PHAST - Hardware Accelerated Shortest path Trees

Daniel Delling   Andrew V. Goldberg
Andreas Nowatzyk   Renato F. Werneck

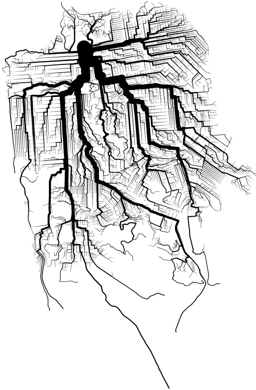
Microsoft Research Silicon Valley

March 25, 2011

Single-Source Shortest Paths

request:

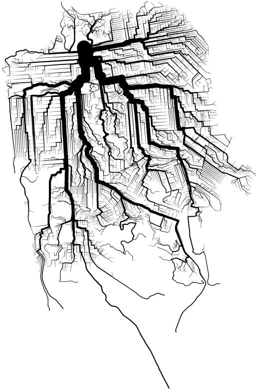
- given a (positively) weighted directed graph $G = (V, E, w)$ and a source node s
- compute distances from s to all other nodes in the graph
- applications: compute many trees for map services (sometimes even all-pairs shortest paths)



Single-Source Shortest Paths

request:

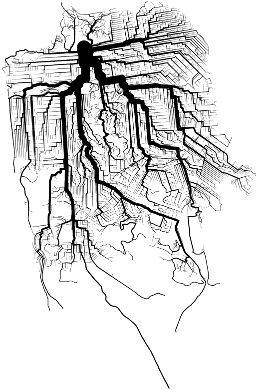
- given a (positively) weighted directed graph $G = (V, E, w)$ and a source node s
- compute distances from s to all other nodes in the graph
- applications: compute many trees for map services (sometimes even all-pairs shortest paths)



solution:

- Dijkstra [Dij59]

Single-Source Shortest Paths



request:

- given a (positively) weighted directed graph $G = (V, E, w)$ and a source node s
- compute distances from s to all other nodes in the graph
- applications: compute many trees for map services (sometimes even all-pairs shortest paths)

solution:

- Dijkstra [Dij59]

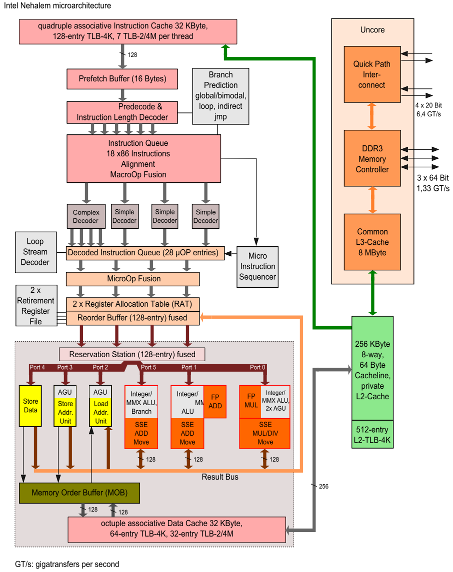
some facts:

- $O(m + n \log n)$ with Fibonacci Heaps [FT87]
- linear (with a small constant) in practice [Gol01]
- exploiting modern hardware architecture is complicated

Modern CPU architecture

some facts:

- multiple cores
- more cores than memory channels
- hyperthreading
- multi-socket systems
- steep memory hierarchy
- cache coherency
- no register coherency



Modern CPU architecture

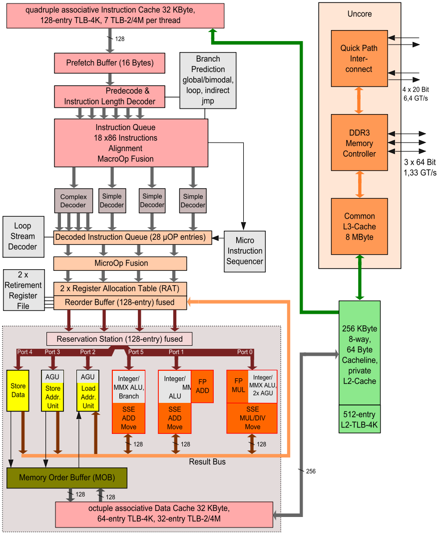
some facts:

- multiple cores
- more cores than memory channels
- hyperthreading
- multi-socket systems
- steep memory hierarchy
- cache coherency
- no register coherency

⇒ **algorithms need to be tailored**

⇒ **speedups of 100x possible**

Intel Nehalem microarchitecture



GT/s: gigatransfers per second

GPU Architecture

some facts:

- many cores (up to 512)
- high memory bandwidth (5x faster than CPU)
- but main $\rightarrow$ GPU memory transfer slow ($\approx$ 20x)
- no cache coherency
- Single Instruction Multiple Threads model
  (thread groups follow same instruction flow)
- barrel processing used to hide DRAM latency
  $\Rightarrow$ need to keep thousands of independent (!) threads busy
- access of a thread group to memory only efficient for certain patterns

GPU Architecture

some facts:

- many cores (up to 512)
- high memory bandwidth (5x faster than CPU)
- but main $\rightarrow$ GPU memory transfer slow ($\approx$ 20x)
- no cache coherency
- Single Instruction Multiple Threads model
  (thread groups follow same instruction flow)
- barrel processing used to hide DRAM latency
  $\Rightarrow$ need to keep thousands of independent (!) threads busy
- access of a thread group to memory only efficient for certain patterns

$\Rightarrow$ **algorithms need to be tailored**

Parallelizing Dijkstra's Algorithm

multiple trees:

- multi-core by source
- instruction-level parallelism exploitable [Yan10]
- approach not applicable for a GPU implementation
  - not enough memory on GPU
  - transfer main $\rightarrow$ GPU memory too slow

Parallelizing Dijkstra's Algorithm

multiple trees:

- multi-core by source
- instruction-level parallelism exploitable [Yan10]
- approach not applicable for a GPU implementation
  - not enough memory on GPU
  - transfer main $\rightarrow$ GPU memory too slow

single tree computation:

- speculation
- Δ-stepping [MS03],[MBBC09]
- more operations than Dijkstra
- no big speedups on sparse networks

Parallelizing Dijkstra's Algorithm

multiple trees:

- multi-core by source
- instruction-level parallelism exploitable [Yan10]
- approach not applicable for a GPU implementation
  - not enough memory on GPU
  - transfer main $\rightarrow$ GPU memory too slow

single tree computation:

- speculation
- Δ-stepping [MS03],[MBBC09]
- more operations than Dijkstra
- no big speedups on sparse networks

other problem:

- data locality

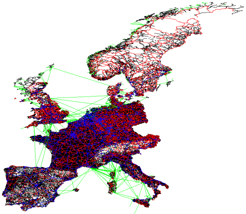
$\Rightarrow$ memory bandwidth bound

PHAST

experiments:

- input: Western European road network
- 18M nodes, 23M road segments

Dijkstra:	≈ 3.0 s	$\Rightarrow$ not real-time
BFS:	≈ 2.0 s	



numbers refer to a Core-i7 workstation (2.66 GHz)

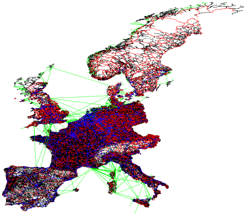
PHAST

experiments:

- input: Western European road network
- 18M nodes, 23M road segments

Dijkstra:	≈ 3.0 s	$\Rightarrow$ not real-time
BFS:	≈ 2.0 s	
$n + m$ clock cycles:	≈ 15 ms	$\Rightarrow$ big gap

- gap does not stem from data structures



numbers refer to a Core-i7 workstation (2.66 GHz)

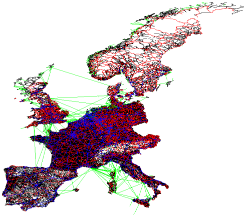
PHAST

experiments:

- input: Western European road network
- 18M nodes, 23M road segments

Dijkstra:	$\approx$ 3.0 s	$\Rightarrow$ not real-time
BFS:	$\approx$ 2.0 s	
$n + m$ clock cycles:	$\approx$ 15 ms	$\Rightarrow$ big gap

- gap does not stem from data structures



numbers refer to a Core-i7 workstation (2.66 GHz)

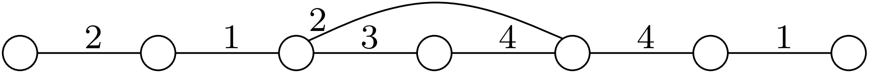
a new 2-phase algorithm for computing shortest path trees: [DGNW11]

- preprocessing:
  - a few minutes
  - works well in graphs with low highway dimension, e.g., road networks
- faster shortest path tree computation:
  - without optimization as fast as BFS
  - allows to exploit hardware architecture on all levels
  - $\Rightarrow$ up to 3 orders of magnitude faster than Dijkstra

Outline

[GSSD08]

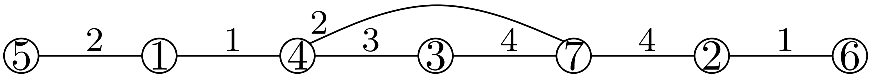
preprocessing:



[GSS**D**08]

preprocessing:

- order nodes by importance (heuristic)

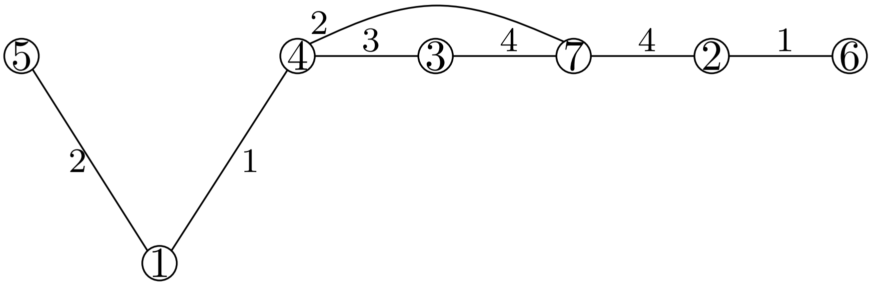


Contraction Hierarchies: A 2-phase algorithm for exact route planning

[GSS**D**08]

preprocessing:

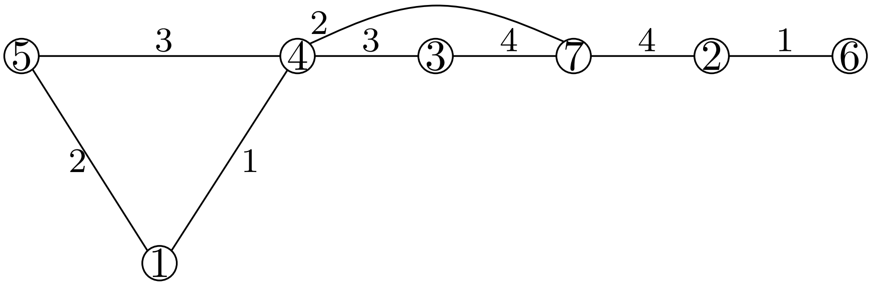
- order nodes by importance (heuristic)
- process in order
- add shortcuts to preserve distances between more important nodes



[GSSD08]

preprocessing:

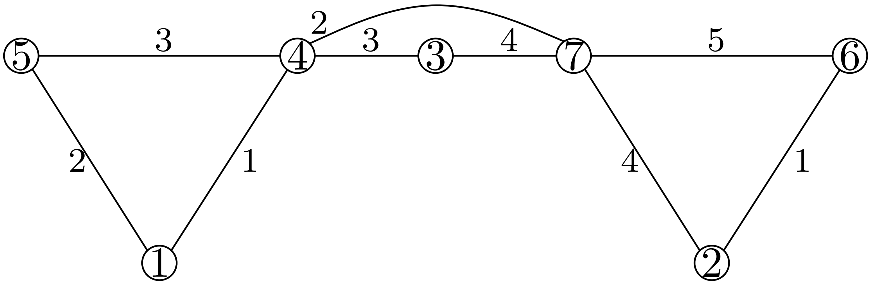
- order nodes by importance (heuristic)
- process in order
- add shortcuts to preserve distances between more important nodes



[GSSD08]

preprocessing:

- order nodes by importance (heuristic)
- process in order
- add shortcuts to preserve distances between more important nodes

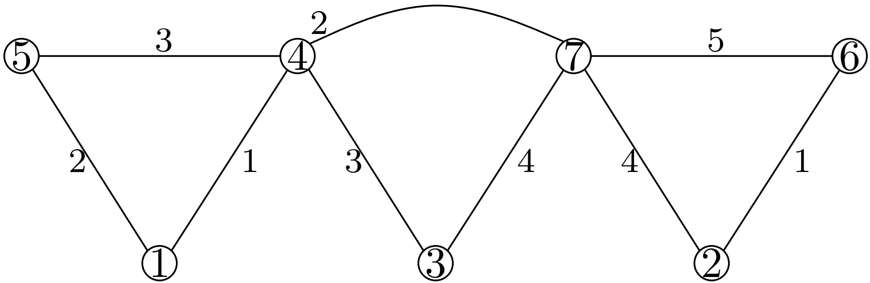


Contraction Hierarchies: A 2-phase algorithm for exact route planning

[GSS**D**08]

preprocessing:

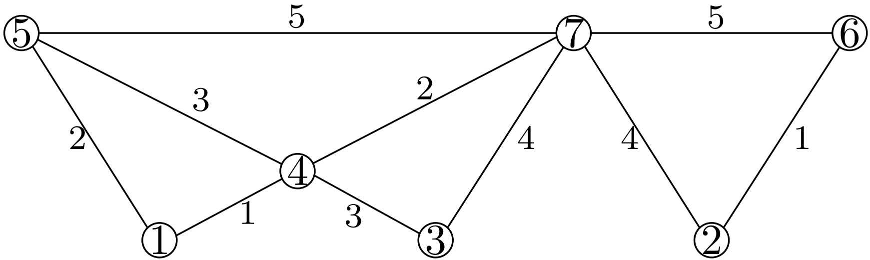
- order nodes by importance (heuristic)
- process in order
- add shortcuts to preserve distances between more important nodes



[GSSD08]

preprocessing:

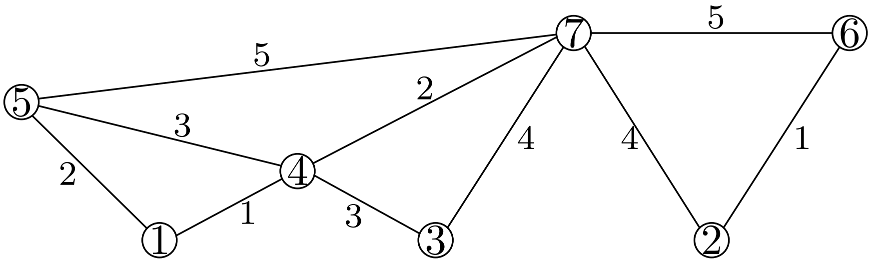
- order nodes by importance (heuristic)
- process in order
- add shortcuts to preserve distances between more important nodes



[GSSD08]

preprocessing:

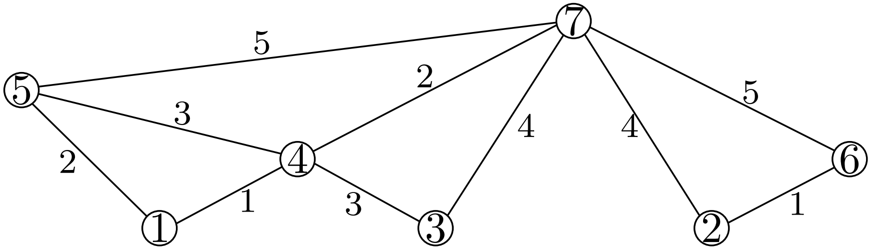
- order nodes by importance (heuristic)
- process in order
- add shortcuts to preserve distances between more important nodes



[GSSD08]

preprocessing:

- order nodes by importance (heuristic)
- process in order
- add shortcuts to preserve distances between more important nodes

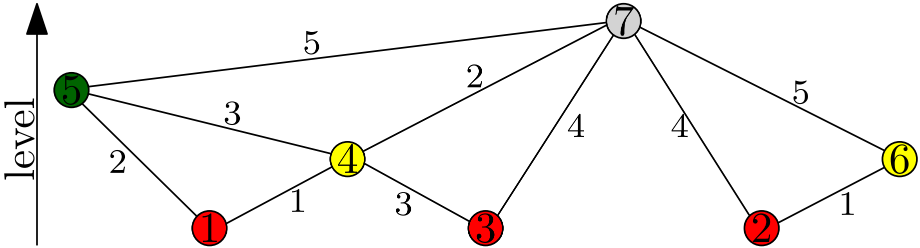


Contraction Hierarchies: A 2-phase algorithm for exact route planning

[GSSD08]

preprocessing:

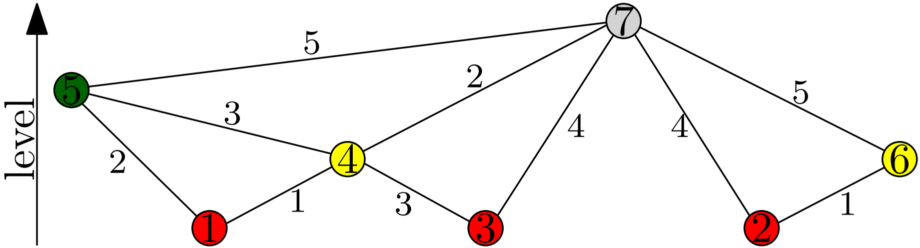
- order nodes by importance (heuristic)
- process in order
- add shortcuts to preserve distances between more important nodes
- assign levels (ca. 150 in road networks)



[GSSD08]

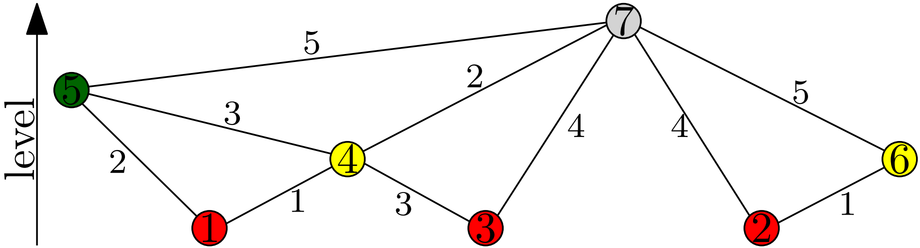
preprocessing:

- order nodes by importance (heuristic)
- process in order
- add shortcuts to preserve distances between more important nodes
- assign levels (ca. 150 in road networks)
- $\approx$ 5 minutes, 75% increase in number of edges
- heavily relies on the metric (assumes a strong hierarchy)



point-to-point query [GSSD08]

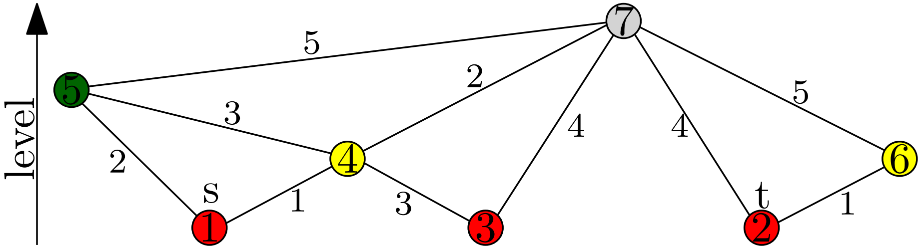
- modified bidirectional Dijkstra
- only follow edges to more important nodes



[GSSD08]

point-to-point query

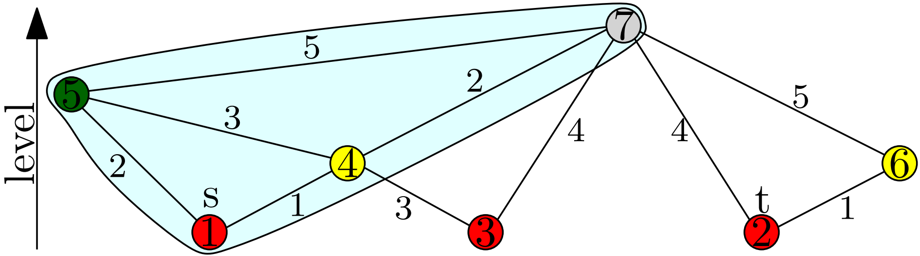
- modified bidirectional Dijkstra
- only follow edges to more important nodes



[GSSD08]

point-to-point query

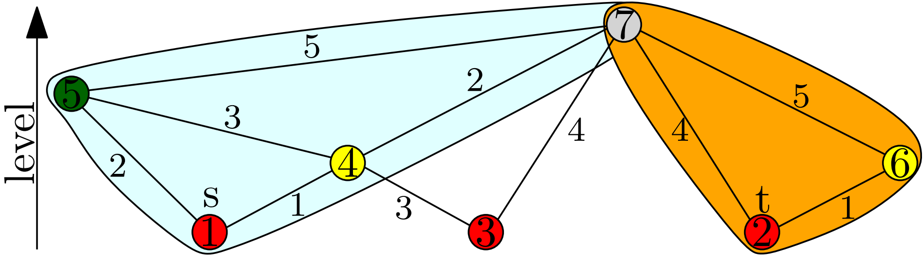
- modified bidirectional Dijkstra
- only follow edges to more important nodes



[GSSD08]

point-to-point query

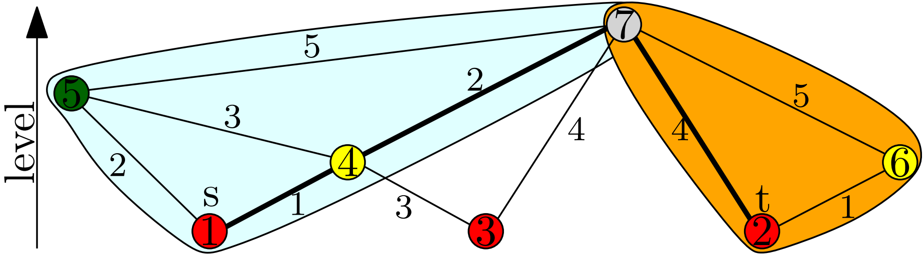
- modified bidirectional Dijkstra
- only follow edges to more important nodes



[GSSD08]

point-to-point query

- modified bidirectional Dijkstra
- only follow edges to more important nodes



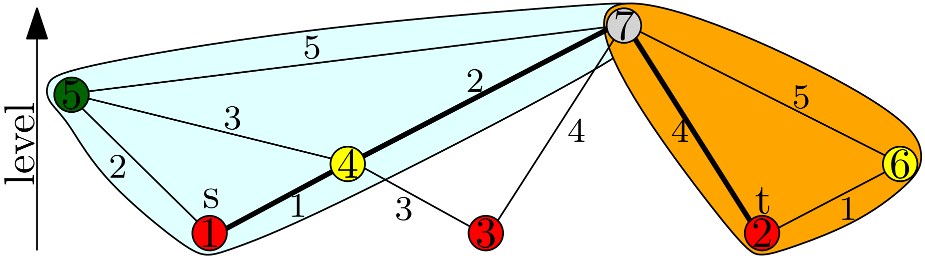
[GSSD08]

point-to-point query

- modified bidirectional Dijkstra
- only follow edges to more important nodes

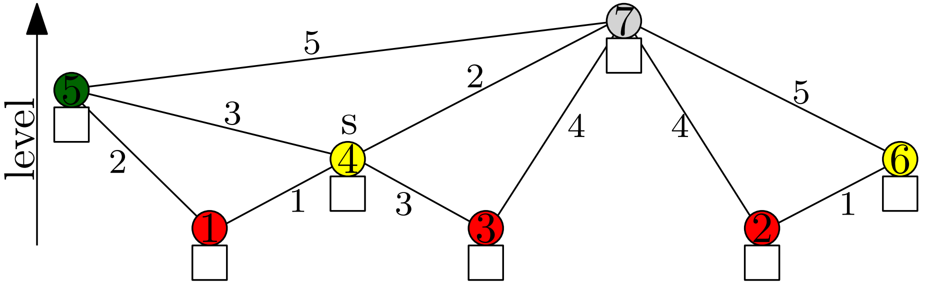
good performance on road networks:

- each upward search scans about 500 nodes
- 10000x faster than bidirectional Dijkstra (point-to-point)



Outline

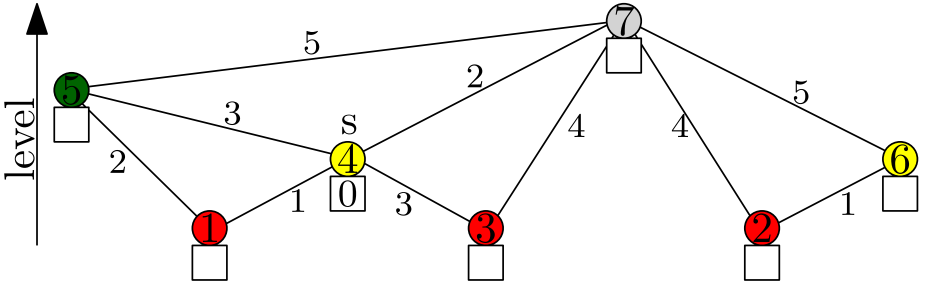
one-to-all search from source *s*:



Replacing Dijkstra

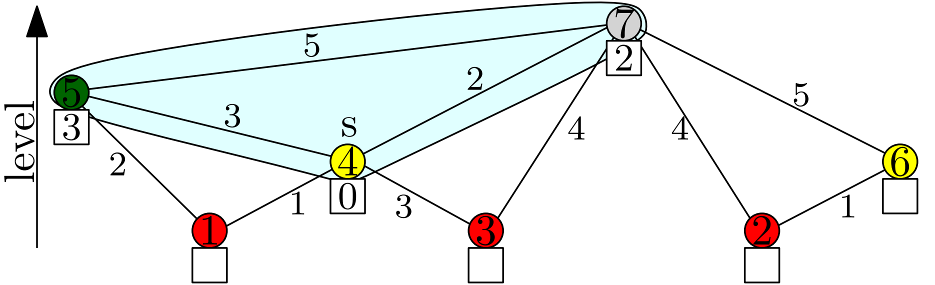
one-to-all search from source *s*:

- run CH forward search from s (≈ 0.05 ms)



one-to-all search from source *s*:

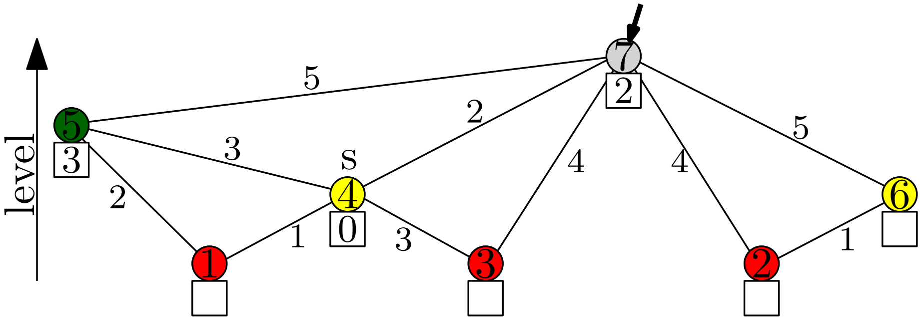
- run CH forward search from *s* ($\approx$ 0.05 ms)
- set distance labels *d* of reached nodes



Replacing Dijkstra

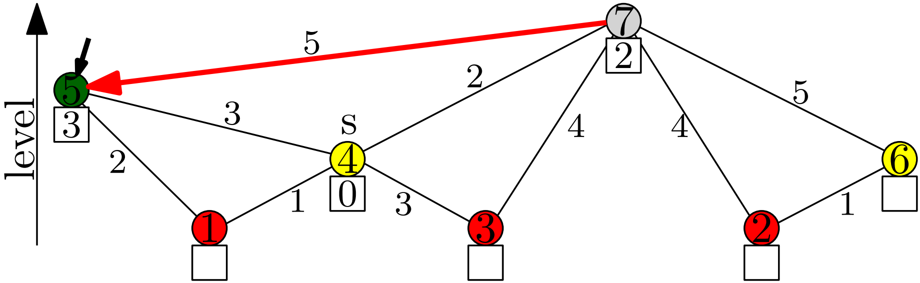
one-to-all search from source *s*:

- run CH forward search from s (≈ 0.05 ms)
- set distance labels d of reached nodes
- process all nodes u in reverse level order:
  - check incoming arcs (v, u) with $lev(v) > lev(u)$
  - set $d(u) = \min\{d(u), d(v) + w(v, u)\}$



one-to-all search from source s:

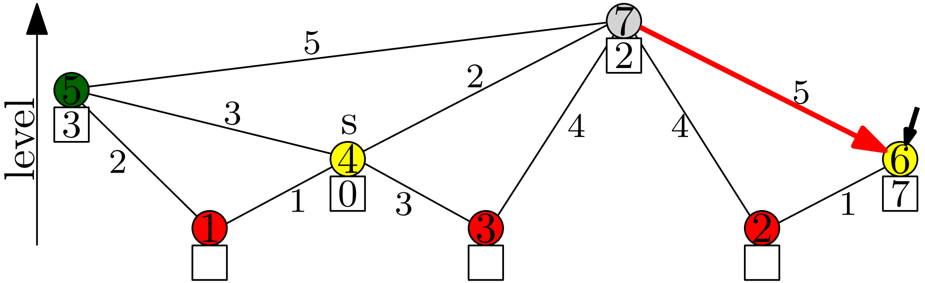
- run CH forward search from s ($\approx$ 0.05 ms)
- set distance labels d of reached nodes
- process all nodes u in reverse level order:
  - check incoming arcs (v, u) with $lev(v) > lev(u)$
  - set $d(u) = \min\{d(u), d(v) + w(v, u)\}$



Replacing Dijkstra

one-to-all search from source *s*:

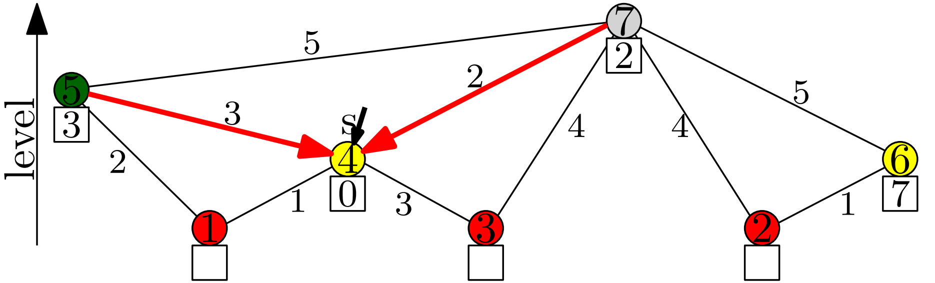
- run CH forward search from s (≈ 0.05 ms)
- set distance labels d of reached nodes
- process all nodes u in reverse level order:
  - check incoming arcs (v, u) with $lev(v) > lev(u)$
  - set $d(u) = \min\{d(u), d(v) + w(v, u)\}$



Replacing Dijkstra

one-to-all search from source s:

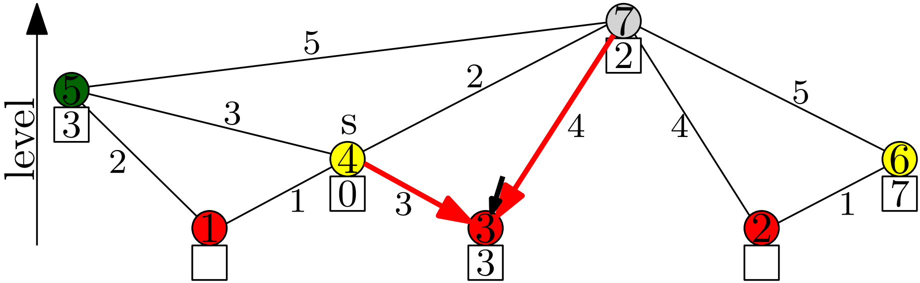
- run CH forward search from s ($\approx$ 0.05 ms)
- set distance labels d of reached nodes
- process all nodes u in reverse level order:
  - check incoming arcs (v, u) with $lev(v) > lev(u)$
  - set $d(u) = \min\{d(u), d(v) + w(v, u)\}$



Replacing Dijkstra

one-to-all search from source s:

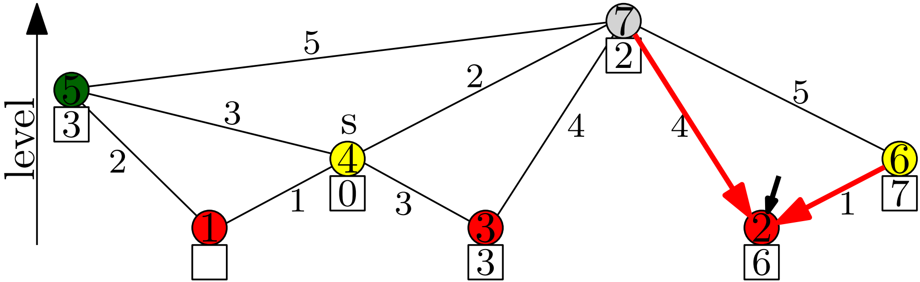
- run CH forward search from s (≈ 0.05 ms)
- set distance labels d of reached nodes
- process all nodes u in reverse level order:
  - check incoming arcs (v, u) with $lev(v) > lev(u)$
  - set $d(u) = \min\{d(u), d(v) + w(v, u)\}$



Replacing Dijkstra

one-to-all search from source s:

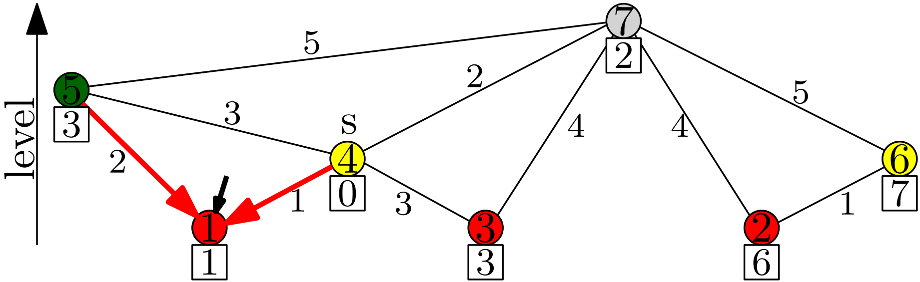
- run CH forward search from s ($\approx$ 0.05 ms)
- set distance labels d of reached nodes
- process all nodes u in reverse level order:
  - check incoming arcs (v, u) with $lev(v) > lev(u)$
  - set $d(u) = \min\{d(u), d(v) + w(v, u)\}$



Replacing Dijkstra

one-to-all search from source s:

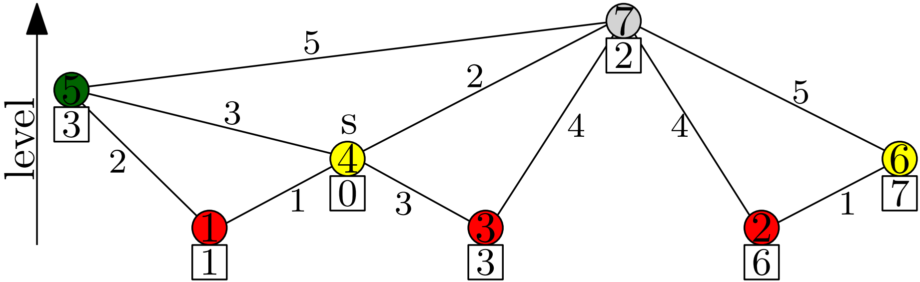
- run CH forward search from s ($\approx$ 0.05 ms)
- set distance labels d of reached nodes
- process all nodes u in reverse level order:
  - check incoming arcs (v, u) with $lev(v) > lev(u)$
  - set $d(u) = \min\{d(u), d(v) + w(v, u)\}$



Replacing Dijkstra

one-to-all search from source s:

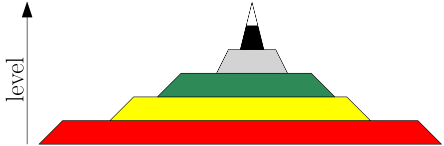
- run CH forward search from s ($\approx$ 0.05 ms)
- set distance labels d of reached nodes
- process all nodes u in reverse level order:
  - check incoming arcs (v, u) with $lev(v) > lev(u)$
  - set $d(u) = \min\{d(u), d(v) + w(v, u)\}$
- top-down processing without priority queue (ca. 2.0 s)



Analysis

observation:

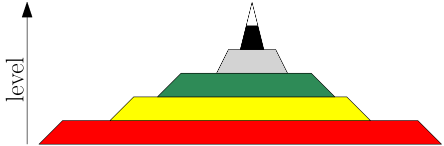
- top-down process is the bottleneck



Analysis

observation:

- top-down process is the bottleneck
- access to the data is inefficient

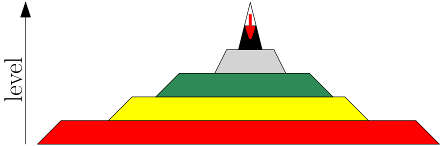


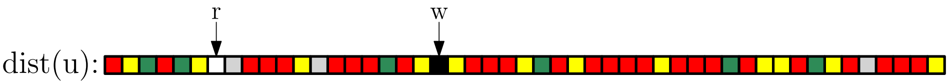
dist(u): 

Analysis

observation:

- top-down process is the bottleneck
- access to the data is inefficient

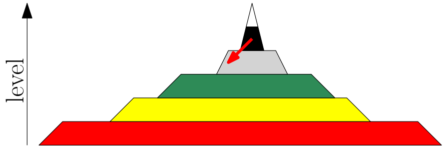


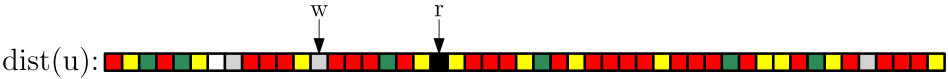


Analysis

observation:

- top-down process is the bottleneck
- access to the data is inefficient

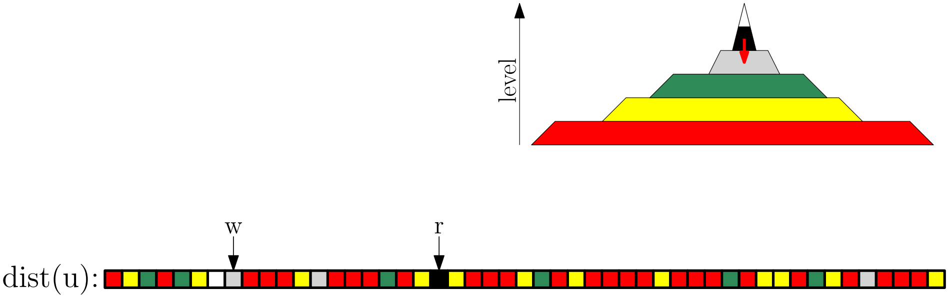




Analysis

observation:

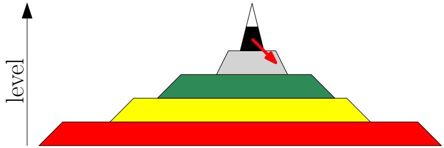
- top-down process is the bottleneck
- access to the data is inefficient

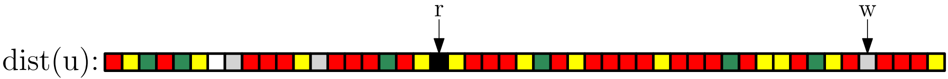


Analysis

observation:

- top-down process is the bottleneck
- access to the data is inefficient

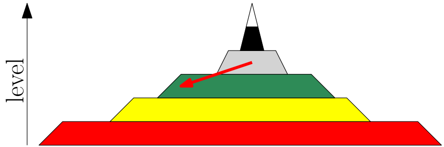




Analysis

observation:

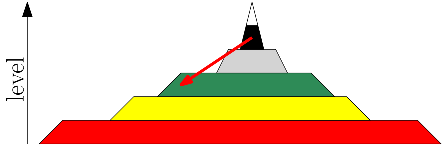
- top-down process is the bottleneck
- access to the data is inefficient

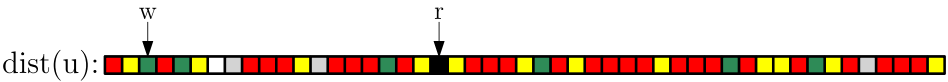


Analysis

observation:

- top-down process is the bottleneck
- access to the data is inefficient

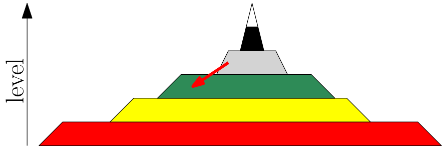




Analysis

observation:

- top-down process is the bottleneck
- access to the data is inefficient
- processing order is independent of the source node



Analysis

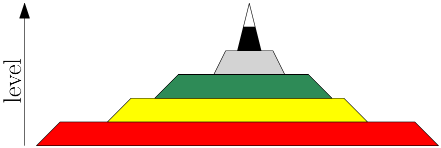
observation:

- top-down process is the bottleneck
- access to the data is inefficient
- processing order is independent of the source node

idea:

- reorder nodes, arcs, distance labels by level

$\Rightarrow$ reading arcs and writing distances become a sequential sweep



dist(u):

Analysis

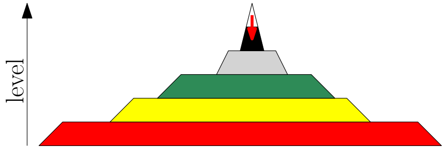
observation:

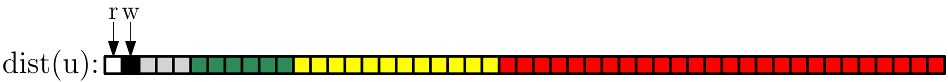
- top-down process is the bottleneck
- access to the data is inefficient
- processing order is independent of the source node

idea:

- reorder nodes, arcs, distance labels by level

⇒ reading arcs and writing distances become a sequential sweep





Analysis

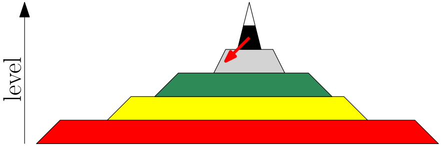
observation:

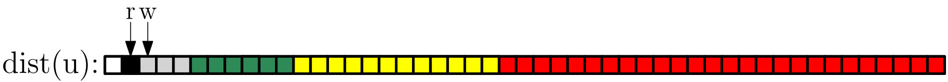
- top-down process is the bottleneck
- access to the data is inefficient
- processing order is independent of the source node

idea:

- reorder nodes, arcs, distance labels by level

$\Rightarrow$ reading arcs and writing distances become a sequential sweep





Analysis

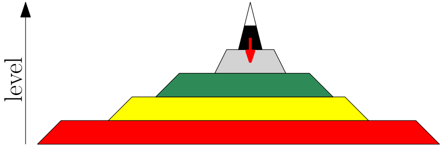
observation:

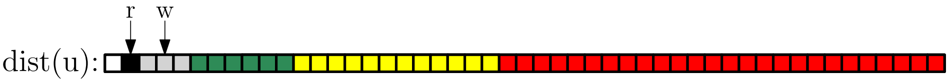
- top-down process is the bottleneck
- access to the data is inefficient
- processing order is independent of the source node

idea:

- reorder nodes, arcs, distance labels by level

⇒ reading arcs and writing distances become a sequential sweep





Analysis

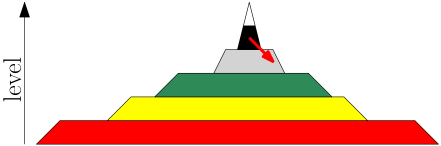
observation:

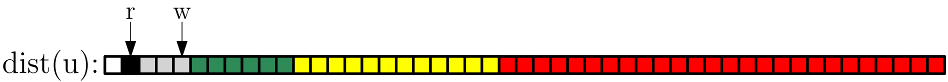
- top-down process is the bottleneck
- access to the data is inefficient
- processing order is independent of the source node

idea:

- reorder nodes, arcs, distance labels by level

⇒ reading arcs and writing distances become a sequential sweep





Analysis

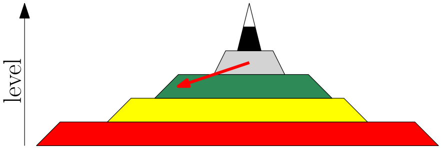
observation:

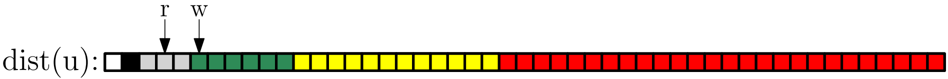
- top-down process is the bottleneck
- access to the data is inefficient
- processing order is independent of the source node

idea:

- reorder nodes, arcs,
  distance labels by level

⇒ reading arcs and writing distances become a sequential sweep





Analysis

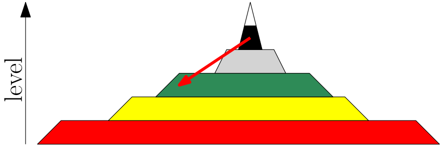
observation:

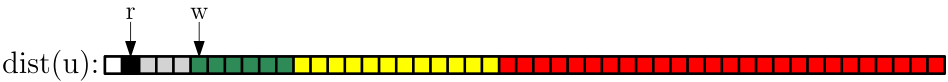
- top-down process is the bottleneck
- access to the data is inefficient
- processing order is independent of the source node

idea:

- reorder nodes, arcs, distance labels by level

⇒ reading arcs and writing distances become a sequential sweep





Analysis

observation:

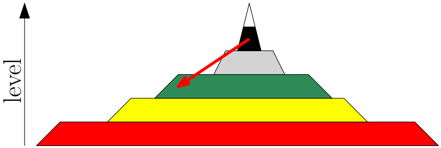
- top-down process is the bottleneck
- access to the data is inefficient
- processing order is independent of the source node

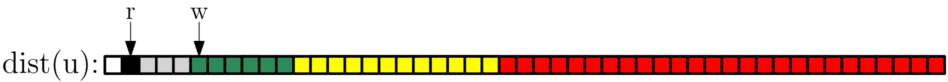
idea:

- reorder nodes, arcs, distance labels by level

⇒ reading arcs and writing distances become a sequential sweep

⇒ 172 ms per tree





Analysis

observation:

- top-down process is the bottleneck
- access to the data is inefficient
- processing order is independent of the source node

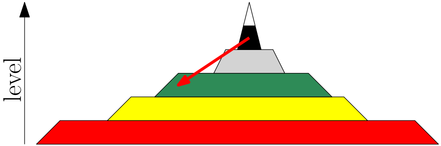
idea:

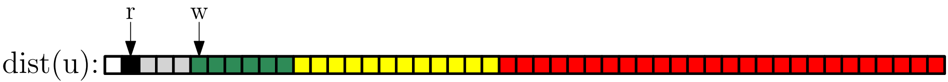
- reorder nodes, arcs, distance labels by level

⇒ reading arcs and writing distances become a sequential sweep

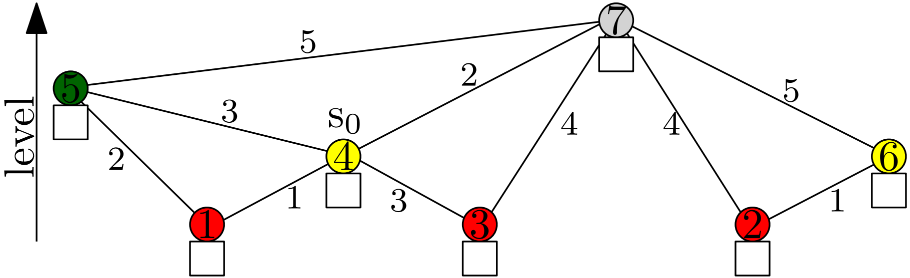
⇒ 172 ms per tree

- but reading distances still inefficient



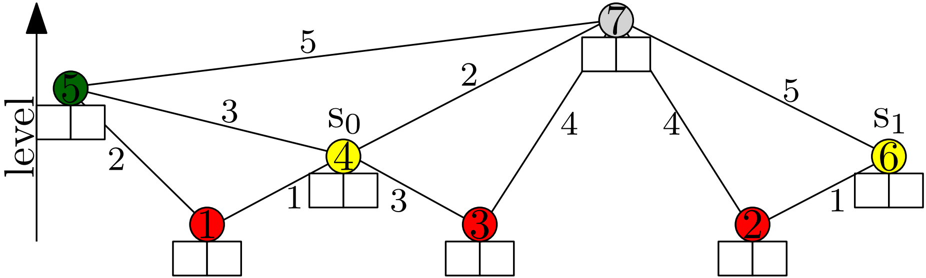


Scenario: Multiple Sources



Scenario: Multiple Sources

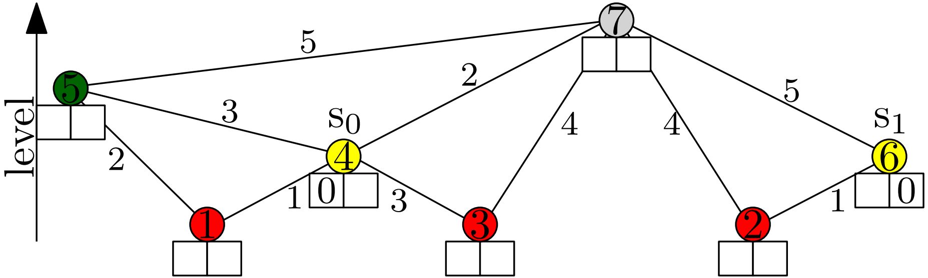
idea:



Scenario: Multiple Sources

idea:

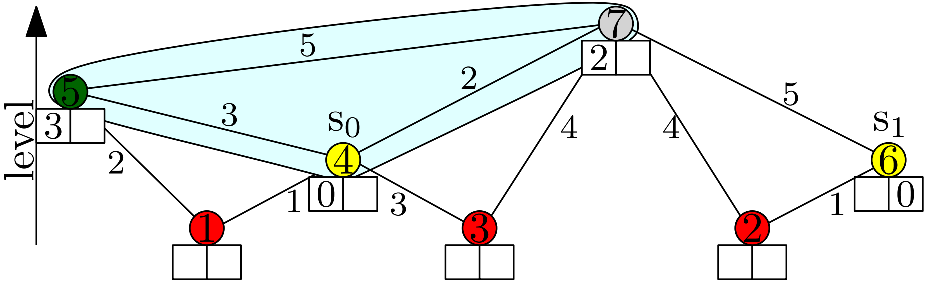
- run k forward searches



Scenario: Multiple Sources

idea:

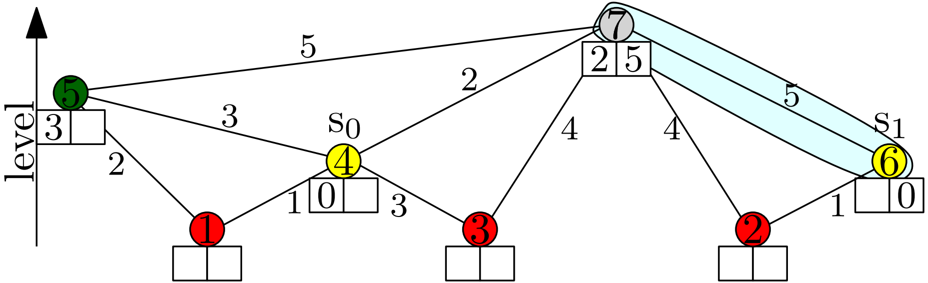
- run k forward searches



Scenario: Multiple Sources

idea:

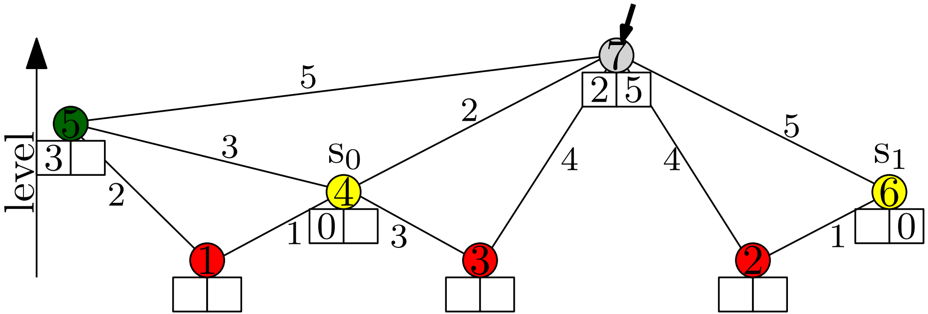
- run k forward searches



Scenario: Multiple Sources

idea:

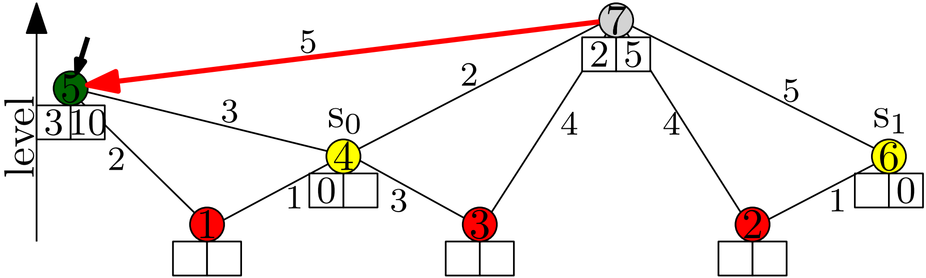
- run k forward searches
- one sweep (update all k values)
- align distance labels per node



Scenario: Multiple Sources

idea:

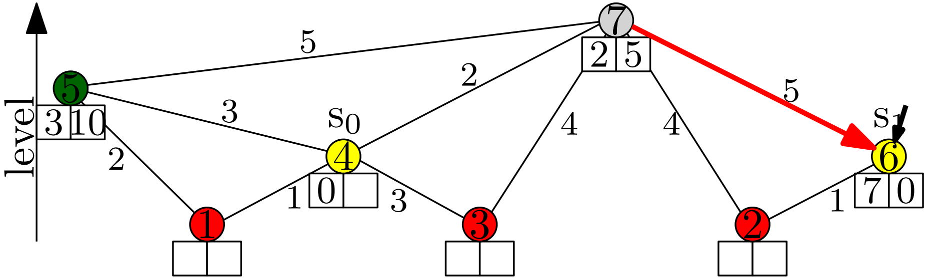
- run k forward searches
- one sweep (update all k values)
- align distance labels per node



Scenario: Multiple Sources

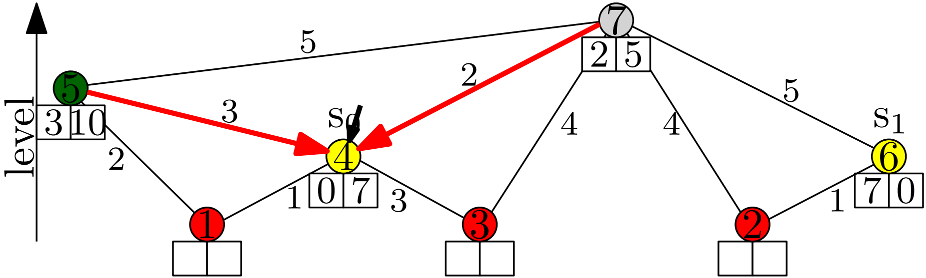
idea:

- run k forward searches
- one sweep (update all k values)
- align distance labels per node



idea:

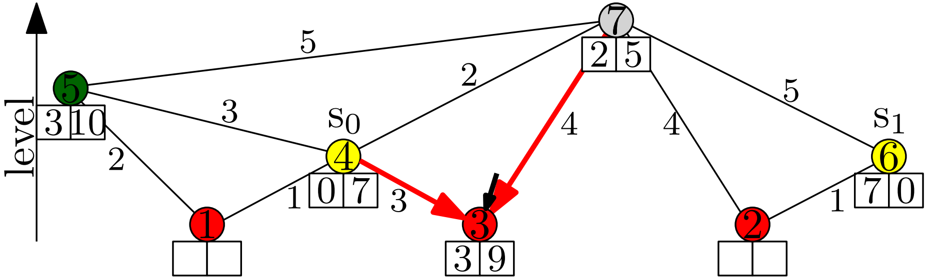
- run k forward searches
- one sweep (update all k values)
- align distance labels per node



Scenario: Multiple Sources

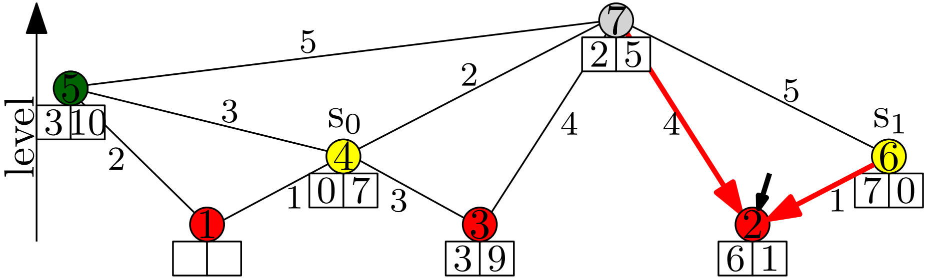
idea:

- run *k* forward searches
- one sweep (update all *k* values)
- align distance labels per node



idea:

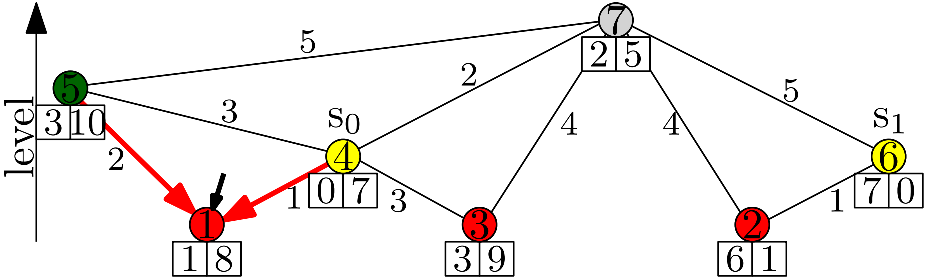
- run k forward searches
- one sweep (update all k values)
- align distance labels per node



Scenario: Multiple Sources

idea:

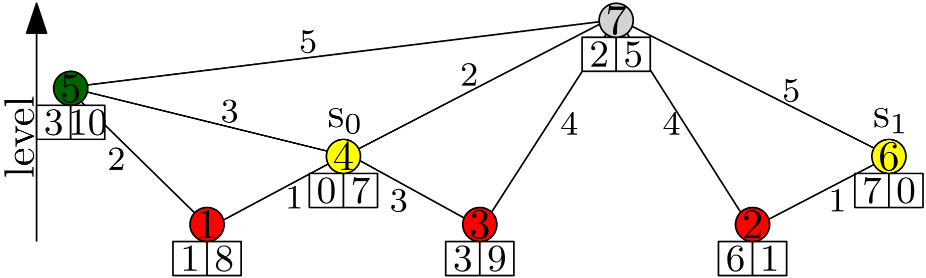
- run k forward searches
- one sweep (update all k values)
- align distance labels per node



Scenario: Multiple Sources

idea:

- run k forward searches
- one sweep (update all k values)
- align distance labels per node
- 96.8 ms per tree ($k = 16$)



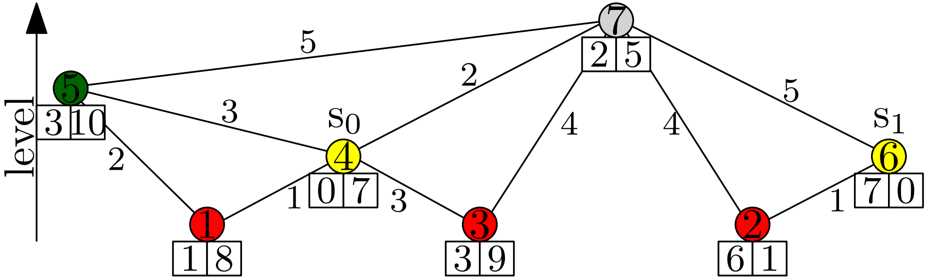
Scenario: Multiple Sources

idea:

- run k forward searches
- one sweep (update all k values)
- align distance labels per node
- 96.8 ms per tree ($k = 16$)

SSE:

- 128-bit registers
- basic operations (min, add) on four 32-bit integers in parallel
- scan 4 sources at once



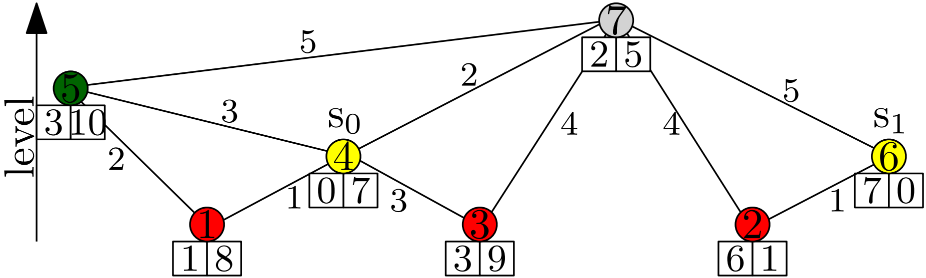
Scenario: Multiple Sources

idea:

- run k forward searches
- one sweep (update all k values)
- align distance labels per node
- 96.8 ms per tree ($k = 16$)

SSE:

- 128-bit registers
- basic operations (min, add) on four 32-bit integers in parallel
- scan 4 sources at once
- 37.1 ms per tree ($k = 16$)

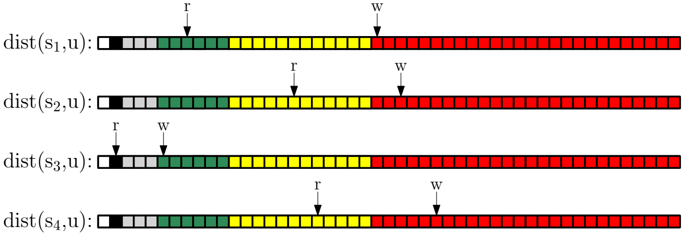


Outline

Parallelization

obvious way of parallelization

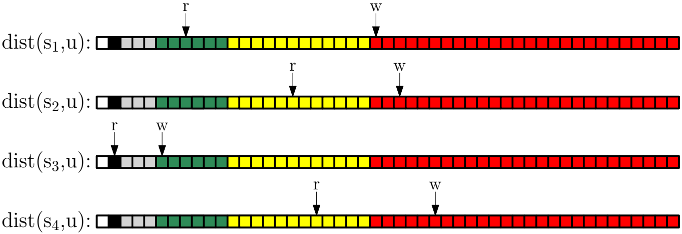
- by sources



Parallelization

obvious way of parallelization

- by sources



results:

- 16 sources per sweep (updating via SSE)
- multi-core by source nodes

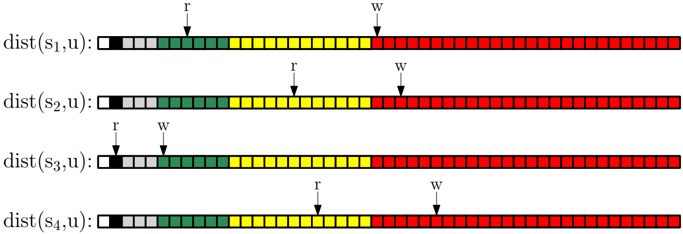
$\Rightarrow$ 64 sources in parallel (4 cores)

- 18.8 ms per tree on average

Parallelization

obvious way of parallelization

- by sources



results:

- 16 sources per sweep (updating via SSE)
- multi-core by source nodes

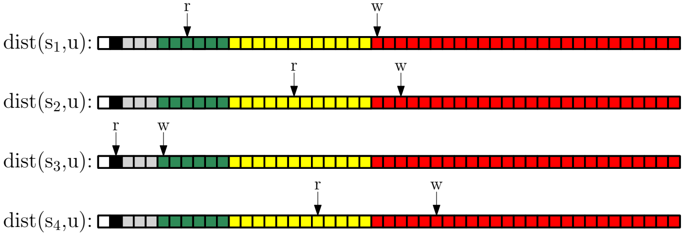
⇒ 64 sources in parallel (4 cores)

- 18.8 ms per tree on average
- why no perfect speedup?

Parallelization

obvious way of parallelization

- by sources



results:

- 16 sources per sweep (updating via SSE)
- multi-core by source nodes

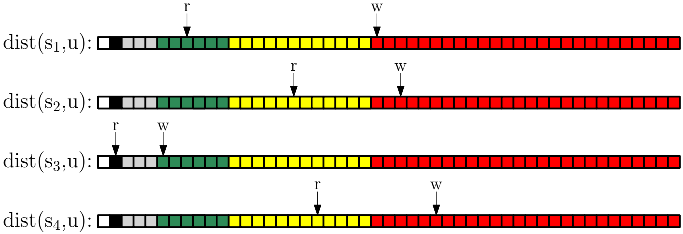
⇒ 64 sources in parallel (4 cores)

- 18.8 ms per tree on average
- why no perfect speedup?
- lower bound tests indicate that we are close to memory bandwidth barrier

Parallelization

obvious way of parallelization

- by sources



results:

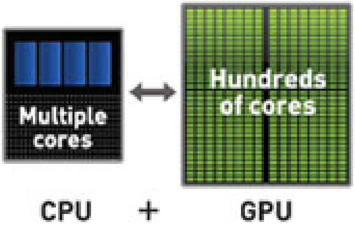
- 16 sources per sweep (updating via SSE)
- multi-core by source nodes

⇒ 64 sources in parallel (4 cores)

- 18.8 ms per tree on average
- why no perfect speedup?
- lower bound tests indicate that we are close to memory bandwidth barrier
- can a GPU help?

Outline

GPU Architecture



Intel Xeon X5680:

- 3.33 GHz
- 32 GB/s memory bandwidth
- 6 cores

NVIDIA GTX 580:

- 772 MHz, 1.5 GB RAM
- 192 GB/s memory bandwidth
- 16 cores, 32 parallel threads (a warp) per core $\Rightarrow$ 512 threads in parallel

GPHAST - Basic Ideas

observation:

- upward search is fast
- bottleneck is the linear sweep
- limited by memory bandwidth

GPHAST - Basic Ideas

observation:

- upward search is fast
- bottleneck is the linear sweep
- limited by memory bandwidth

idea:

- run upward seach on the CPU
- copy search space to GPU (less than 2 kB)
- do linear sweep on the GPU

GPHAST - Basic Ideas

observation:

- upward search is fast
- bottleneck is the linear sweep
- limited by memory bandwidth

idea:

- run upward seach on the CPU
- copy search space to GPU (less than 2 kB)
- do linear sweep on the GPU

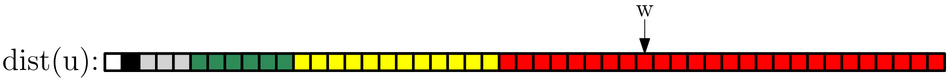
problem:

- not enough memory on GPU to compute thousands of trees in parallel
- we need to parallelize a single tree computation

Parallel Linear Sweep

observation:

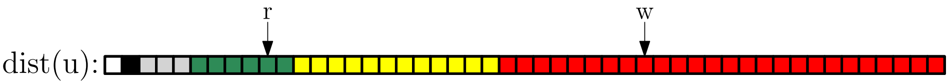
- when scanning level i:
  - only incoming arcs from level $> i$ are relevant
  - writing distance labels in level i, read from level $> i$
  - distance labels for level $> i$ are correct
- scanning a level-i node is independent from other level-i nodes



Parallel Linear Sweep

observation:

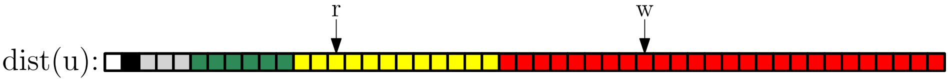
- when scanning level i:
  - only incoming arcs from level $> i$ are relevant
  - writing distance labels in level i, read from level $> i$
  - distance labels for level $> i$ are correct
- scanning a level-i node is independent from other level-i nodes



Parallel Linear Sweep

observation:

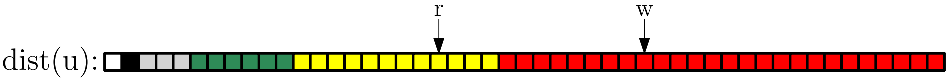
- when scanning level *i*:
  - only incoming arcs from level $> i$ are relevant
  - writing distance labels in level *i*, read from level $> i$
  - distance labels for level $> i$ are correct
- scanning a level-*i* node is independent from other level-*i* nodes



Parallel Linear Sweep

observation:

- when scanning level i:
  - only incoming arcs from level $> i$ are relevant
  - writing distance labels in level i, read from level $> i$
  - distance labels for level $> i$ are correct
- scanning a level-i node is independent from other level-i nodes



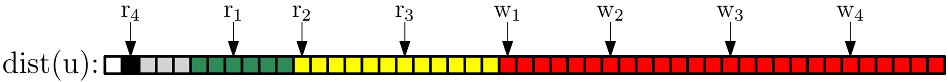
Parallel Linear Sweep

observation:

- when scanning level i:
  - only incoming arcs from level $> i$ are relevant
  - writing distance labels in level i, read from level $> i$
  - distance labels for level $> i$ are correct
- scanning a level-i node is independent from other level-i nodes

idea:

- scan all nodes on level i in parallel
- synchronization after each level
- one thread per node



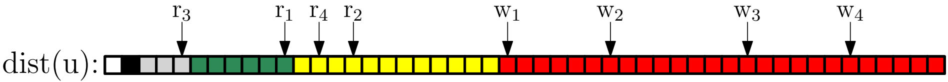
Parallel Linear Sweep

observation:

- when scanning level i:
  - only incoming arcs from level $> i$ are relevant
  - writing distance labels in level i, read from level $> i$
  - distance labels for level $> i$ are correct
- scanning a level-i node is independent from other level-i nodes

idea:

- scan all nodes on level i in parallel
- synchronization after each level
- one thread per node



Parallel Linear Sweep

observation:

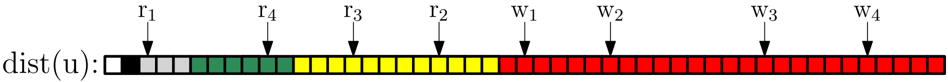
- when scanning level i:
  - only incoming arcs from level $> i$ are relevant
  - writing distance labels in level i, read from level $> i$
  - distance labels for level $> i$ are correct
- scanning a level-i node is independent from other level-i nodes

idea:

- scan all nodes on level i in parallel
- synchronization after each level
- one thread per node

results:

- 5.5 ms on an NVIDIA GTX 480
- 511 speedup over Dijkstra



Parallel Linear Sweep

observation:

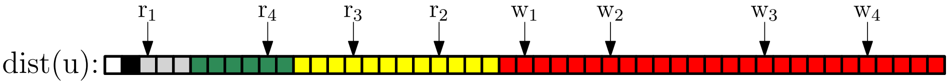
- when scanning level i:
  - only incoming arcs from level $> i$ are relevant
  - writing distance labels in level i, read from level $> i$
  - distance labels for level $> i$ are correct
- scanning a level-i node is independent from other level-i nodes

idea:

- scan all nodes on level i in parallel
- synchronization after each level
- one thread per node

results:

- 5.5 ms on an NVIDIA GTX 480
- 511 speedup over Dijkstra
- (multiple trees: 2.2 ms)



All-Pairs Shortest Paths

algorithm	device	time	energy [MJ]
Dijkstra	4-core workstation	197d	
	12-core server	60d	
	48-core server	35d	
PHAST	4-core workstation	94h	
	12-core server	36h	
	48-core server	20h	
GPHAST	GTX 580		

All-Pairs Shortest Paths

algorithm	device	time	energy [MJ]
Dijkstra	4-core workstation	197d	
	12-core server	60d	
	48-core server	35d	
PHAST	4-core workstation	94h	
	12-core server	36h	
	48-core server	20h	
GPHAST	GTX 580	11h	

All-Pairs Shortest Paths

algorithm	device	time	energy [MJ]
Dijkstra	4-core workstation	197d	
	12-core server	60d	
	48-core server	35d	
PHAST	4-core workstation	94h	
	12-core server	36h	
	48-core server	20h	
GPHAST	GTX 580	11h	

4-core workstation without GPU: 163 watts
4-core workstation with GPU: 375 watts
12-core server: 332 watts
48-core server: 747 watts

All-Pairs Shortest Paths

algorithm	device	time	energy [MJ]
Dijkstra	4-core workstation	197d	2780.6
	12-core server	60d	1725.9
	48-core server	35d	2265.5
PHAST	4-core workstation	94h	55.2
	12-core server	36h	43.0
	48-core server	20h	54.2
GPHAST	GTX 580	11h	14.9

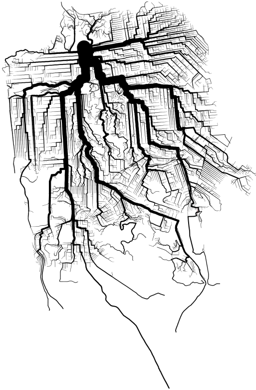
4-core workstation without GPU: 163 watts
4-core workstation with GPU: 375 watts
12-core server: 332 watts
48-core server: 747 watts

Outline

Conclusion

summary:

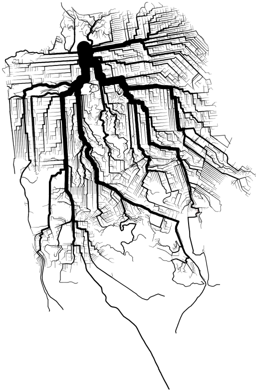
- one tree on a GPU: 5.5 ms
  (about 0.31 ns per entry)
- real-time computation of shortest path trees
- 16 trees on a GPU at once: 2.2 ms per tree
  (about 0.13 ns per entry)
- APSP in 11 hours (on a workstation with one GPU),
  instead of half a year (on 4 cores)
- APSP-based computation becomes practical
- 150 times more energy-efficient than Dijkstra's algorithm



Conclusion

summary:

- one tree on a GPU: 5.5 ms
  (about 0.31 ns per entry)
- real-time computation of shortest path trees
- 16 trees on a GPU at once: 2.2 ms per tree
  (about 0.13 ns per entry)
- APSP in 11 hours (on a workstation with one GPU),
  instead of half a year (on 4 cores)
- APSP-based computation becomes practical
- 150 times more energy-efficient than Dijkstra's algorithm



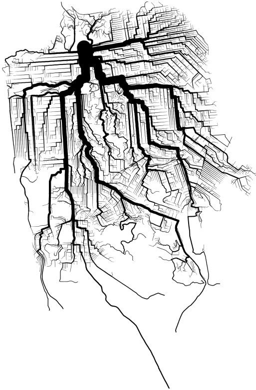
other recent results:

- point-to-point shortest paths with a few memory accesses
- refinement of highway dimension
- graph partitioning
- fully realistic driving directions

Conclusion

summary:

- one tree on a GPU: 5.5 ms
  (about 0.31 ns per entry)
- real-time computation of shortest path trees
- 16 trees on a GPU at once: 2.2 ms per tree
  (about 0.13 ns per entry)
- APSP in 11 hours (on a workstation with one GPU), instead of half a year (on 4 cores)
- APSP-based computation becomes practical
- 150 times more energy-efficient than Dijkstra's algorithm



other recent results:

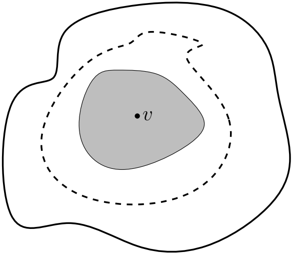
- point-to-point shortest paths with a few memory accesses
- refinement of highway dimension
- graph partitioning
- fully realistic driving directions

Thank you for your attention!

Appendix

Graph Partitioning I: Filtering

[DGRW11]

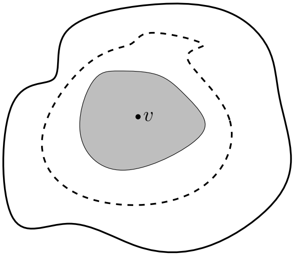


1. natural cut detection

- pick a random center
- use BFS to define a core and a ring
- find minimum cut between them
- repeat multiple times

Graph Partitioning I: Filtering

[DGRW11]

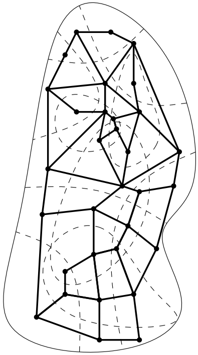


1. natural cut detection

- pick a random center
- use BFS to define a core and a ring
- find minimum cut between them
- repeat multiple times

2. contraction

- keep only edges that appeared in some cut
- contract the rest into fragments
- reduces graph by several orders of magnitude
- preserves natural cuts between dense regions
  (e.g., bridges, national borders, mountain passes...)



Graph Partitioning II: Assembly

[DGRW11]

1. run greedy algorithm

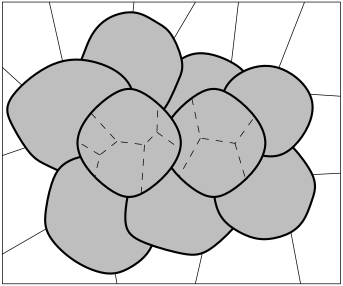
- join well-connected fragments
- find maximal solution

2. run local search

- reoptimize pairs of adjacent cells
- fragments can move to neighboring cells

3. enhanced optimizations (optional)

- multistart, recombination, branch-and-bound



$\Rightarrow$ **yields best known solutions for road networks**

Case Study: Point-to-Point Shortest Paths

two phase approach:

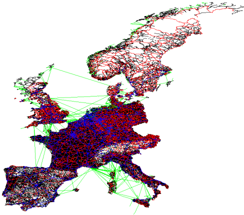
- preprocess network to compute auxillary data
- use data to speed up queries
- three-criteria optimization (preprocessing time, space, query times)

Case Study: Point-to-Point Shortest Paths

two phase approach:

- preprocess network to compute auxillary data
- use data to speed up queries
- three-criteria optimization (preprocessing time, space, query times)

	preprocessing		query
method	time [h:m]	space [GB]	time [μs]
Reach	0:15	1.5	1253.5
CH	0:05	0.4	93.5
TNR	1:52	3.7	1.8
Table Lookup	$>$ 11:03	1 208 358.7	0.056

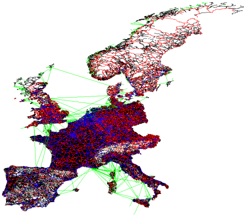


Case Study: Point-to-Point Shortest Paths

two phase approach:

- preprocess network to compute auxillary data
- use data to speed up queries
- three-criteria optimization (preprocessing time, space, query times)

	preprocessing		query
method	time [h:m]	space [GB]	time [μs]
Reach	0:15	1.5	1253.5
CH	0:05	0.4	93.5
TNR	1:52	3.7	1.8
Table Lookup	$>$ 11:03	1 208 358.7	0.056



observation:

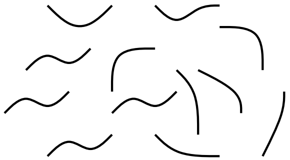
- excellent performance in practice
- used in production
- prime example for algorithm engineering
- but for a long time: no theoretical justification

A Theoretical Justification: Highway Dimension

[AFGW10]

(r, k) **shortest path cover**

- all shortest paths with length between r and $2r$ are hit

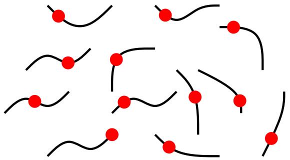


A Theoretical Justification: Highway Dimension

[AFGW10]

(r, k) **shortest path cover**

- all shortest paths with length between r and $2r$ are hit

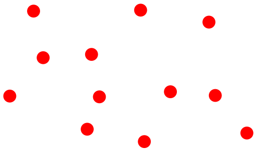


A Theoretical Justification: Highway Dimension

[AFGW10]

(r, k) **shortest path cover**

- all shortest paths with length between r and $2r$ are hit
- locally sparse
  ($\leq k$ vertices in any ball of radius $O(r)$)

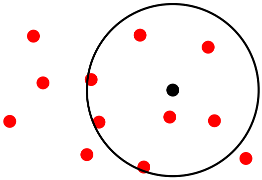


A Theoretical Justification: Highway Dimension

[AFGW10]

(r, k) **shortest path cover**

- all shortest paths with length between r and $2r$ are hit
- locally sparse
  ($\leq k$ vertices in any ball of radius $O(r)$)

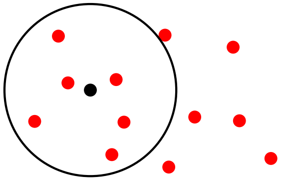


A Theoretical Justification: Highway Dimension

[AFGW10]

(r, k) **shortest path cover**

- all shortest paths with length between r and $2r$ are hit
- locally sparse
  ($\leq k$ vertices in any ball of radius $O(r)$)

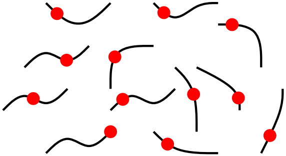


A Theoretical Justification: Highway Dimension

[AFGW10]

(r, k) **shortest path cover**

- all shortest paths with length between r and $2r$ are hit
- locally sparse
  ($\leq k$ vertices in any ball of radius $O(r)$)



Highway Dimension

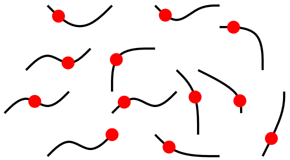
A graph with highway dimension h has an (r, h)-SPC for all r.

A Theoretical Justification: Highway Dimension

[AFGW10]

(r, k) **shortest path cover**

- all shortest paths with length between r and $2r$ are hit
- locally sparse
  ($\leq k$ vertices in any ball of radius $O(r)$)



Highway Dimension

A graph with highway dimension h has an (r, h)-SPC for all r.

results:

- sublinear query bounds for many algorithms
- best query bound: a labeling algorithm
- has not been considered in practical implementations

A Labeling Algorithm

[CPPR04]

preprocessing:

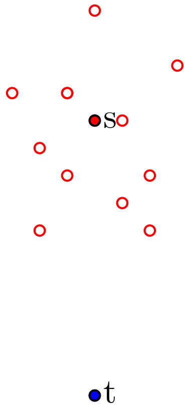
- compute a label $L(v)$ for each vertex v
- compute $\mathrm{dist}(v, w)$ for each vertex $w \in L(v)$
- obey the label property:
  for all s, t a shortest s–t path intersects $L(s) \cap L(t)$

A Labeling Algorithm

[CPPR04]

preprocessing:

- compute a label $L(v)$ for each vertex v
- compute $\text{dist}(v, w)$ for each vertex $w \in L(v)$
- obey the label property:
  for all s, t a shortest s–t path intersects $L(s) \cap L(t)$

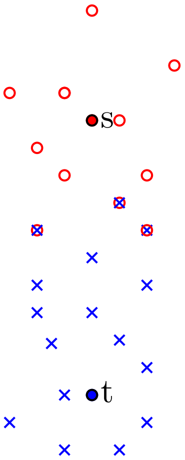


A Labeling Algorithm

[CPPR04]

preprocessing:

- compute a label $L(v)$ for each vertex v
- compute $\text{dist}(v, w)$ for each vertex $w \in L(v)$
- obey the label property:
  for all s, t a shortest s–t path intersects $L(s) \cap L(t)$



A Labeling Algorithm

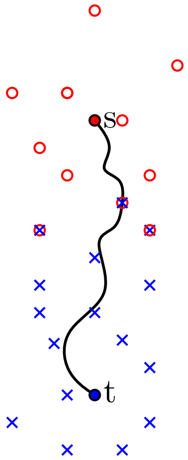
[CPPR04]

preprocessing:

- compute a label $L(v)$ for each vertex v
- compute $\text{dist}(v, w)$ for each vertex $w \in L(v)$
- obey the label property:
  for all s, t a shortest s–t path intersects $L(s) \cap L(t)$

s–t **queries:**

- find vertex $w \in L(s) \cap L(t)$ ...



A Labeling Algorithm

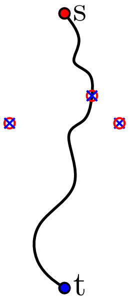
[CPPR04]

preprocessing:

- compute a label $L(v)$ for each vertex v
- compute $\mathrm{dist}(v, w)$ for each vertex $w \in L(v)$
- obey the label property:
  for all s, t a shortest s–t path intersects $L(s) \cap L(t)$

s–t **queries:**

- find vertex $w \in L(s) \cap L(t)$ ...

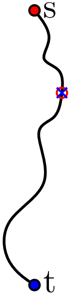


A Labeling Algorithm

[CPPR04]

preprocessing:

- compute a label $L(v)$ for each vertex v
- compute $\text{dist}(v, w)$ for each vertex $w \in L(v)$
- obey the label property:
  for all s, t a shortest s–t path intersects $L(s) \cap L(t)$



s–t **queries:**

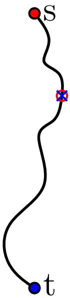
- find vertex $w \in L(s) \cap L(t)$ ...
- ... that minimizes $\text{dist}(s, v) + \text{dist}(v, t)$

A Labeling Algorithm

[CPPR04]

preprocessing:

- compute a label $L(v)$ for each vertex v
- compute $\text{dist}(v, w)$ for each vertex $w \in L(v)$
- obey the label property:
  for all s, t a shortest s–t path intersects $L(s) \cap L(t)$



s–t **queries:**

- find vertex $w \in L(s) \cap L(t)$ ...
- ... that minimizes $\text{dist}(s, v) + \text{dist}(v, t)$

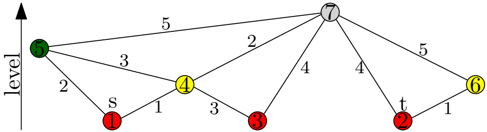
observation:

- practical if labels are small
- how to compute labels efficiently?
- SPC algorithms currently are too slow
  (maybe PHAST can help)

Practical Implementation: HubLabels

[ADGW11]

idea:

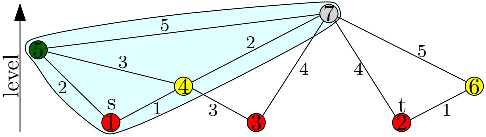


Practical Implementation: HubLabels

[ADGW11]

idea:

- search spaces of contraction hierarchies form valid labels
- run upward (forward and backward) search from each vertex, store label
- sort label entries by node id



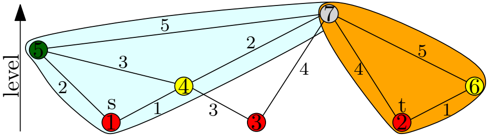
$L(s)$				
	1,0	4,1	5,2	7,3

Practical Implementation: HubLabels

[ADGW11]

idea:

- search spaces of contraction hierarchies form valid labels
- run upward (forward and backward) search from each vertex, store label
- sort label entries by node id



$L(s)$	1,0	4,1	5,2	7,3

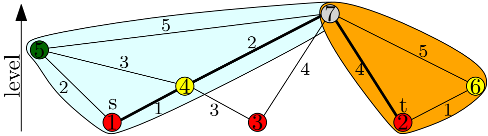
$L(t)$	2,0	6,1	7,4

Practical Implementation: HubLabels

[ADGW11]

idea:

- search spaces of contraction hierarchies form valid labels
- run upward (forward and backward) search from each vertex, store label
- sort label entries by node id



$L(s)$	1,0	4,1	5,2	7,3

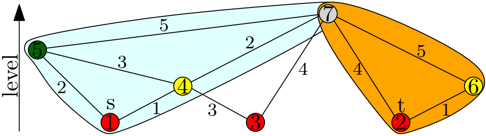
$L(t)$	2,0	6,1	7,4

Practical Implementation: HubLabels

[ADGW11]

idea:

- search spaces of contraction hierarchies form valid labels
- run upward (forward and backward) search from each vertex, store label
- sort label entries by node id



$L(s)$ | 1,0 | 4,1 | 5,2 | 7,3 |

$L(t)$ | 2,0 | 6,1 | 7,4 |

$\boxed{\infty}$

query:

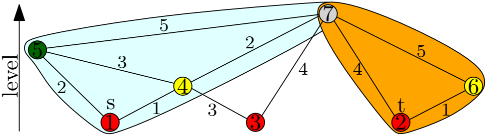
- process like merge sort

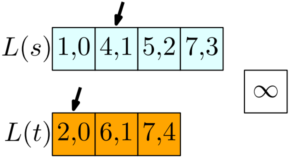
Practical Implementation: HubLabels

[ADGW11]

idea:

- search spaces of contraction hierarchies form valid labels
- run upward (forward and backward) search from each vertex, store label
- sort label entries by node id





query:

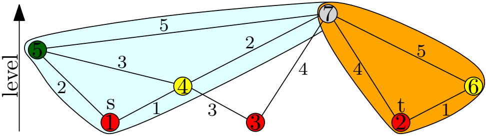
- process like merge sort

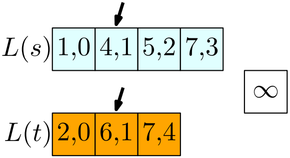
Practical Implementation: HubLabels

[ADGW11]

idea:

- search spaces of contraction hierarchies form valid labels
- run upward (forward and backward) search from each vertex, store label
- sort label entries by node id





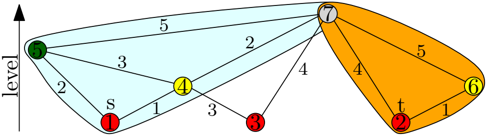
query:

- process like merge sort

[A**D**GW11]

idea:

- search spaces of contraction hierarchies form valid labels
- run upward (forward and backward) search from each vertex, store label
- sort label entries by node id



$L(s)$	1,0	4,1	5,2	7,3
$L(t)$	2,0	6,1	7,4	

$\boxed{\infty}$

query:

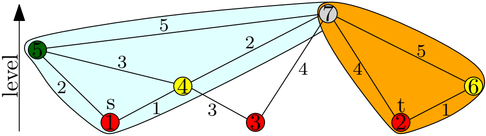
- process like merge sort

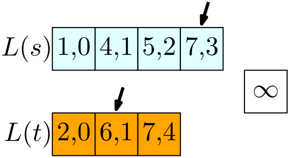
Practical Implementation: HubLabels

[ADGW11]

idea:

- search spaces of contraction hierarchies form valid labels
- run upward (forward and backward) search from each vertex, store label
- sort label entries by node id





query:

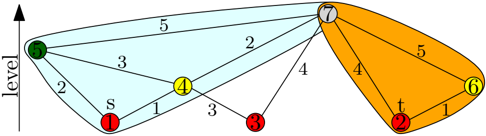
- process like merge sort

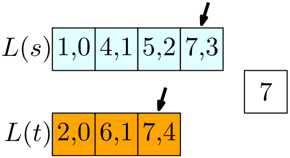
Practical Implementation: HubLabels

[ADGW11]

idea:

- search spaces of contraction hierarchies form valid labels
- run upward (forward and backward) search from each vertex, store label
- sort label entries by node id





query:

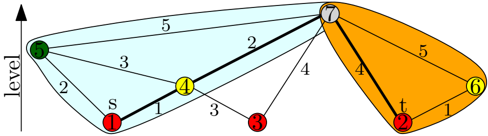
- process like merge sort
- update whenever the ids match

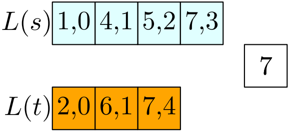
Practical Implementation: HubLabels

[ADGW11]

idea:

- search spaces of contraction hierarchies form valid labels
- run upward (forward and backward) search from each vertex, store label
- sort label entries by node id





query:

- process like merge sort
- update whenever the ids match
- very cache-efficient

problem:

- average label sizes of around 500 ⇒ 150 GB of data

Optimizations

[ADGW11]

label sizes:

- 80% of the nodes in search spaces unnecessary
- prune by bootstrapping
- SPC algorithms on small important subgraph

$\Rightarrow$ average label size shrinks to 85 ($\rightarrow$ 24 GB)

Optimizations

[ADGW11]

label sizes:

- 80% of the nodes in search spaces unnecessary
- prune by bootstrapping
- SPC algorithms on small important subgraph

$\Rightarrow$ average label size shrinks to 85 ($\rightarrow$ 24 GB)

reduce number of cache lines read:

- use compression ($\rightarrow$ 6 GB)
- define partition oracle to accelerate long-range queries
- many algorithmic low-level optimizations

$\Rightarrow$ we fetch only a few cache lines from memory

Results

	preprocessing		query
method	time [h:m]	space [GB]	time [μs]
Reach	0:15	1.5	1253.5
CH	0:05	0.4	93.5
TNR	1:52	3.7	1.8
Table Lookup	$>$ 14:01	1 208 358.7	0.056

Results

method	preprocessing time [h:m]	preprocessing space [GB]	query time [μs]
Reach	0:15	1.5	1253.5
CH	0:05	0.4	93.5
TNR	1:52	3.7	1.8
HL	2:14	21.3	0.276
Table Lookup	$>$ 14:01	1 208 358.7	0.056

Results

method	preprocessing time [h:m]	preprocessing space [GB]	query time [μs]
Reach	0:15	1.5	1253.5
CH	0:05	0.4	93.5
TNR	1:52	3.7	1.8
HL	2:14	21.3	0.276
HL compressed	2:45	5.7	0.527
Table Lookup	$>$ 14:01	1 208 358.7	0.056

Results

method	preprocessing time [h:m]	space [GB]	query time [μs]
Reach	0:15	1.5	1253.5
CH	0:05	0.4	93.5
TNR	1:52	3.7	1.8
HL	2:14	21.3	0.276
HL compressed	2:45	5.7	0.527
Table Lookup	$>$ 14:01	1 208 358.7	0.056

scientific method at work:

- observation: practical algorithms are empirically fast
- theory: highway dimension and sublinear query bounds
- prediction: the labeling algorithm is the fastest
- verification: engineered implementation guided by theory

Results

	preprocessing		query
method	time [h:m]	space [GB]	time [μs]
Reach	0:15	1.5	1253.5
CH	0:05	0.4	93.5
TNR	1:52	3.7	1.8
HL	2:14	21.3	0.276
HL compressed	2:45	5.7	0.527
Table Lookup	$>$ 14:01	1 208 358.7	0.056

scientific method at work:

- observation: practical algorithms are empirically fast
- theory: highway dimension and sublinear query bounds
- prediction: the labeling algorithm is the fastest
- verification: engineered implementation guided by theory

$\Rightarrow$ **new running time record**