Karlsruhe Institute of Technology

Algorithmen / Algorithms II

Peter Sanders

Exercise:

Daniel Seemaier, Tobias Heuer

Institute of Theoretical Informatics

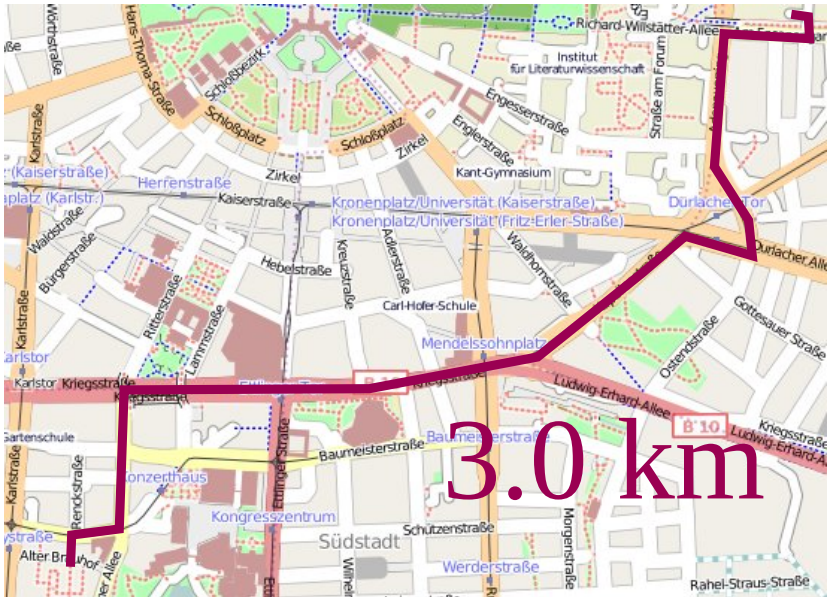
Web:

`http://algo2.iti.kit.edu/AlgorithmenII_WS20.php`

Advanced Graph Algorithms

3 Shortest Paths

Slides partially due to Rob van Stee



Input: Graph $G = (V, E)$

Cost function/edge weights $c : E \to \mathbb{R}$

Source vertex s.

Output: for all $v \in V$

Length $\mu(v)$ of the shortest path from s to v,

$\mu(v) := \min\{c(p) : p \text{ is path from } s \text{ to } v\}$

where $c(\langle e_1, \ldots, e_k \rangle) := \sum_{i=1}^{k} c(e_i)$.

Often we want a "suitable" representation of the shortest path.

General Definitions

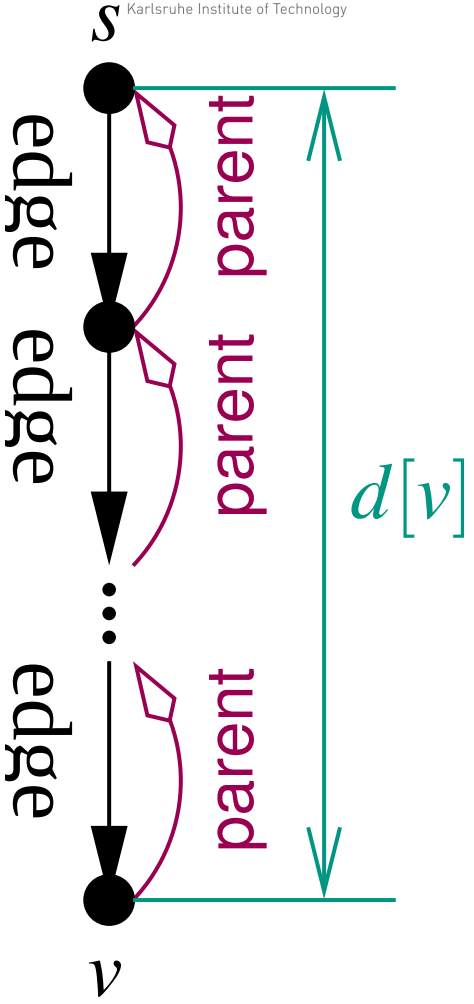
Similar to BFS we use two vertex arrays:

- $d[v]$ = current (preliminary) distance from s to v
  Invariant: $d[v] \geq \mu(v)$
- parent$[v]$ = predecessor of v
  on the (preliminary) shortest path from s to v
  Invariant: this path attests $d[v]$

Initialization:

$d[s] = 0$, parent[s] $= s$

$d[v] = \infty$, parent$[v] = \perp$



Relaxation of an edge (u,v)

if $d[u]+c(u,v)<d[v]$

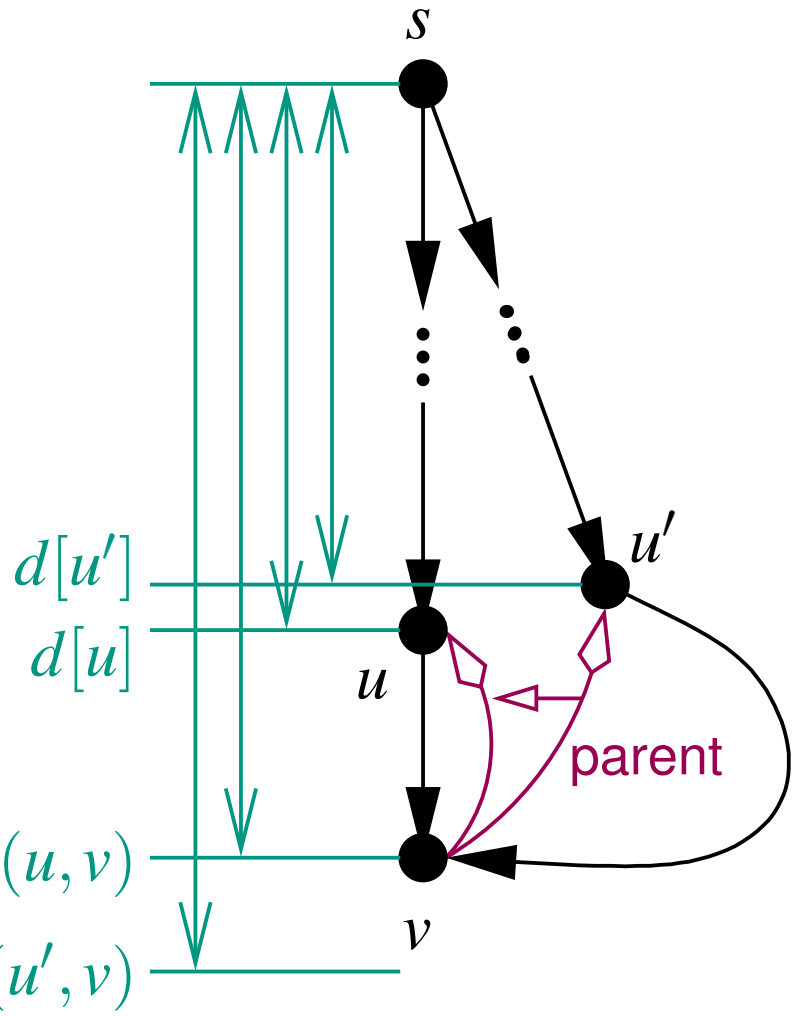
possibly $d[v]=\infty$

set $d[v]:=d[u]+c(u,v)$ and parent$[v]:=u$

Invariants are maintained!

Observation:

$d[v]$ can change several times!



Dijkstra's Algorithm: Pseudocode

initialize d, parent

all nodes are non-scanned

while $\exists$ non-scanned node u with $d[u] < \infty$

$u :=$ non-scanned node v with minimal $d[v]$

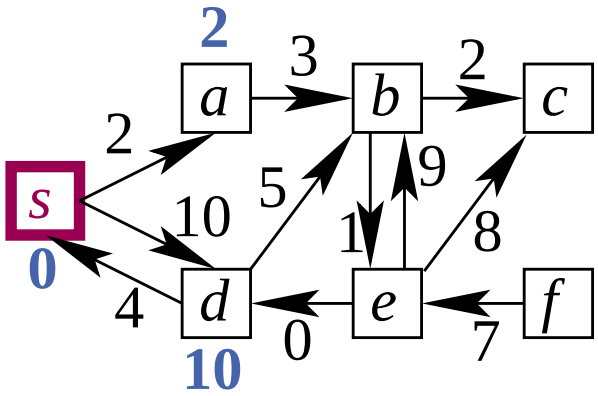
relax all edges (u,v) out of u

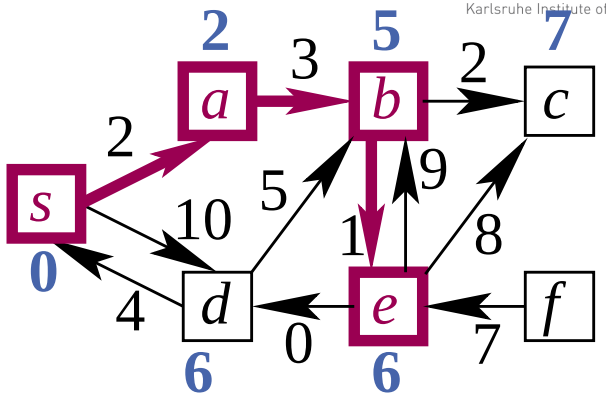
u is scanned now

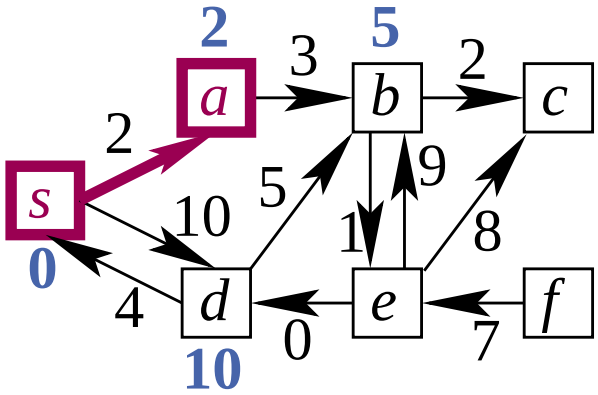
Claim: At the end, d defines the optimal distances and parent the corresponding paths (see Algo I:)

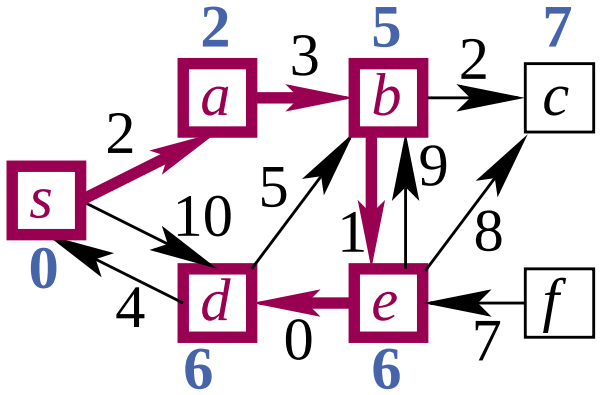
- v reachable $\Longrightarrow$ v will be scanned at some point
- v scanned $\Longrightarrow \mu(v) = d[v]$

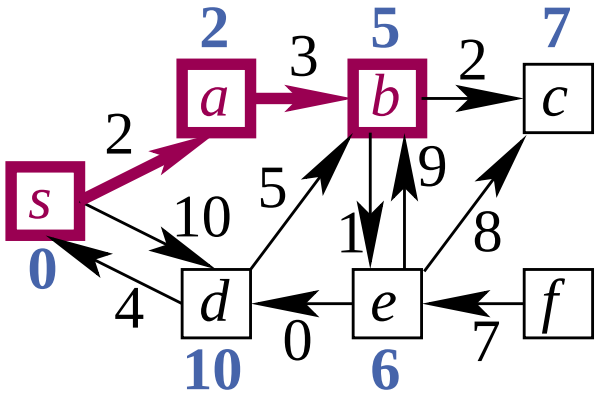
Example

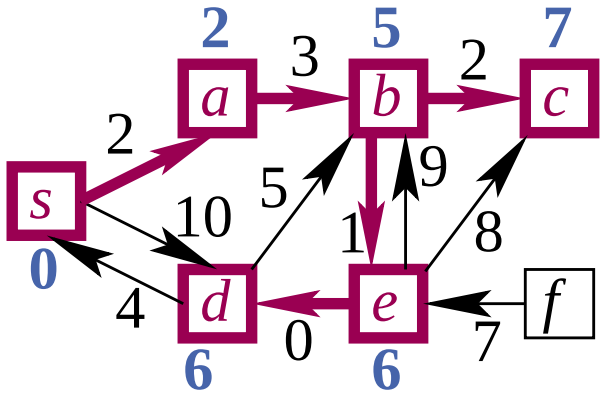

2
a
3
b
2
c
2
5
9
s
10
1
8
0
d
e
f
4
0
7
10


2
5
7
a
3
b
2
c
2
s
10
5
9
1
8
0
d
e
f
4
0
7
6
6


2
5
a
3
b
2
c
2
s
10
5
9
1
8
0
d
e
f
4
0
7
10


2
5
7
a
3
b
2
c
2
s
10
5
9
1
8
0
d
e
f
4
0
7
6
6


2
5
7
a
3
b
2
c
2
s
10
5
9
1
8
0
d
e
f
4
0
7
10
6


2
5
7
a
3
b
2
c
2
s
10
5
9
1
8
0
d
e
f
4
0
7
6
6

Running Time

$$T_{\text{Dijkstra}} = \mathrm{O}(m \cdot T_{\text{decreaseKey}}(n) + n \cdot (T_{\text{deleteMin}}(n) + T_{\text{insert}}(n)))$$

Using Fibonacci heap priority queues:

- insert $\mathrm{O}(1)$
- decreaseKey $\mathrm{O}(1)$
- deleteMin $\mathrm{O}(\log n)$ (amortized)

$$\begin{aligned} T_{\text{DijkstraFib}} &= \mathrm{O}(m \cdot 1 + n \cdot (\log n + 1)) \\ &= \mathrm{O}(m + n \log n) \end{aligned}$$

However: constant factors in $\mathrm{O}(\cdot)$ are larger when compared to binary heaps!

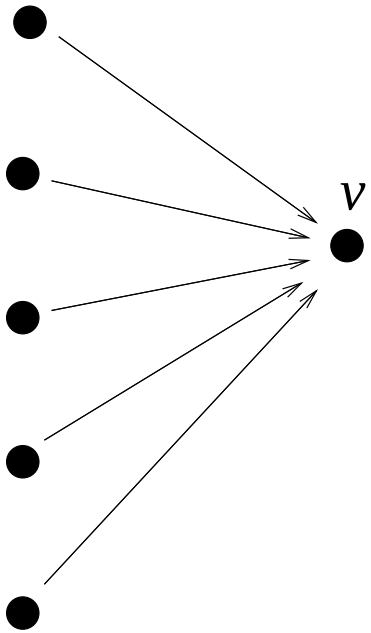
Running Time on Average

So far: $\leq m$ decreaseKeys ($\leq 1\times$ per edge)

How many decreaseKeys on average?

Model:

- Arbitrary graph G
- Arbitrary source vertex s
- Arbitrary sets $C(v)$
  of edge weights for
  incoming edges of vertex v



indegree(v)=5
C(v)={c_1,..., c_5}

Average over all possible assignments

$C(v) \rightarrow$ incoming edges of v

Example: all costs are independent and uniformly distributed

Running Time on Average

Probabilistic view: Random selection of the uniformly distributed inputs that are to be averaged

$\Longrightarrow$

we look for the expected value of the running time

Question: Difference to the expected running time for randomized algorithms?

Running Time on Average

Theorem 1. $\mathbb{E}[\#\textit{decreaseKey operations}] = \mathrm{O}\left(n \log \frac{m}{n}\right)$

Then

$$\begin{aligned}
\mathbb{E}(T_{\text{DijkstraBHeap}}) &= \mathrm{O}\left(m + n\log\frac{m}{n} \cdot T_{\text{decreaseKey}}(n)\right. \\
&\qquad \left. + n \cdot (T_{\text{deleteMin}}(n) + T_{\text{insert}}(n))\right) \\
&= \mathrm{O}\left(m + n\log\frac{m}{n}\log n + n\log n\right) \\
&= \mathrm{O}\left(m + n\log\frac{m}{n}\log n\right)
\end{aligned}$$

(previously, we had $T_{\text{DijkstraBHeap}} = \mathrm{O}((m+n)\log n)$)

($T_{\text{DijkstraFib}} = \mathrm{O}(m + n\log n)$ worst case)

Linear Running Time for Dense Graphs

$m = \Omega(n \log n \log\log n) \Rightarrow$ linear running time.

(verify)

Thus, in this case might be better than Fibonacci heaps

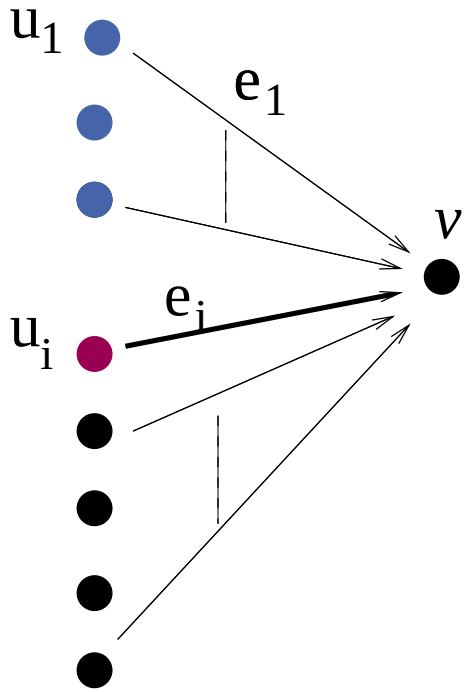
Karlsruhe Institute of Technology

Theorem 1. $\mathbb{E}[\#\textit{decreaseKey operations}] = \mathrm{O}\left(n \log \frac{m}{n}\right)$

Theorem 1. $\mathbb{E}[\#\textit{decreaseKey operations}] = O\left(n \log \frac{m}{n}\right)$

decreaseKey only performed during processing of e_i if

$$\mu(u_i) + c(e_i) < \min_{j<i}(\mu(u_j) + c(e_j)).$$



However, $\mu(u_i) \geq \mu(u_j)$ for $j < i$, and thus:

$$c(e_i) < \min_{j<i} c(e_j)$$

is a necessary condition.

Prefix minimum

Theorem 1. $\mathbb{E}[\#\textit{decreaseKey operations}] = \mathrm{O}\left(n \log \frac{m}{n}\right)$

Costs in $C(v)$ appear in random order

How many times does one find a new minimum in a random order?

Harmonic number H_k (see below)

First minimum: leads to insert(v).

Thus $\leq H_k - 1 \leq (\ln k + 1) - 1 = \ln k$ expected decreaseKeys

Theorem 1. $\mathbb{E}[\#\textit{decreaseKey operations}] = \mathrm{O}\left(n \log \frac{m}{n}\right)$

For each vertex $v \leq H_k - 1 \leq \ln k$ (expected) decreaseKeys where $k = \text{indegree}(v)$.

In total

$$\sum_{v \in V} \ln \text{indegree}(v) \leq n \ln \frac{m}{n}$$

(due to concavity of $\ln x$)

Prefix Minima in a Random Sequence

Define random variable M_n as the number of prefix minima in a sequence of n different numbers (dependent on a random permutation)

Define indicator variables $I_i := 1$ iff the i-th number is a prefix minimum.

$$\mathrm{E}[M_n] = \mathrm{E}[\sum_{i=1}^{n} I_i] \overset{\text{Lin. E}[\cdot]}{=} \sum_{i=1}^{n} \mathrm{E}[I_i]$$

$$= \sum_{i=1}^{n} \frac{1}{i} = H_n \text{ due to } \mathbb{P}[I_i = 1] = \frac{1}{i}$$

$$\underbrace{x_1, \ldots, x_{i-1}}_{<x_i?}, x_i, x_{i+1}, \ldots, x_n$$

Monotone Integer Priority Queues

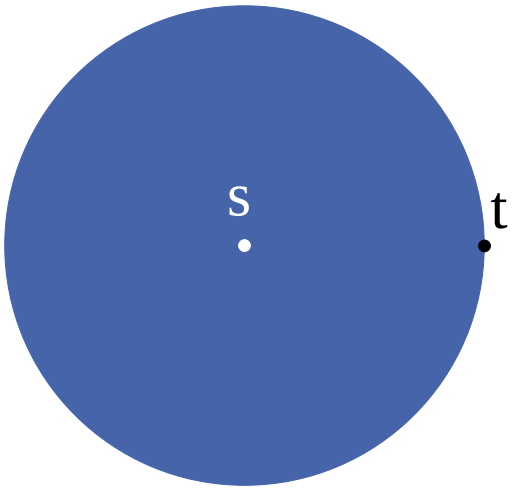
Basic idea: Application tailored data structure

Dijkstra's algorithm uses priority queue monotonically:

Operations insert and decreaseKey use distances in the form of

$d[u] + c(e)$

This value is continiously increasing



Monotone Integer Priority Queues

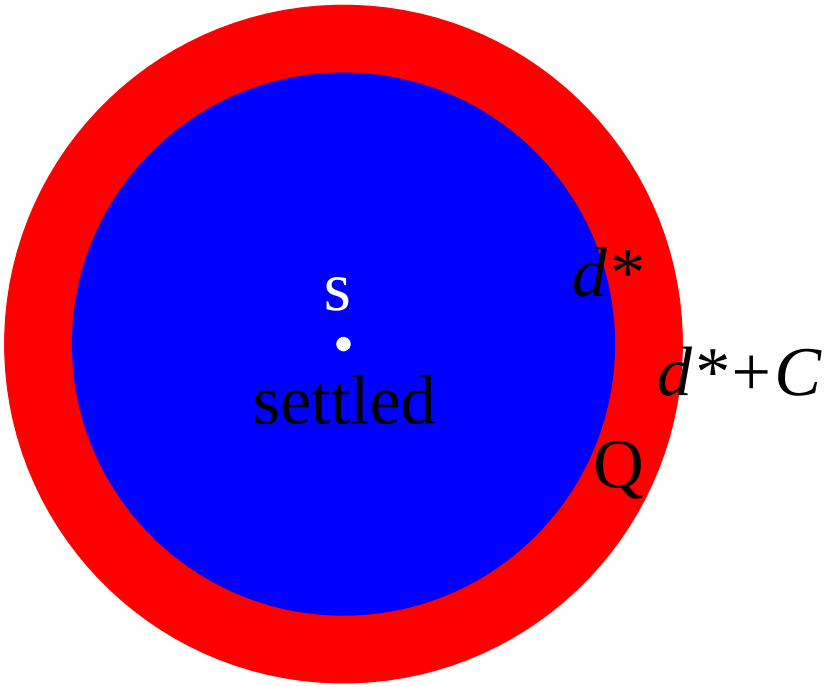
Assumption: All edge weights are integers in the interval $[0, C]$

$\Longrightarrow \forall v \in V : d[v] \leq (n-1)C$

Furthermore:

Let d^* be the last value
that was removed from Q.

In Q there are always
only vertices with distances in the
interval $[d^*, d^* + C]$.

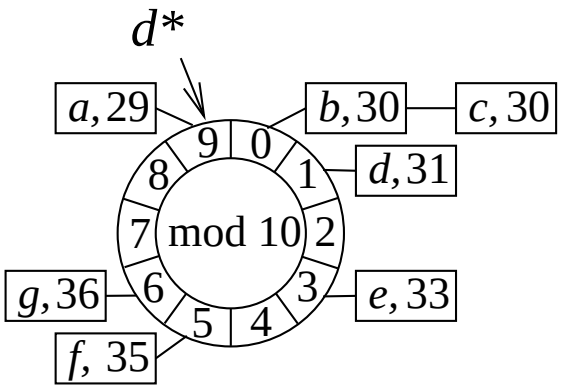


Bucket Queue

Cyclic array B of $C+1$ doubly linked lists

Vertex with distance $d[v]$ is stored in $B[d[v] \mod (C+1)]$.

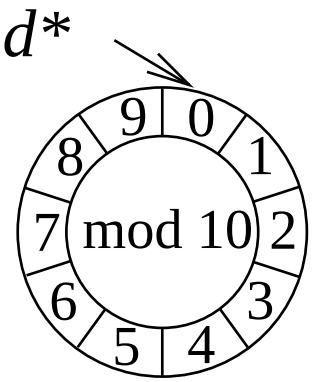
Bucket queue with C = 9



Content=

<(a,29), (b,30), (c,30), (d,31)
(e,33), (f,35), (g,36)>

Operations



Initialization: $C+1$ empty lists, $d^* = 0$

insert(v): inserts v in $B[d[v] \mod (C+1)]$ $O(1)$

decreaseKey(v): removes v from its list and adds it to $B[d[v] \mod (C+1)]$ $O(1)$

deleteMin: starts at bucket $B[d^* \mod (C+1)]$. If empty, set $d^* := d^* + 1$ and repeat. requires monotonicity!

d^* is increased at most nC times, at most n elements in total are removed from Q $\Rightarrow$

Total costs of deleteMin operations $= O(n + nC) =$ $O(nC)$.

More specifically: $O(n +$ maxPathLength$)$

Running Time Dijkstra with Bucket Queues

$$
\begin{aligned}
T_{\text{Dijkstra}} &= \mathrm{O}(m \cdot T_{\text{decreaseKey}}(n) \\
&\quad + \text{Costs of deleteMin operations} \\
&\quad + n \cdot T_{\text{insert}}(n))) \\
T_{\text{DijkstraBQ}} &= \mathrm{O}(m \cdot 1 + nC + n \cdot 1)) \\
&= \mathrm{O}(m + nC) \text{ or} \\
&= \mathrm{O}(m + \text{maxPathLength})
\end{aligned}
$$

Using radix heaps we can achieve $T_{\text{DijkstraRadix}} = \mathrm{O}(m + n \cdot \log C)$

Idea: use buckets of different sizes

Radix Heaps

We use buckets -1 till K, where $K = 1 + \lfloor \log C \rfloor$

d^* = distance that was previously removed from Q

For each vertex $v \in Q$ holds $d[v] \in [d^*, \ldots, d^* + C]$.

Consider binary representation of the possible distances in Q.

For example, let $C = 9$, in binary 1001. Then $K = 4$.

Example 1: $d^* = 10000$, then $\forall v \in Q : d[v] \in [10000, 11001]$

Example 2: $d^* = 11101$, then $\forall v \in Q : d[v] \in [11101, 100110]$

Store v in bucket $B[i]$ if $d[v]$ and d^* first differ at the i-th position,
(use $B[K]$ if $i > K$, and $B[-1]$ if they are the same)

Definition $msd(a,b)$

The position of the highest valued binary digit where a and b are different

a	$1100\mathbf{1}010$	$1010\mathbf{1}00$	1110110
b	$1100\mathbf{0}101$	$1010\mathbf{1}10$	1110110
$\text{msd}(a,b)$	3	1	-1

Using a machine instruction, we can evaluate $msd(a,b)$ very fast

Radix Heap Invariant

v is stored in bucket $B[i]$ where $i = \min(msd(d^*, d[v]), K)$.

Example 1: $d^* = 10000$, $C = 9$, $K = 4$

Bucket	$d[v]$ binary	$d[v]$
-1	10000	16
0	10001	17
1	1001*	18,19
2	101**	20–23
3	11***	24–25
4	-	-

(Nothing is stored in bucket 4)

Radix Heap Invariant

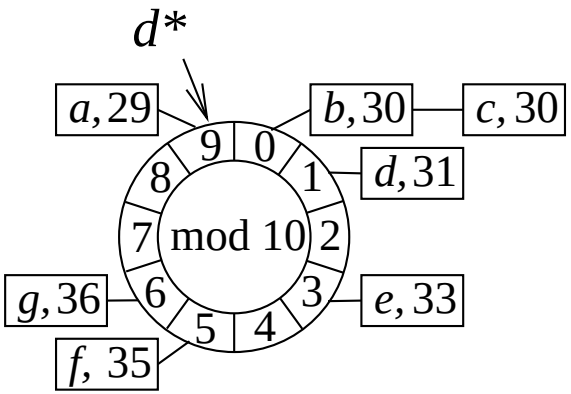
v is stored in bucket $B[i]$ where $i = \min(msd(d^*, d[v]), K)$.

Example 2: $d^* = 11101, C = 9, K = 4$

Bucket	$d[v]$ binary	$d[v]$
-1	11101	29
0	-	-
1	1111*	30,31
2	-	-
3	-	-
4	100000 and higher	32 and higher

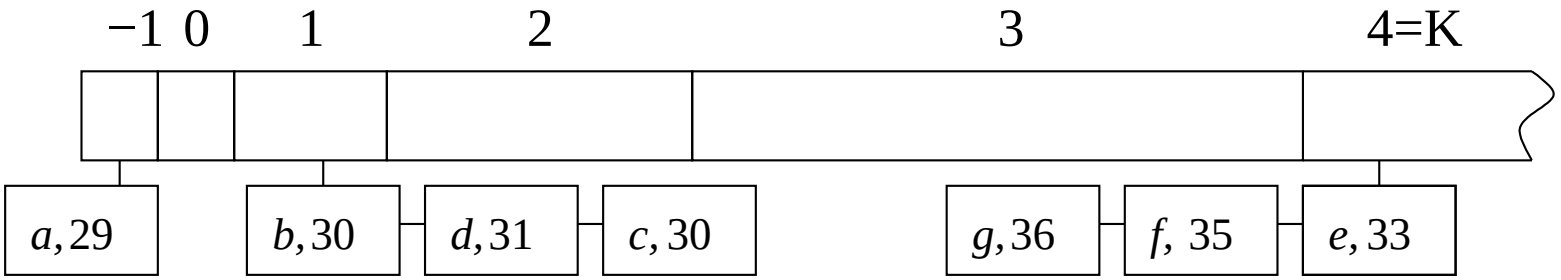
If $d[v] \geq 32$, then $msd(d^*, d[v]) > 4$!

Bucket Queues and Radix Heaps



Bucket queue with C = 9

Content=
<(a,29), (b,30), (c,30), (d,31), (e,33), (f,35), (g,36)>



Binary Radix Heap

Radix Heap: deleteMin

Function deleteMin: Element

  if $B[-1] = \emptyset$

    $i := \min\{j \in 0..K : B[j] \neq \emptyset\}$

    move $\min B[i]$ to $B[-1]$ and to d^*

    foreach $e \in B[i]$ **do** // exactly here invariant is violated !

      move e to $B[\min(msd(d^*, d[e]), K)]$

  result:= $B[-1]$.popFront

  return result

$B[0], \ldots, B[i-1]$: empty, thus do nothing.

$B[i+1], \ldots, B[K]$: msd is preserved, because old and new d^*

same for all bits $j > i$

Buckets $j > i$ when changing d^*

Example: $d^* = 10000$, $C = 9$, $K = 4$.

New $d^* = 10010$, was stored in bucket 1

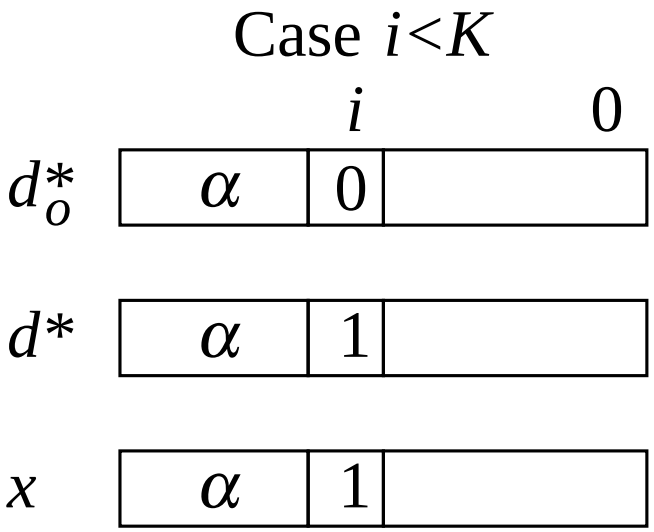
	$d^* = 10000$		$d^* = 10010$	
Bucket	$d[v]$ binary	$d[v]$	$d[v]$ binary	$d[v]$
-1	10000	16	10010	18
0	10001	17	10011	19
1	1001*	18,19	-	-
2	101**	20–23	101**	20-23
3	11***	24–25	11***	24-27
4	-	-	-	-

Bucket $B[i]$ when changing d^*

Lemma: Elements x of $B[i]$ move to buckets with smaller indices

We only cover the case $i < K$.

Let d_o^* be the old value of d^*.



Costs of deleteMin Operations

Find bucket $B[i]$: $\mathrm{O}(i)$

Shift elements from $B[i]$: $\mathrm{O}(|B[i]|)$

In total $\mathrm{O}(K + |B[i]|)$ if $i \geq 0$, $\mathrm{O}(1)$ if $i = -1$

Always shift in the direction of smaller indices

We already pay for this during insert (amortized analysis):

there are at most K shifts of an element

Running Time Dijkstra with Radix Heaps

In total we get (amortized)

- $T_{\text{insert}}(n) = \mathrm{O}(K)$
- $T_{\text{deleteMin}}(n) = \mathrm{O}(K)$
- $T_{\text{decreaseKey}}(n) = \mathrm{O}(1)$

$$T_{\text{Dijkstra}} = \mathrm{O}(m \cdot T_{\text{decreaseKey}}(n) + n \cdot (T_{\text{deleteMin}}(n) + T_{\text{insert}}(n)))$$

$$T_{\text{DijkstraRadix}} = \mathrm{O}(m + n \cdot (K + K)) = \mathrm{O}(m + n \cdot \log C)$$

Linear Running Time for Random Edge Weights

Previously: Dijkstra with binary heaps has linear running time for dense graphs ($m > n \log n \log\log n$)

Previous slide: $T_{\text{DijkstraRadix}} = \mathrm{O}(m + n \cdot \log C)$

Now: Dijkstra with radix heaps has linear running time ($\mathrm{O}(m+n)$) if Edge weights are identically uniformly distributed in $0..C$

– we only need a small adjustment in the algorithm

Modification of Algorithm for Random Edge Weights

Precomputation of lightest incoming edge weight

$c^{in}_{\min}(v) := \min\{c((u,v) : (u,v) \in E\}$

Observation: $d[v] \leq d^* + c^{in}_{\min}(v)$

$\Longrightarrow d[v] = \mu(v).$

$\Longrightarrow$ place v in set F of unscanned vertices with correct distance

Vertices in F will be scanned at the next possibility.

($\approx F$ as an extension of von $B[-1]$.)

Analysis

A vertex v is never put in a bucket i with $i < \log c^{in}_{\min}(v)$

Thus v is shifted at most $K + 1 - \log c^{in}_{\min}(v)$ times

Total cost for shifting then is at most

$$\sum_v (K - \log c^{\text{in}}_{\min}(v) + 1) = n + \sum_v (K - \log c^{\text{in}}_{\min}(v)) \leq n + \sum_e (K - \log c(e)).$$

$K - \log c(e) =$ number of zeroes at the beginning of the binary representation of $c(e)$ as a K-bit number.

$\mathbb{P}(K - \log c(e) = i) = 2^{-i} \quad \Rightarrow \quad \mathbb{E}(K - \log c(e)) = \sum_{i \geq 0} i 2^{-i} \leq 2$

Running time $= \mathrm{O}(m + n)$

All-Pairs Shortest Paths

For now there has always been a dedicated source vertex s

How can we find the shortest paths for all pairs (u, v) in G?

Assumption: negative costs allowed, but no negative cycles

Solution 1: Run Bellman-Ford n times

... Running time $O(n^2 m)$

Solution 2: Vertex potentials

... Running time $O(nm + n^2 \log n)$, significantly faster

Vertex Potentials

Each vertex gets a potential $\text{pot}(v)$

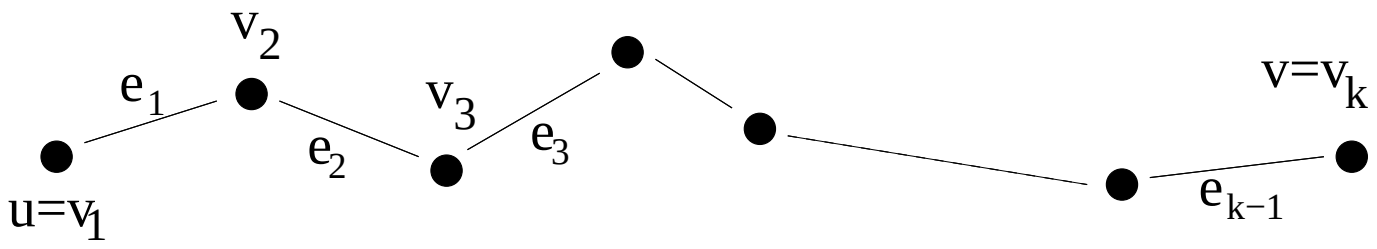
With the help of these potentials we define the reduced cost $\bar{c}(e)$ of an edge $e = (u, v)$ to be

$$\bar{c}(e) = \text{pot}(u) + c(e) - \text{pot}(v).$$

Using these costs we find the same shortest paths as before!

This holds for all possible potentials – we can define them freely

Vertex Potentials



Let p be a path from u to v with cost $c(p)$. Then

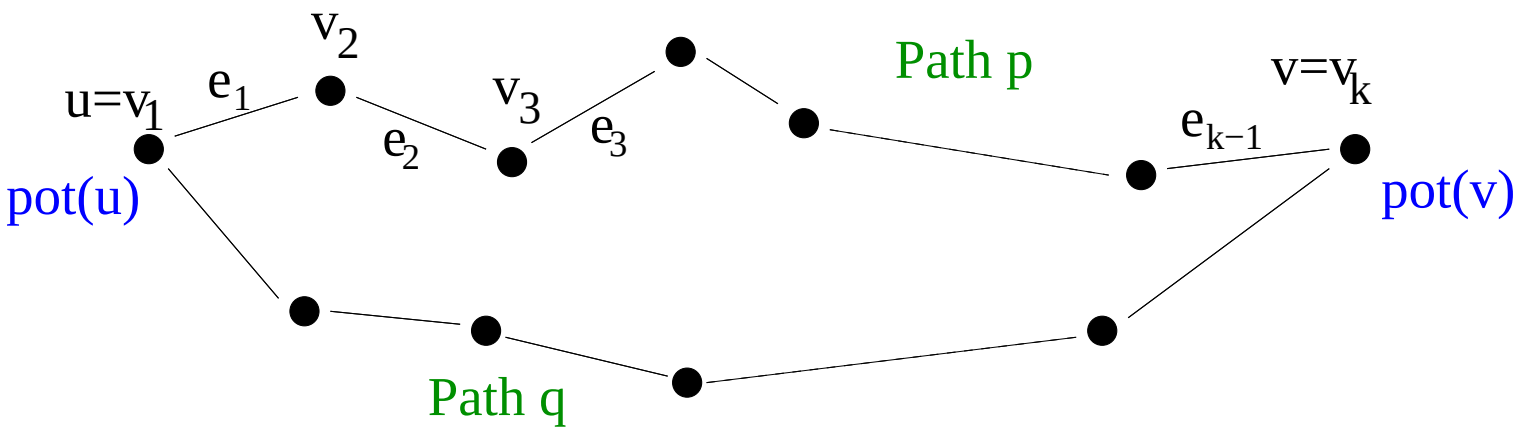
$$\begin{aligned}
\bar{c}(p) &= \sum_{i=1}^{k-1} \bar{c}(e_i) = \sum_{i=1}^{k-1} (\text{pot}(v_i) + c(e_i) - \text{pot}(v_{i+1})) \\
&= \text{pot}(v_1) + \sum_{i=1}^{k-1} c(e_i) - \text{pot}(v_k) \\
&= \text{pot}(v_1) + c(p) - \text{pot}(v_k).
\end{aligned}$$

Node Potentials

Let p be a path from u to v with cost $c(p)$. Then

$$\bar{c}(p) = \text{pot}(v_1) + c(p) - \text{pot}(v_k).$$

Let q be another u-v-path, then $c(p) \leq c(q) \Leftrightarrow \bar{c}(p) \leq \bar{c}(q)$.



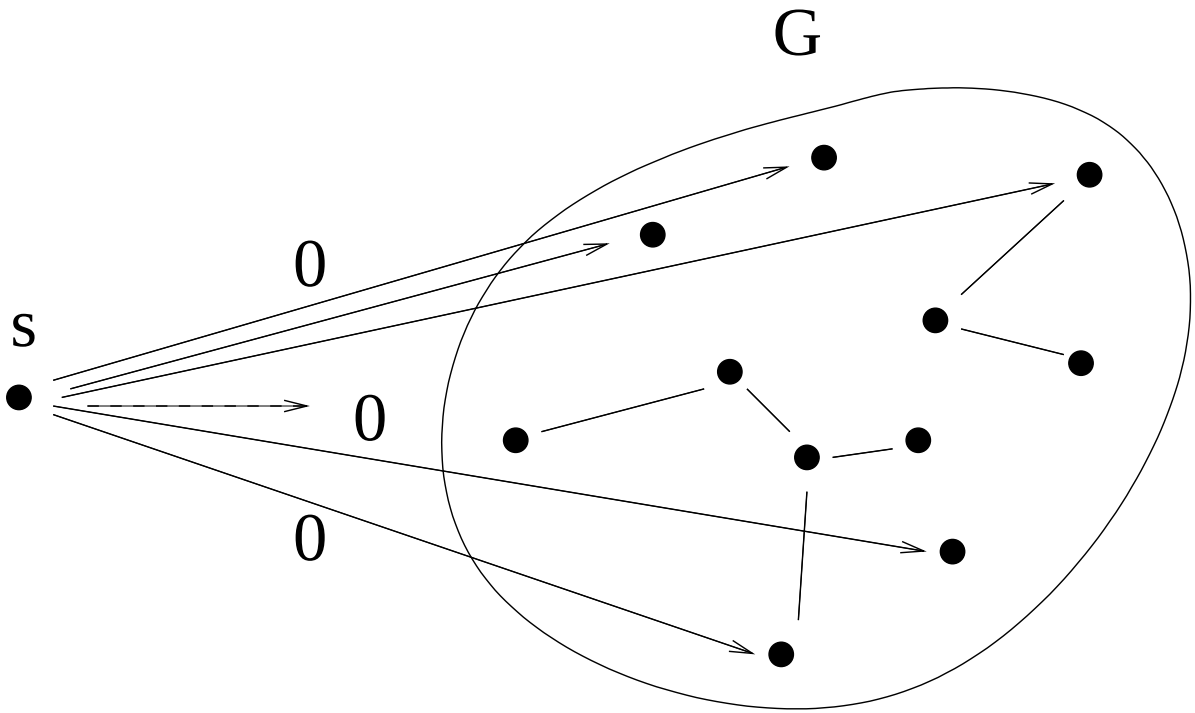
Definition: $\mu(u,v) =$ shortest distance from u to v

Auxiliary Vertices

We add an auxiliary vertex s to G

For all $v \in V$ we add an edge (s, v) with cost 0

Calculate shortest paths from s using Bellman-Ford



Definition of Potentials

Define $\text{pot}(v) := \mu(v)$ for all $v \in V$

Now the reduced costs are all non negative: we can use Dijkstra!

(Possibly remove s...)

- No negative cycles, thus $\text{pot}(v)$ well-defined
- For arbitray edge $e = (u, v)$ it holds that

$$\mu(u) + c(e) \geq \mu(v)$$

and therefore

$$\bar{c}(e) = \underbrace{\mu(u) + c(e)}_{\geq \mu(v)} - \mu(v) \geq 0$$

Algorithm

All-Pairs Shortest Paths in the Absence of Negative Cycles

new vertex s

foreach $v \in V$ **do** add edge (s,v) (cost 0) // $O(n)$

pot:= μ:= BellmanFordSSSP(s,c) // $O(nm)$

foreach vertex $x \in V$ **do** // $O(n(m+n\log n))$

$\bar{\mu}(x,\cdot)$:= DijkstraSSSP$(x,\bar{c})$

//return to original cost function

foreach $e=(v,w) \in V \times V$ **do** // $O(n^2)$

$\mu(v,w) := \bar{\mu}(v,w) + \text{pot}(w) - \text{pot}(v)$

Running Time

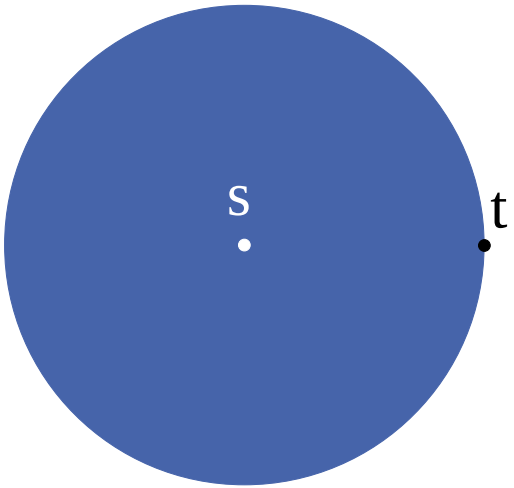
- Add s: $\mathrm{O}(n)$
- Postprocessing: $\mathrm{O}(n^2)$ (return to original cost function)
- Running Dijkstra n times dominates

Running time $\mathrm{O}(n(m + n\log n)) = \mathrm{O}(nm + n^2 \log n)$

Possible parallelization: par. Bellman–Ford + independent SSSP searches. Memory consumption?

Distance to a Target Vertex t

What to do if we are only interested in the distance from s to a specific target vertex t?



Trick 0:

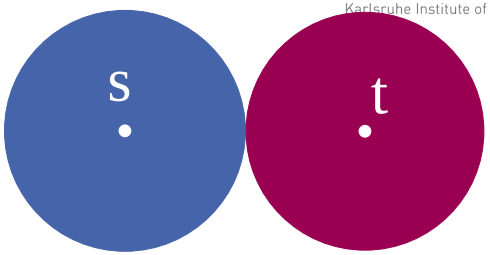
Dijkstra stops once t is removed from Q

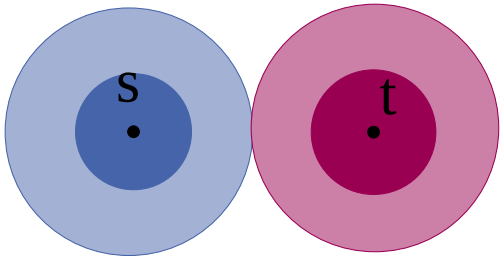
"On average" saves half of the scans

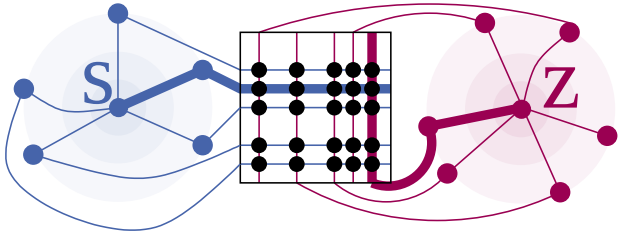
Question: How much does it save for navigation?

Ideas for Route Planning

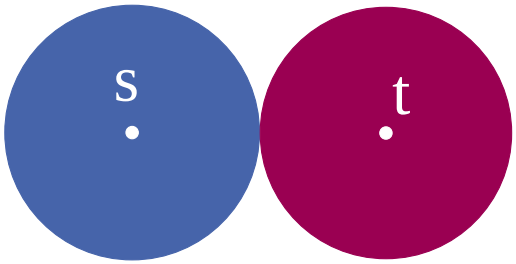
- Forward + backward search
- **Target-oriented search**
- Exploit hierarchies 
- Tabulate segments 







Bidirectional Search



Idea: Alternate searching for s and t

Forward search on original graph $G = (V, E)$

Backward search on backward graph $G^r = (V, E^r)$

(Search direction switches at every step)

Preliminary shortest distance is stored at every step:

$d[s,t] = \min(d[s,t],\ d_{\text{forward}}[u] + d_{\text{backward}}[u])$

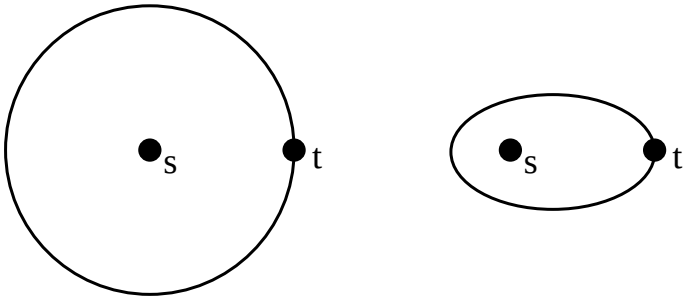
Stopping criterion:

Search scans vertex that has already been scanned in other direction.

$d[s,t] \Rightarrow\ \mu(s,t)$

A^* Search

Idea: Search "in the direction of t"

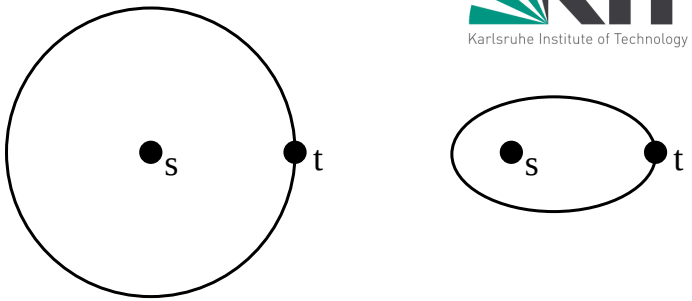


Assumption: We know a function $f(v)$ which estimates $\mu(v,t)$ $\forall v$

Define pot$(v) = f(v)$ and $\bar{c}(u,v) = c(u,v) + f(v) - f(u)$

[Or: in Dijkstra's algorithm, do not remove v that has minimum $d[v]$ from Q, but v that has minimum $d[v] + f[v]$]

A^* Search



Idea: Search "in the direction of t"

Assumption: We know a function $f(v)$ which estimates $\mu(v,t)$ $\forall v$

Define pot$(v) = f(v)$ and $\bar{c}(u,v) = c(u,v) + f(v) - f(u)$

Example: $f(v) = \mu(v,t)$.

Then: $\bar{c}(u,v) = c(u,v) + \mu(v,t) - \mu(u,t) = 0$ if (u,v) is part of the shortest path from s to t.

$$s \to \cdots \to \underbrace{u \xrightarrow{c(u,v)} \overbrace{v \to \cdots \to t}^{\mu(v,t)}}_{\mu(u,t)}$$

Therefore, Dijkstra only scans vertices on this path!

Required Properties of $f(v)$

- Consistency (reduced costs not negative):
  $c(e) + f(v) \geq f(u) \; \forall e = (u,v)$
- $f(v) \leq \mu(v,t) \; \forall v \in V$
- $f(t) = 0$ then we can stop once t is removed from Q

Let p be any path from s to t.

Are all edges along p relaxed? $\Rightarrow d[t] \leq c(p)$.

Otherwise: $\exists v \in p \cap Q$, and $d[t] + f(t) \leq d[v] + f(v)$ because t was already removed. Thus

$$d[t] = d[t] + f(t) \leq d[v] + f(v) \leq d[v] + \mu(v,t) \leq c(p)$$

How do we find $f(v)$?

We need heuristics for $f(v)$.

Route in road network: $f(v) =$ Euclidean distance $||v-t||_2$

results in noticable but not outstanding speedup

Travel time: $\dfrac{||v-t||_2}{\text{Maximum speed}}$ practically useless

Even better but requires precomputation: Landmarks

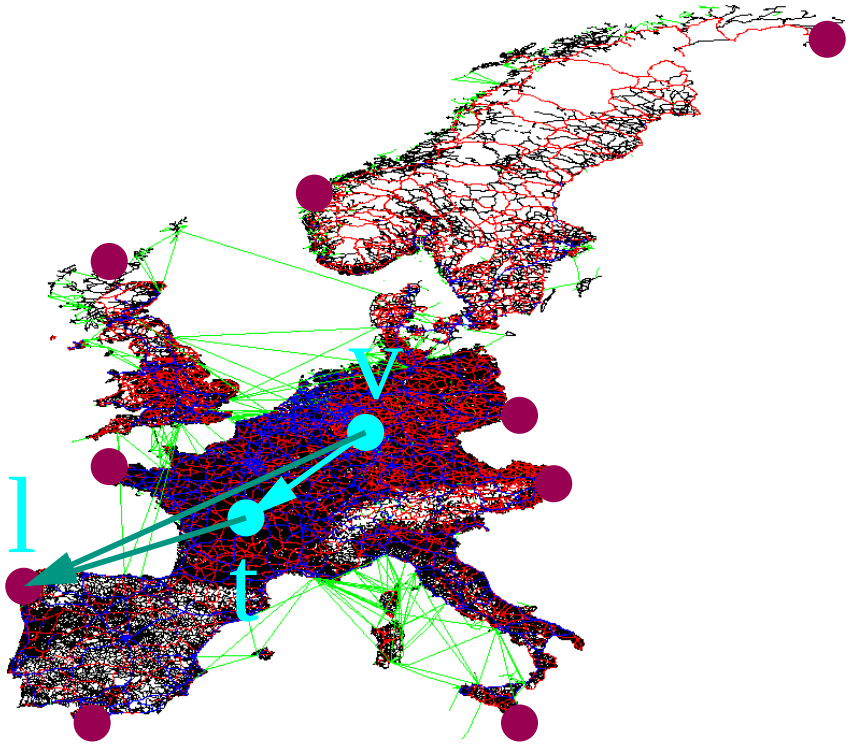
Landmarks [Goldberg Harrelson 2003]

Precomputation: Choose landmarkset L. $\forall \ell \in L, v \in V$ compute/store $\mu(v,\ell)$.

Query: Search for landmark $\ell \in L$ "behind" the target.

Use lower bound $f_\ell(v) = \mu(v,\ell) - \mu(t,\ell)$

- $+$ Conceptually simple
- $+$ Significant speedup
  ($\approx$ factor of 20 on average)
- $+$ Combinable with other techniques
- $-$ Landmark selection complicated
- $-$ High memory consumption



Summary Shortest Paths

- Non-trivial examples for average-case analysis. Similar to MST
- Monotone integer priority queues as an example for data structures that are tailored for an algorithm
- Vertex potentials generally useful for graph algorithms
- State-of-the-art research meets classic algorithmics