Updated 20 October, 2013

Seeking for the best priority queue: Lessons learnt

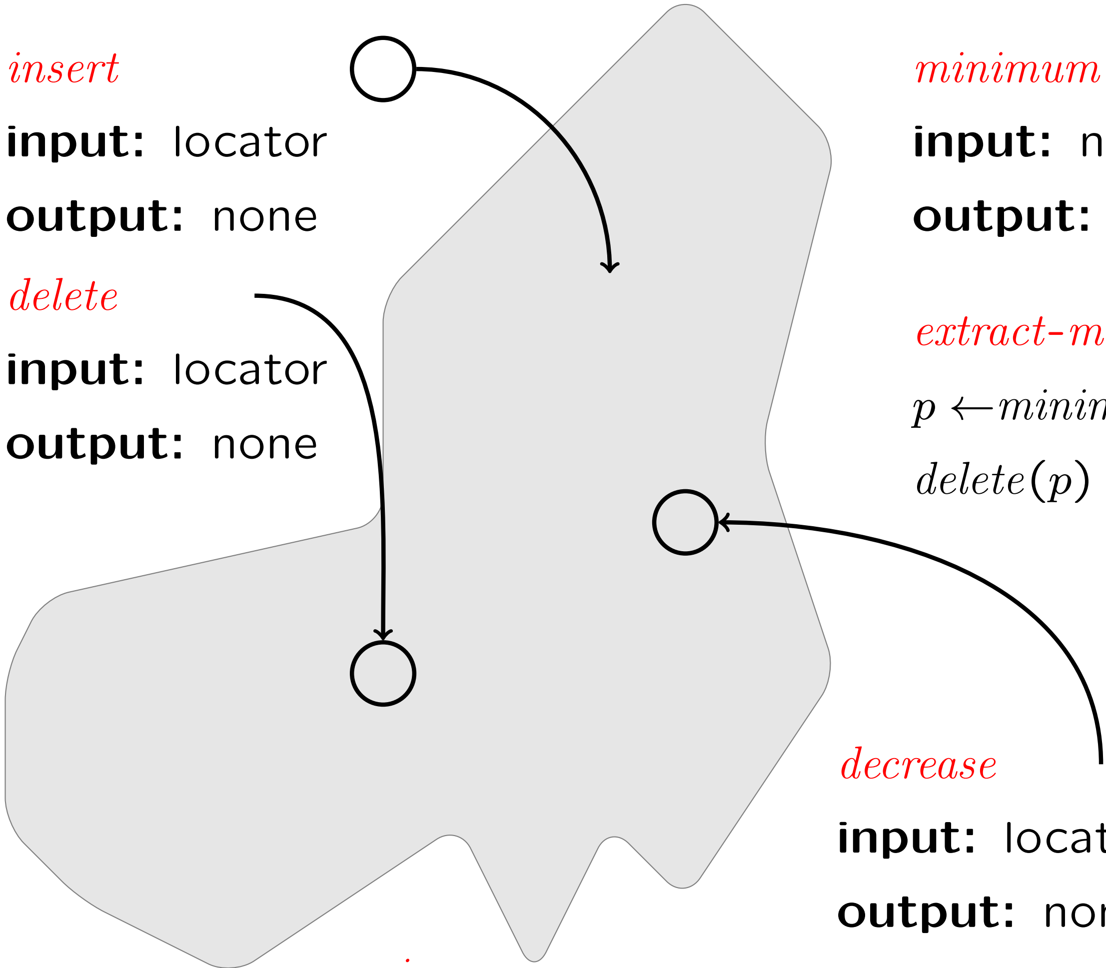
Jyrki Katajainen[1,2]

Stefan Edelkamp[3], **Amr Elmasry**[4], **Claus Jensen**[5]

[1] University of Copenhagen
[2] Jyrki Katajainen and Company
[3] University of Bremen
[4] Alexandria University
[5] The Royal Library

These slides are available from my research information system (see `http://www.diku.dk/~jyrki/` under Presentations).

Categorization of priority queues



insert

input: locator

output: none

delete

input: locator

output: none

minimum

input: none

output: locator

extract-min

$p \leftarrow minimum()$

$delete(p)$

decrease

input: locator, value

output: none

union

input: two priority queues

output: one priority queue

Elementary

minimum

insert

extract-min

Addressable

delete

decrease

Mergeable

union

Structure of this talk: Research question

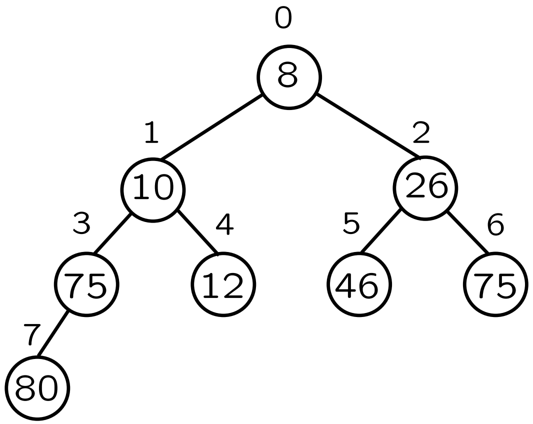
What is the "best" priority queue?

I: Elementary

II: Addressable

III: Mergeable

Binary heaps



$N = 8$

a	8	10	26	75	12	46	75	80
	0	1	2	3	4	5	6	7

left-child(i)
  return $2i + 1$

right-child(i)
  return $2i + 2$

parent(i)
  return $\lfloor (i-1)/2 \rfloor$

minimum$()$
  return $a[0]$

insert(x)
  $a[N] = x$
  $\mathit{siftup}(N)$
  $N \mathrel{+}= 1$

extract-min$()$
  $\mathit{min} = a[0]$
  $N \mathrel{-}= 1$
  $a[0] = a[N]$
  $\mathit{siftdown}(0)$
  return *min*

Elementary: array-based

Addressable: pointer-based

Mergeable: forest of heaps

Market analysis

Efficiency / Operation	binary heap [Wil64] worst case	binomial queue [Vui78] worst case	Fibonacci heap [FT87] amortized	run-relaxed heap [DGST88] worst case
minimum	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
insert	$\Theta(\lg N)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
decrease	$\Theta(\lg N)$	$\Theta(\lg N)$	$\Theta(1)$	$\Theta(1)$
delete	$\Theta(\lg N)$	$\Theta(\lg N)$	$\Theta(\lg N)$	$\Theta(\lg N)$
union	$\Theta(\lg M \times \lg N)$	$\Theta(\min\{\lg M, \lg N\})$	$\Theta(1)$	$\Theta(\min\{\lg M, \lg N\})$

Here M and N denote the number of elements in the priority queues just prior to the operation.

I

What is the best solution when handling a request sequence consisting of N *insert* and N *extract-min* operations?

Best: Comparison complexity / running time

In-place: $O(1)$ extra words

element comparisons

Data structure	*insert*	*extract-min*
binary heaps [Wil64]	$\lg N + O(1)$	$2 \lg N + O(1)$
heaps on heaps [GM86]	$\lg \lg N \pm O(1)^{a)}$	$\lg N + \log^* N \pm O(1)^{b)}$
optimal in-place heaps [EEK13]	$O(1)$	$\lg N + O(1)$
lower bounds	$\Omega(1)$	$\lg N - O(1)$

minimum: $O(1)$ worst-case time

a) Binary search on the *siftup* path (also in [Car87])

b) $\lg N - \lg \lg N$ levels down along the *siftdown* path, *siftup* in a binary-search manner, or recur further down

**Optimal in-place heaps are galactic;
only a masochist would implement them**

Ideas are interesting though . . .

Binary-heap lower bounds—assumptions

1. You should not help others

2. You should keep your house clean after each visit

What is the importance of other complexity measures?

1. # element moves
2. # instructions
3. # branch mispredictions
4. # cache misses

Theory

element moves

Source	*insert*	*extract-min*
[Wil64]	$\sim \lg N$	$\sim \lg N$
[LL96]	$\sim \frac{1}{2} \lg N$	$\sim \frac{1}{2} \lg N$

pure-C instructions

Source	*insert*	*extract-min*
[Wil64]	$\sim 9 \lg N$	$\sim 12 \lg N$
[BKS00]	$\sim 5 \lg N$	$\sim 9 \lg N$

branch mispredictions

Source	*insert*	*extract-min*
[Wil64]	$O(1)$	$\sim \frac{1}{2} \lg N$
[EK12]	$O(1)$	$O(1)$

```
< if (less(a[j], a[j + 1])) {
<    j += 1;
< }
---
> j += less(a[j], a[j + 1]);
```

cache misses

Source	*insert*	*extract-min*
[Wil64]	$\sim \lg \left(\frac{N}{M}\right)$	$\sim \lg \left(\frac{N}{M}\right)$
[WT89]	$\sim \frac{1}{B} \lg \left(\frac{N}{M}\right)^{a)}$	$\sim \frac{2}{B} \lg \left(\frac{N}{M}\right)^{a)}$

a) amortized

M: # elements in the cache

B: # elements in a cache line

Experimental environment for sanity checks

Processor

Intel® Core™ i5-2520M CPU @ 2.50GHz $\times$ 4

Memory system

8-way-associative L1 cache: 32 KB
12-way-associative L3 cache: 3 MB
cache lines: 64 B
main memory: 3.8 GB

Operating system

Ubuntu 13.04 (Linux kernel 3.5.0-37-generic)

Compiler

`g++` compiler (`gcc` version 4.7.3) with optimization `-O3`

Profiler

`valgrind` simulators (version 3.8.1)

Practice: Key performance indicators

element comparisons

Source	2^{10}	2^{15}	2^{20}	2^{25}
[Wil64]	1.73	1.82	1.87	1.89
[Car87]	1.49	1.37	1.37	1.3

element moves

Source	2^{10}	2^{15}	2^{20}	2^{25}
[Wil64]	1.96	1.64	1.48	1.39
[LL96]	1.41	1.11	0.95	0.86

instructions

Source	2^{10}	2^{15}	2^{20}	2^{25}
[Wil64]	15.1	14.9	14.8	14.7
[BKS00]	15.5	14.8	14.5	14.3

branch mispredictions

Source	2^{10}	2^{15}	2^{20}	2^{25}
[Wil64]	0.59	0.56	0.54	0.53
[EK12]	0.20	0.14	0.10	0.08

L1 cache misses

Source	2^{10}	2^{15}	2^{20}	2^{25}
[Wil64]	1.00	13.1	19.1	19.7
[WT89]	1.06	6.83	4.26	3.63

Request: $N \times \mathit{insert} + N \times \mathit{extract\text{-}min}$

Repetitions: $r = 2^{26}/N$

Input: Random `int`'s

Reported: Grand total divided by $r \times N \lg N$ or $r \times \frac{N}{B} \lg \left(\max\left\{2, \frac{N}{M}\right\}\right)$

Practice: CPU times

Source \ N	2^{10}	2^{15}	2^{20}	2^{25}
`std` library [g++]	6.41	6.09	6.96	12.6
original [Wil64]	5.59	5.45	6.47	12.2
comparison opt. [Car87]	9.69	8.34	9.02	14.5
instruction opt. [BKS00]	5.44	5.41	6.31	11.8
misprediction opt. [EK12]	3.92	4.16	7.2	25.8
bottom-up [Weg93]	6.77	6.63	7.34	13.1
4-ary [LL96]	5.47	5.49	6.3	10.4
external [WT89]	5.23	5.27	5.69	**6.04**

Request: $N \times \mathit{insert} + N \times \mathit{extract\text{-}min}$
Repetitions: $r = 2^{26}/N$; each for a different array
Input: Random permutation of $\{0, 1, \ldots, N-1\}$; type `int`
Reported: Running time divided by $r \times N \lg N$ [ns]

II

What is the best solution when handling a request sequence consisting of N *insert*, N *extract-min*, and M *decrease* operations?

Best: Comparison complexity / running time

Theory

Rank-relaxed weak heaps are **better** than Fibonacci heaps! [EEK12]

Data structure	**# element comparisons**
Fibonacci heap	$2M + 2.89N \lg N$
Rank-relaxed weak heap	$2M + 1.5N \lg N$

But they are not **simpler**!

Data structure	**Lines of code**
Binary heap	205
Fibonacci heap	296
Rank-relaxed weak heap	883

Does a factor of two matter?

T vs. $2T$

Parameterized design

weak heap	
element type	int, double, ...
comparator type	std::less, std::greater, ...
node type	weak-heap node, combined weak-heap node & graph node, ...
modifier type	relaxed heap modifier, ...
level-registry type	leaf registry, ...
mark-registry type	naive mark registry, eager mark registry, lazy mark registry, ...

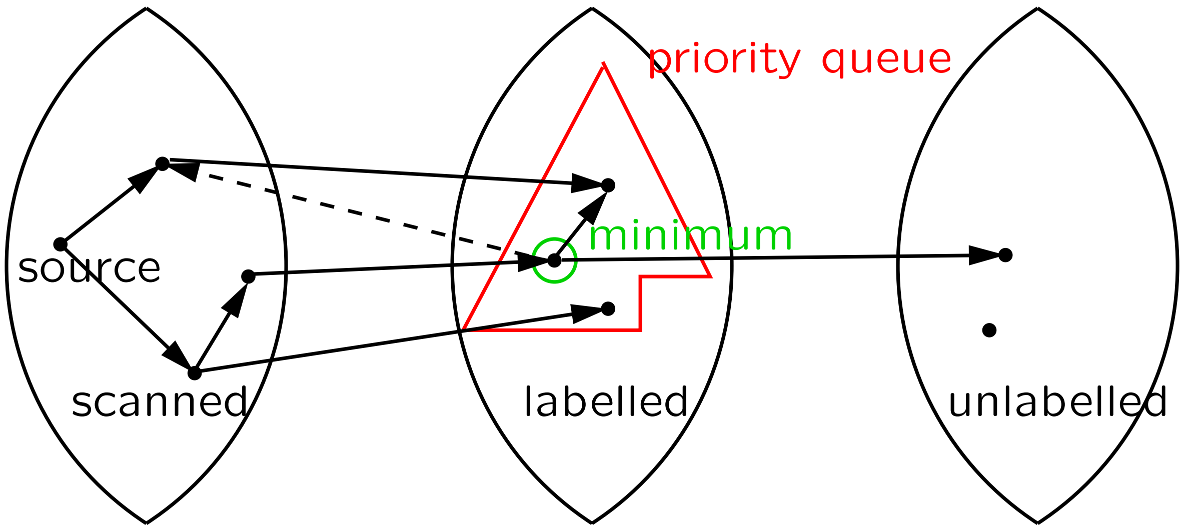
- comparators shared
- nodes shared
- transformations shared
- level registries shared
- mark registries shared

a factor of two less code

Our play with Dijkstra's algorithm

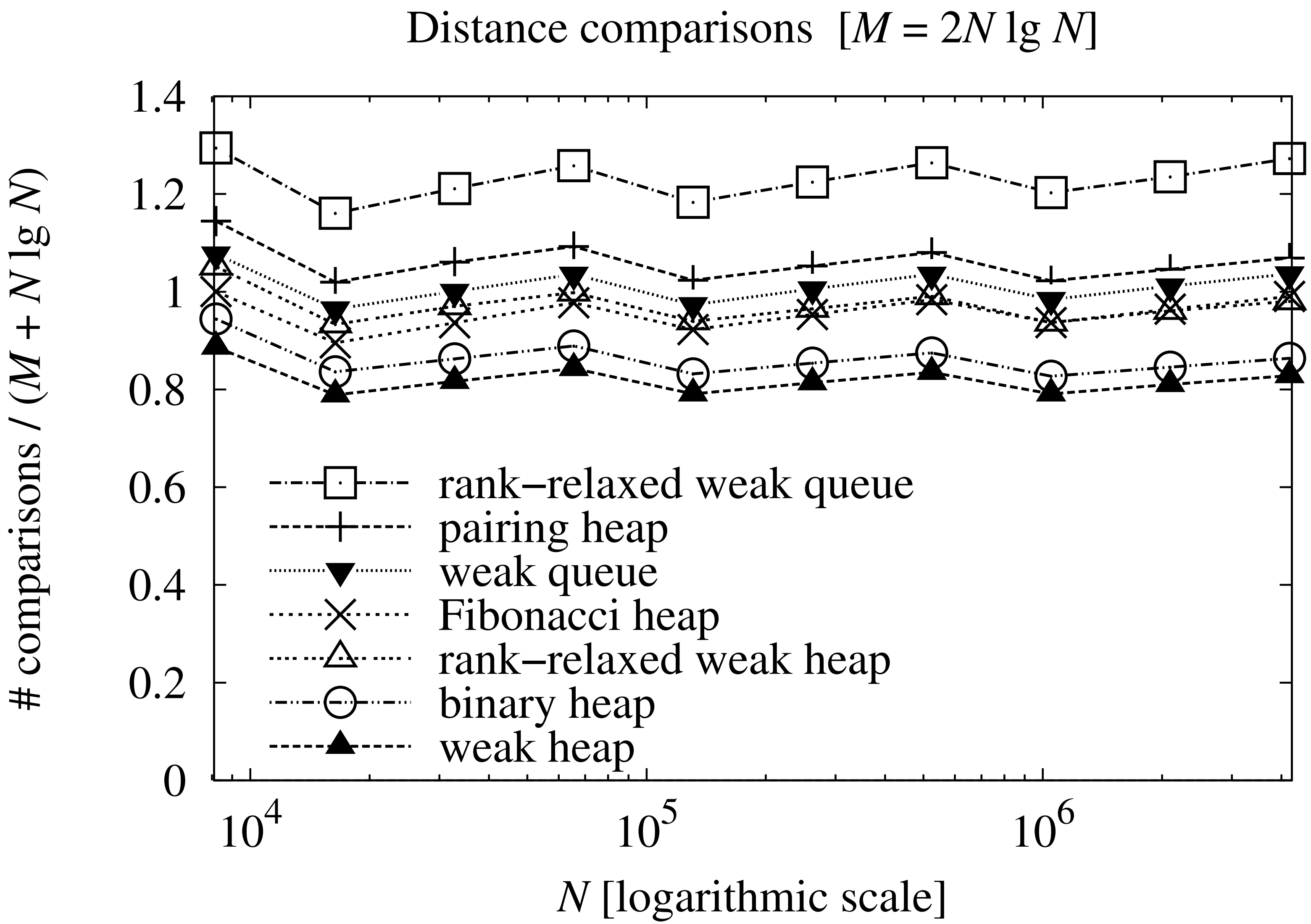
With your search engine, you will find many experimental studies on Dijkstra's algorithm. Be critical when you read the results.

- Which algorithm
- Which graph representation
- Which priority queue
- Which tuning level

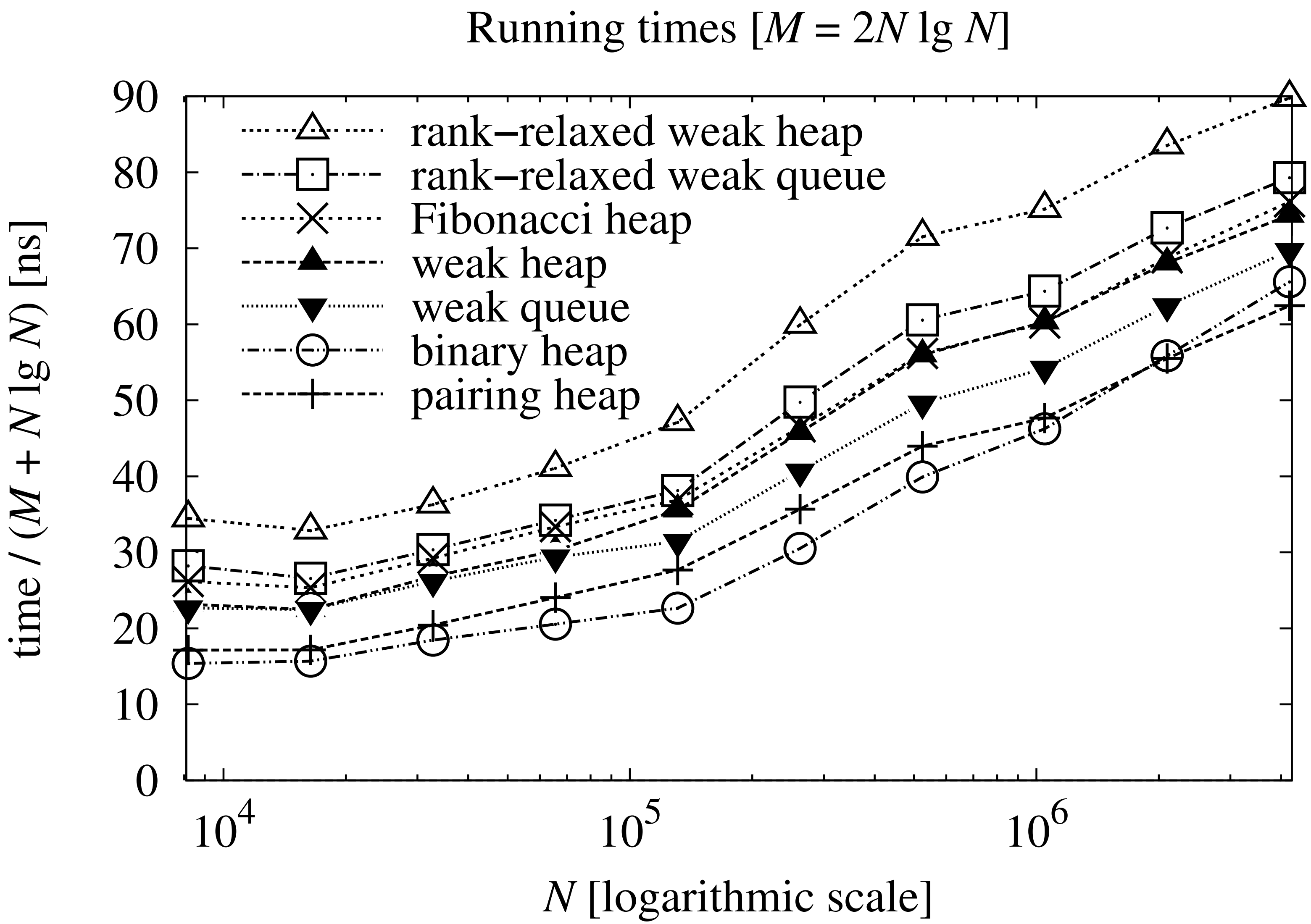


a factor of two speed-up

Policy-based benchmarking



Policy-based benchmarking



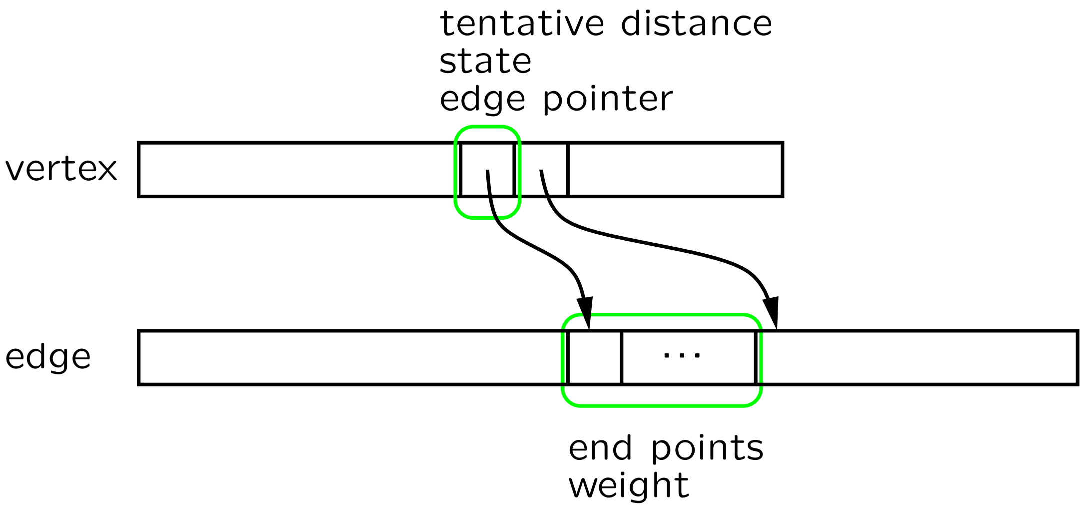
Graph representation

CPH STL

- adjacency arrays
- simple
- static
- $16M + 16N + O(1)$ bytes for a graph with M edges and N vertices

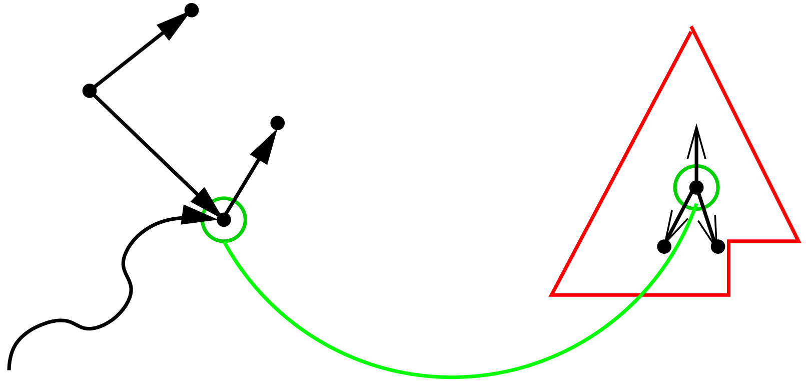
LEDA

- adjacency lists
- nice interface
- fully dynamic
- parameterized
- $52M + 60N + O(1)$ bytes [LEDA Book, § 6.14]



a factor of two speed-up

Avoiding indirection



Combine the graph vertex and the priority-queue node [Knu94] $\rightarrow$ improves cache behaviour

a factor of two speed-up

Tuning

Running time / N [μs]

Structure / Operation	CPH STL Fibonacci heap	LEDA 6.2 Fibonacci heap
insert		
N: 10 000	0.10	0.18
N: 100 000	0.09	0.15
N: 1 000 000	0.09	0.15
decrease		
N: 10 000	0.03	0.06
N: 100 000	0.05	0.22
N: 1 000 000	0.06	0.31
extract-min		
N: 10 000	0.7	1.2
N: 100 000	1.4	2.7
N: 1 000 000	2.8	4.5

 # element comparisons / N

Structure / Operation	CPH STL Fibonacci heap	LEDA 6.2 Fibonacci heap
insert		
N: 10 000	0	1
N: 100 000	0	1
N: 1 000 000	0	1
decrease		
N: 10 000	0	2
N: 100 000	0	2
N: 1 000 000	0	2
extract-min		
N: 10 000	16.2	29.9
N: 100 000	21.2	38.3
N: 1 000 000	26.2	46.5

a factor of two speed-up

On my computer (Ubuntu, `g++`, with `-O3`)

Best of the bests

Our reference sequence

Theory: rank-relaxed weak heap

Dijkstra—time: binary heap [Wil64]

Dijkstra—comps: weak heap [Dut93]

Worst case / operation

insert—**time:** Fibonacci heap [FT87]

insert—**comps:** Fibonacci heap

decrease—**time:** Fibonacci heap

decrease—**comps:** Fibonacci heap

extract-min—**time:** weak queue [Vui78]

extract-min—**comps:** weak heap

Mistakes

Analysis: Worst-case bounds

- The correlation between # element comparisons and running time can be poor
- Some people even analyse the constant factors in lower-order terms

Experiments: Randomly-generated data

- Too much focus on numeric data
- For dense graphs, few $decrease$ operations executed
- We tried to force theory and practice meet

III

What is the best mergeable heap?

Best: Comparison complexity / running time

element comparisons

Data structure	*delete / extract-min*	Other
meldable priority queues [Bro96]	$\beta \lg N^{a)}$	$O(1)$
optimal priority queues [EK12]	$\approx 70 \lg N$	$O(\kappa)^{b)}$
strict Fibonacci heaps [BLT12]	$\tau \lg N^{c)}$	$O(1)$
lower bounds	$\lg N - O(1)$	$\Omega(1)$

$^{a)}$ Brodal's constant β high

$^{b)}$ Katajainen's constant κ high

$^{c)}$ Tarjan's constant τ unknown

Optimal mergeable heaps are galactic; only a masochist would implement them

Larkin, Sen, and Tarjan have implemented strict Fibonacci heaps . . . **SLOW** . . .

Conjectures

1st conjecture: Without *decrease*; $\lg N + O(\lg \lg N)$ element comparisons per *delete* possible

2nd conjecture: The full monty; $20 \lg N$ element comparisons per *delete* possible

3rd conjecture: Our reference sequence can be handled with $2M + N \lg N + o(N \lg N)$ element comparisons in $O(M + N \lg N)$ worst-case time

Next steps

- Galactic algorithms vs. working programs
- $O(\lg N)$ and $O(1)$ vs. $O(\lg N)$ and $O(1)$
- Memory management vs. data structures
- Constant factors for cache-efficient algorithms
- Abstraction overhead in policy-based benchmarking

J. Bojesen, J. Katajainen, and M. Spork. Performance engineering case study: Heap construction. *ACM J. Exp. Algorithmics* **5**:Article 15, 2000

G. S. Brodal. Worst-case efficient priority queues. *SODA 1996*, pp. 52–58. ACM/SIAM, 1996

G. S. Brodal, G. Lagogiannis, and R. E. Tarjan. Strict Fibonacci heaps. *STOC 2012*, pp. 1177–1184. ACM, 2012

A. Bruun, S. Edelkamp, J. Katajainen, and J. Rasmussen. Policy-based benchmarking of weak heaps and their relatives. *SEA 2010*, LNCS **6049**, pp. 459–435. Springer, 2010

S. Carlsson. A variant of Heapsort with almost optimal number of comparisons. *Inform. Process. Lett. 24*(4):247–250, 1987

J. Chen, S. Edelkamp, A. Elmasry, and J. Katajainen. In-place heap construction with optimized comparisons, moves, and cache misses. *MFCS 2012*, LNCS **7464**, pp. 259–270, Springer, 2012

J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan. Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. *Commun. ACM* **31**(11):1343–1354, 1988

R. D. Dutton. Weak-heap sort. *BIT* **33**(3):372–381, 1993

S. Edelkamp, A. Elmasry, and J. Katajainen. The weak-heap data structure: Variants and applications. *J. Discrete Algorithms* **16**:187–205, 2012

S. Edelkamp, A. Elmasry, and J. Katajainen. Optimal in-place heaps. 2013

A. Elmasry, C. Jensen, and J. Katajainen. Multipartite priority queues. *ACM Trans. Algorithms* **5**(1):Article 14, 2008

A. Elmasry and J. Katajainen. Lean programs, branch mispredictions, and sorting. *FUN 2012*, LNCS **7288**, pp. 119–130, Springer, 2012

A. Elmasry and J. Katajainen. Worst-case optimal priority queues via extended regular counters. *CSR 2012*, LNCS **7353**, pp. 130–142, Springer, 2012

M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM* **34**(3):596–615, 1987

G. H. Gonnet and J. I. Munro. Heaps on heaps. *SIAM J. Comput.* **15**(4):964–971, 1986

D. E. Knuth. *The Stanford GraphBase: A Platform for Combinatorial Computing*. ACM Press, 1994

A. LaMarca and R. Ladner . The influence of caches on the performance of heaps. *ACM J. Exp. Algorithmics* **1**:Article 4, 1996

K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999

J. Vuillemin. A data structure for manipulating priority queues. *Commun. ACM* **21**(4):309–315, 1978

I. Wegener. Bottom-Up-Heapsort, a new variant of Heapsort beating, on an average, Quicksort (if n is not very small). *Theoret. Comput. Sci.* **118**(1):81–98, 1993

L. M. Wegner and J. I. Teuhola. The external heapsort. *IEEE Trans. Softw. Eng.* **15**(7):917–925, 1989

J. W. J. Williams. Algorithm 232: Heapsort. *Commun. ACM* **7**(6):347–348, 1964