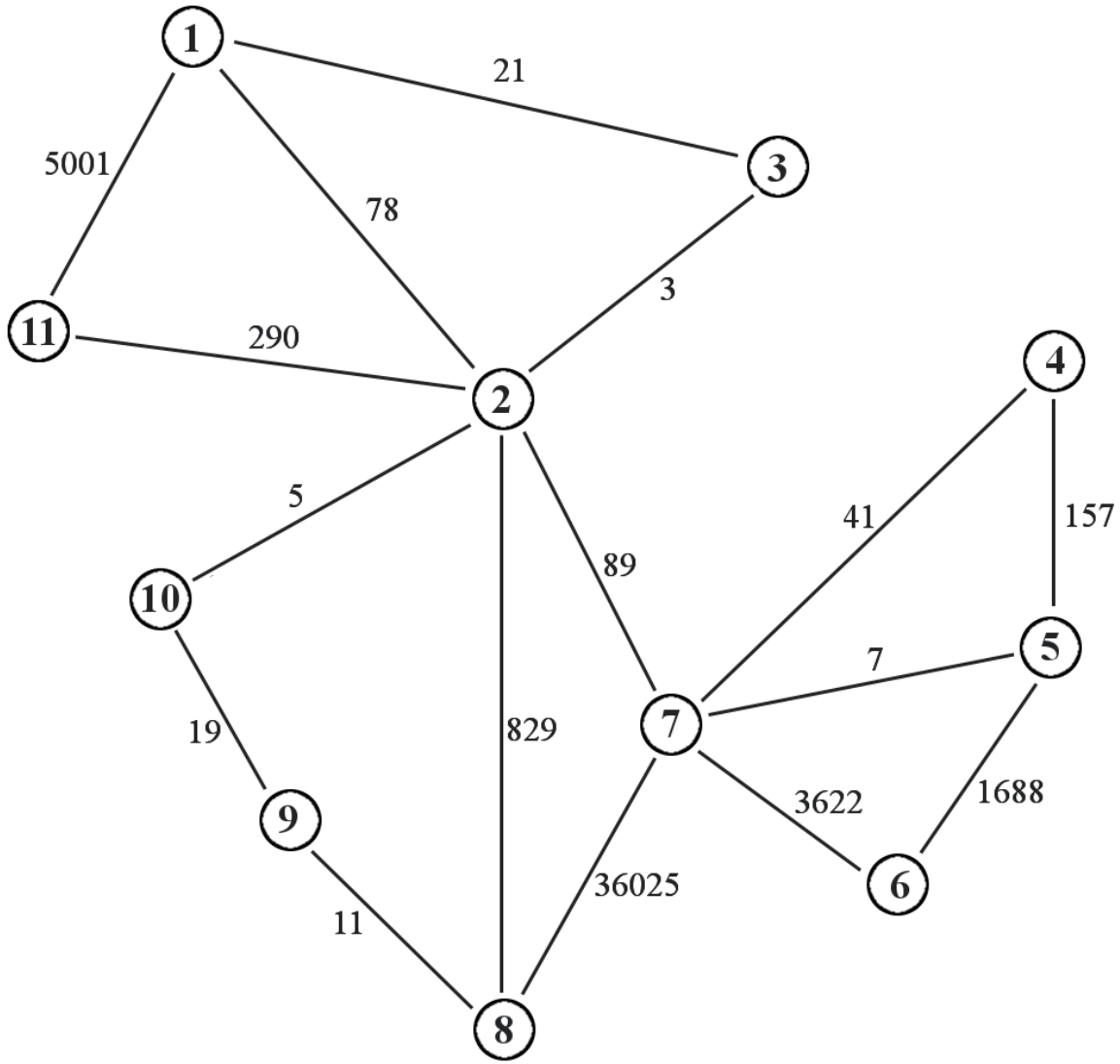
Thorup's algorithm

The *msb*-minimum spanning tree $\mathscr{M}$

- let $msb(x) = \lfloor \log_2 x \rfloor$ denote the position of the most significant bit of $x \in \mathbb{N}$
- an *msb-minimum spanning tree* of a graph G is a spanning tree that is minimal in G where each weight x is replaced by $msb(x)$
- use such an *msb*-minimum spanning tree for constructing the component tree

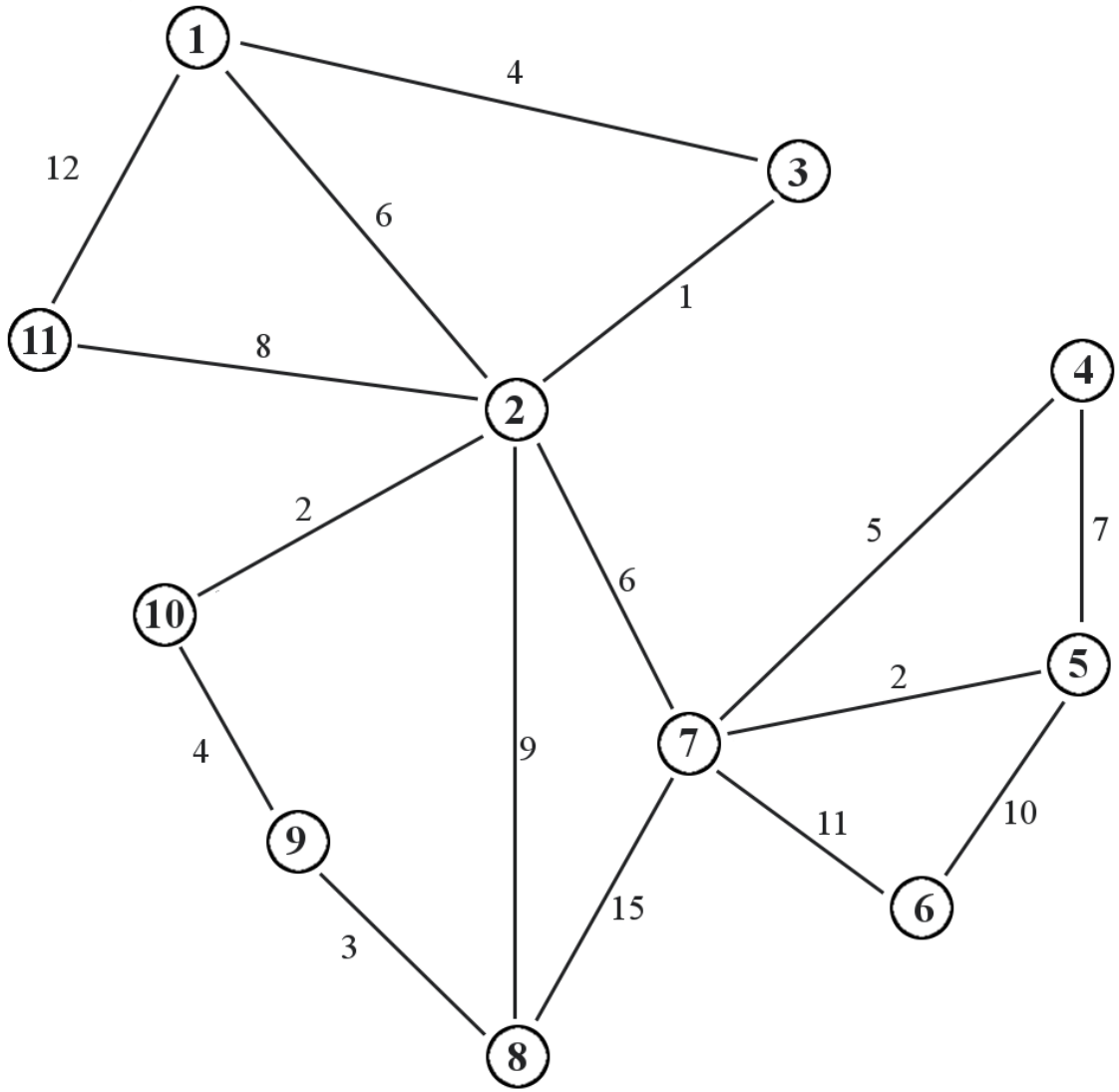
Thorup's algorithm

An example of an input graph G ($n = 11, m = 16$)



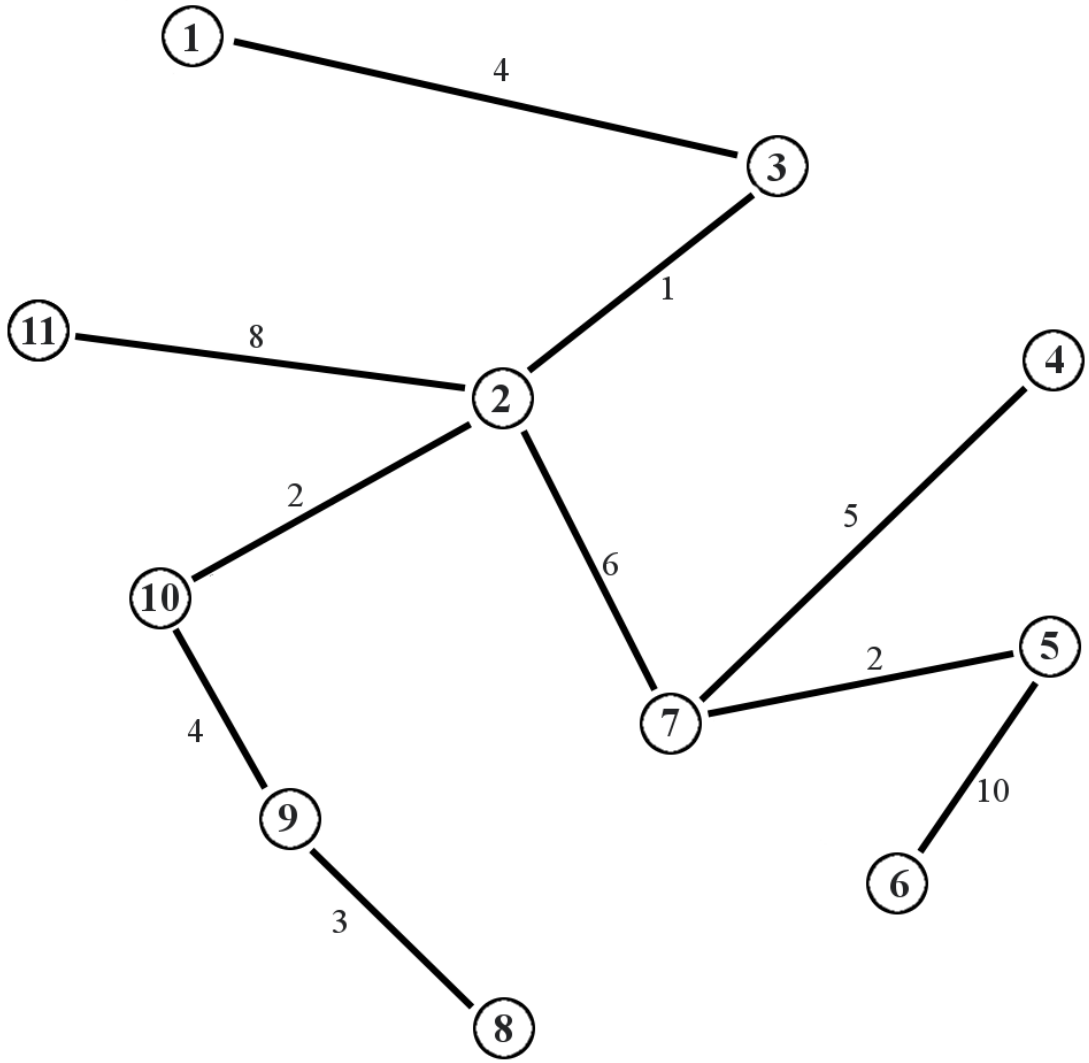
Thorup's algorithm

The input graph G after having replaced each weight x by $msb(x)$



Thorup's algorithm

An *msb*-minimum spanning tree of G



Thorup's algorithm

How to construct an *msb*-minimum spanning tree

1. sort all edges according to their *msb*-weights in linear time using simple bucketing

Thorup's algorithm

How to construct an *msb*-minimum spanning tree

1. sort all edges according to their *msb*-weights in linear time using simple bucketing
2. compute a minimum spanning tree with Kruskal's algorithm [Kru56], using the pre-sorted edges

Thorup's algorithm

How to construct an *msb*-minimum spanning tree

- the clustering is done with Tarjan's union-find algorithm [Tar75]

Thorup's algorithm

How to construct an *msb*-minimum spanning tree

- the clustering is done with Tarjan's union-find algorithm [Tar75]
- the use of *union with size*, and *find with path compression* leads to a time bound of $O(\alpha(m, n)m)$

Thorup's algorithm

The component hierarchy

- let G_i be the subgraph of G containing all edges $e \in E$ with $w(e) < 2^i$

Thorup's algorithm

The component hierarchy

- let G_i be the subgraph of G containing all edges $e \in E$ with $w(e) < 2^i$
- Level i of Thorup's *component hierarchy* consists of the *components* of G_i

Thorup's algorithm

The component hierarchy

- let G_i be the subgraph of G containing all edges $e \in E$ with $w(e) < 2^i$
- Level i of Thorup's *component hierarchy* consists of the *components* of G_i
  - use the *msb*-minimum spanning tree here

Thorup's algorithm

The component hierarchy

- let G_i be the subgraph of G containing all edges $e \in E$ with $w(e) < 2^i$
- Level i of Thorup's *component hierarchy* consists of the *components* of G_i
  - use the *msb*-minimum spanning tree here
- let $[v]_i$ denote the component on level i containing the vertex $v \in V$

Thorup's algorithm

The component hierarchy

- let G_i be the subgraph of G containing all edges $e \in E$ with $w(e) < 2^i$
- Level i of Thorup's *component hierarchy* consists of the *components* of G_i
  - use the *msb*-minimum spanning tree here
- let $[v]_i$ denote the component on level i containing the vertex $v \in V$
- the children of a component $[v]_i$ are all components $[w]_{i-1}$ with $[w]_i = [v]_i$, in other words with $w \in [v]_i$

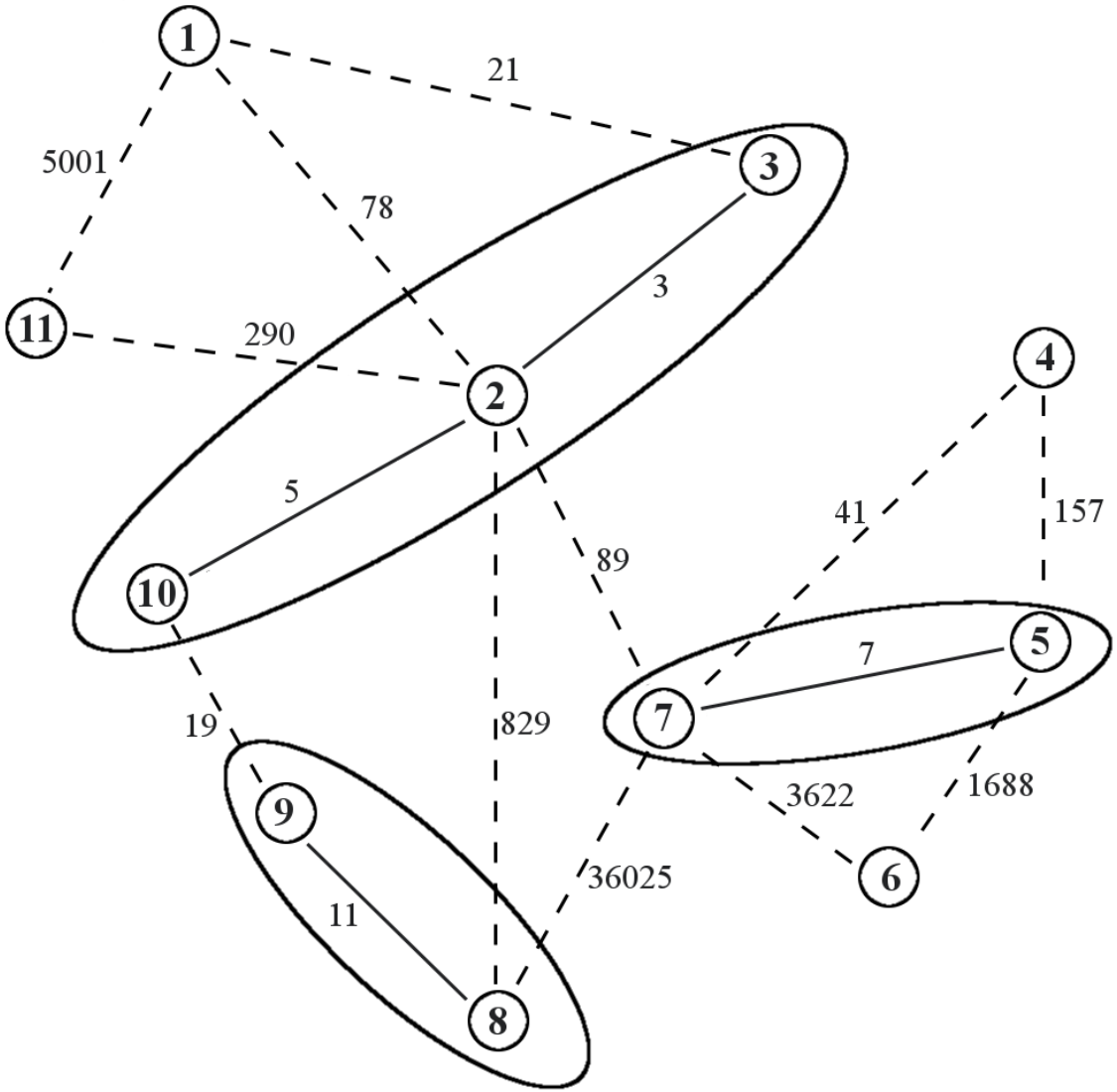
Thorup's algorithm

The component hierarchy

- let G_i be the subgraph of G containing all edges $e \in E$ with $w(e) < 2^i$
- Level i of Thorup's *component hierarchy* consists of the *components* of G_i
  - use the *msb*-minimum spanning tree here
- let $[v]_i$ denote the component on level i containing the vertex $v \in V$
- the children of a component $[v]_i$ are all components $[w]_{i-1}$ with $[w]_i = [v]_i$, in other words with $w \in [v]_i$
- $[v]_i \neq [w]_i \Rightarrow dist(v, w) \geq 2^i$

Thorup's algorithm

The subgraph G_4



Thorup's algorithm

The component tree $\mathscr{T}$

- the *component tree* $\mathscr{T}$ skips all nodes $[v]_i = [v]_{i-1}$:

Thorup's algorithm

The component tree $\mathscr{T}$

- the *component tree* $\mathscr{T}$ skips all nodes $[v]_i = [v]_{i-1}$:
  - Every *leaf* of $\mathscr{T}$ is a singleton component $[v]_0 = \{v\}$, $v \in V$.

Thorup's algorithm

The component tree $\mathscr{T}$

- the *component tree* $\mathscr{T}$ skips all nodes $[v]_i = [v]_{i-1}$:
  - Every *leaf* of $\mathscr{T}$ is a singleton component $[v]_0 = \{v\}$, $v \in V$.
  - Every *internal node* of $\mathscr{T}$ is a component $[v]_i$, $v \in V$, with $i > 0$ and $[v]_{i-1} \subsetneq [v]_i$.

Thorup's algorithm

The component tree $\mathscr{T}$

- the *component tree* $\mathscr{T}$ skips all nodes $[v]_i = [v]_{i-1}$:
  - Every *leaf* of $\mathscr{T}$ is a singleton component $[v]_0 = \{v\}$, $v \in V$.
  - Every *internal node* of $\mathscr{T}$ is a component $[v]_i$, $v \in V$, with $i > 0$ and $[v]_{i-1} \subsetneq [v]_i$.
  - The *root* of $\mathscr{T}$ is the node $[v]_r = G$ with r minimized.

Thorup's algorithm

The component tree $\mathscr{T}$

- the *component tree* $\mathscr{T}$ skips all nodes $[v]_i = [v]_{i-1}$:
  - Every *leaf* of $\mathscr{T}$ is a singleton component $[v]_0 = \{v\}$, $v \in V$.
  - Every *internal node* of $\mathscr{T}$ is a component $[v]_i$, $v \in V$, with $i > 0$ and $[v]_{i-1} \subsetneq [v]_i$.
  - The *root* of $\mathscr{T}$ is the node $[v]_r = G$ with r minimized.
- the parent of a node $[v]_i$ is its nearest ancestor in the component hierarchy with at least two children

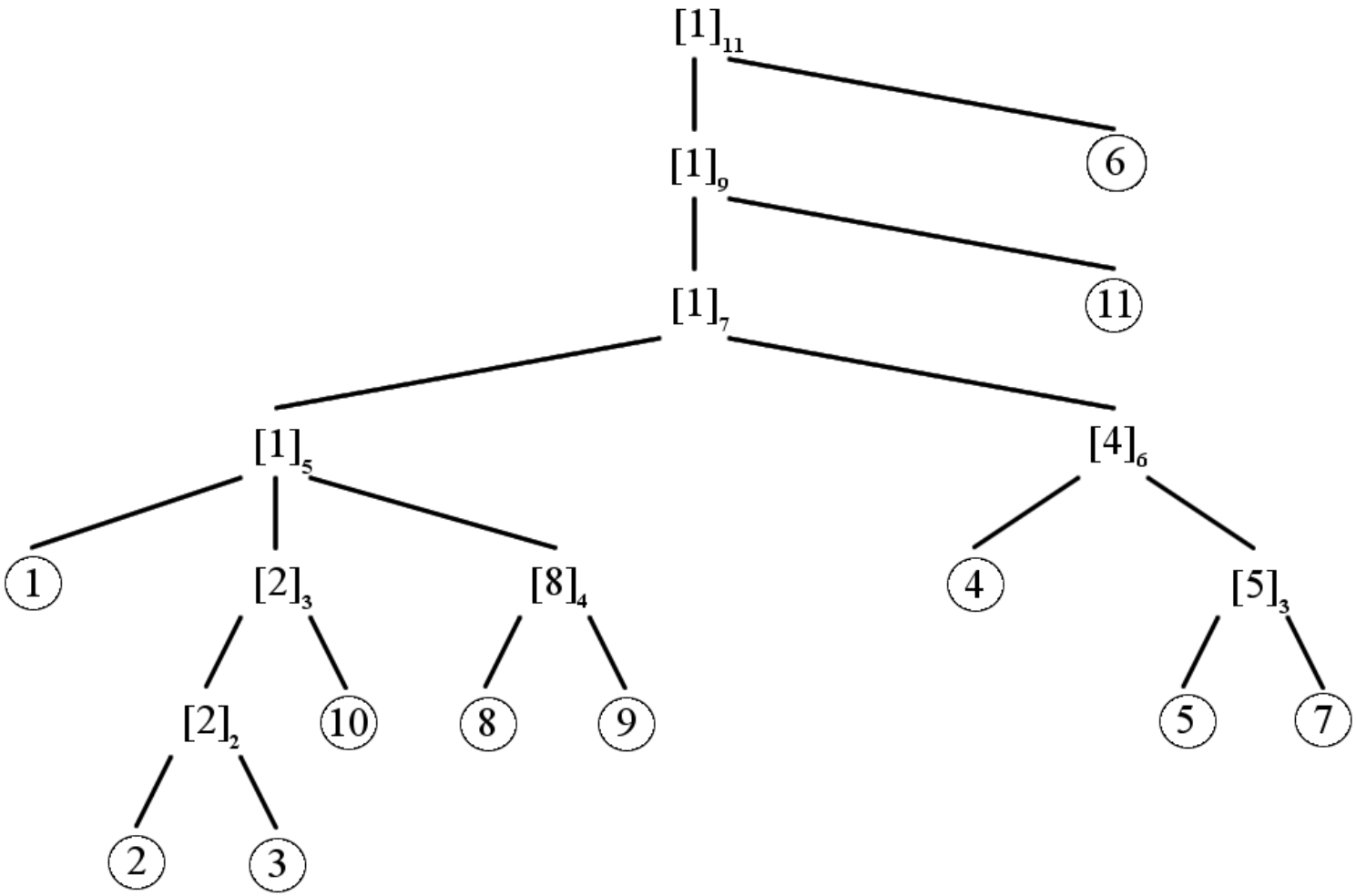
Thorup's algorithm

The component tree $\mathscr{T}$

- the *component tree* $\mathscr{T}$ skips all nodes $[v]_i = [v]_{i-1}$:
  - Every *leaf* of $\mathscr{T}$ is a singleton component $[v]_0 = \{v\}$, $v \in V$.
  - Every *internal node* of $\mathscr{T}$ is a component $[v]_i$, $v \in V$, with $i > 0$ and $[v]_{i-1} \subsetneq [v]_i$.
  - The *root* of $\mathscr{T}$ is the node $[v]_r = G$ with r minimized.
- the parent of a node $[v]_i$ is its nearest ancestor in the component hierarchy with at least two children
- $\mathscr{T}$ has no nodes with exactly one child $\Rightarrow$ the total number of nodes is bounded by $2n \in O(n)$

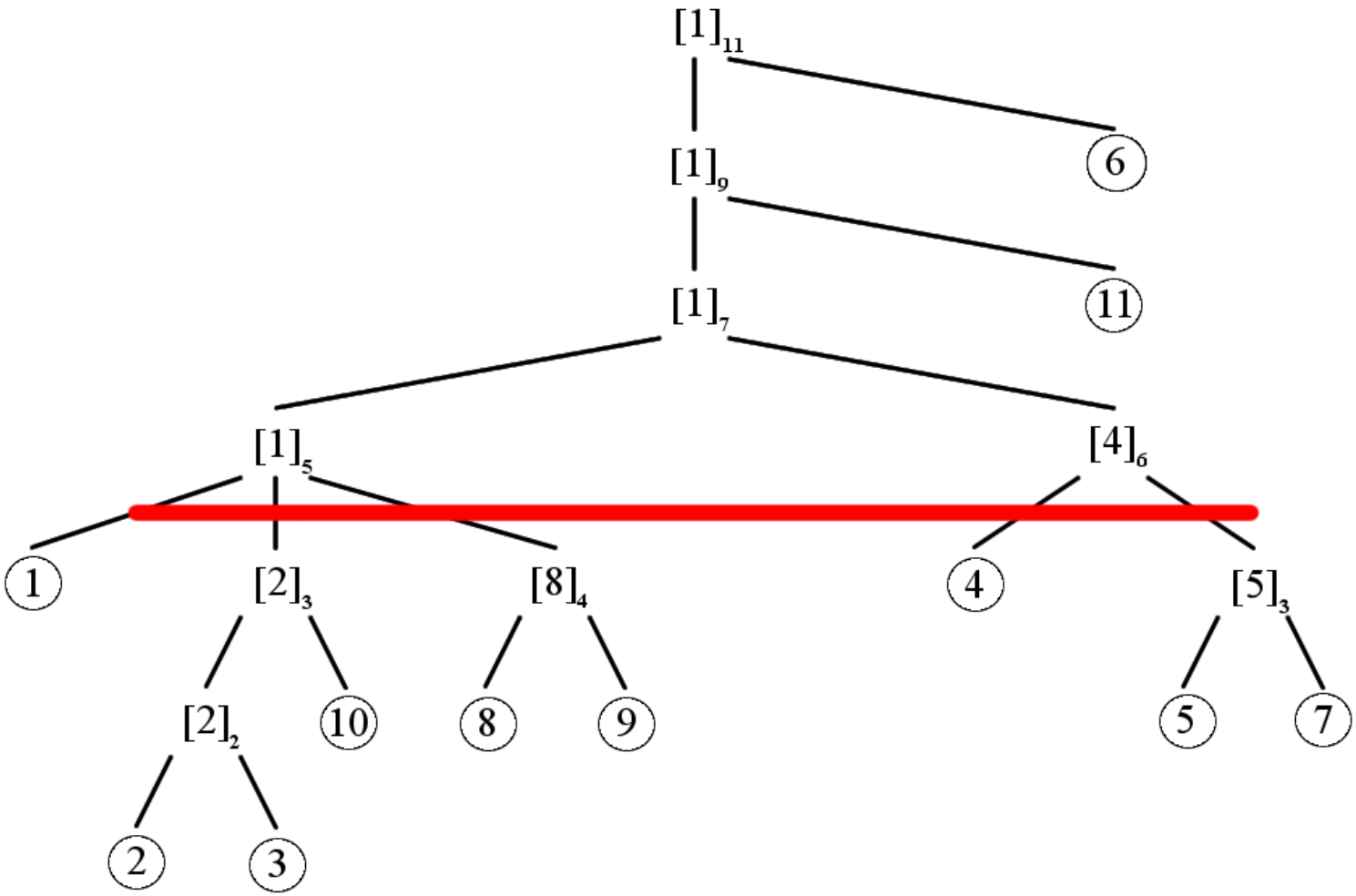
Thorup's algorithm

The component tree $\mathscr{T}$ of G



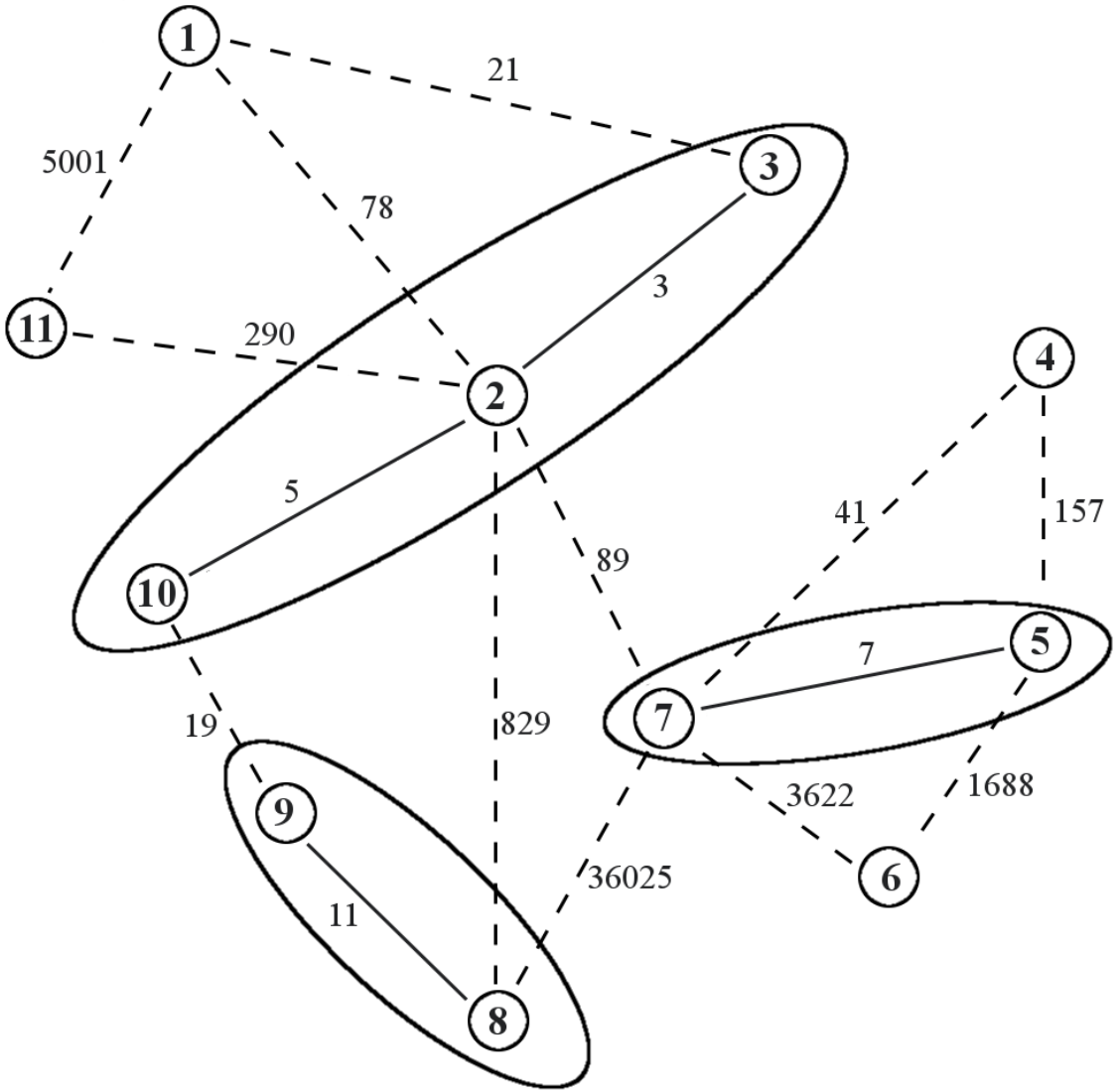
Thorup's algorithm

The components of G_4



Thorup's algorithm

The subgraph G_4



Thorup's algorithm

How to construct the component tree $\mathscr{T}$

1. sort the edges of $\mathscr{M}$ according to the most significant bits of their weights

Thorup's algorithm

How to construct the component tree $\mathscr{T}$

1. sort the edges of $\mathscr{M}$ according to the most significant bits of their weights
2. process the resulting sequence of edges $e_1, \ldots, e_{n-1}$ in the following way: For $i = 1$ to $n - 1$:

Thorup's algorithm

How to construct the component tree $\mathscr{T}$

1. sort the edges of $\mathscr{M}$ according to the most significant bits of their weights
2. process the resulting sequence of edges $e_1, \ldots, e_{n-1}$ in the following way: For $i = 1$ to $n - 1$:
   - 2.1 Let $(v, w) = e_i$.
   - 2.2 Call $union(v, w)$.
   - 2.3 If $msb(w(e_i)) < msb(w(e_{i+1}))$: Insert all new components of the union-find structure into $\mathscr{T}$.

Thorup's algorithm

How to construct the component tree $\mathscr{T}$

- using Tarjan's union-find algorithm [Tar75] again, the running time is $O(\alpha(m, n)m)$

Thorup's algorithm

How to construct the component tree $\mathscr{T}$

- using Tarjan's union-find algorithm [Tar75] again, the running time is $O(\alpha(m, n)m)$
- construction $\mathscr{T}$ in linear time is also possible:

Thorup's algorithm

How to construct the component tree $\mathscr{T}$

- using Tarjan's union-find algorithm [Tar75] again, the running time is $O(\alpha(m, n)m)$
- construction $\mathscr{T}$ in linear time is also possible:
  - requires the tabulation-based union-find algorithm by Gabow and Tarjan which runs in $O(m + n)$ time [GT85]

Thorup's algorithm

How to construct the component tree $\mathscr{T}$

- using Tarjan's union-find algorithm [Tar75] again, the running time is $O(\alpha(m, n)m)$
- construction $\mathscr{T}$ in linear time is also possible:
  - requires the tabulation-based union-find algorithm by Gabow and Tarjan which runs in $O(m + n)$ time [GT85]
  - much more complicated than the one by Tarjan
  - we use the simpler one here

Thorup's algorithm

The bucketing structure $\mathscr{B}$

- visit the nodes of the component tree $\mathscr{T}$ in the right order

Thorup's algorithm

The bucketing structure $\mathscr{B}$

- visit the nodes of the component tree $\mathscr{T}$ in the right order
- whenever a component $[v]_i$ is visited, so are all its ancestors in $\mathscr{T}$

Thorup's algorithm

The bucketing structure $\mathscr{B}$

- visit the nodes of the component tree $\mathscr{T}$ in the right order
- whenever a component $[v]_i$ is visited, so are all its ancestors in $\mathscr{T}$
- bucket the children $[w]_h$ of a component $[v]_i$ into $B([v]_i, \min D([w]_h^-) \gg i - 1)$

Thorup's algorithm

The bucketing structure $\mathscr{B}$

- visit the nodes of the component tree $\mathscr{T}$ in the right order
- whenever a component $[v]_i$ is visited, so are all its ancestors in $\mathscr{T}$
- bucket the children $[w]_h$ of a component $[v]_i$ into $B([v]_i, \min D([w]_h^-) \gg i - 1)$
- maintain two additional properties for every component:
  - $ix([v]_i) \leq$ the smallest index of a nonempty bucket of $[v]_i$
  - $\Delta([v]_i) =$ number buckets of $[v]_i$

Thorup's algorithm

The bucketing structure $\mathscr{B}$

- visit the nodes of the component tree $\mathscr{T}$ in the right order
- whenever a component $[v]_i$ is visited, so are all its ancestors in $\mathscr{T}$
- bucket the children $[w]_h$ of a component $[v]_i$ into $B([v]_i, \min D([w]_h^-) \gg i - 1)$
- maintain two additional properties for every component:
  - $ix([v]_i) \leq$ the smallest index of a nonempty bucket of $[v]_i$
  - $\Delta([v]_i) =$ number buckets of $[v]_i$
- the total number of buckets is bounded by $8n$

Thorup's algorithm

The unvisited data structure $\mathscr{U}$

- The *unvisited data structure* $\mathscr{U}$ ...

Thorup's algorithm

The unvisited data structure $\mathscr{U}$

- The *unvisited data structure* $\mathscr{U}$ . . .
  - . . . represents the unvisited subforest of the component tree $\mathscr{T}$

Thorup's algorithm

The unvisited data structure $\mathscr{U}$

- The *unvisited data structure* $\mathscr{U}$ ...
  - ... represents the unvisited subforest of the component tree $\mathscr{T}$
  - ... is required for maintaining the changing values $\min D([v]_i^-)$ for the changing set of roots $[v]_i$ in the unvisited part of $\mathscr{T}$ in linear total time

Thorup's algorithm

The unvisited data structure $\mathscr{U}$

- The *unvisited data structure* $\mathscr{U}$ ...
  - ... represents the unvisited subforest of the component tree $\mathscr{T}$
  - ... is required for maintaining the changing values $\min D([v]_i^-)$ for the changing set of roots $[v]_i$ in the unvisited part of $\mathscr{T}$ in linear total time
- $[v]_i$ is a root of a tree in $\mathscr{U}$ if and only if $[v]_i$ is an unvisited child of a visited component in $\mathscr{T}$

Thorup's algorithm

The unvisited data structure $\mathscr{U}$

- The *unvisited data structure* $\mathscr{U}$ ...
  - ... represents the unvisited subforest of the component tree $\mathscr{T}$
  - ... is required for maintaining the changing values $\min D([v]_i^-)$ for the changing set of roots $[v]_i$ in the unvisited part of $\mathscr{T}$ in linear total time
- $[v]_i$ is a root of a tree in $\mathscr{U}$ if and only if $[v]_i$ is an unvisited child of a visited component in $\mathscr{T}$
- $[v]_i = [v]_i^-$ for each of these roots, because they are unvisited

Thorup's algorithm

The problem: Operations of the unvisited data structure

- two operations in amortized constant time:
  1. Update $\min D([v]_i)$ whenever $D(v)$ is decreased for some vertex $v \in V$ with unvisited root $[v]_i$.
  2. Turn all children $[w]_h$ of $[v]_i$ in $\mathscr{T}$ into roots in $\mathscr{U}$ and compute $\min D([w]_h^-)$ for all of them whenever an unvisited root $[v]_i$ is visited.

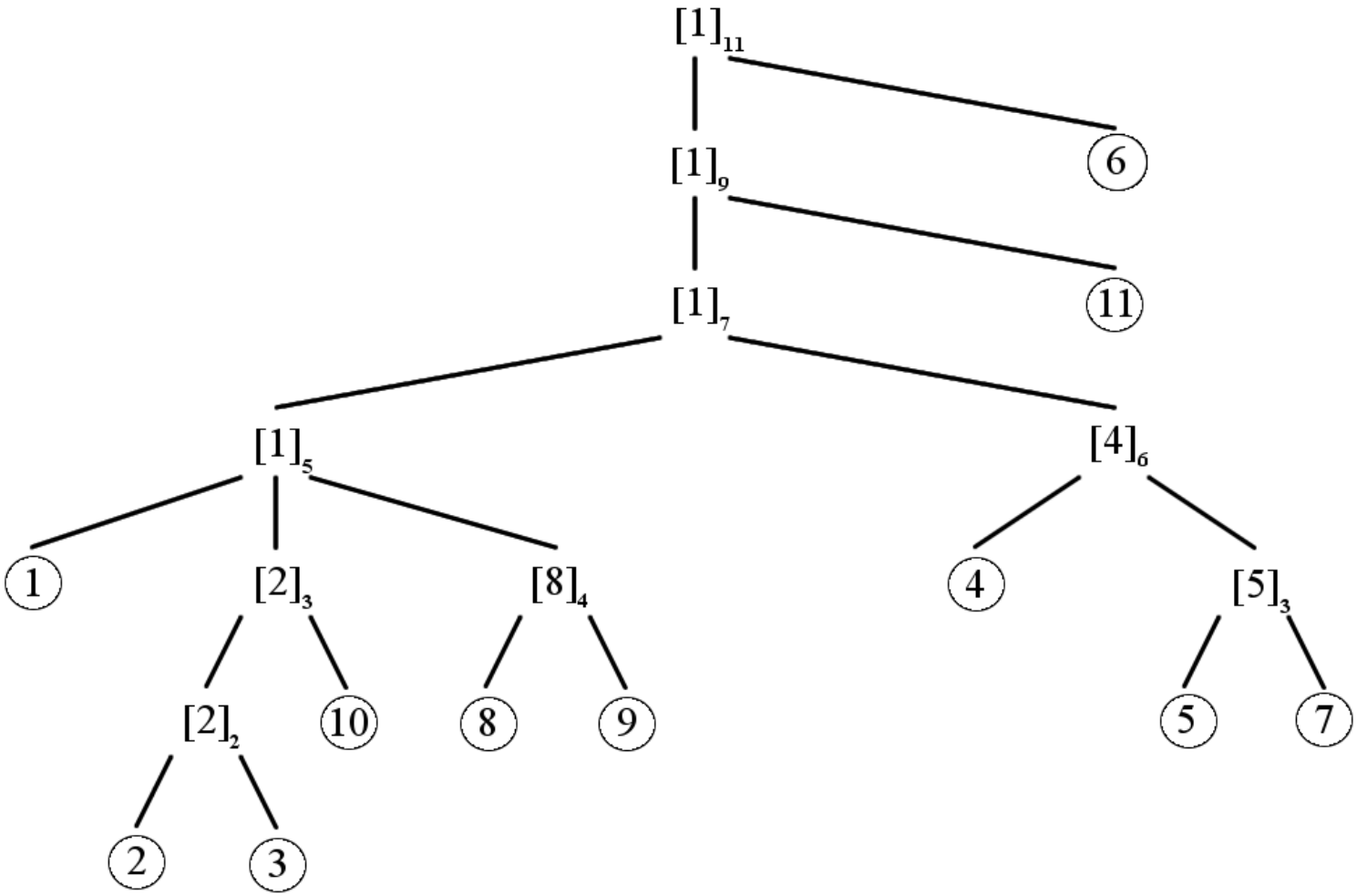
Thorup's algorithm

The problem: Operations of the unvisited data structure

- two operations in amortized constant time:
  1. Update min $D([v]_i)$ whenever $D(v)$ is decreased for some vertex $v \in V$ with unvisited root $[v]_i$.
  2. Turn all children $[w]_h$ of $[v]_i$ in $\mathscr{T}$ into roots in $\mathscr{U}$ and compute min $D([w]_h^-)$ for all of them whenever an unvisited root $[v]_i$ is visited.
- transform this problem into another one that can be solved by the *split-findmin* structure by Gabow [Gab85]

Thorup's algorithm

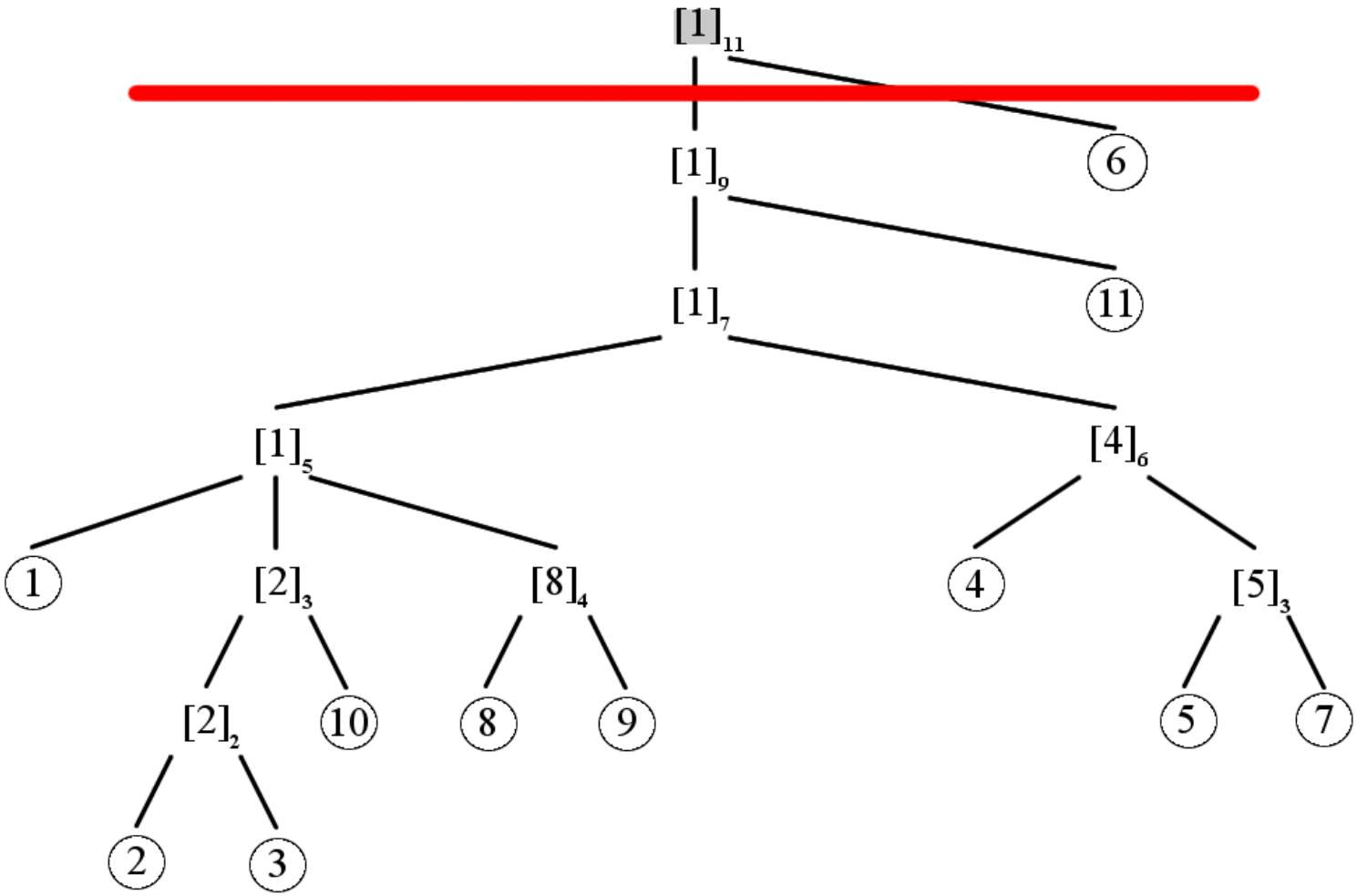
The initial unvisited data structure $\mathscr{U}$ of G



[1, 2, 3, 10, 8, 9, 4, 5, 7, 11, 6]

Thorup's algorithm

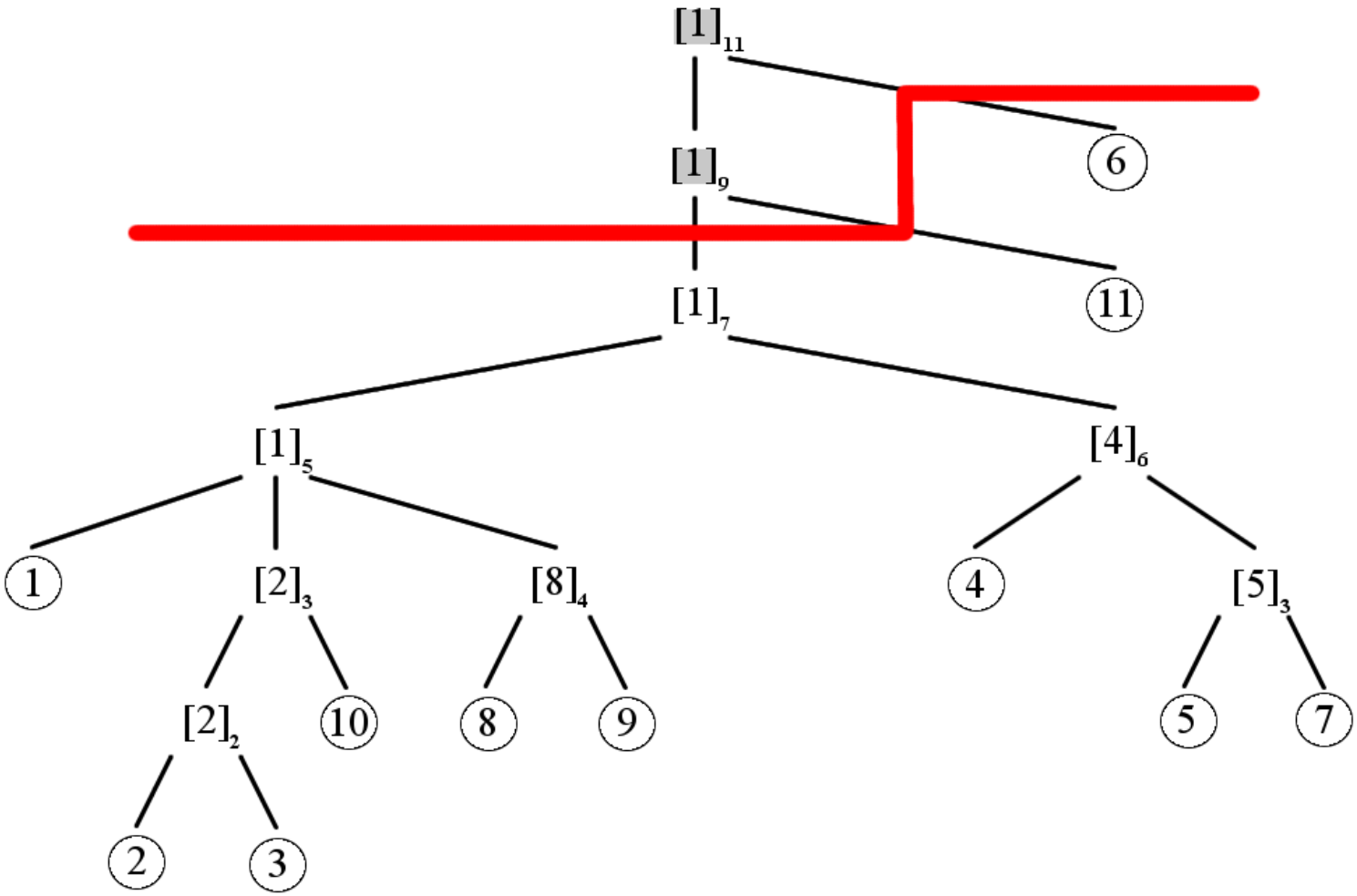
The unvisited data structure $\mathscr{U}$ after having called *visit*($[1]_{11}$)



[1, 2, 3, 10, 8, 9, 4, 5, 7, 11] [6]

Thorup's algorithm

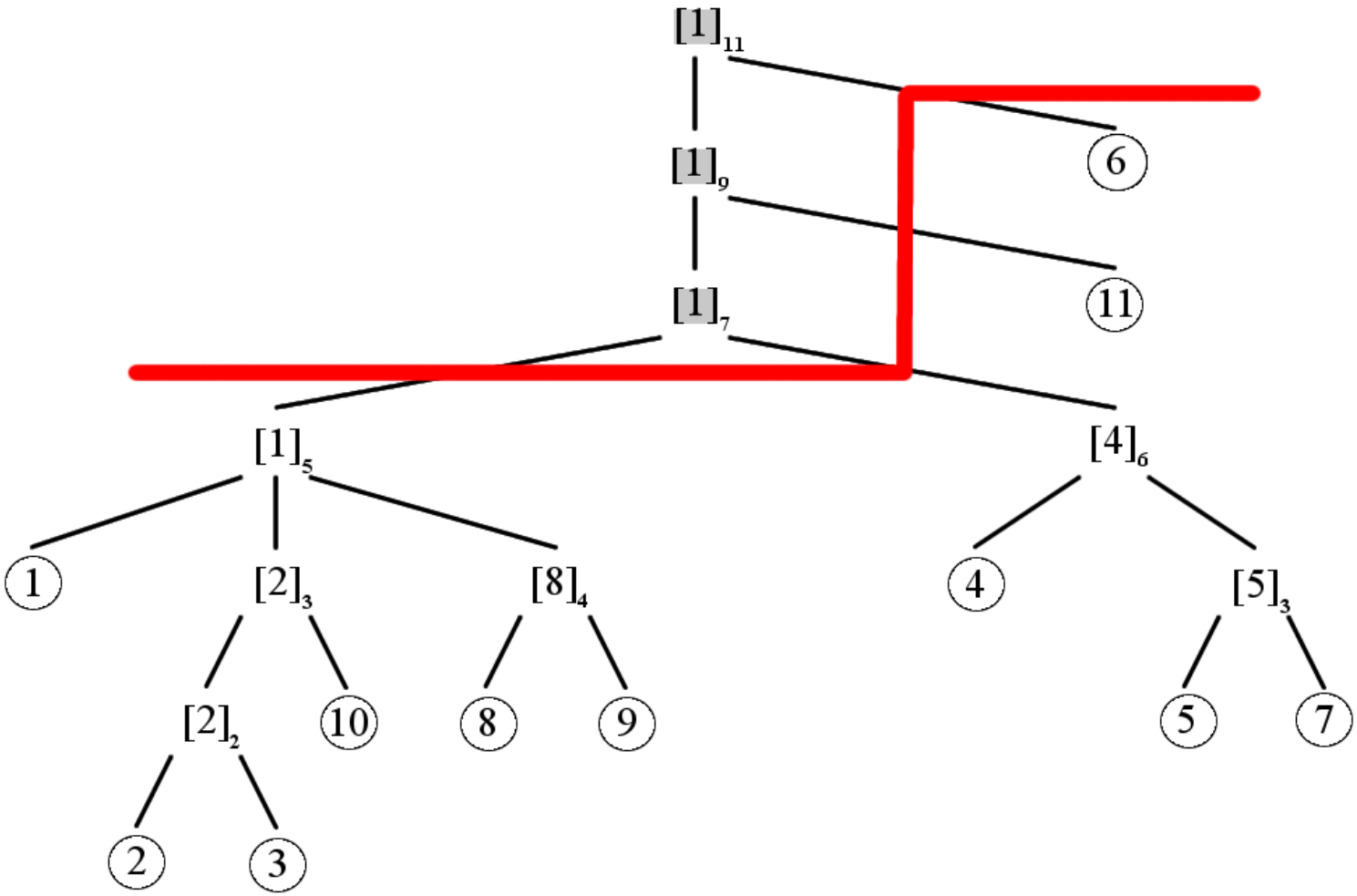
The unvisited data structure $\mathscr{U}$ after having called *visit*$([1]_9)$



[1, 2, 3, 10, 8, 9, 4, 5, 7] [11] [6]

Thorup's algorithm

The unvisited data structure $\mathscr{U}$ after having called *visit*$([1]_7)$



[1, 2, 3, 10, 8, 9] [4, 5, 7] [11] [6]

Thorup's algorithm

Total running time of the operations of the unvisited data structure

- for $k \in \mathbb{N}$, $k > 2$, each split in k can be implemented by $k - 1$ splits in two

Thorup's algorithm

Total running time of the operations of the unvisited data structure

- for $k \in \mathbb{N}$, $k > 2$, each split in k can be implemented by $k - 1$ splits in two
- at most $n - 1$ splits in two, because $|V| = n$

Thorup's algorithm

Total running time of the operations of the unvisited data structure

- for $k \in \mathbb{N}$, $k > 2$, each split in k can be implemented by $k - 1$ splits in two
- at most $n - 1$ splits in two, because $|V| = n$
- at most m decreases, one for each edge in G

Thorup's algorithm

Total running time of the operations of the unvisited data structure

- for $k \in \mathbb{N}$, $k > 2$, each split in k can be implemented by $k - 1$ splits in two
- at most $n - 1$ splits in two, because $|V| = n$
- at most m decreases, one for each edge in G
- Gabow's *split-findmin* data structure supports exactly these two operations

Thorup's algorithm

Total running time of the operations of the unvisited data structure

- for $k \in \mathbb{N}$, $k > 2$, each split in k can be implemented by $k - 1$ splits in two
- at most $n - 1$ splits in two, because $|V| = n$
- at most m decreases, one for each edge in G
- Gabow's *split-findmin* data structure supports exactly these two operations
- the total runinng time for $n - 1$ splits and m decreases is $O(\alpha(m, n)m)$

Thorup's algorithm

Operations of the unvisited data structure in linear total time?

- Thorup presented an $O(m+n)$ solution [Tho99]), which is based on the atomic heaps by Fredman and Willard

Thorup's algorithm

Operations of the unvisited data structure in linear total time?

- Thorup presented an $O(m+n)$ solution [Tho99]), which is based on the atomic heaps by Fredman and Willard
- as mentioned above, these priority queues require $n > 2^{12^{20}}$

Thorup's algorithm

Operations of the unvisited data structure in linear total time?

- Thorup presented an $O(m+n)$ solution [Tho99]), which is based on the atomic heaps by Fredman and Willard
- as mentioned above, these priority queues require $n > 2^{12^{20}}$
- fall back to the solution by Gabow and to a total running time of $O(\alpha(m,n)m)$

Thorup's algorithm

The main routine revisited

1. Construct an *msb*-minimum spanning tree $\mathscr{M}$ in $O(m)$. *(done)*

Thorup's algorithm

The main routine revisited

1. Construct an *msb*-minimum spanning tree $\mathscr{M}$ in $O(m)$. *(done)*
2. Construct the component tree $\mathscr{T}$ in $O(m)$. *(done)*

Thorup's algorithm

The main routine revisited

1. Construct an *msb*-minimum spanning tree $\mathscr{M}$ in $O(m)$. *(done)*
2. Construct the component tree $\mathscr{T}$ in $O(m)$. *(done)*
3. Construct the unvisited data structure $\mathscr{U}$ in $O(n)$. *(done)*

Thorup's algorithm

The main routine revisited

1. Construct an *msb*-minimum spanning tree $\mathscr{M}$ in $O(m)$. *(done)*
2. Construct the component tree $\mathscr{T}$ in $O(m)$. *(done)*
3. Construct the unvisited data structure $\mathscr{U}$ in $O(n)$. *(done)*
4. Set $S = \{s\}$. *(done)*
5. Set $D(s) = 0$. *(done)*
6. For all $v \in V$ with $v \neq s$: Set $D(v) = w(s, v)$. *(done)*

Thorup's algorithm

The main routine revisited

1. Construct an *msb*-minimum spanning tree $\mathscr{M}$ in $O(m)$. *(done)*
2. Construct the component tree $\mathscr{T}$ in $O(m)$. *(done)*
3. Construct the unvisited data structure $\mathscr{U}$ in $O(n)$. *(done)*
4. Set $S = \{s\}$. *(done)*
5. Set $D(s) = 0$. *(done)*
6. For all $v \in V$ with $v \neq s$: Set $D(v) = w(s, v)$. *(done)*
7. Visit the root of the component tree $\mathscr{T}$.
8. Return D.

Thorup's algorithm

Visit($[v]_i$): Step 1

If $[v]_i$ is the root of $\mathscr{T}$,

1. then: Set $j = \omega + 1$.
2. else: Let $[v]_j$ be the parent of $[v]_i$ in $\mathscr{T}$.

Thorup's algorithm

Visit($[v]_i$): Step 2

If $i = 0$:

1. Add v to S.
2. For all edges $(u, v) \in E$:
   - 2.1 If $D(v) + w(u, v) < D(w)$:
     - 2.1.1 Let $[u]_h$ be the unvisited root of $[u]_0$ in $\mathscr{U}$.
     - 2.1.2 Let $[u]_i$ be the visited parent of $[u]_h$ in $\mathscr{T}$.
     - 2.1.3 Set $oldMin = \min D([u]_h^-) \gg i - 1$.
     - 2.1.4 Decrease $D(u)$ to $D(v) + w(u, v)$.
     - 2.1.5 If $\min D([u]_h^-) \gg i - 1 < oldMin$: Move $[u]_h$ to bucket $B([u]_i, \min D([u]_h) \gg i - 1)$.
3. Remove $[v]_i$ from its bucket of $[v]_j$.
4. Return.

Thorup's algorithm

Visit($[v]_i$): Step 3

If $[v]_i$ is visited for the first time:

1. Construct the $\Delta([v]_i)$ buckets of $[v]_i$.
2. Delete $[v]_i$ from $\mathscr{U}$, turning its children into roots in $\mathscr{U}$.
3. For all children $[w]_h$ of $[v]_i$:
   - 3.1 Bucket $[w]_h$ in $B([v]_i, \min D([w]_h^-) \gg i - 1)$.

Thorup's algorithm

Visit($[v]_i$): Steps 4-6

- Set $oldIndex = ix([v]_i) \gg j - i$.

Thorup's algorithm

Visit($[v]_i$): Steps 4-6

- Set $oldIndex = ix([v]_i) \gg j - i$.
- While $[v]_i^- \neq \emptyset$ and $oldIndex = ix([v]_i) \gg j - i$:
  1. While $B([v]_i, ix([v]_i)) \neq \emptyset$:
     1.1 Let $[w]_h \in B([v]_i, ix([v]_i))$.
     1.2 Call $visit([w]_h)$.
  2. Increment $ix([v]_i)$.

Thorup's algorithm

Visit($[v]_i$): Steps 4-6

- Set $oldIndex = ix([v]_i) \gg j - i$.
- While $[v]_i^- \neq \emptyset$ and $oldIndex = ix([v]_i) \gg j - i$:
  1. While $B([v]_i, ix([v]_i)) \neq \emptyset$:
     1.1 Let $[w]_h \in B([v]_i, ix([v]_i))$.
     1.2 Call $visit([w]_h)$.
  2. Increment $ix([v]_i)$.
- If $[v]_i^- = \emptyset$,
  1. then: If $[v]_i$ is not the root of $\mathscr{T}$, remove it from its bucket of $[v]_j$.
  2. else: Move $[v]_i$ to bucket $B([v]_j, ix([v]_i) \gg j - i)$.

Implementation details

The strategy pattern

- Java implementation:
  - imperative programming language
  - fits the required RAM model
  - object-oriented
  - word length $\omega = 32$, number of vertices $n \leq 2^{32}$
  - allows the extensive use of the *strategy pattern*

Implementation details

Implemented data structures

- an undirected, weighted graph using adjacency lists
- an array priority queue
- a Fibonacci heap [FT84]
- a split-findmin structure [GT85]
- a union-find structure [Tar75]

Implementation details

Implemented algorithms

- Kruskal [Kru56]
- Dijkstra [Dij59]
- Thorup [Tho99]

Performance

The performance tests

- average of five passes

Performance

The performance tests

- average of five passes
- the test system:
  - Intel Core 2 Duo E6300 at 1,86 GHz
  - 2048 MB DDR2 667 PC2-5300 RAM
  - Java 1.6.0_15

Performance

Varying the number of vertices

- start with $n = 1000$ and increase it in steps of 1000 until $n = 10000$
- about $m = 5n$ edges
- edge weights $1 \leq w \leq 100000$

Performance

Varying the number of vertices: Results

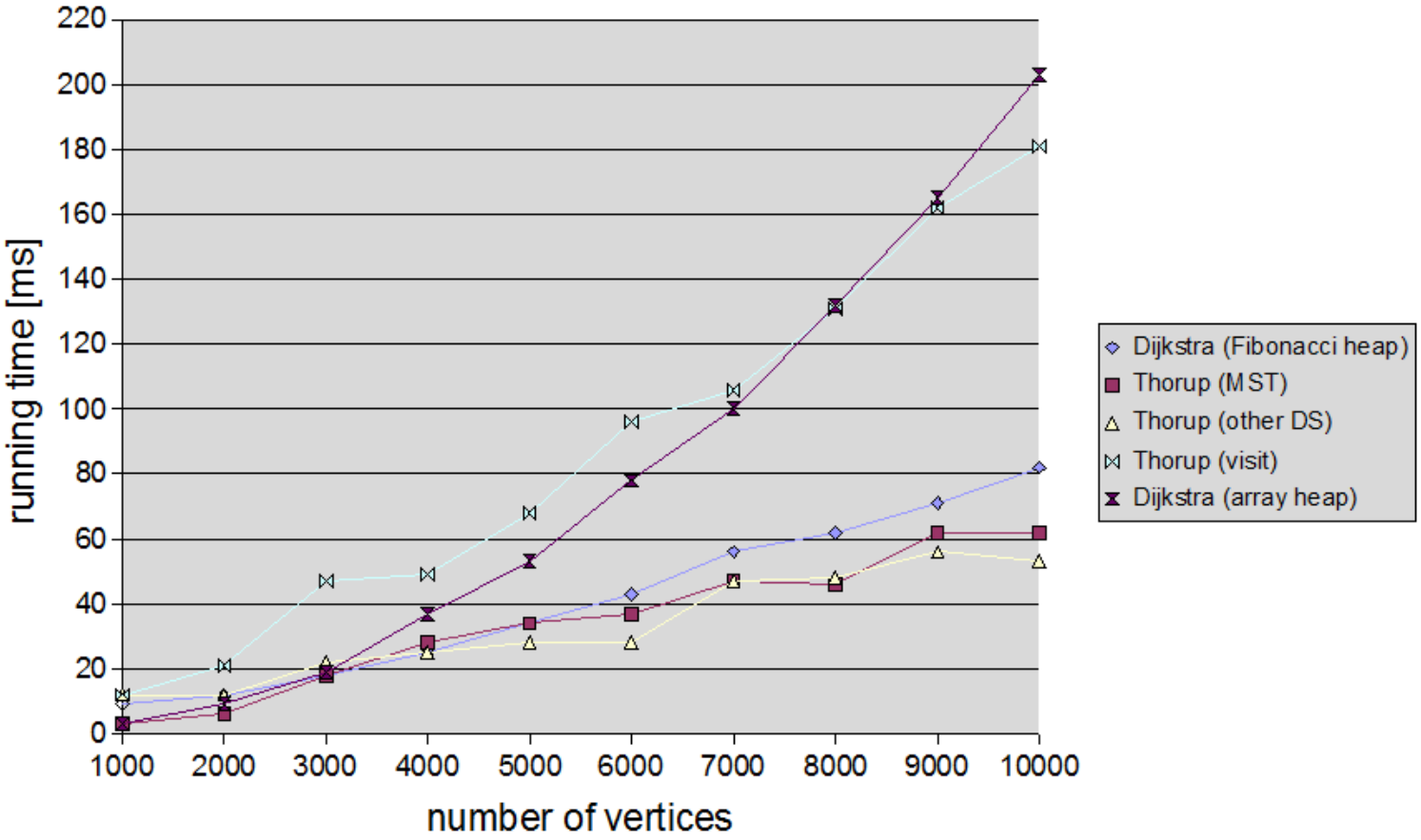


Figure: Running times for $1000 \leq n \leq 10000, m = 5n$.

Performance

Varying the number of vertices

- start with $n = 4000$ and increase it in steps of 4000 until $n = 40000$
- about $m = 5n$ edges again
- edge weights $1 \leq w \leq 100000$ again

Performance

Varying the number of vertices

- start with $n = 4000$ and increase it in steps of 4000 until $n = 40000$
- about $m = 5n$ edges again
- edge weights $1 \leq w \leq 100000$ again
- leave out the naive implementation of Dijkstra's algorithm and focus on the faster one

Performance

Varying the number of vertices

- start with $n = 4000$ and increase it in steps of 4000 until $n = 40000$
- about $m = 5n$ edges again
- edge weights $1 \leq w \leq 100000$ again
- leave out the naive implementation of Dijkstra's algorithm and focus on the faster one
- find out which n is required for Thorup to catch up with Dijkstra

Performance

Varying the number of vertices: Results

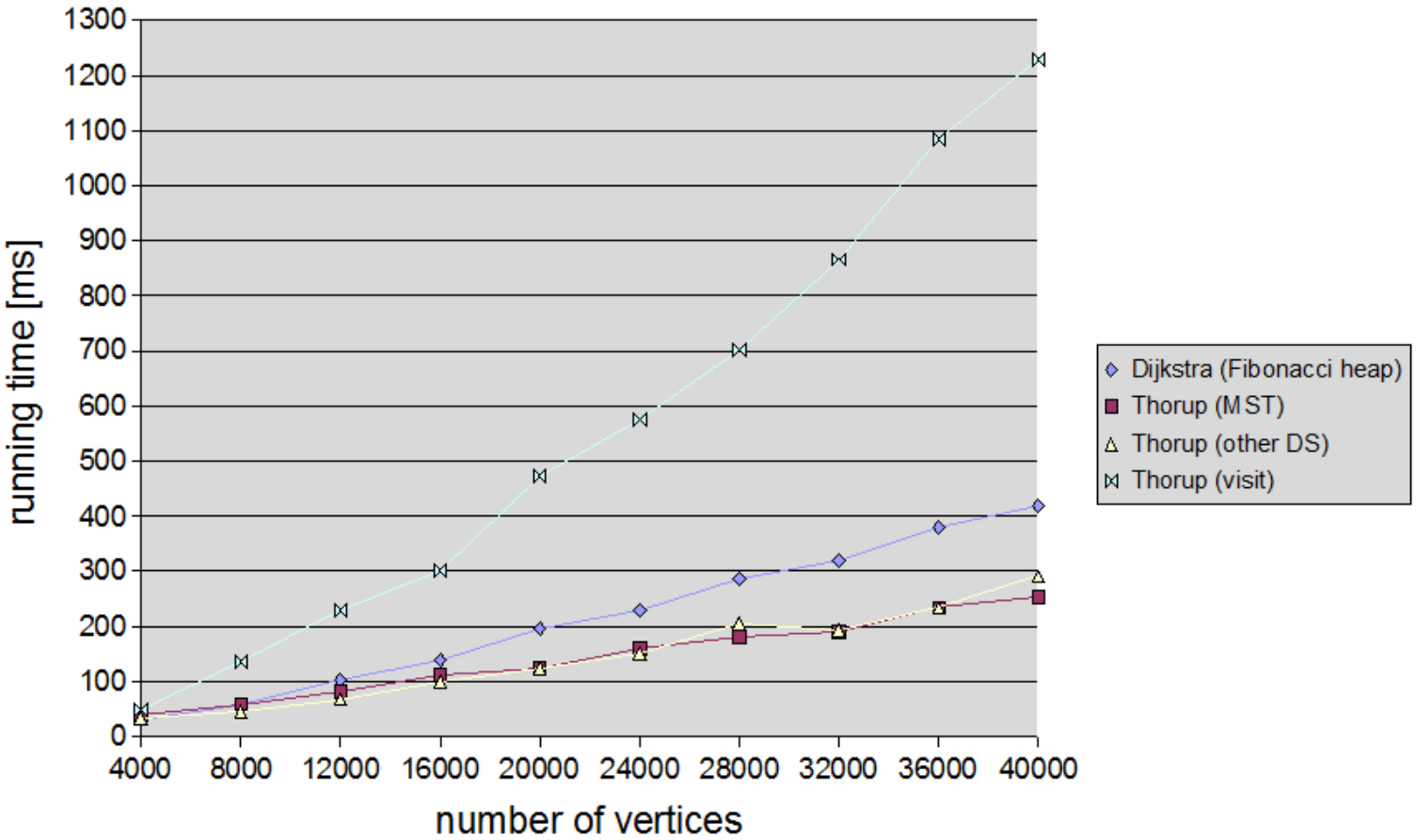


Figure: Running times for $4000 \leq n \leq 40000$, $m = 5n$.

Performance

Varying the number of edges per vertex

- fix the number of vertices $n = 20000$
- start with $m = 3n$ edges and increase it in steps of $3n$ until $m = 24n$
- edge weights $1 \leq w \leq 100000$ again

Performance

Varying the number of edges per vertex: Results

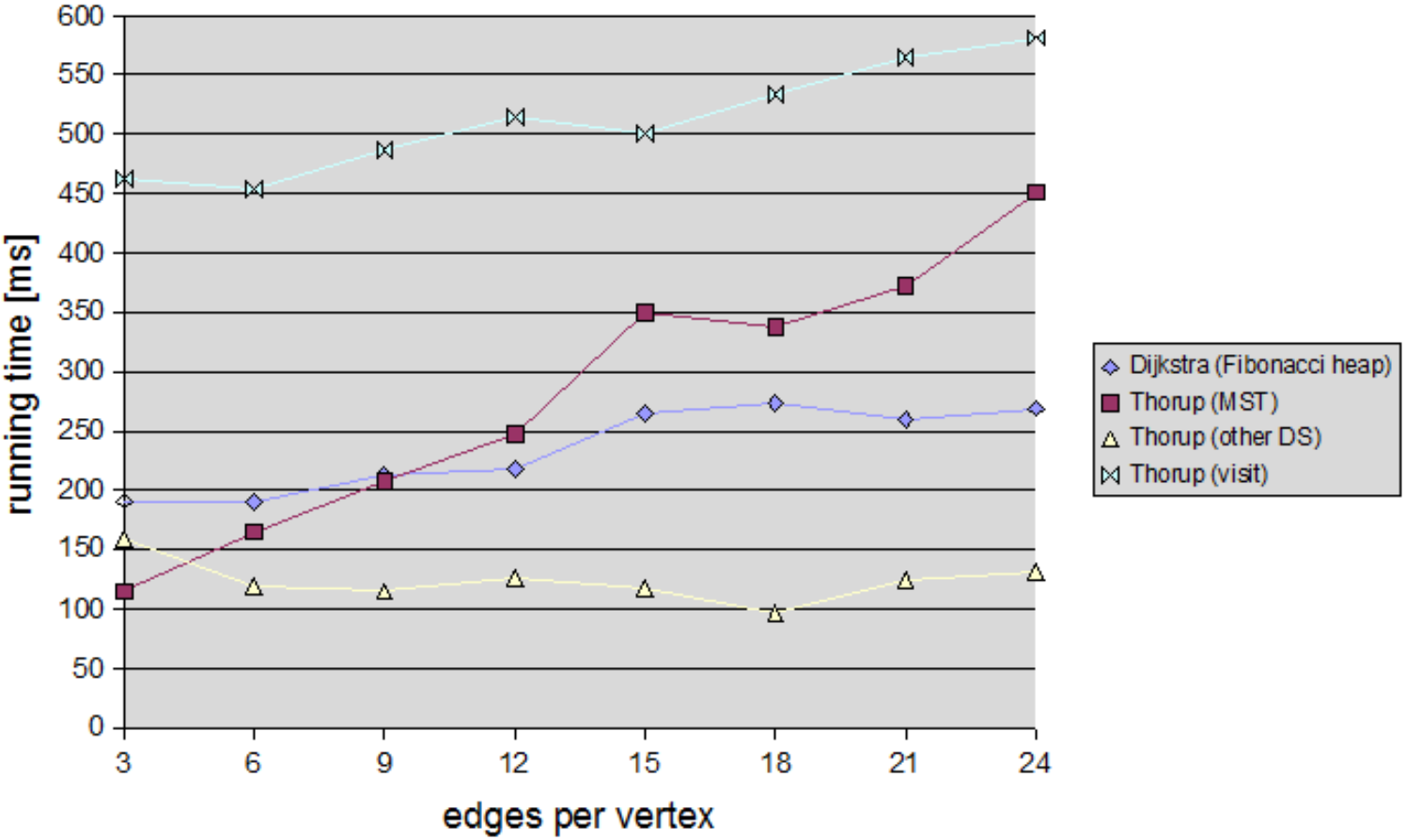


Figure: Running times for $n = 20000, 3n \leq m \leq 24n$.

Performance

Varying the maximum edge weight

- fix the number of vertices $n = 20000$
- fix the number of edges $m = 5n$
- start by choosing all edge weights $1 \leq w \leq 5$ and increase the maximum edge weight in steps of 5 until $1 \leq w \leq 100$

Performance

Varying the maximum edge weight: Results

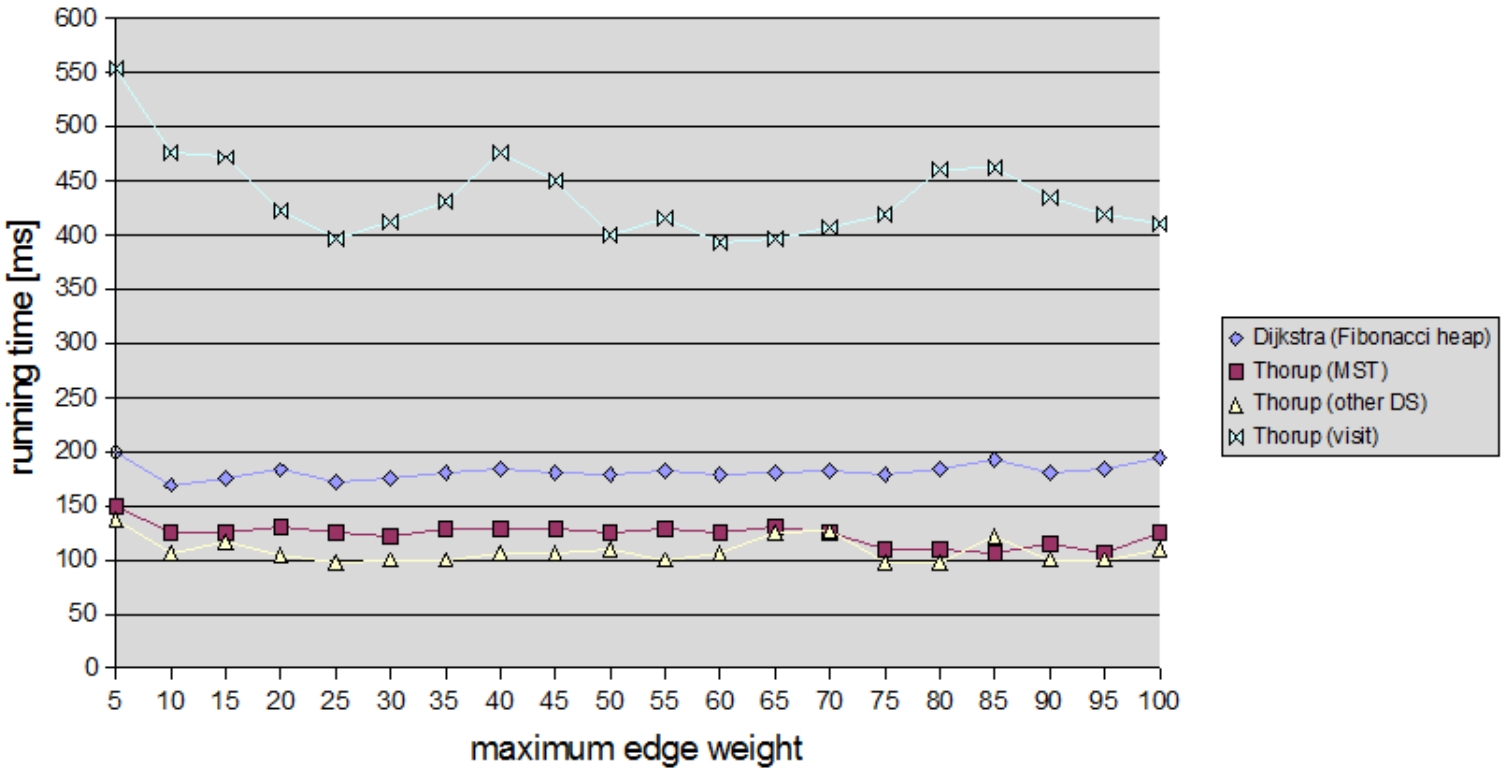


Figure: Running times for $n = 20000, m = 5n$, and maximum edge weights between 5 and 100.

Performance

Varying the maximum edge weight: Results

maximum edge weight	256	1024	16384	262144
Dijkstra (Fibonacci heap)	212	209	212	219
Thorup (MST)	148	153	145	168
Thorup (other DS)	153	171	156	140
Thorup (visit)	461	423	430	403

Table: Running times for $n = 20000, m = 5n$ and maximum edge weights $2^8, 2^{10}, 2^{14}$ and 2^{18}.

Performance

Repetitive queries

- find out whether Thorup's algorithm can catch up with the one by Dijkstra making repetive queries

Performance

Repetitive queries

- find out whether Thorup's algorithm can catch up with the one by Dijkstra making repetive queries
- at the first query, all required data structures are computed once
  - this is the initial lead Dijkstra's algorithm has over the one by Thorup

Performance

Repetitive queries

- find out whether Thorup's algorithm can catch up with the one by Dijkstra making repetive queries
- at the first query, all required data structures are computed once
  - this is the initial lead Dijkstra's algorithm has over the one by Thorup
- *clean up* between two queries:
  1. Reset the set S of visited vertices.
  2. Clear all buckets.
  3. Reset the unvisited data structure $\mathscr{U}$, making it contain only the root of $\mathscr{T}$.

Performance

Repetitive queries

- find out whether Thorup's algorithm can catch up with the one by Dijkstra making repetive queries
- at the first query, all required data structures are computed once
  - this is the initial lead Dijkstra's algorithm has over the one by Thorup
- *clean up* between two queries:
  1. Reset the set S of visited vertices.
  2. Clear all buckets.
  3. Reset the unvisited data structure $\mathscr{U}$, making it contain only the root of $\mathscr{T}$.
- still takes significantly less time than before the first query

Performance

Repetitive queries

- test instance: the road network of New York City
  - number of vertices $n = 264,346$
  - number of edges $m = 733,846$ (about $m = 3n$)

Performance

Repetitive queries

- test instance: the road network of New York City
  - number of vertices $n = 264,346$
  - number of edges $m = 733,846$ (about $m = 3n$)
- source: *9th DIMACS Implementation Challenge - Shortest Paths* [Dem06]

Performance

Repetitive queries

- test instance: the road network of New York City
  - number of vertices $n = 264,346$
  - number of edges $m = 733,846$ (about $m = 3n$)
- source: *9th DIMACS Implementation Challenge - Shortest Paths* [Dem06]
- find the shortest paths from the vertices with indices 0 to 9 to all other ones
- accumulate the resulting running times

Performance

Repetitive queries

- test instance: the road network of New York City
  - number of vertices $n = 264,346$
  - number of edges $m = 733,846$ (about $m = 3n$)
- source: *9th DIMACS Implementation Challenge - Shortest Paths* [Dem06]
- find the shortest paths from the vertices with indices 0 to 9 to all other ones
- accumulate the resulting running times
- the initialization of Thorup's algorithm takes about 2100 ms and is added to the first accumulated time
- the time required for cleaning up all data structures is always about 200 ms and is added to the accumulated time of the query the clean-up has been done before

Performance

Repetitive queries: Results

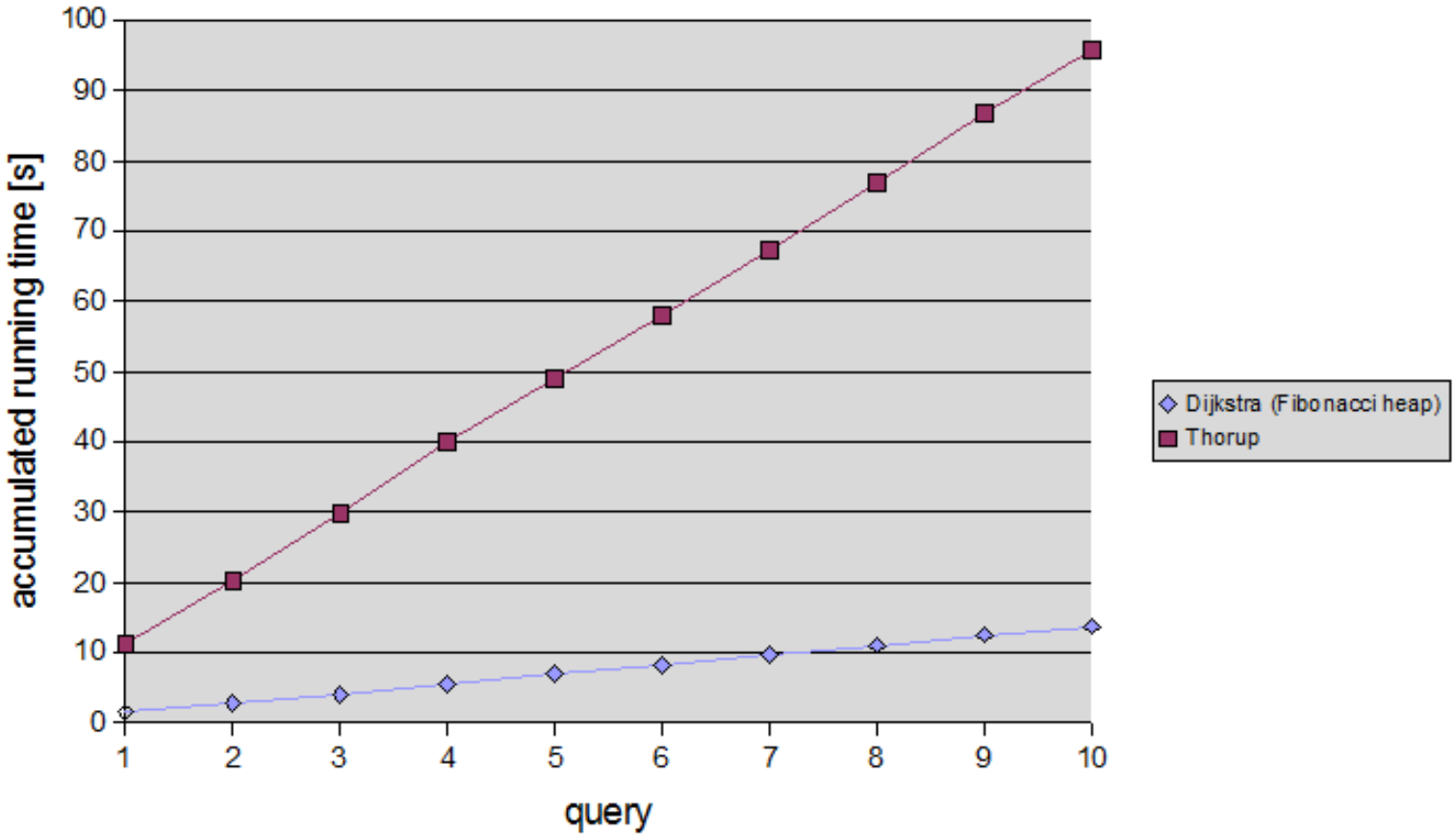


Figure: Accumulated running times for ten queries on the road network of New York City.

Conclusion

- Thorup's algorithm does not require comparison-based sorting and is today's theoretically fastest SSSP algorithm, ...

Conclusion

- Thorup's algorithm does not require comparison-based sorting and is today's theoretically fastest SSSP algorithm, ...
- ... but our implementation of the algorithm is still significantly slower than our implementation of Dijkstra's algorithm using a Fibonacci heap

Conclusion

- Thorup's algorithm does not require comparison-based sorting and is today's theoretically fastest SSSP algorithm, ...
- ... but our implementation of the algorithm is still significantly slower than our implementation of Dijkstra's algorithm using a Fibonacci heap
- possbile reasons are:
  - an inefficient implementation
  - a word length which is still too small for realizing the full potential of Thorup's algorithm

Conclusion

- Thorup's algorithm does not require comparison-based sorting and is today's theoretically fastest SSSP algorithm, ...
- ... but our implementation of the algorithm is still significantly slower than our implementation of Dijkstra's algorithm using a Fibonacci heap
- possbile reasons are:
  - an inefficient implementation
  - a word length which is still too small for realizing the full potential of Thorup's algorithm
- these observations essentially equal the conclusion made by Yasuhito Asano and Hiroshi Imai [AI00]

Conclusion

Future work

- implement the strictly linear time parts of Thorup's algorithm:
  - the linear time algorithm for constructing minimum spanning trees [FW90]
  - the linear time union-find [GT85] structure
  - the linear time split-findmin structure [Tho99]

Conclusion

Future work

- implement the strictly linear time parts of Thorup's algorithm:
  - the linear time algorithm for constructing minimum spanning trees [FW90]
  - the linear time union-find [GT85] structure
  - the linear time split-findmin structure [Tho99]
- see how well Thorup's algorithm does compared to fast all shortest paths algorithms

Conclusion

Future work

- implement the strictly linear time parts of Thorup's algorithm:
  - the linear time algorithm for constructing minimum spanning trees [FW90]
  - the linear time union-find [GT85] structure
  - the linear time split-findmin structure [Tho99]
- see how well Thorup's algorithm does compared to fast all shortest paths algorithms
- see if his component tree can be useful for other applications

Conclusion

Future work

- implement the strictly linear time parts of Thorup's algorithm:
  - the linear time algorithm for constructing minimum spanning trees [FW90]
  - the linear time union-find [GT85] structure
  - the linear time split-findmin structure [Tho99]
- see how well Thorup's algorithm does compared to fast all shortest paths algorithms
- see if his component tree can be useful for other applications
- run the algorithm on significantly larger graphs and check whether it is as attractive for repetitive queries as expected, as soon as we are able to

Conclusion

Thank you for your attention!


Yasuhito Asano and Hiroshi Imai.
Practical efficiency of the linear-time algorithm for the single source shortest path problem.
Journal of the Operations Research, 43:431–447, 2000.


Camil Demetrescu.
9th dimacs implementation challenge - shortest paths.
`http://www.dis.uniroma1.it/~challenge9/download.shtml`, 2006.


Edsger W. Dijkstra.
A note on two problems in connection with graphs.
Numer. Math., 1:269–271, 1959.


Michael L. Fredman and Robert Endre Tarjan.
Fibonacci heaps and their uses in improved network optimization algorithms.
In *FOCS*, pages 338–346. IEEE, 1984.


Michael L. Fredman and Dan E. Willard.

Trans-dichotomous algorithms for minimum spanning trees and shortest paths.
In *FOCS*, volume II, pages 719–725. IEEE, 1990.


Harold N. Gabow.
A scaling algorithm for weighted matching on general graphs.
In *FOCS*, pages 90–100. IEEE, 1985.


Harold N. Gabow and Robert Endre Tarjan.
A linear-time algorithm for a special case of disjoint set union.
J. Comput. Syst. Sci., 30(2):209–221, 1985.


Joseph B. Kruskal.
On the shortest spanning subtree of a graph and the traveling salesman problem.
Proc. Am. Math. Soc, 7:48–50, 1956.


Robert Endre Tarjan.
Efficiency of a good but not linear set union algorithm.
J. ACM, 22(2):215–225, 1975.

Mikkel Thorup.
Undirected single-source shortest paths with positive integer weights in linear time.
J. ACM, 46(3):362–394, 1999.