Karlsruhe Institute of Technology

Algorithmen für Routenplanung

8. Sitzung, Sommersemester 2014

Ben Strasser | 12. Mai 2014

INSTITUT FÜR THEORETISCHE INFORMATIK · ALGORITHMIK · PROF. DR. DOROTHEA WAGNER

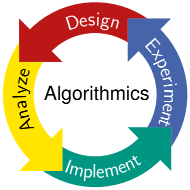
A Bit of History

- Contraction Hierarchies (CH) are a speedup technique for shortest paths in graphs.

- ”First” introduced by [GSSD08].
- Exploits the edge weights.

- Generalized to Weak CHs by [BCRW13].
- Basis for edge weight independent CH.
- Pure theoretic work.
- Connection to speed up technique for Gaussian Elimination [Geo73] discovered.

- Customizable CH (CCH) by [DSW14].
- Continuation of [BCRW13].



For didactic reasons in this course:

- First CCH / weakCH
- Then metric-dependent CH

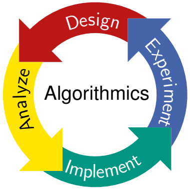
A Bit of History

- Contraction Hierarchies (CH) are a speedup technique for shortest paths in graphs.

- ”First” introduced by [GSSD08].
- Exploits the edge weights.

- Generalized to Weak CHs by [BCRW13].
- Basis for edge weight independent CH.
- Pure theoretic work.
- Connection to speed up technique for Gaussian Elimination [Geo73] discovered.

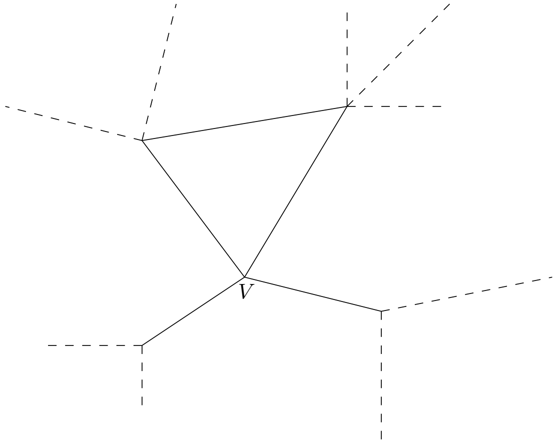
- Customizable CH (CCH) by [DSW14].
- Continuation of [BCRW13].



For didactic reasons in this course:

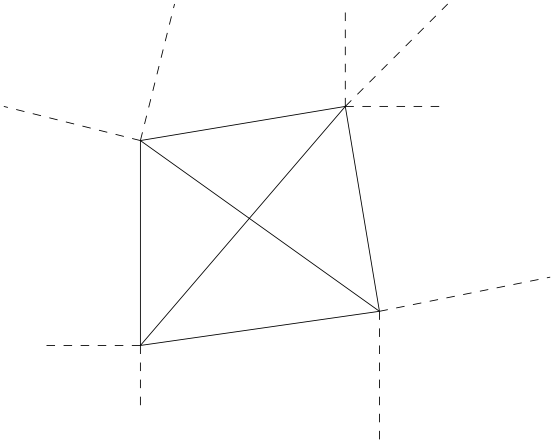
- First CCH / weakCH
- Then metric-dependent CH

Vertex-Contraction



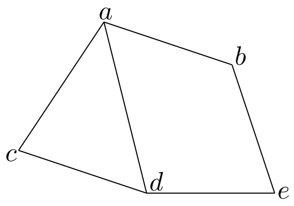
Contraction of vertex *v*: Replace *v* with a full clique.

Vertex-Contraction



Contraction of vertex *v*: Replace *v* with a full clique.

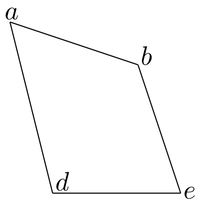
Contraction-Order



Order π: c, d, a, e, b

Contract vertices along the order π.

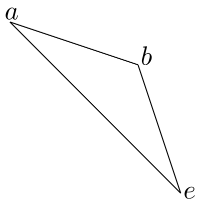
Contraction-Order



Order π: c, d, a, e, b

Contract vertices along the order π.

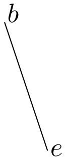
Contraction-Order



Order π: c, d, a, e, b

Contract vertices along the order π.

Contraction-Order



Order π: c, d, a, e, b

Contract vertices along the order π.

Contraction-Order

Order π: c, d, a, e, b

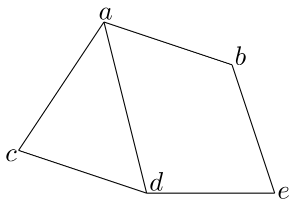
Contract vertices along the order π.

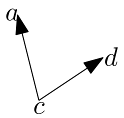
Contraction-Order

Order π: c, d, a, e, b

Contract vertices along the order π.

Contraction-Order

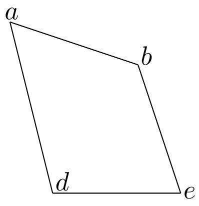


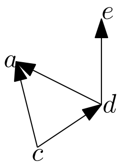


Order π: c, d, a, e, b

Build the search graph.

Contraction-Order

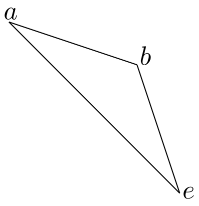


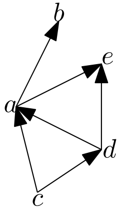


Order π: c, d, a, e, b

Build the search graph.

Contraction-Order

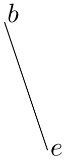


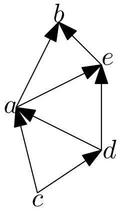


Order π: c, d, a, e, b

Build the search graph.

Contraction-Order



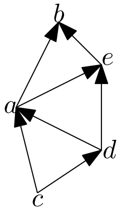


Order π: c, d, a, e, b

Build the search graph.

Contraction-Order

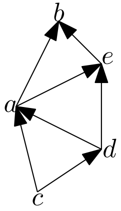
b



Order π: c, d, a, e, b

Build the search graph.

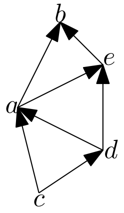
Contraction-Order



Order π: c, d, a, e, b

Build the search graph.

Contraction-Order

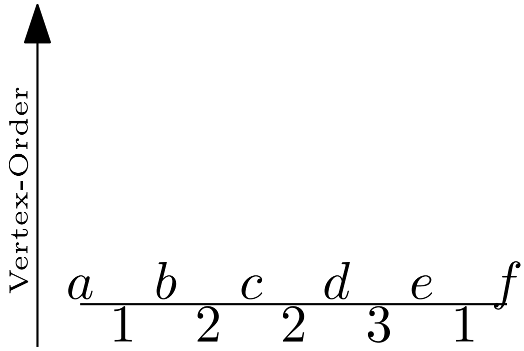


Order π: c, d, a, e, b

Definitions:

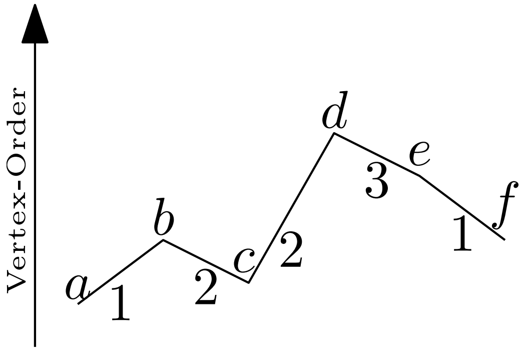
- The *search space* of a vertex *v* is the subgraph of the search graph induced by the vertices reachable from *v*.
- The remaining graph is called the *core*.
- π^{-1} is called *rank*.
- Inserted arcs are called *shortcuts*.

Contraction-Order



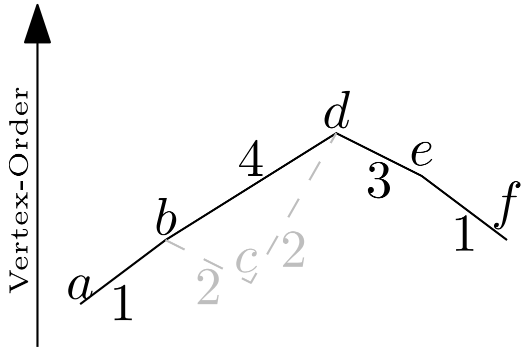
For each source-target-pair there is an up-down-path.

Contraction-Order



For each source-target-pair there is an up-down-path.

Contraction-Order



For each source-target-pair there is an up-down-path.

Dijkstra-Query

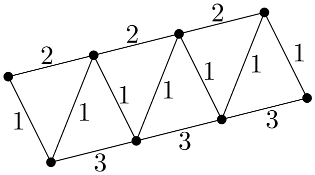
Question:
How to efficiently find an *st*-up-down-path?

Solution 1:
Run a bidirectional Dijkstra in the search-graph from *s* and *t*.

Stop Criterion:
The search in one direction can be stopped if its radius is larger than the current shortest path.
Warning: This is weaker than the general stopping criterion, because it is possible that *t* is in the search space of *s* and therefore the whole up-down path is found by the forward search from *s*.

Stall-on-Demand

Observation:
Some tentative distances are larger than needed.



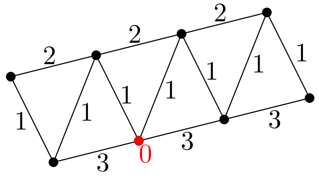
Optimization:
Denote by $d(v)$ the tentative distances and by x the vertex removed from the queue. Do not relax the outgoing arcs if an arc (x, y) exists for which

$$\exists(x, y) \in \text{search graph} : d(x) > w(x, y) + d(y)$$

holds.

Stall-on-Demand

Observation:
Some tentative distances are larger than needed.



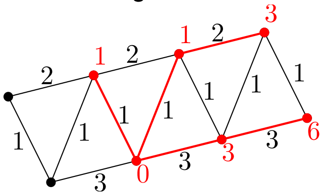
Optimization:
Denote by $d(v)$ the tentative distances and by x the vertex removed from the queue. Do not relax the outgoing arcs if an arc (x, y) exists for which

$$\exists (x, y) \in \text{search graph} : d(x) > w(x, y) + d(y)$$

holds.

Stall-on-Demand

Observation:
Some tentative distances are larger than needed.



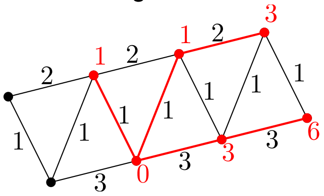
Optimization:
Denote by $d(v)$ the tentative distances and by x the vertex removed from the queue. Do not relax the outgoing arcs if an arc (x, y) exists for which

$$\exists(x, y) \in \text{search graph} : d(x) > w(x, y) + d(y)$$

holds.

Stall-on-Demand

Observation:
Some tentative distances are larger than needed.



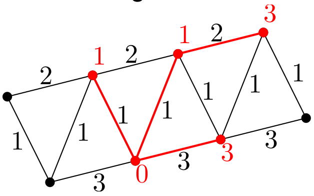
Optimization:
Denote by $d(v)$ the tentative distances and by x the vertex removed from the queue. Do not relax the outgoing arcs if an arc (x, y) exists for which

$$\exists(x, y) \in \text{search graph} : d(x) > w(x, y) + d(y)$$

holds.

Stall-on-Demand

Observation:
Some tentative distances are larger than needed.



Optimization:
Denote by $d(v)$ the tentative distances and by x the vertex removed from the queue. Do not relax the outgoing arcs if an arc (x, y) exists for which

$$\exists(x, y) \in \text{search graph} : d(x) > w(x, y) + d(y)$$

holds.

Elimination-Tree

The elimination-tree is defined as following:

$$\mathrm{parent}(x) = \underset{(x,y) \text{ is in search graph}}{\arg\min} \left(\pi^{-1}(y)\right)$$

Ancestors of x = vertices reachable from x in the search graph.

Proof by contraction:

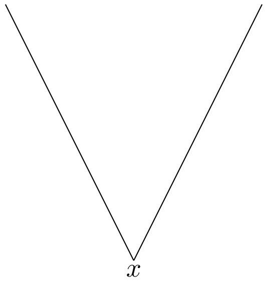
Elimination-Tree

The elimination-tree is defined as following:

$$\text{parent}(x) = \underset{(x,y)\text{ is in search graph}}{\arg\min} \left(\pi^{-1}(y)\right)$$

Ancestors of x = vertices reachable from x in the search graph.

Proof by contraction:



The search space of x.

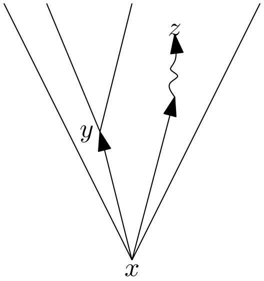
Elimination-Tree

The elimination-tree is defined as following:

$$\text{parent}(x) = \underset{(x,y)\text{ is in search graph}}{\arg\min} \left(\pi^{-1}(y)\right)$$

Ancestors of x = vertices reachable from x in the search graph.

Proof by contraction:



Let $\text{parent}(x) = y$.
Assume that z is in the search space of x but not of y.

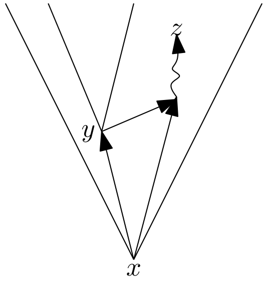
Elimination-Tree

The elimination-tree is defined as following:

$$\text{parent}(x) = \underset{(x,y)\text{ is in search graph}}{\arg\min} \left(\pi^{-1}(y)\right)$$

Ancestors of x = vertices reachable from x in the search graph.

Proof by contraction:



By construction this arc must exist.

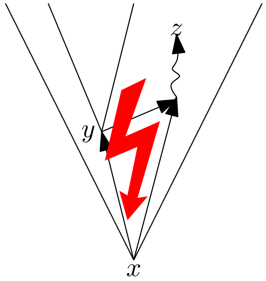
Elimination-Tree

The elimination-tree is defined as following:

$$\mathrm{parent}(x) = \underset{(x,y)\text{ is in search graph}}{\arg\min} \left(\pi^{-1}(y)\right)$$

Ancestors of x = vertices reachable from x in the search graph.

Proof by contraction:



z must be in the search space of y.

Elimination-Tree-Query

While not at root:

- If s comes before t in order:
  - Relax all outgoing arcs of s in the search graph.
  - $s \leftarrow \mathrm{parent}(s)$
- Else:
  - Relax all outgoing arcs of t in the search graph.
  - $t \leftarrow \mathrm{parent}(t)$

Bonus:

- No priority queue.
- Works with negative weights.

Vertex Separators

Vertex Separator

A *vertex separator* S of a graph $G = (V, E)$ is a vertex subset such that S's removal divides G into two parts $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$.

Balanced Vertex Separator

For a *balanced vertex separator* we require $|V_1| \leq \frac{2}{3}n$ and $|V_2| \leq \frac{2}{3}n$.

Recursive Balanced Vertex Separator

A graph G has *recursive balanced vertex separators* of size $O(|V|^\alpha)$ if G has a balanced vertex separator with at most $O(|V|^\alpha)$ vertices and G_1 and G_2 have recursive balanced vertex separators of sizes $O(|V_1|^\alpha)$ and $O(|V_2|^\alpha)$.

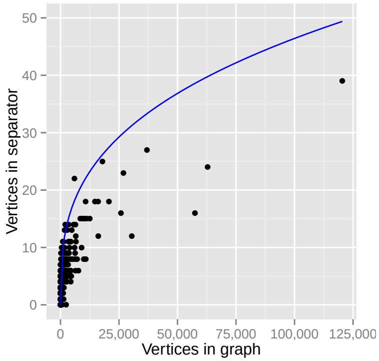
Vertex Separators

Example

All planar graphs have $O(n^{\frac{1}{2}})$ recursive balanced vertex separators.

Proof: See planar separator theorem in lecture on planar graphs.

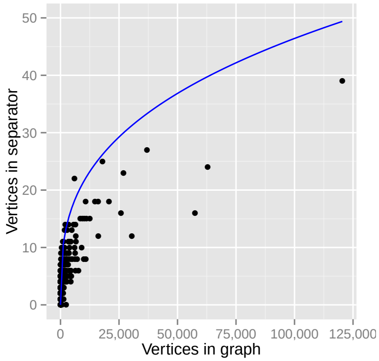
Vertex Separators



Separators for road graph around Karlsruhe.
The blue function is $y = \sqrt[3]{x}$.

Assumption: road graphs have $O(\sqrt[3]{x})$ recursive balanced separators.

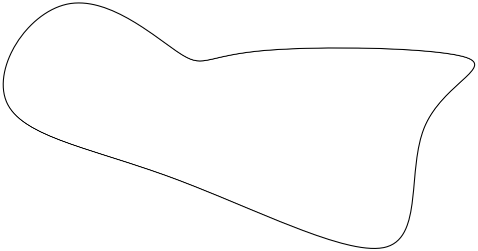
Vertex Separators



Separators for road graph around Karlsruhe.
The blue function is $y = \sqrt[3]{x}$.

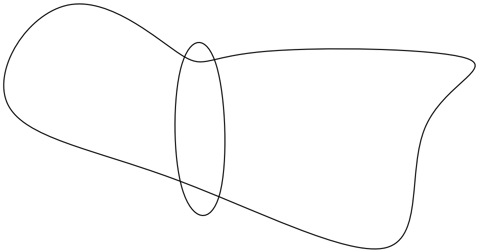
Assumption: road graphs have $O(\sqrt[3]{x})$ recursive balanced separators.

Nested Dissection



Order:

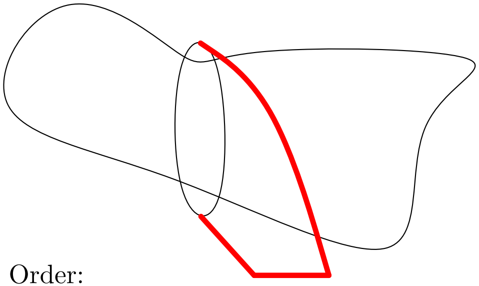
Nested Dissection



Order:

Find a small balanced vertex separator.

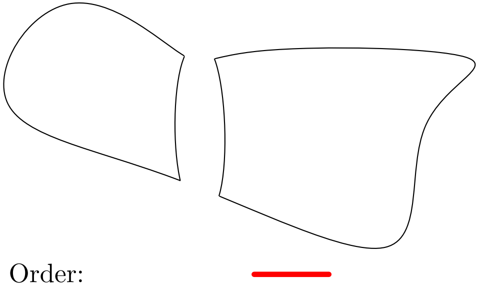
Nested Dissection



Order:

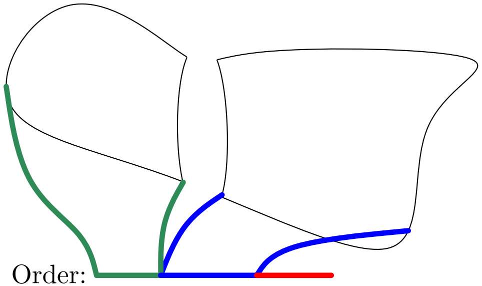
The last vertices in the order are the separator.

Nested Dissection



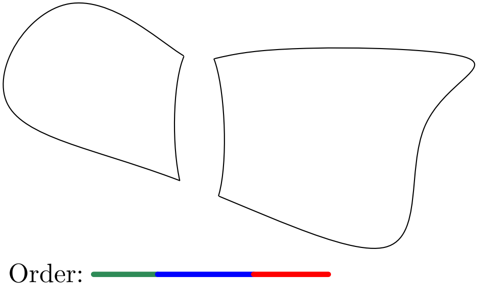
Remove the separator.

Nested Dissection



Recurse on both parts.

Nested Dissection



The contraction order.

Search Space Sizes

At each dissection level the search space contains at most one separator. The number of vertices in the search space is therefore bounded by:

$$\begin{aligned} & \sum_{i=0}^{\infty} O\left(\sqrt[3]{\left(\frac{2}{3}\right)^i n}\right) \\ = \quad & O\left(\sqrt[3]{n} \cdot \sum_{i=0}^{\infty} \left(\sqrt[3]{\frac{2}{3}}\right)^i\right) \\ = \quad & O\left(\sqrt[3]{n}\right) \end{aligned}$$

Nested Dissection

Approximation

Let G be a n-vertex graph with

- a minimum balanced separator with $\Theta(n^\alpha)$ vertices
- recursive minimum balanced separators with $O(n^\alpha)$ vertices

then a ND-order approximates the average and maximum search space in terms of vertices and arcs within a constant factor.

Key ideas:

- The separators form full cliques in the CH.
- The top level separator is the biggest and dominates all other separators.

Detailed in Section 4 of [DSW14].

Constructing the Search Graph

Option 1:
Use a dynamic adjacency array (i.e. one where arcs can be inserted).

Works, but one can do better.

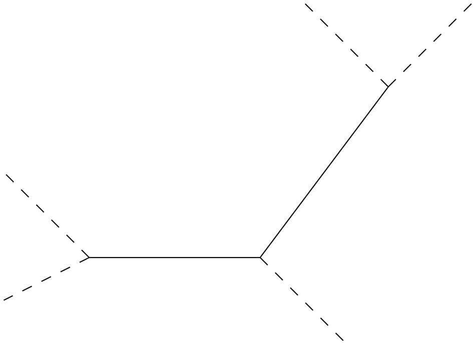
Constructing the Search Graph

Option 1:
Use a dynamic adjacency array (i.e. one where arcs can be inserted).

Works, but one can do better.

Constructing the Search Graph

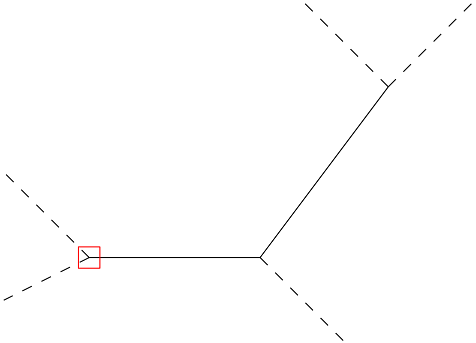
Option 2: Maintain an independent set of virtually contracted vertices. The neighborhood of v in the core is the union of v's actual neighborhood and the neighborhoods of all virtually contracted neighbors of v.



When two virtual vertices become neighbors, then remove one and rewire the edges. **Advantage**: Faster than Option 1 and linear memory usage as no edges are inserted.

Constructing the Search Graph

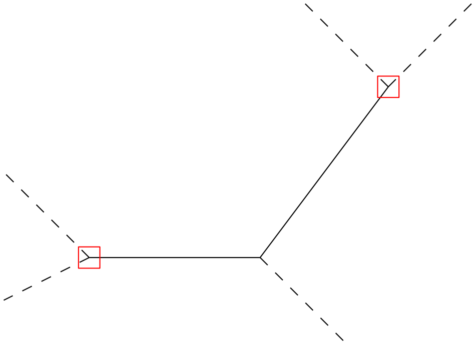
Option 2: Maintain an independent set of virtually contracted vertices. The neighborhood of *v* in the core is the union of *v*'s actual neighborhood and the neighborhoods of all virtually contracted neighbors of *v*.



When two virtual vertices become neighbors, then remove one and rewire the edges. **Advantage**: Faster than Option 1 and linear memory usage as no edges are inserted.

Constructing the Search Graph

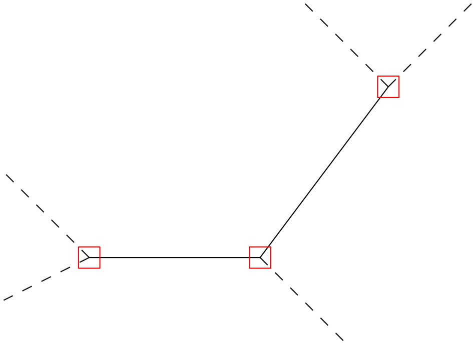
Option 2: Maintain an independent set of virtually contracted vertices. The neighborhood of *v* in the core is the union of *v*'s actual neighborhood and the neighborhoods of all virtually contracted neighbors of *v*.



When two virtual vertices become neighbors, then remove one and rewire the edges. **Advantage**: Faster than Option 1 and linear memory usage as no edges are inserted.

Constructing the Search Graph

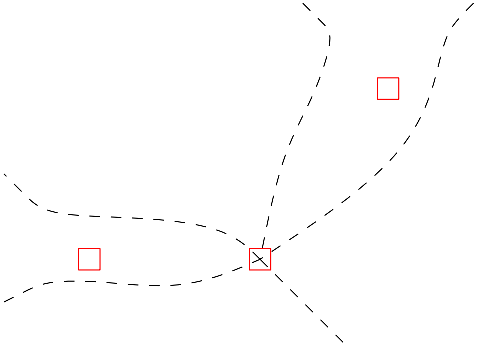
Option 2: Maintain an independent set of virtually contracted vertices. The neighborhood of *v* in the core is the union of *v*'s actual neighborhood and the neighborhoods of all virtually contracted neighbors of *v*.



When two virtual vertices become neighbors, then remove one and rewire the edges. **Advantage**: Faster than Option 1 and linear memory usage as no edges are inserted.

Constructing the Search Graph

Option 2: Maintain an independent set of virtually contracted vertices. The neighborhood of *v* in the core is the union of *v*'s actual neighborhood and the neighborhoods of all virtually contracted neighbors of *v*.



When two virtual vertices become neighbors, then remove one and rewire the edges. **Advantage**: Faster than Option 1 and linear memory usage as no edges are inserted.

Triangles

Triangle

A *triangle* (x, y, z) is a subgraph such that edges $\{x, y\}$, $\{x, z\}$ and $\{y, z\}$ exist.

Lower Triangle

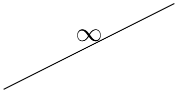
A *lower triangle* (x, y, z) of the edge $\{x, y\}$ is one where z has a rank smaller than x and y.

Intermediate & Upper Triangles

Similar but z's rank is between those of x and y or the largest.

Customization

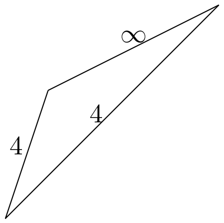
Normal Customization



Some arc in the CH.

Customization

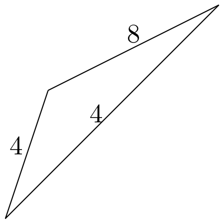
Normal Customization



For every lower triangle the triangle inequality must hold.

Customization

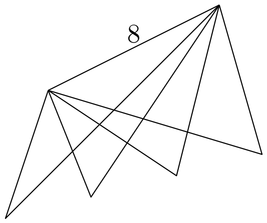
Normal Customization



For every lower triangle the triangle inequality must hold.

Customization

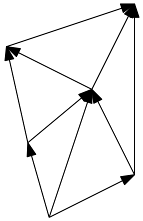
Normal Customization



For every arc enumerate all lower triangles.

Customization

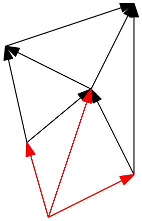
Normal Customization



Iterate over all arcs increasing by level.

Customization

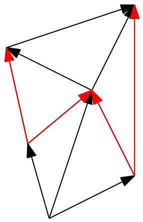
Normal Customization



Iterate over all arcs increasing by level.

Customization

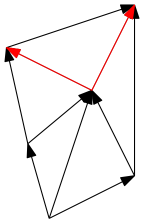
Normal Customization



Iterate over all arcs increasing by level.

Customization

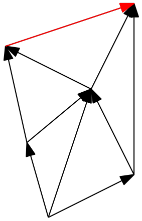
Normal Customization



Iterate over all arcs increasing by level.

Customization

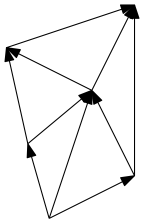
Normal Customization



Iterate over all arcs increasing by level.

Customization

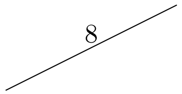
Normal Customization



CH-query now works.

Customization

Perfect Customization

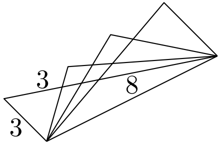


For every arc enumerate all intermediate & upper triangles.
Iterate over all arcs decreasing by level.

If there is an arc, its weight is the shortest path.
(A perfect customization is not needed for correct queries).

Customization

Perfect Customization

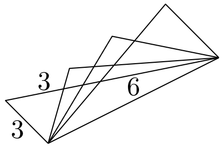


For every arc enumerate all intermediate & upper triangles.
Iterate over all arcs decreasing by level.

If there is an arc, its weight is the shortest path.
(A perfect customization is not needed for correct queries).

Customization

Perfect Customization



For every arc enumerate all intermediate & upper triangles.
Iterate over all arcs decreasing by level.

If there is an arc, its weight is the shortest path.
(A perfect customization is not needed for correct queries).

Enumerating all Lower Triangles

Option 1: Store for each arc a lower triangle list.
Problem: Needs a lot of memory.

Option 2:

- Store the reverse search graph.
- Order the outgoing arcs by head node ID.
- To enumerate all lower triangles *xyz* of the arc *xy* do a merge-sort-like scan over the outgoing arcs of *x* and *y*. If the arcs *xz* and *yz* exist the a new triangle has been found.

Problem: Somewhat slower than Option 1.

Path Unpacking

Basic Algorithm:

- Extract an up-down path.
- Iteratively replace all shortcuts with the original arcs.

Question: Which are the two original arcs of a shortcut?

- Option 1: Store for each shortcut the pair of original arcs.
  Problem: Depends on the metric.
- Option 2: Enumerate for each arc the lower triangles, and pick one such that the weights match.

One-Way Streets

To support one-way streets:

- Store two metrics.
- Use one in the forward search and the other one in the backward search.
- One-way streets have weight ∞.
- The search graph is stored only once. Each arc has 2 weights.

Sparse Matrices

Goal: Upper triangle matrix using Gauß elinimation

$$\begin{bmatrix} 1 & -1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & -2 \\ 1 & 0 & -1 & -1 & 0 \\ -1 & 0 & -1 & -2 & 0 \\ 1 & -1 & 0 & 0 & 1 \end{bmatrix}$$

Notice: 4 zeros in upper triangle part.

Sparse Matrices

Goal: Upper triangle matrix using Gauß elinimation

$$\begin{bmatrix} 1 & -1 & 1 & 1 & 1 \\ 0 & 2 & -1 & -1 & -3 \\ 0 & 1 & -2 & -2 & -1 \\ 0 & -1 & 0 & -1 & 1 \\ 0 & 0 & -1 & -1 & 0 \end{bmatrix}$$

Sparse Matrices

Goal: Upper triangle matrix using Gauß elinimation

$$\begin{bmatrix} 1 & -1 & 1 & 1 & 1 \\ 0 & 2 & -1 & -1 & -3 \\ 0 & 0 & -\frac{3}{2} & -\frac{3}{2} & \frac{1}{2} \\ 0 & 0 & -\frac{1}{2} & -\frac{3}{2} & -\frac{1}{2} \\ 0 & 0 & -1 & -1 & 0 \end{bmatrix}$$

Sparse Matrices

Goal: Upper triangle matrix using Gauß elinimation

$$\begin{bmatrix} 1 & -1 & 1 & 1 & 1 \\ 0 & 2 & -1 & -1 & -3 \\ 0 & 0 & -\frac{3}{2} & -\frac{3}{2} & \frac{1}{2} \\ 0 & 0 & 0 & -1 & -\frac{2}{3} \\ 0 & 0 & 0 & 0 & -\frac{1}{3} \end{bmatrix}$$

Notice: No zero left in upper triangle part. $\rightarrow$:(

Sparse Matrices

Idea: Reorder first and last column and row.

$$\begin{bmatrix} 1 & -1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & -2 \\ 1 & 0 & -1 & -1 & 0 \\ -1 & 0 & -1 & -2 & 0 \\ 1 & -1 & 0 & 0 & 1 \end{bmatrix}$$

Notice: All 4 zeros conserved $\rightarrow$:)

Sparse Matrices

Idea: Reorder first and last column and row.

$$\begin{bmatrix} 1 & -1 & 0 & 0 & 1 \\ -2 & 1 & 0 & 0 & 1 \\ 0 & 0 & -1 & -1 & 1 \\ 0 & 0 & -1 & -2 & -1 \\ 1 & -1 & 1 & 1 & 1 \end{bmatrix}$$

Notice: All 4 zeros conserved $\rightarrow$:)

Sparse Matrices

Idea: Reorder first and last column and row.

$$\begin{bmatrix} 1 & -1 & 0 & 0 & 1 \\ 0 & -1 & 0 & 0 & 3 \\ 0 & 0 & -1 & -1 & 1 \\ 0 & 0 & -1 & -2 & -1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

Notice: All 4 zeros conserved $\rightarrow$:)

Sparse Matrices

Idea: Reorder first and last column and row.

$$\begin{bmatrix} 1 & -1 & 0 & 0 & 1 \\ 0 & -1 & 0 & 0 & 3 \\ 0 & 0 & -1 & -1 & 1 \\ 0 & 0 & 0 & -1 & -2 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Notice: All 4 zeros conserved $\rightarrow$:)

Sparse Matrices

Idea: Reorder first and last column and row.

$$\begin{bmatrix} 1 & -1 & 0 & 0 & 1 \\ 0 & -1 & 0 & 0 & 3 \\ 0 & 0 & -1 & -1 & 1 \\ 0 & 0 & 0 & -1 & -2 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Notice: All 4 zeros conserved $\rightarrow$:)

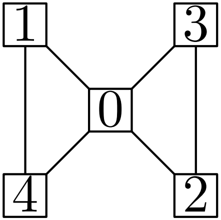
Sparse Matrices

Why does this work?

Sparse Matrices

Interpret every non-zero as edge of an adjacency matrix.

$$\begin{bmatrix} 1 & -1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & -2 \\ 1 & 0 & -1 & -1 & 0 \\ -1 & 0 & -1 & -2 & 0 \\ 1 & -1 & 0 & 0 & 1 \end{bmatrix}$$

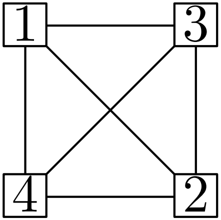


Column elimination $\leftrightarrow$ Vertex contraction
Every shortcut destroys a zero.

Sparse Matrices

Interpret every non-zero as edge of an adjacency matrix.

$$\begin{bmatrix} 1 & -1 & 1 & 1 & 1 \\ 0 & 2 & -1 & -1 & -3 \\ 0 & 1 & -2 & -2 & -1 \\ 0 & -1 & 0 & -1 & 1 \\ 0 & 0 & -1 & -1 & 0 \end{bmatrix}$$

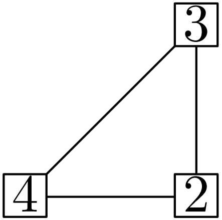


Column elimination $\leftrightarrow$ Vertex contraction
Every shortcut destroys a zero.

Sparse Matrices

Interpret every non-zero as edge of an adjacency matrix.

$$\begin{bmatrix} 1 & -1 & 1 & 1 & 1 \\ 0 & 2 & -1 & -1 & -3 \\ 0 & 0 & -\frac{3}{2} & -\frac{3}{2} & \frac{1}{2} \\ 0 & 0 & -\frac{1}{2} & -\frac{3}{2} & -\frac{1}{2} \\ 0 & 0 & -1 & -1 & 0 \end{bmatrix}$$

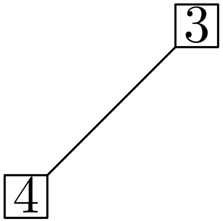


Column elimination $\leftrightarrow$ Vertex contraction
Every shortcut destroys a zero.

Sparse Matrices

Interpret every non-zero as edge of an adjacency matrix.

$$\begin{bmatrix} 1 & -1 & 1 & 1 & 1 \\ 0 & 2 & -1 & -1 & -3 \\ 0 & 0 & -\frac{3}{2} & -\frac{3}{2} & \frac{1}{2} \\ 0 & 0 & 0 & -1 & -\frac{2}{3} \\ 0 & 0 & 0 & 0 & -\frac{1}{3} \end{bmatrix}$$



Column elimination $\leftrightarrow$ Vertex contraction
Every shortcut destroys a zero.

Sparse Matrices

Interpret every non-zero as edge of an adjacency matrix.

$$\begin{bmatrix} 1 & -1 & 1 & 1 & 1 \\ 0 & 2 & -1 & -1 & -3 \\ 0 & 0 & -\frac{3}{2} & -\frac{3}{2} & \frac{1}{2} \\ 0 & 0 & 0 & -1 & -\frac{2}{3} \\ 0 & 0 & 0 & 0 & -\frac{1}{3} \end{bmatrix}$$

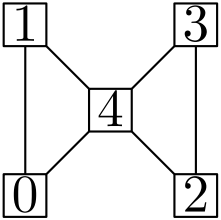
$\boxed{4}$

Column elimination $\leftrightarrow$ Vertex contraction
Every shortcut destroys a zero.

Sparse Matrices

Interpret every non-zero as edge of an adjacency matrix.

$$\begin{bmatrix} 1 & -1 & 0 & 0 & 1 \\ -2 & 1 & 0 & 0 & 1 \\ 0 & 0 & -1 & -1 & 1 \\ 0 & 0 & -1 & -2 & -1 \\ 1 & -1 & 1 & 1 & 1 \end{bmatrix}$$

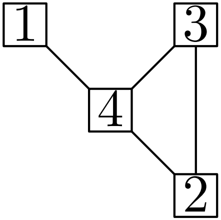


Column elimination $\leftrightarrow$ Vertex contraction
Every shortcut destroys a zero.

Sparse Matrices

Interpret every non-zero as edge of an adjacency matrix.

$$\begin{bmatrix} 1 & -1 & 0 & 0 & 1 \\ 0 & -1 & 0 & 0 & 3 \\ 0 & 0 & -1 & -1 & 1 \\ 0 & 0 & -1 & -2 & -1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

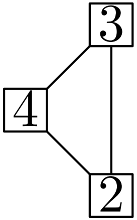


Column elimination $\leftrightarrow$ Vertex contraction
Every shortcut destroys a zero.

Sparse Matrices

Interpret every non-zero as edge of an adjacency matrix.

$$\begin{bmatrix} 1 & -1 & 0 & 0 & 1 \\ 0 & -1 & 0 & 0 & 3 \\ 0 & 0 & -1 & -1 & 1 \\ 0 & 0 & -1 & -2 & -1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

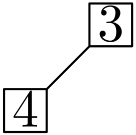


Column elimination $\leftrightarrow$ Vertex contraction
Every shortcut destroys a zero.

Sparse Matrices

Interpret every non-zero as edge of an adjacency matrix.

$$\begin{bmatrix} 1 & -1 & 0 & 0 & 1 \\ 0 & -1 & 0 & 0 & 3 \\ 0 & 0 & -1 & -1 & 1 \\ 0 & 0 & 0 & -1 & -2 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$



Column elimination $\leftrightarrow$ Vertex contraction
Every shortcut destroys a zero.

Sparse Matrices

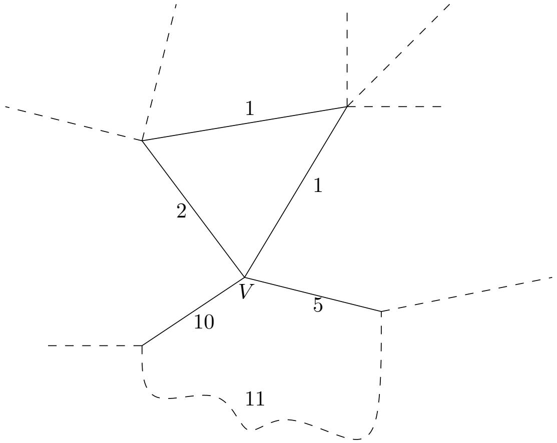
Interpret every non-zero as edge of an adjacency matrix.

$$\begin{bmatrix} 1 & -1 & 0 & 0 & 1 \\ 0 & -1 & 0 & 0 & 3 \\ 0 & 0 & -1 & -1 & 1 \\ 0 & 0 & 0 & -1 & -2 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$\boxed{4}$

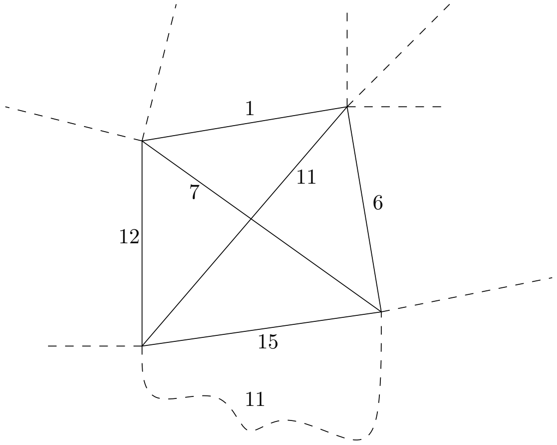
Column elimination $\leftrightarrow$ Vertex contraction
Every shortcut destroys a zero.

Are all shortcuts needed?



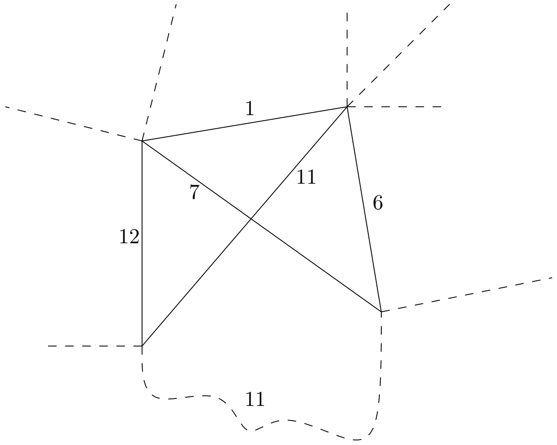
Insert an only if all paths through the core are longer.

Are all shortcuts needed?



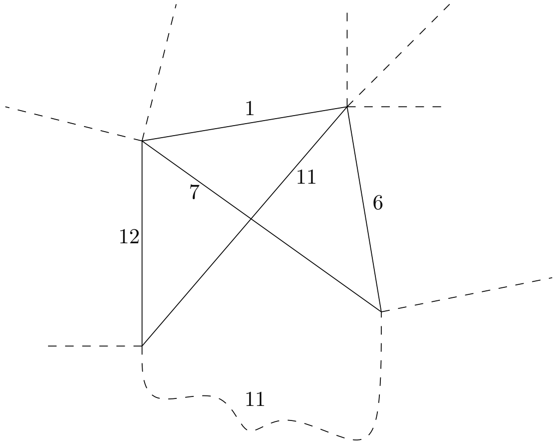
Insert an only if all paths through the core are longer.

Are all shortcuts needed?



Insert an only if all paths through the core are longer.

Are all shortcuts needed?



Insert an only if all paths through the core are longer.

WeakCH

Observation:

- For a fixed metric not all search graph arcs are needed.

Definitions:

- A *weakCH* is a CH that contains for a given order and metric at least the arcs needed for query.
- A *maximum weakCH* contain all arcs inserted for a given order.
  Notice: A maximum weakCH is metric independent.
  (This is what we computed up to now.)

For the rest of this lecture:

- We merge preprocessing and customization.
- In the preprocessing phase the metric and graph are known.

Witness Search

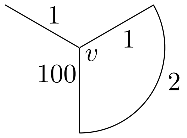
Algorithm:

- Before inserting an arc (x, y) run a bidirectional Dijkstra restricted to the core.
- If the shortest xy-path is longer than the new arc's weight, then the new arc is not needed.

Optimization:

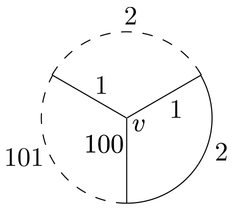
- Stop the search after a fixed amount of steps.
- CH bigger than needed but witness search is faster.
- Common optimization, but the algorithm is feasible without.

Witness Search



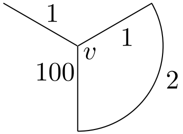
Consider this graph.
Contract *v*.

Witness Search



Two potential shortcuts.
In which order should we test them?

Witness Search

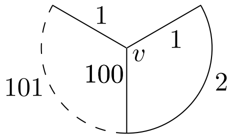


Option 1: First the long arc then the short one.

No witness for the long one $\rightarrow$ insert

No witness for the short one $\rightarrow$ insert

Witness Search

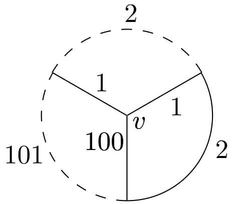


Option 1: First the long arc then the short one.

No witness for the long one → insert

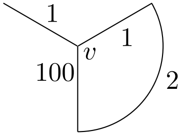
No witness for the short one → insert

Witness Search



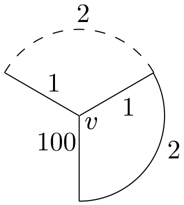
Option 1: First the long arc then the short one.
No witness for the long one $\rightarrow$ insert
No witness for the short one $\rightarrow$ insert

Witness Search



Option 2: First the short arc then the long one.
No witness for the short one $\rightarrow$ insert
Witness found for the long one $\rightarrow$ prune

Witness Search

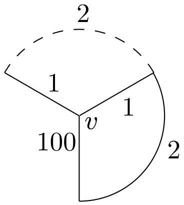


Option 2: First the short arc then the long one.

No witness for the short one $\rightarrow$ insert

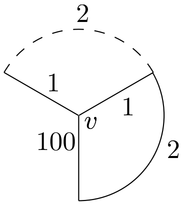
Witness found for the long one $\rightarrow$ prune

Witness Search



Option 2: First the short arc then the long one.
No witness for the short one $\rightarrow$ insert
Witness found for the long one $\rightarrow$ prune

Witness Search



$\rightarrow$ Test the arcs increasing by weight to insert as few arcs as possible.

Metric-Dependent-Orders

Question: Can we get better orders for a fixed metric?

Intuition:

- The node contracted first (last) is the least (most) important one.

Two approaches:

- Bottom-Up [GSSD08]: Guess the least important vertex.
- Top-Down [ADGW12]: Guess the most important vertex.

Bottom-Up

For each vertex we define its importance as:

$$I(v) = \ell(v) + \frac{|A(v)|}{|D(v)|} + \frac{\sum_{a \in A(v)} h(a)}{\sum_{a \in D(v)} h(a)}$$

where

- $\ell(v)$ is an estimate for the level of v in the search graph.
- $A(v)$ is the set of arcs inserted if we would contract v.
- $D(v)$ is the set of arcs removed if we would contract v.
- $h(a)$ is the number of original arcs in the path represented by the shortcut a.

Details:

- Initially $\ell(v) = 0$.
- If v is contracted then for each neighbor u we do $\ell(u) \leftarrow \max(\ell(u), \ell(v) + 1)$.
- To determine $A(v)$ and $D(v)$ we need to run witness searches.
- Many other ”importance” heuristics exist.

Bottom-Up

Algorithm:

- Maintain a priority queue of vertices weighted by their importance.
- Pop the min vertex v.
- Put v into the order.
- Contract v.
- Recompute the importances of all neighbors of v and decrease/increase their key in the priority queue.
- Repeat until the queue is empty.

Why does this work?

- No one knows for sure. . .

but it works . . .

and scales to large road graphs.

Bottom-Up

Algorithm:

- Maintain a priority queue of vertices weighted by their importance.
- Pop the min vertex v.
- Put v into the order.
- Contract v.
- Recompute the importances of all neighbors of v and decrease/increase their key in the priority queue.
- Repeat until the queue is empty.

Why does this work?

- No one knows for sure...

but it works ...

and scales to large road graphs.

Top-Down

Idea:

- The most important vertex is the one that is part of the most shortest paths.
- We say that v covers a st-shortest path if v is on the path.

Basic Algorithm:

- Denote by U the set of *uncovered* paths. (Initially U is the set of all shortest paths.)
- Find the vertex v that covers the most paths in U.
- Put v into the order.
- Remove all paths from U covered by v.

Running Time:

- Basic algorithm has running time in $O(n^3)$.
- Can be improved to $O(n^2)$. (See [ADGW12])

One-Way Streets

To support one-way streets:

- Some arcs are pruned only for the forward or the backward search.
- Store two different search graphs.

Experiments

average travel-time metric query running times

CCH-Dijkstra :	0.81 ms
CCH-Stall :	0.85 ms
CCH-Tree :	0.41 ms
CH-Dijkstra :	0.28 ms
CH-Stall :	0.11 ms

average distance metric query running times

CCH-Dijkstra :	0.87 ms
CCH-Stall :	1.00 ms
CCH-Tree :	0.42 ms
CH-Dijkstra :	2.66 ms
CH-Stall :	0.54 ms

as always: instance is DIMACS Europe
sequential unless mentioned

Experiments

customization

- running time: 0.4s
- parallelized with 16 cores and SIMD/SSE.
- details not covered in this course, see [DSW14]

search graph construction time (no order)

dynamic adjacency array :	305.8 s
contraction graph :	15.5 s

as always: instance is DIMACS Europe
sequential unless mentioned

Experiments

nested dissection order computation time

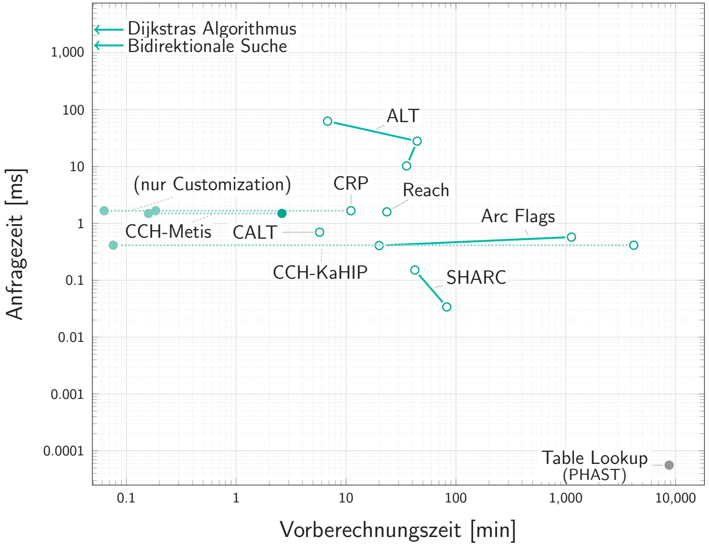
- Depends on the partitioner used to do the bisection.
- Metis is faster than KaHip but separators are larger.
- KaHip: 2.8 days
- Metis: 2.2 min
- details not covered in this course, see [DSW14]

metric-dependent order computation time

- Bottom-Up: 10-100 min (depending on importance heuristic)
- Top-Down: 29.75 h (uses tricks not covered, see [ADGW12])

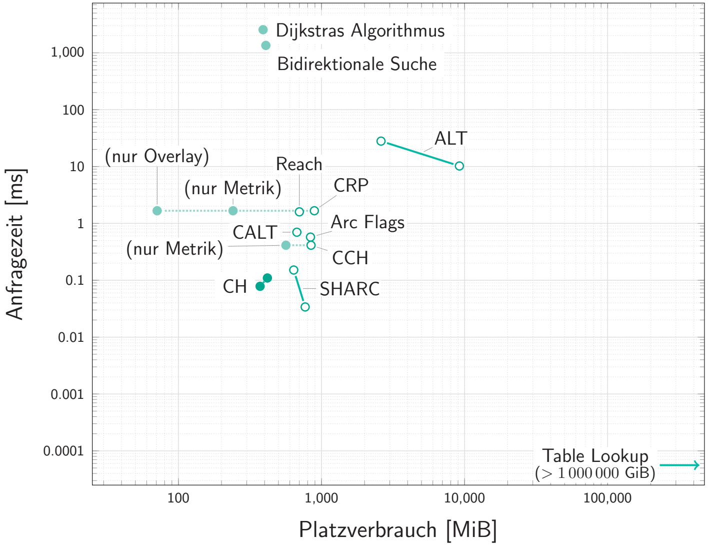
as always: instance is DIMACS Europe
sequential unless mentioned

Overview



Dijkstras Algorithmus
Bidirektionale Suche
ALT
(nur Customization)
CRP
Reach
Arc Flags
CCH-Metis
CALT
CCH-KaHIP
SHARC
Table Lookup
(PHAST)
Anfragezeit [ms]
1,000
100
10
1
0.1
0.01
0.001
0.0001
0.1
1
10
100
1,000
10,000
Vorberechnungszeit [min]

Overview



Dijkstras Algorithmus
Bidirektionale Suche
ALT
Reach
CRP
(nur Overlay)
(nur Metrik)
Arc Flags
CALT
(nur Metrik)
CCH
CH
SHARC
Table Lookup
($>$ 1 000 000 GiB)
Anfragezeit [ms]
1,000
100
10
1
0.1
0.01
0.001
0.0001
Platzverbrauch [MiB]
100
1,000
10,000
100,000

Literatur I

Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck.
Hierarchical hub labelings for shortest paths.
In *Proceedings of the 20th Annual European Symposium on Algorithms (ESA'12)*, volume 7501 of *Lecture Notes in Computer Science*, pages 24–35. Springer, 2012.

Reinhard Bauer, Tobias Columbus, Ignaz Rutter, and Dorothea Wagner.
Search-space size in contraction hierarchies.
In *Proceedings of the 40th International Colloquium on Automata, Languages, and Programming (ICALP'13)*, volume 7965 of *Lecture Notes in Computer Science*, pages 93–104. Springer, 2013.

Julian Dibbelt, Ben Strasser, and Dorothea Wagner.
Customizable contraction hierarchies.
Technical report, ITI Wagner, Department of Informatics, Karlsruhe Institute of Technology (KIT), 2014.
Technical Report on arXiv.

Literatur II

Alan George.
Nested dissection of a regular finite element mesh.
SIAM Journal on Numerical Analysis, 1973.

Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling.
Contraction hierarchies: Faster and simpler hierarchical routing in road networks.
In *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science*, pages 319–333. Springer, June 2008.