Algorithm Engineering for Sorting and Searching, and All That

Stefan Edelkamp
Universität Koblenz-Landau

Symposium of Experimental Algorithms
June 16-18, 2020, Catania, Italy
Overview

• AE 4 Sorting
• AE 4 Searching
• Application Area
Overview

• AE 4 Sorting
• AE 4 Searching
• Application Area
AE 4 Sorting

Task: Sort a sequence of elements of some totally ordered universe.

5 1 8 7 6 2 3 10 9 11 12 4

~

1 2 3 4 5 6 7 8 9 10 11 12

General purpose sorting algorithm:
- for any data type
- use only pairwise comparisons
- able to handle duplicate elements

Standard algorithms: Quicksort, Mergesort, Heapsort
Quicksort is considered to be the fastest.
Aim: Improve Quicksort for random inputs.
Quicksort

1: procedure QUICKSORT(A[\ell, \ldots, r])
2: if \( r > \ell \) then
3:     pivot \leftarrow \text{choosePivot}(A[\ell, \ldots, r])
4:     cut \leftarrow \text{partition}(A[\ell, \ldots, r], \text{pivot})
5:     Quicksort(A[\ell, \ldots, \text{cut} - 1])
6:     Quicksort(A[\text{cut}, \ldots, r])
7: end if
8: end procedure

- After line 4:
  \[ \leq \text{pivot} \quad \geq \text{pivot} \]

- After line 6: both parts sorted recursively with Quicksort
Properties

- In-place algorithm: additional space $O(\log n)$ for recursion stack.
- $1.38n \log n$ comparisons on average with a fixed or random element as pivot.
- $1.18n \log n$ comparisons on average with median of three.
- $n \log n + o(n \log n)$ comparisons on avg. with median of $\sqrt{n}$.
- Insertionsort for small arrays improves running time
- quadratic worst case.

Solution to quadratic worst case: Introsort (Musser 1997) or similar approaches (switch to other algorithm in really bad cases).

- Quicksort suffers from branch mispredictions in an essential way (Kalogis, Sanders, 2006).

Solution: Tuned Quicksort (Elmasry, Katajainen, Stenmark, 2014) – much faster than other Quicksorts, but does not allow duplicates.
Pipelining – Branch Mispredictions

- Current processors have long pipelines. Pipeline stages include
  - Instruction fetch
  - Instruction decode and register fetch
  - Actual execution
  - Memory access
  - Register write back
- 14 stages for Intel Haswell, Broadwell, Skylake processors
- For every conditional statement (if, loops), the processor has to decide in advance which branch to follow.
  \[ \sim \] branch prediction
- Easiest branch prediction scheme: static predictor, e.g.
  - for if statements take the if branch,
  - for loops, assume the loop is repeated.
- If the execution follows the wrong branch, the content of the pipeline has to be discarded and the pipeline filled again.
  \[ \sim \] many clock-cycles wasted
Partitioning

1: procedure \textsc{Partition}(A[\ell, \ldots, r], \text{pivot})
2: \hspace{1em} \textbf{while} \ \ell < r \ \textbf{do}
3: \hspace{2em} \textbf{while} \ A[\ell] < \text{pivot} \ \textbf{do} \ \ell \text{++) end while}
4: \hspace{2em} \textbf{while} \ A[r] > \text{pivot} \ \textbf{do} \ r \text{-- end while}
5: \hspace{2em} \textbf{if} \ \ell < r \ \textbf{then} \ \text{swap}(A[\ell], A[r]); \ \ell++; \ r--; \ \textbf{end if}
6: \hspace{1em} \textbf{end while}
7: \hspace{1em} \textbf{return} \ \ell
8: \hspace{1em} \textbf{end procedure}

pivot = 5:

\[\begin{array}{cccccccc}
4 & 1 & 8 & 7 & 6 & 2 & 3 & 10 & 9 & 11 & 12 & 5 \\
\end{array}\]
Branch Mispredictions

1: procedure PARTITION(A[ℓ, ..., r], pivot)
2: while ℓ < r do
3:     while A[ℓ] < pivot do ℓ++ end while
4:     while A[r] > pivot do r-- end while
5:     if ℓ < r then swap(A[ℓ], A[r]); ℓ++; r-- end if
6: end while
7: return ℓ
8: end procedure

- Whenever the execution reaches the comparison $A[\ell] < \text{pivot}$, there are two possibilities
  - continue the loop
  - exit the loop

- for an optimal pivot (median), there is a 50% chance for each.

- processor (branch predictor) “guesses” which branch would be followed and decodes the respective instructions

- with a 50% chance it decodes the wrong instructions and the pipeline has to be filled anew.
Block Quicksort

Choose block size $B$ (we use $B = 128$)

1: procedure BLOCK_PARTITION($A[\ell, \ldots, r]$, pivot)
2:  integer offsets$_L[0, \ldots, B - 1]$, offsets$_R[0, \ldots, B - 1]$
3:  integer start$_L$, start$_R$, num$_L$, num$_R \leftarrow 0$
4:  while $r - \ell + 1 > 2B$ do $\quad$ $\triangleright$ start main loop
5:     ScanLeft
6:     ScanRight
7:     Rearrange
8:  end while $\quad$ $\triangleright$ end main loop
9:  scan and rearrange remaining elements
10: end procedure

Block size $B = 4$, pivot = 5:

\[
\begin{array}{cccccccccccc}
5 & 1 & 8 & 7 & 6 & 2 & 3 & 10 & 9 & 11 & 12 & 4 \\
\end{array}
\]

\[
\begin{array}{c}
0 \\
2 \\
3 \\
\end{array} \quad \begin{array}{c}
0 \\
\end{array}
\]

offsets$_L$  offsets$_R$
Block Partitioning

1: procedure SCANLEFT
2: if num_L = 0 then ▷ if left buffer is empty, refill it
3: start_L ← 0
4: for i = 0, ..., B − 1 do
5: offsets_L[num_L] ← i
6: num_L += (pivot ≥ A[ℓ + i]) ▷ comparison returns 0 or 1
7: end for
8: end if
9: end procedure

- Current offset is always written into offset array – without considering outcome of the comparison.
- Conversion from Boolean (processor flag) to integer – supported in hardware on x86 and many other processors (setcc)

〜 no unpredictable conditional statements

Other sub-procedures:
- ScanRight symmetric to ScanLeft
- Rearrange: swap elements and update pointers
Experiments
Research Questions

What is the "best" heap-construction algorithm?

What is the "best" sorting algorithm?

What is "best" priority queue?
What is the best in-place heap-construction algorithm?

Best $\sim$ In terms of Comparisons and Practical Runtime

In-Place $\sim \Theta(1)$ extra words
Some Options

Number of element comparisons

<table>
<thead>
<tr>
<th>Inventor</th>
<th>Abbreviation</th>
<th>Worst</th>
<th>Average</th>
<th>Extra Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floyd</td>
<td>Alg. F</td>
<td>$2n$</td>
<td>$\sim 1.88n$</td>
<td>$\Theta(1)$ words</td>
</tr>
<tr>
<td>Gonnet &amp; Munro</td>
<td>Alg. GM</td>
<td>$\sim 1.625n$</td>
<td>$\sim 1.625n$</td>
<td>$\Theta(n)$ words</td>
</tr>
<tr>
<td>McDiarmid &amp; Reed</td>
<td>Alg. MR</td>
<td>$2n$</td>
<td>$\sim 1.52n$</td>
<td>$\Theta(n)$ bits</td>
</tr>
<tr>
<td>Li &amp; Reed</td>
<td>Lower bound</td>
<td>$\sim 1.37n$</td>
<td>$\sim 1.37n$</td>
<td>$\Omega(1)$ words</td>
</tr>
</tbody>
</table>

Average-case results assume that the input is a random permutation of $n$ distinct elements.
Construction

Weak Heap  Lower bound $n - 1$ (comparisons)

Weak-2-Heap  $\sim 0.625n$ [IWOCA-12, MCTS-12]

$\sim 1.625n$ heap construction, $n$ bits (worst case).

Bottom trees  $\sim 1.625n$ in-place heap construction (worst case).
Weak-2-Heap

GM Build a binary heap in 2 phases: 1) Construct heap-ordered binomial tree 2) Convert this tree into binary heap

We use complete weak heaps instead – number of comparisons:

\[
C'(8) = 1, \\
C(2^k) = 2C(2^{k-1}) + k - 1.
\]

For \( n = 2^k \geq 8 \), the solution of this relation is \( C(n) = 5/8 \cdot n - \lg n - 1 \).

Alternative using less moves: navigation piles
Bottom Tree Conversion

bottom trees  All complete binary trees of size \( m = 2^{\lfloor \log_2 n \rfloor + 1} - 1 \).

1) Convert all bottom trees to bottom heaps

2) Ensure heap order at upper levels by using sift-down procedure of Floyd

3) Optimize number of element moves, by handling complete binary micro trees of size 7 differently

1) \# elements involved in all bottom heap constructions is bounded by \( n \Rightarrow \# \) element comparisons in \( 1.625n + o(n) \).

2-3) At most \( n/2^h + 1 \) nodes at height \( h \), process nodes at height \( \lfloor \log_\log n \rfloor + 1 \) upwards
## Element Comparisons

<table>
<thead>
<tr>
<th>Program $n$</th>
<th>std</th>
<th>F</th>
<th>BKS</th>
<th>in-situ GM</th>
<th>in-situ MR</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{10} - 1$</td>
<td>1.64</td>
<td>1.86</td>
<td>1.86</td>
<td>1.74</td>
<td>1.52</td>
</tr>
<tr>
<td>$2^{15} - 1$</td>
<td>1.64</td>
<td>1.88</td>
<td>1.88</td>
<td>1.65</td>
<td>1.54</td>
</tr>
<tr>
<td>$2^{20} - 1$</td>
<td>1.64</td>
<td>1.88</td>
<td>1.88</td>
<td>1.63</td>
<td>1.53</td>
</tr>
<tr>
<td>$2^{25} - 1$</td>
<td>1.65</td>
<td>1.88</td>
<td>1.88</td>
<td>1.63</td>
<td>1.53</td>
</tr>
</tbody>
</table>

**std**  Bottom-up STL heap construction (*make_heap*, Williams, Wegener)

**BKS**  Improved version of Floyd’s algorithm (Bojesen et al.)

## Execution Times

<table>
<thead>
<tr>
<th>Program $n$</th>
<th>std</th>
<th>F</th>
<th>BKS</th>
<th>in-situ GM</th>
<th>in-situ MR</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{10} - 1$</td>
<td>22.3</td>
<td>14.6</td>
<td>17.1</td>
<td>21.3</td>
<td>26.2</td>
</tr>
<tr>
<td>$2^{15} - 1$</td>
<td>22.2</td>
<td>14.6</td>
<td>17.4</td>
<td>23.0</td>
<td>24.4</td>
</tr>
<tr>
<td>$2^{20} - 1$</td>
<td>29.3</td>
<td>21.9</td>
<td>17.8</td>
<td>22.9</td>
<td>23.6</td>
</tr>
<tr>
<td>$2^{25} - 1$</td>
<td>29.8</td>
<td>21.7</td>
<td>17.5</td>
<td>22.9</td>
<td>23.6</td>
</tr>
</tbody>
</table>
What is the best constant-factor-optimal internal/adaptive sorting algorithm?

Best \sim In terms of Comparisons and Practical Runtime

Internal \sim \Theta(\lg n) extra words
Sequential Sorting

Lower bound \( \lg n! = n \lg n - n/\ln 2 + O(\lg n) \), where \( 1/\ln 2 = 1.4426 \)

- **Worst Case** \( n \lg n + 0.1n \) [Dutton 1993, BIT]
- **Best Case / Index Sorting** : \( n \lg n - 0.9n \) [STACS-00, JEA-02]
- **QuickWeakHeapsort**: \( \leq n \lg n + 0.2n \) on average [EA-02]
## Constant-Factor-Optimal Algorithms

<table>
<thead>
<tr>
<th>Method</th>
<th>Mem.</th>
<th>Other</th>
<th>Worst</th>
<th>Avg.</th>
<th>Exper.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Lower bound</strong></td>
<td>$O(1)$</td>
<td>$O(n \log n)$</td>
<td>-1.44</td>
<td>-1.44</td>
<td></td>
</tr>
<tr>
<td>BUHeapsort [Weg93]</td>
<td>$O(1)$</td>
<td>$O(n \log n)$</td>
<td>$\omega(1)$</td>
<td>$-$</td>
<td>[0.35, 0.39]</td>
</tr>
<tr>
<td>WeakHeapsort [Dut93]</td>
<td>$O(n/w)$</td>
<td>$O(n \log n)$</td>
<td>0.09</td>
<td>$-$</td>
<td>[-0.46, -0.42]</td>
</tr>
<tr>
<td>RWeakHeapsort [ES02]</td>
<td>$O(n)$</td>
<td>$O(n \log n)$</td>
<td>-0.91</td>
<td>-0.91</td>
<td>-0.91</td>
</tr>
<tr>
<td>Mergesort [Knu73]</td>
<td>$O(n)$</td>
<td>$O(n \log n)$</td>
<td>-0.91</td>
<td>-1.26</td>
<td>$-$</td>
</tr>
<tr>
<td>EWWeakHeapsort</td>
<td>$O(n)$</td>
<td>$O(n \log n)$</td>
<td>-0.91</td>
<td>-1.26</td>
<td>$-$</td>
</tr>
<tr>
<td>Insertionsort [Knu73]</td>
<td>$O(1)$</td>
<td>$O(n^2)$</td>
<td>-0.91</td>
<td>-1.38</td>
<td>-</td>
</tr>
<tr>
<td>MergeInsertion [Knu73]</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>-1.32</td>
<td>-1.3999</td>
<td>[-1.43, -1.41]</td>
</tr>
<tr>
<td>InPlaceMergesort [R92]</td>
<td>$O(1)$</td>
<td>$O(n \log n)$</td>
<td>-1.32</td>
<td>$-$</td>
<td>$-$</td>
</tr>
<tr>
<td>QuickHeapsort [DW13]</td>
<td>$O(n/w)$</td>
<td>$O(n \log n)$</td>
<td>$\omega(1)$</td>
<td>-0.03</td>
<td>$\approx 0.20$</td>
</tr>
<tr>
<td>QuickMergesort (IS)</td>
<td>$O(\log n)$</td>
<td>$O(n \log n)$</td>
<td>$\omega(1)$</td>
<td>-0.99</td>
<td>$\approx -1.24$</td>
</tr>
<tr>
<td>QuickMergesort</td>
<td>$O(1)$</td>
<td>$O(n \log n)$</td>
<td>-0.32</td>
<td>-1.38</td>
<td>$-$</td>
</tr>
<tr>
<td>QuickMergesort (MI)</td>
<td>$O(\log n)$</td>
<td>$O(n \log n)$</td>
<td>-0.32</td>
<td>-1.26</td>
<td>[-1.29, -1.27]</td>
</tr>
</tbody>
</table>

-1.3999 -> -1.4112 [E. Weiß, Wild, Algorithmica 2020]
Idea QuickXsort

As in Quicksort the array is partitioned into the elements greater and less than some pivot element.

Then one part of the array is sorted by some algorithm \( X \) and the other part is sorted recursively.

The advantage of this procedure is that, if \( X \) is an external algorithm, then in QuickXsort the part of the array which is not currently being sorted may be used as temporary space, what yields an internal variant of \( X \).

By taking with \( \Theta(\sqrt{n}) \) elements as sample for pivot selection, QuickXsort performs up to \( o(n) \) terms on average the same number of comparisons as \( X \).
Results Small Datasets

**Small-Scale Comparison Experiment**
- Lower Bound
- Insertionsort
- Merge Insertion Improved
- Merge Insertion

**Small-Scale Runtime Experiment**
- Insertionsort
- Merge Insertion Improved
- Merge Insertion
Results Large Datasets
Adaptive Sorting

- A sorting algorithm is **adaptive** with respect to a measure of disorder, if it sorts all input sequences, but performs particularly well on those that have a low amount of disorder.

- The running time of such algorithm is measured as a function of the length of the input, $n$, and the amount of disorder. Hence, the running time varies between $O(n)$ time and $O(n \lg n)$ depending on the amount of disorder.

- The algorithm should be adaptive without knowing the amount of disorder beforehand.

Let $(x_1, x_2, \ldots, x_n)$ be a sequence of $n$ elements. For simplicity, assume that all elements are distinct.

$$Inv := \left| \{(i, j) \mid 1 \leq i < j \leq n \text{ and } x_i > x_j \} \right|$$

is one measure of disorder.
Adaptive Heapsort

**input**: sequence \( \langle x_1, x_2, \ldots, x_n \rangle \) of \( n \) elements

1. Construct an empty Cartesian tree \( C \)
2. \( \text{hint} \leftarrow 0 \)
3. \( \text{for} \ i \in \{1, 2, \ldots, n\} \)
   4. \( \text{hint} \leftarrow C.\text{insert}(x_i, \text{hint}) \)
5. Construct an empty priority queue \( Q \)
6. \( Q.\text{insert}(C.\text{minimum}()) \)
7. \( \text{for} \ j \in \{1, 2, \ldots, n\} \)
   8. \( x_j \leftarrow Q.\text{extract-min}() \)
9. Let \( Y \) be the set of children \( x_j \) has in \( C \)
10. \( \text{for each} \ y \in Y \)
11. \( Q.\text{insert}(y) \)

**Idea**: Keep \( Q \) small.

[Levcopoulos & Petersson 1993]
Theoretical Race

For priority queue $Q$, the number of element comparisons performed is bounded by $\beta n \lg (\text{Inv}/n) + O(n)$.

<table>
<thead>
<tr>
<th>$Q$</th>
<th>$\beta$</th>
<th>reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary heap</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>combined extract-min insert</td>
<td>2.5</td>
<td>[Levcopoulos &amp; Petersson 1993]</td>
</tr>
<tr>
<td>binomial queue</td>
<td>2</td>
<td>[folklore]</td>
</tr>
<tr>
<td>weak heap</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>combined extract-min insert</td>
<td>1.5</td>
<td>[folklore]</td>
</tr>
<tr>
<td>multipartite priority queue</td>
<td>1</td>
<td>[Elmasry, Jensen &amp; Katajainen 2008]</td>
</tr>
</tbody>
</table>

Goal: Achieve the constant-factor optimality, i.e. $\beta = 1$, and in the meantime ensure practicality!
Array-based Solution

**Task** insert in $O(1)$ amortized time; extract-min in $O(\lg n)$ worst-case time including at most $\lg n + O(1)$ element comparisons

**Idea** Temporarily store inserted elements in a buffer and, once it is full, move elements to the main structure using bulk-insertion
Experiments

CPU time used and the number of element comparisons performed by different sorting algorithms for \( n = 10^8 \).
Overview

• AE 4 Sorting
• AE 4 Searching
• Application Area
What is best basic heap priority queue?

Best $\sim$ In terms of Comparisons and Practical Runtime

Basic $\sim$ no handles, no support of decrease and delete
Basic Heap Priority Queues

Lower bound \( \Omega(\text{lg} \text{lg} \, n) \) for insert if \( \text{lg} \, n \) for delete-min

Bulk-Insertion in Weak Heaps
\( O(1) \) amortized for insert, \( \text{lg} \, n \) amortized for delete-min

\( \sim \) Engineered Weak Heaps
\( O(1) \) for insert, \( \text{lg} \, n \) for delete-min, memory \( n/w + O(1) \)

Bulk-Insertion in Heaps
\( O(1) \) amort. for insert, \( \text{lg} \, n \) amort. for delete-min

\( \sim \) Optimal In-Place Heaps
\( O(1) \) for insert, \( \text{lg} \, n \) for delete-min, memory \( O(1) \) 

\[E., \text{Elmasry, Katajainen, TCS 2017}\]
# Complexity of Some Priority Queues

<table>
<thead>
<tr>
<th>Data structure</th>
<th>Space</th>
<th>insert</th>
<th>extract-min</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary heaps [Wil64]</td>
<td>$O(1)$</td>
<td>$\lg n + O(1)$</td>
<td>$2 \lg n + O(1)$</td>
</tr>
<tr>
<td>bin. queues [Bro78,Vui78]</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$2 \lg n + O(1)$</td>
</tr>
<tr>
<td>heaps on heaps [GM86]</td>
<td>$O(1)$</td>
<td>$\lg \lg n + O(1)$</td>
<td>$\lg n + \log^* n + O(1)$</td>
</tr>
<tr>
<td>queue of pennants [CMP88]</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$\lg n + O(1)$</td>
</tr>
<tr>
<td>multipartite PQs [EJK08]</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$\lg n + O(1)$</td>
</tr>
<tr>
<td>engin. weak heaps [EEK12]</td>
<td>$n/w + O(1)$</td>
<td>$O(1)$</td>
<td>$\lg n + O(1)$</td>
</tr>
<tr>
<td>optimal in-place heaps</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$\lg n + O(1)$</td>
</tr>
</tbody>
</table>

All data structures support *construct* in $O(n)$ and *minimum* in $O(1)$ worst-case time.
Strong Heaps

A strong heap is a heap together where nodes dominate their right siblings

Rotating Sift-Down:
Strong Heaps - Strong Sift-Down
In-Place Heaps

\[ \lceil \lg n_0 \rceil - \lceil \lg \lg n_0 \rceil \]

\[ \lceil \lg \lg n_0 \rceil \]

top heap

bottom heaps

submersion area
both of size \( O(\lg^2 n_0) \)

insertion buffer
What is the best bound when handling a request sequence consisting of $n$ insert, $n$ delete-min, and $m$ decrease operations?

Best ~ In terms of Comparisons and Practical Runtime
Advanced Priority Queues

- **insert**
  - input: element
  - output: locator

- **find-min**
  - input: none
  - output: locator

- **delete-min**
  - p ← find-min()
  - delete(p)

- **delete**
  - input: locator, element
  - output: none

- **decrease**
  - input: locator, element
  - output: none

- **meld**
  - input: two priority queues
  - output: one priority queue
## Market Analysis

<table>
<thead>
<tr>
<th>Efficiency method</th>
<th>binary heap worst case</th>
<th>binomial queue worst case</th>
<th>Fibonacci heap amortized</th>
<th>run-relaxed heap worst case</th>
</tr>
</thead>
<tbody>
<tr>
<td>find-min</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>insert</td>
<td>$\Theta(lg n)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>decrease</td>
<td>$\Theta(lg n)$</td>
<td>$\Theta(lg n)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>delete</td>
<td>$\Theta(lg n)$</td>
<td>$\Theta(lg n)$</td>
<td>$\Theta(lg n)$</td>
<td>$\Theta(lg n)$</td>
</tr>
<tr>
<td>meld</td>
<td>$\Theta(lg m \times lg n)$</td>
<td>$\Theta(lg m)$</td>
<td>$\Theta(lg n)$</td>
<td>$\Theta(lg n)$</td>
</tr>
</tbody>
</table>

Here $m$ and $n$ denote the number of elements in the priority queues just prior to the operation.
Result

Rank-relaxed weak heaps are better than Fibonacci heaps!

<table>
<thead>
<tr>
<th>Data structure</th>
<th># element comparisons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fibonacci heap</td>
<td>$2m + 2.89n \lg n$</td>
</tr>
<tr>
<td>Rank-relaxed weak heap</td>
<td>$2m + 1.5n \lg n$</td>
</tr>
</tbody>
</table>

But they are not simpler!

<table>
<thead>
<tr>
<th>Data structure</th>
<th>Lines of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary heap</td>
<td>205</td>
</tr>
<tr>
<td>Fibonacci heap</td>
<td>296</td>
</tr>
<tr>
<td>Rank-relaxed weak heap</td>
<td>883</td>
</tr>
</tbody>
</table>
Joins

Joining and splitting two perfect weak heaps of the same size:

Note that for a binary heap a join may take logarithmic time.
Heap Registry

- **Inject**: input: $H_i$, $i \leq j$; output: none
- **Eject**: input: none; output: $H_j$
- **Replace**: input: $H_\ell$ and $H_\ell'$; output: none

Size: $O(\lg n)$
Mark(ed Node) Registry

- **mark**
  - input: a node
  - output: none

- **unmark**
  - input: a node
  - output: none

- **reduce**
  - input: none
  - output: none
  - effect: Unmark at least one arbitrary marked node.
Transformations

a) cleaning transformation,

b) parent transformation,

c) sibling transformation,

d) pair transformation.

Gray nodes are marked.
Rank-relaxed Weak Heap

- $\lambda \leq \lceil \log n \rceil - 1$ nodes marked; they may violate the weak-heap ordering

**insert**: Insert a leaf, mark it, apply $\lambda$-reducing transformations as long as possible.

**decrease-key**: Decrease the value in the given node, mark it, apply $\lambda$-reducing transformations as long as possible.

**extract-min**: Find the minimum (at the root or one of the marked nodes), borrow a leaf, fix the structure of the subtree that lost its root, mark the root of the fixed subtree, apply $\lambda$-reducing transformations as long as possible.

**Improvement in extract-min**: If the mark registry is more than half full before the minimum finding, empty it.
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 - Comparisons

SSSP \(\text{normal graphs}\)

- 2–3 heap
- violation heap
- pairing heap
- Fibonacci heap
- tuned–relaxed WQ
- run–relaxed WQ
- weak queue
- bu–heap
- weak–heap
- LEDA pairing heap
- LEDA Fibonacci heap

number of element comparisons

number of vertices \(n\)
Policy-Based Benchmarking - Time

SSSP normal graphs

CPU time in s

number of vertices \( n \)

- 2–3 heap
- violation heap
- pairing heap
- Fibonacci heap
- tuned rank–relaxed WQs
- rank–relaxed WQ
- weak queue
- bu–heaps
- weak–heaps
- LEDA pairing heap
- LEDA Fibonacci heap
Graph Representation

Combine the graph vertex and the priority-queue node [Knuth 1994]
→ improves cache behaviour

a factor of two speed-up
## Running time per $n$ [μs]

<table>
<thead>
<tr>
<th>Structure</th>
<th>Operation</th>
<th>CPH STL Fibonacci heap</th>
<th>LEDA 6.2 Fibonacci heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>$n$: 10 000</td>
<td>0.10</td>
<td>0.18</td>
</tr>
<tr>
<td></td>
<td>$n$: 100 000</td>
<td>0.09</td>
<td>0.15</td>
</tr>
<tr>
<td></td>
<td>$n$: 1 000 000</td>
<td>0.09</td>
<td>0.15</td>
</tr>
<tr>
<td>decrease-key</td>
<td>$n$: 10 000</td>
<td>0.03</td>
<td>0.06</td>
</tr>
<tr>
<td></td>
<td>$n$: 100 000</td>
<td>0.05</td>
<td>0.22</td>
</tr>
<tr>
<td></td>
<td>$n$: 1 000 000</td>
<td>0.06</td>
<td>0.31</td>
</tr>
<tr>
<td>extract-min</td>
<td>$n$: 10 000</td>
<td>0.7</td>
<td>1.2</td>
</tr>
<tr>
<td></td>
<td>$n$: 100 000</td>
<td>1.4</td>
<td>2.7</td>
</tr>
<tr>
<td></td>
<td>$n$: 1 000 000</td>
<td>2.8</td>
<td>4.5</td>
</tr>
</tbody>
</table>

## Element comparisons per $n$

<table>
<thead>
<tr>
<th>Structure</th>
<th>Operation</th>
<th>CPH STL Fibonacci heap</th>
<th>LEDA 6.2 Fibonacci heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>$n$: 10 000</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>$n$: 100 000</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>$n$: 1 000 000</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>decrease-key</td>
<td>$n$: 10 000</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>$n$: 100 000</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>$n$: 1 000 000</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>extract-min</td>
<td>$n$: 10 000</td>
<td>16.2</td>
<td>29.9</td>
</tr>
<tr>
<td></td>
<td>$n$: 100 000</td>
<td>21.2</td>
<td>38.3</td>
</tr>
<tr>
<td></td>
<td>$n$: 1 000 000</td>
<td>26.2</td>
<td>46.5</td>
</tr>
</tbody>
</table>

On my computer (Ubuntu, gcc, with -O3)

---

*university logo*
## What is the Best?

<table>
<thead>
<tr>
<th>Our reference sequence</th>
<th>Worst case per operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Theory:</strong> rank-relaxed weak heap</td>
<td><em>insert</em>—<strong>time:</strong> Fibonacci heap</td>
</tr>
<tr>
<td><strong>Dijkstra</strong>—<strong>time:</strong> binary heap</td>
<td>[Fredman &amp; Tarjan 1987]</td>
</tr>
<tr>
<td>[Williams 1964]</td>
<td><strong>insert</strong>—<strong>comps:</strong> Fibonacci heap</td>
</tr>
<tr>
<td><strong>Dijkstra</strong>—<strong>comps:</strong> weak heap</td>
<td><strong>decrease-key</strong>—<strong>time:</strong> Fibonacci heap</td>
</tr>
<tr>
<td>[Dutton 1993]</td>
<td><strong>decrease-key</strong>—<strong>comps:</strong> Fibonacci heap</td>
</tr>
<tr>
<td></td>
<td><strong>extract-min</strong>—<strong>time:</strong> weak queue</td>
</tr>
<tr>
<td></td>
<td>[Vuillemin 1978]</td>
</tr>
<tr>
<td></td>
<td><strong>extract-min</strong>—<strong>comps:</strong> weak heap</td>
</tr>
</tbody>
</table>
A* = Dijkstra + Reweighting

h consistent \( 0 \leq w(u,v) + h(v) - h(u) \)

Initialisierung: \( f(s) = h(s), f(u) = \text{infinity}, \text{if } u <> s \)

Select: \( u \) mit \( f(u) = \min \{ f(v) \mid v \text{ ist in Open} \} \)

Update: \( f(v) = \min \{ f(v), f(u) + w(u,v) +h(v) - h(u) \mid v \text{ is Successor of } u \} \)
Reweighting

\[ h = 6, \quad h^* = 13 \]

\[ h = 1, \quad h^* = 5 \]

\[ h = 0, \quad h^* = 0 \]

\[ f = 11, \quad f = 12 \]

\[ f = 6 \]

\[ f = 12 \]
Set-Based Dijkstra

open_0 ← I, closed ← ⊥, g ← 0
repeat
  if (open_g ∧ G ≠ ⊥) STOP
  open_g ← open_g ∧ !closed
  für c ← 1, ..., C
  open_{g+c} ← open_{g+c} V image_c(open_g)
  closed ← closed V open_g
  g ← g + 1
Set A*
Overview

• AE 4 Sorting
• AE 4 Searching
• Application Area
Cache-Efficient SSSP

• Flood-filling algorithms as used for coloring images and shadow casting show that improved locality greatly increases the cache performance and, in turn, reduces the running time of an algorithm.

• In [E., KI-2017] I look at Dijkstra’s method to compute the shortest paths for example to generate pattern databases.

• As cache-improving contributions, I propose edge-cost factorization and flood-filling the memory layout of the graph.
Idea for PTSP
Traveling Salesman Tours

- Given a map, compute a minimum-cost round trip visiting certain cities
- Shortest paths graph reduction: precompute all-pairs-shortest-paths with
  - Dijkstra’s algorithm (be smart: employ radix heaps)
  - Model problem as an IP and call solver (CPLEX, IPSolve, ...)
  - Neighborhood search (xOPT: SA; GA; AA; PSO; LNS, ...)
  - Depth-First Branch and Bound with
    - DFBnB\(_0\): No Heuristic – incremental O(1) time
    - DFBnB\(_1\): Column/Row Minima – incremental O(1) time
    - DFBnB\(_2\): Assignment Problem – incremental O(n\(^2\)) time
  - ... New in the arena: Monte-Carlo Search
Nested Rollout Policy Adaptation

**Input**: Iteration width (exploitation), nestedness (exploration)

**Policy**: (city-to-city) mapping $N\times N \rightarrow IR$ to be learnt
Constraints

- Time Windows, Capacities, Premium Services, Pickup and Deliveries
- TSP+TW: Restricted time intervals / service times
- C+TSP: Limited vehicle load
- TSP+PD: Pickup and deliveries (PDP)
- TSP*: Premium service – same-day delivery preferred
- VRP: Vehicle routing – several vehicles
Physical TSP
Multi-Goal Motion Planning with Dynamics
Dynamics

• Express relation between input controls and resulting motions
• Necessary to plan motions that can be executed
• Impose significant challenges
  ➢ Constrain the feasible motions
  ➢ Often are nonlinear and high-dimensional
  ➢ Give rise to nonholonomic systems
  ➢ State and control spaces are continuous
  ➢ Solution trajectories are often long

• Computational complexity of motion planning with dynamics
  • Point with Newtonian dynamics NP-Hard [DXCR 1993]
  • Polygon Dubin’s car Decidable [CPK 2008]
  • General nonlinear dynamics Undecidable [Branicky 1995]
Dynamics

- Express relation between input controls and resulting motions

- Modeled via physics-based engines
  - ROS/Gazebo, ODE, Bullet, PhysX
  - General rigid-body dynamics
  - Friction and gravity

\[
\begin{align*}
\dot{s} &= f(s, u) \\
\dot{s} &= (x, y, \theta_0, v, \psi, \theta_1, \ldots, \theta_n) \\
u &= (a, \omega) \\
\dot{x} &= v \cos(\theta_0) \\
\dot{y} &= v \sin(\theta_0) \\
\dot{\theta}_0 &= v \tan(\psi) \\
v &= a \\
\dot{\psi} &= \omega \\
\dot{\theta}_i &= \frac{v}{d} \left( \prod_{j=1}^{i-1} \cos(\theta_{i-1} - \theta_j) \right) (\sin(\theta_{i-1}) - \sin(\theta))
\end{align*}
\]
Introduce Discrete Layer to Guide the Search

• Workspace decomposition provides
• discrete layer as adjacency graph $G = (R,E)$
• $R$ denotes the regions of the decomposition
• $E = \{(r_i,r_j) \mid r_i, r_j \in R \text{ are physically adjacent}\}$
• $h_{cost}(r)$ estimates the difficulty of reaching the goal region from $r$
• defined as length of shortest path in $G = (R,E)$ from $r$ to goal
• $[h_{cost}(r_1), h_{cost}(r_2),\ldots, h_{cost}(r_n)]$
• computed by running BFS/A* on $G$ backwards from goal
Sampling Based Motion Planning

- Expand a tree $T$ of collision-free and dynamically-feasible motions
- select a state $s$ from which to expand the tree
- sample control input $u$
- generate new trajectory by applying $u$ to $s$
Sampling Based Motion Planning

- Expand a tree $T$ of collision-free and dynamically-feasible motions
  - select a state $s$ from which to expand the tree
  - sample control input $u$
  - generate new trajectory by applying $u$ to $s$
Sampling Based Motion Planning

- Expand a tree $T$ of collision-free and dynamically-feasible motions
  - select a state $s$ from which to expand the tree
  - sample control input $u$
  - generate new trajectory by applying $u$ to $s$
Guided Expansion of Motion Tree

- **Sampling-based motion planning**
  - generality: dynamics as black-box function \( s' = \text{MOTION}(s, u, dt) \)
  - continuous state/control spaces: probabilistic sampling to make it feasible
  - high-dimensionality: search to find solution

- **coupled with discrete abstractions**
  - provide simplified planning layer
  - guide search in the continuous state/control spaces

- **and motion controllers**
  - open up the black-box MOTION function
  - facilitate search expansion

- **Formal guarantees**
  - probabilistic completeness

selecting an equivalence class from which to expand motion tree \( T \)

sampling-based motion planning to expand \( T \)
Architecture

- motion-planning problem with dynamics
  - discrete layer
    - workspace decomposition
    - discrete search
    - lead: regions to explore
    - progress estimation
  - continuous layer: sampling-based motion planning
    - solution trajectory

Stefan Edelkamp
Abstraction

Used to induce partition of motion tree into equivalence classes

\( v_i = v_j \) iff

\( \text{TRAJ}(T,v_i) \) provides same abstract information as \( \text{TRAJ}(T,v_j) \) iff

\( \text{region}(v_i) = \text{region}(v_j) \)

\( \rightarrow \) equivalence class corresponding to abstract state \(<r>\)

\( \Gamma_{<r>} = \{v \mid v \text{ in } T \text{ and } \text{region}(v) = r\} \)

\( \rightarrow \) partition of motion tree \( T \) into equivalence classes

\( \Gamma = \{\Gamma_{<r>} : \Gamma_{<r>} > 0\} \)
Coloring and Inspection

Stefan Edelkamp
Inspection Problem
3D Inside / Outside Inspection
Questions