

# 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

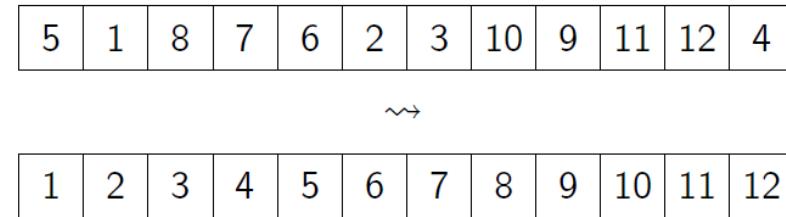
- AE 4 Sorting
- AE 4 Searching
- Application Area

# Overview

- AE 4 Sorting
- AE 4 Searching
- Application Area

# AE 4 Sorting

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



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

- After line 4:



- After line 6: both parts sorted recursively with Quicksort



# Properties

- In-place algorithm: additional space  $\mathcal{O}(\log n)$  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 (Kaligosi, 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.  
~~ 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.  
~~ many clock-cycles wasted

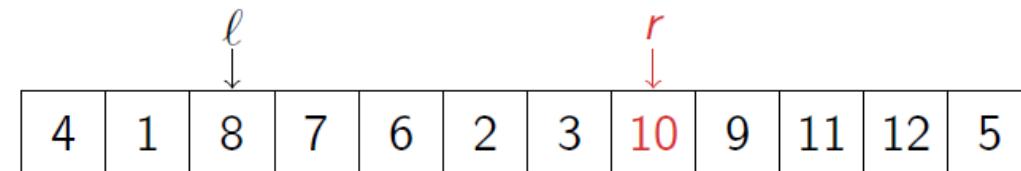
# Partitioning

```

1: procedure PARTITION( $A[\ell, \dots, r]$ , pivot)
2:   while  $\ell < r$  do
3:     while  $A[\ell] < \text{pivot}$  do  $\ell++$  end while
4:     while  $A[r] > \text{pivot}$  do  $r--$  end while
5:     if  $\ell < r$  then swap( $A[\ell], A[r]$ );  $\ell++$ ;  $r--$  end if
6:   end while
7:   return  $\ell$ 
8: end procedure

```

$\text{pivot} = 5$ :



# Branch Mispredictions

```
1: procedure PARTITION( $A[\ell, \dots, r]$ , pivot)
2:   while  $\ell < r$  do
3:     while  $A[\ell] < \text{pivot}$  do  $\ell++$  end while
4:     while  $A[r] > \text{pivot}$  do  $r--$  end while
5:     if  $\ell < r$  then swap( $A[\ell], A[r]$ );  $\ell++$ ;  $r--$  end if
6:   end while
7:   return  $\ell$ 
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

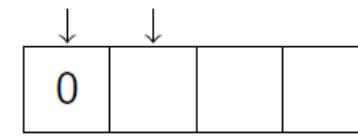
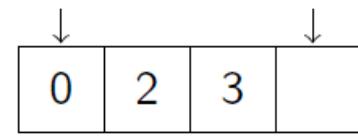
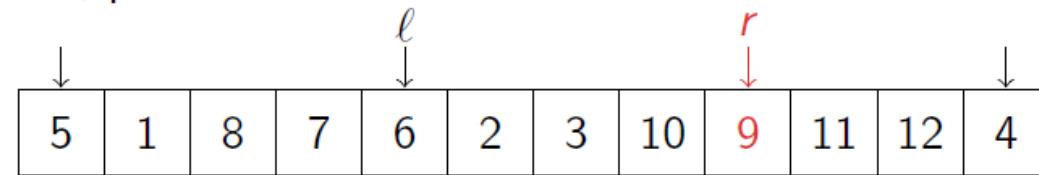
Choose block size  $B$  (we use  $B = 128$ )

```

1: procedure BLOCKPARTITION( $A[\ell, \dots, r]$ , pivot)
2:   integer offsetsL[0, ...,  $B - 1$ ], offsetsR[0, ...,  $B - 1$ ]
3:   integer startL, startR, numL, numR  $\leftarrow 0$ 
4:   while  $r - \ell + 1 > 2B$  do                                 $\triangleright$  start main loop
5:     ScanLeft
6:     ScanRight
7:     Rearrange
8:   end while                                          $\triangleright$  end main loop
9:   scan and rearrange remaining elements
10: end procedure

```

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



# Block Partitioning

```

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

```

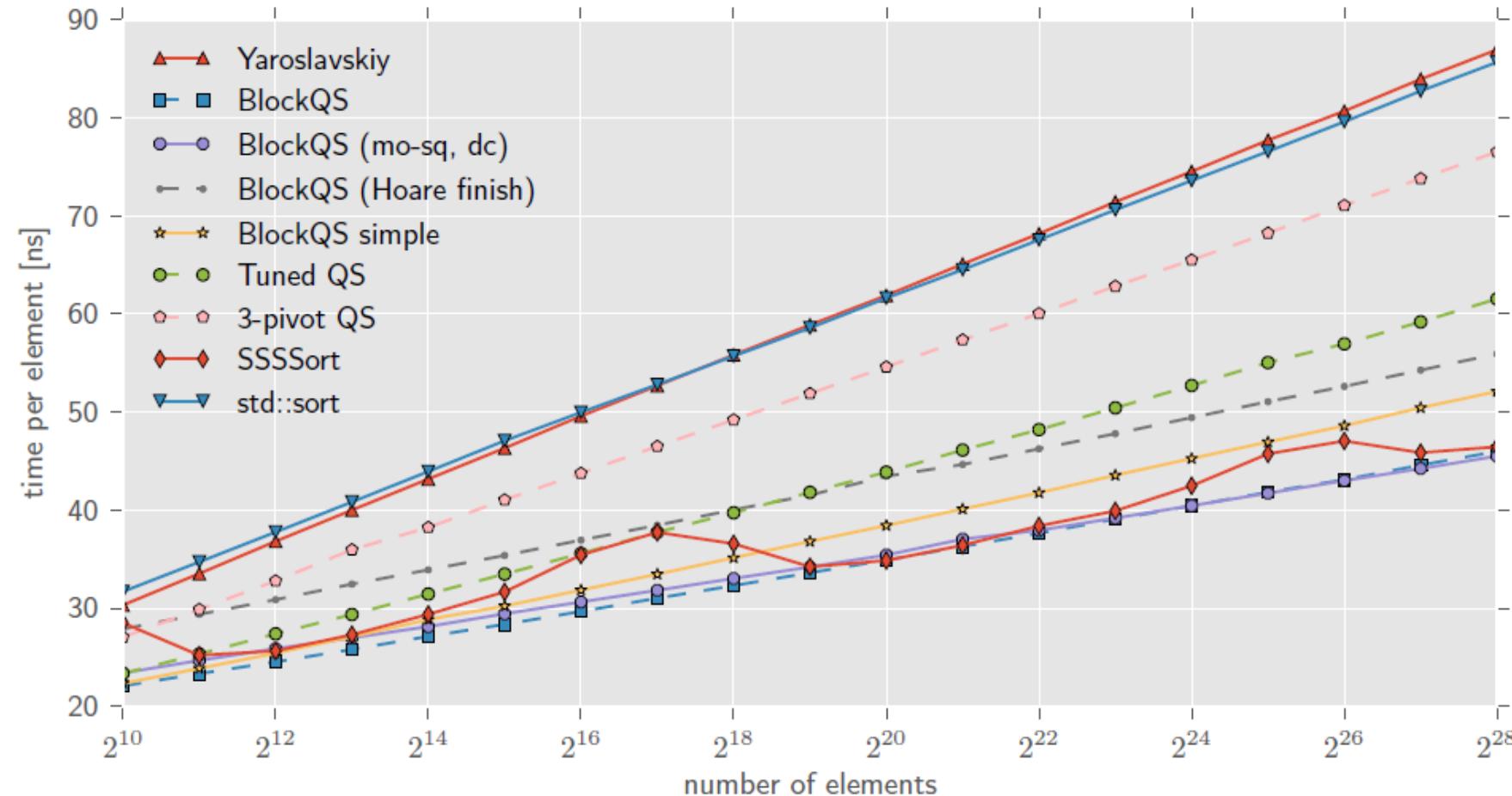
- Current offset is always written into offset array – without considering outcome of the comparison.
- Conversion from Boolean (processor flag) to integer – supported in hardware on x86 and many other processors (setcc)

⇒ 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 ~ In terms of Comparisons and Practical  
Runtime

In-Place ~  $\Theta(1)$  extra words

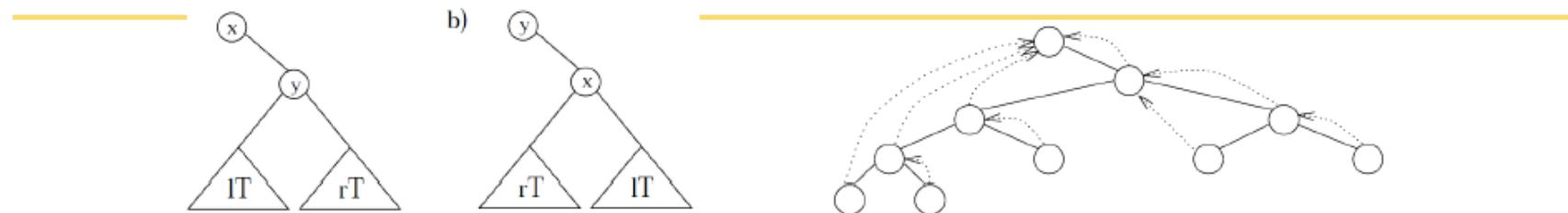
# Some Options

Number of element comparisons

| Inventor         | Abbreviation | Worst         | Average       | Extra Space       |
|------------------|--------------|---------------|---------------|-------------------|
| Floyd            | Alg. F       | $2n$          | $\sim 1.88n$  | $\Theta(1)$ words |
| Gonnet & Munro   | Alg. GM      | $\sim 1.625n$ | $\sim 1.625n$ | $\Theta(n)$ words |
| McDiarmid & Reed | Alg. MR      | $2n$          | $\sim 1.52n$  | $\Theta(n)$ bits  |
| Li & Reed        | Lower bound  | $\sim 1.37n$  | $\sim 1.37n$  | $\Omega(1)$ words |

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**  $\sim .625n$  [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

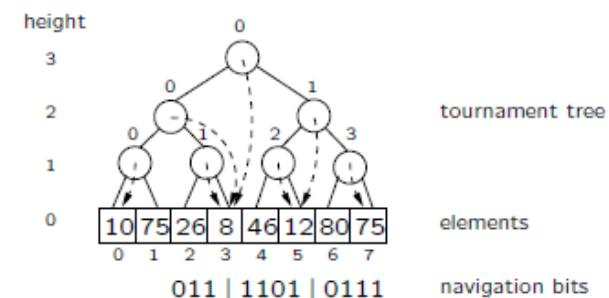
---

**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:

$$\begin{aligned} C(8) &= 1, \\ C(2^k) &= 2C(2^{k-1}) + k - 1. \end{aligned}$$

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  $m = 2^{\lfloor \lg \lg n \rfloor + 1} - 1$ .

- 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  $\lfloor \lg \lg n \rfloor + 1$  upwards

| Program<br>$n$ | std  | F    | BKS  | in-situ<br>GM | in-situ<br>MR |
|----------------|------|------|------|---------------|---------------|
| $2^{10} - 1$   | 1.64 | 1.86 | 1.86 | 1.74          | 1.52          |
| $2^{15} - 1$   | 1.64 | 1.88 | 1.88 | 1.65          | 1.54          |
| $2^{20} - 1$   | 1.64 | 1.88 | 1.88 | 1.63          | 1.53          |
| $2^{25} - 1$   | 1.65 | 1.88 | 1.88 | 1.63          | 1.53          |

**std** Bottom-up STL heap construction (`make_heap`, Williams, We-  
gener)

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

## Execution Times

| Program<br>$n$ | std  | F    | BKS  | in-situ<br>GM | in-situ<br>MR |
|----------------|------|------|------|---------------|---------------|
| $2^{10} - 1$   | 22.3 | 14.6 | 17.1 | 21.3          | 26.2          |
| $2^{15} - 1$   | 22.2 | 14.6 | 17.4 | 23.0          | 24.4          |
| $2^{20} - 1$   | 29.3 | 21.9 | 17.8 | 22.9          | 23.6          |
| $2^{25} - 1$   | 29.8 | 21.7 | 17.5 | 22.9          | 23.6          |

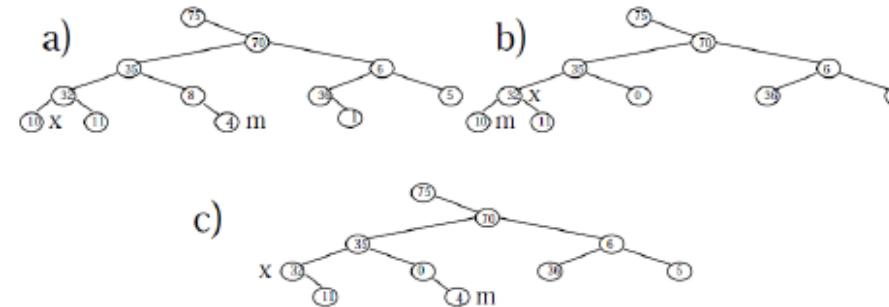
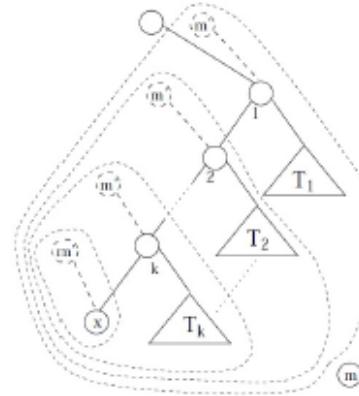
## What is the best constant-factor-optimal internal/adaptive sorting algorithm?

Best ~ In terms of Comparisons and Practical Runtime

Internal ~  $\Theta(\lg n)$  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]

|                        | Mem.        | Other         | Worst       | Avg.           | Exper.               |
|------------------------|-------------|---------------|-------------|----------------|----------------------|
| Lower bound            | $O(1)$      | $O(n \log n)$ | -1.44       | -1.44          |                      |
| BUHeapsort [Weg93]     | $O(1)$      | $O(n \log n)$ | $\omega(1)$ | —              | [0.35,0.39]          |
| WeakHeapsort [Dut93]   | $O(n/w)$    | $O(n \log n)$ | 0.09        | —              | [-0.46,-0.42]        |
| RWeakHeapsort [ES02]   | $O(n)$      | $O(n \log n)$ | -0.91       | -0.91          | -0.91                |
| Mergesort [Knu73]      | $O(n)$      | $O(n \log n)$ | -0.91       | -1.26          | —                    |
| EWeakHeapsort          | $O(n)$      | $O(n \log n)$ | -0.91       | <b>-1.26</b>   | —                    |
| Insertionsort [Knu73]  | $O(1)$      | $O(n^2)$      | -0.91       | <b>-1.38</b>   | —                    |
| MergeInsertion [Knu73] | $O(n)$      | $O(n^2)$      | -1.32       | <b>-1.3999</b> | <b>[-1.43,-1.41]</b> |
| InPlaceMergesort [R92] | $O(1)$      | $O(n \log n)$ | -1.32       | —              | —                    |
| QuickHeapsort [DW13]   | $O(1)$      | $O(n \log n)$ | $\omega(1)$ | -0.03          | $\approx 0.20$       |
|                        | $O(n/w)$    | $O(n \log n)$ | $\omega(1)$ | -0.99          | $\approx -1.24$      |
| QuickMergesort (IS)    | $O(\log n)$ | $O(n \log n)$ | -0.32       | <b>-1.38</b>   | —                    |
| QuickMergesort         | $O(1)$      | $O(n \log n)$ | -0.32       | <b>-1.26</b>   | <b>[-1.29,-1.27]</b> |
| QuickMergesort (MI)    | $O(\log n)$ | $O(n \log n)$ | -0.32       | <b>-1.3999</b> | <b>[-1.41,-1.40]</b> |

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

---

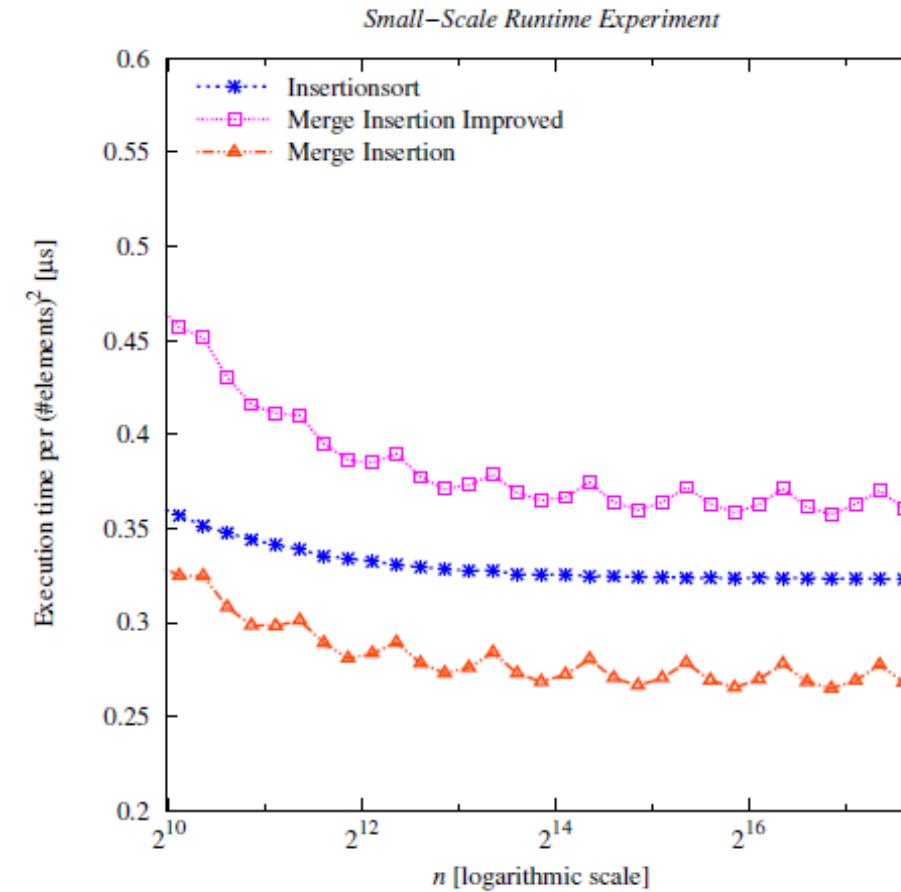
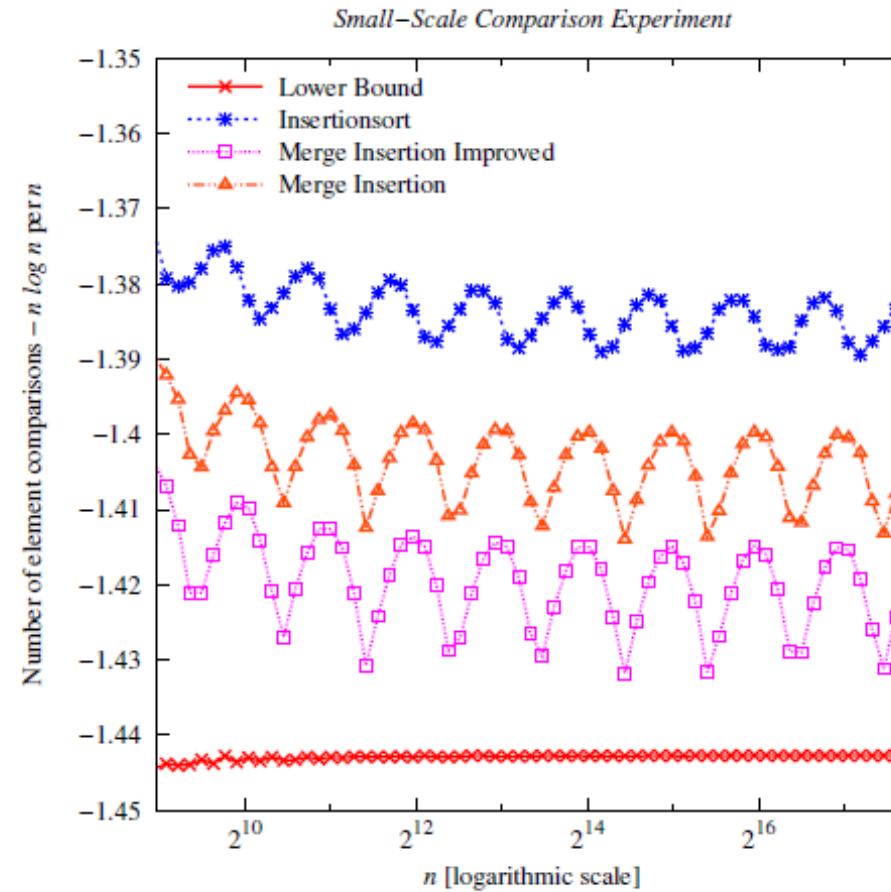
As in Quicksort the array is partitioned into the elements greater and less than some pivot element

Then one part of the array is sorted by some algorithm  $X$  and the other part is sorted recursively

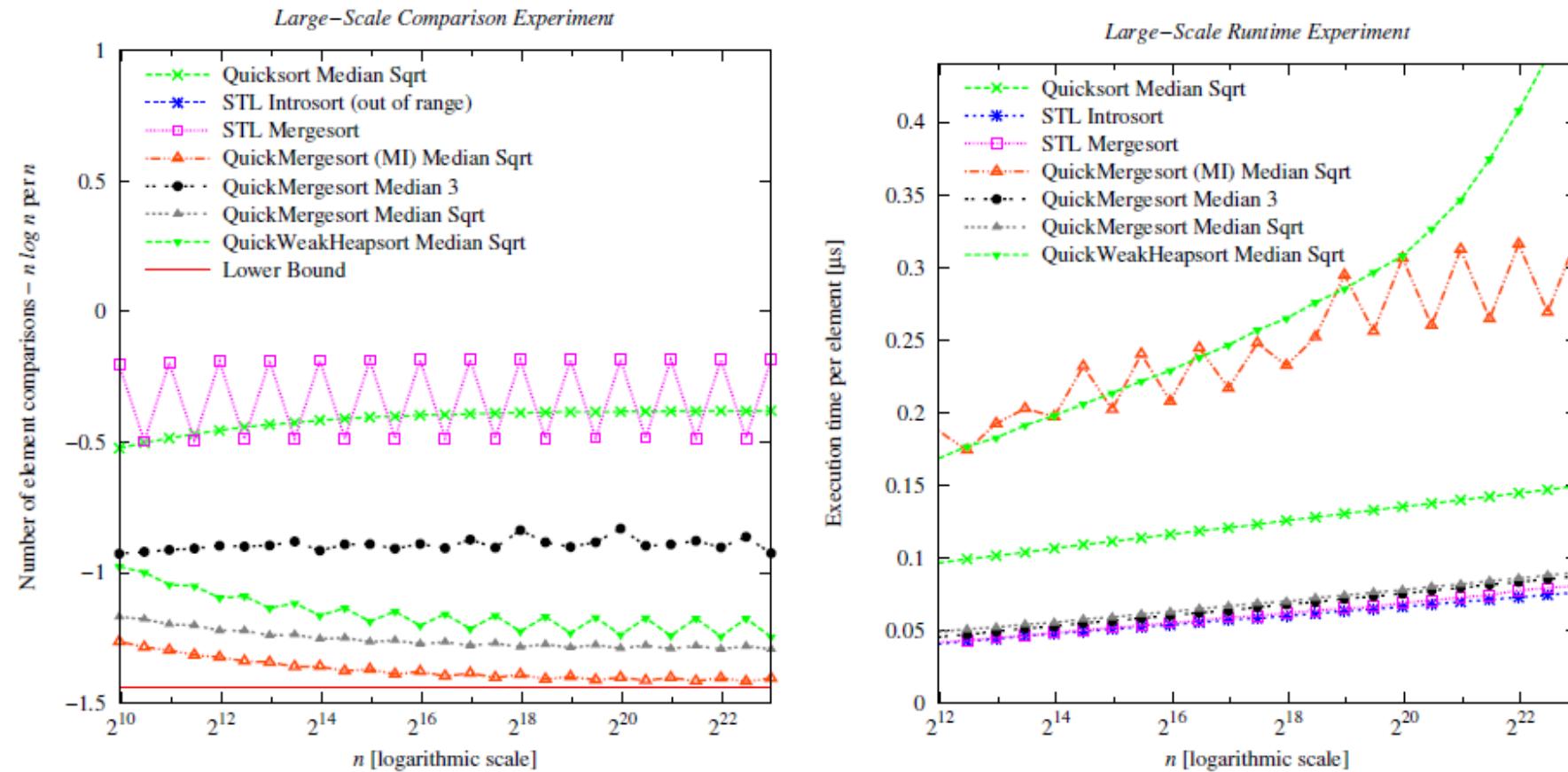
The advantage of this procedure is that, if  $X$  is an external algorithm, then in QuickXsort the part of the array which is not currently being sorted may be used as temporary space, what yields an internal variant of  $X$

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



# 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  $\langle x_1, x_2, \dots, x_n \rangle$  be a sequence of  $n$  elements. For simplicity, assume that all elements are distinct.

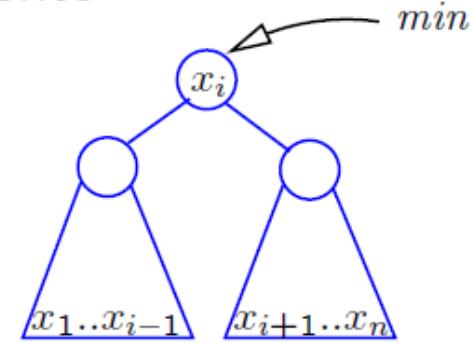
$Inv := \left| \{(i, j) \mid 1 \leq i < j \leq n \text{ and } x_i > x_j\} \right|$  is one measure of disorder

# Adaptive Heapsort

---

**input:** sequence  $\langle x_1, x_2, \dots, x_n \rangle$  of  $n$  elements

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



**Idea:** Keep  $\mathcal{Q}$  small.

[Levcopoulos & Petersson 1993]

# Theoretical Race

---

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

| $\mathcal{Q}$                                     | $\beta$  | reference                           |
|---|----------|-------------------------------------|
| binary heap<br>combined <i>extract-min insert</i> | 3<br>2.5 | [Levcopoulos & Petersson 1993]      |
| binomial queue                                    | 2        | [folklore]                          |
| weak heap<br>combined <i>extract-min insert</i>   | 2<br>1.5 | [folklore]                          |
| multipartite priority queue                       | 1        | [Elmasry, Jensen & Katajainen 2008] |

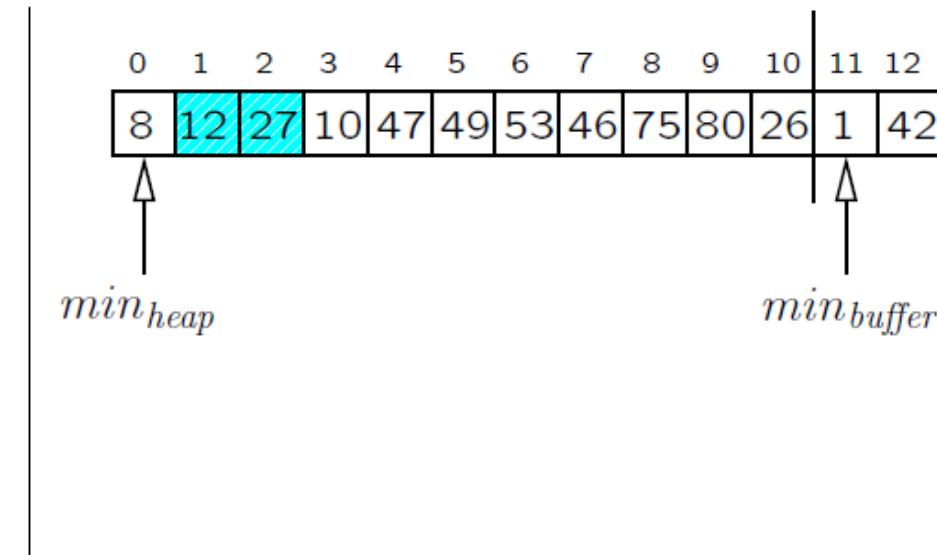
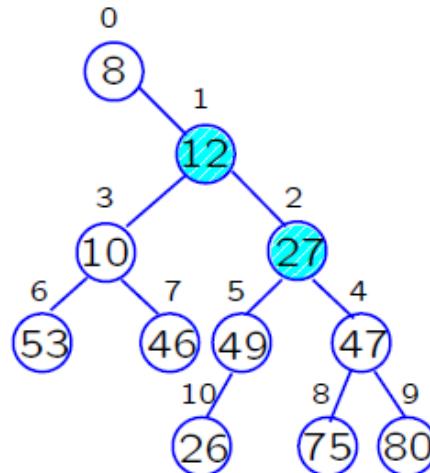
**Goal:** Achieve the constant-factor optimality, i.e.  $\beta = 1$ , and in the meantime ensure practicality!

## 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

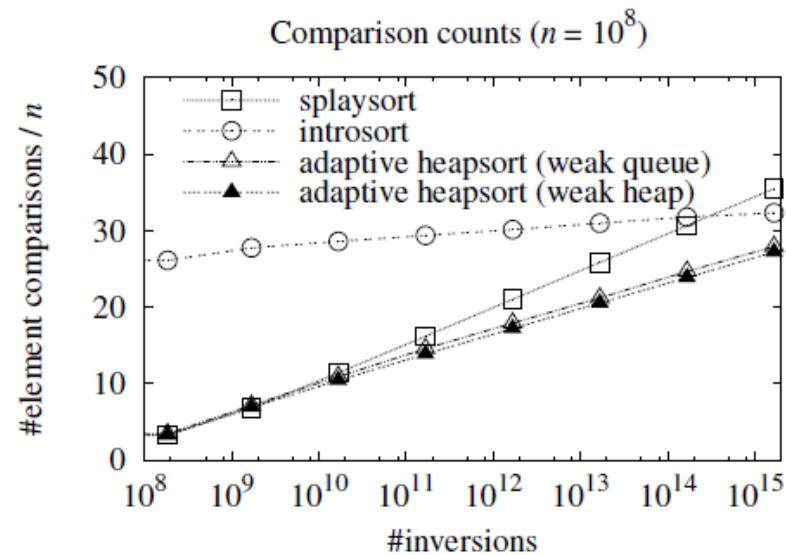
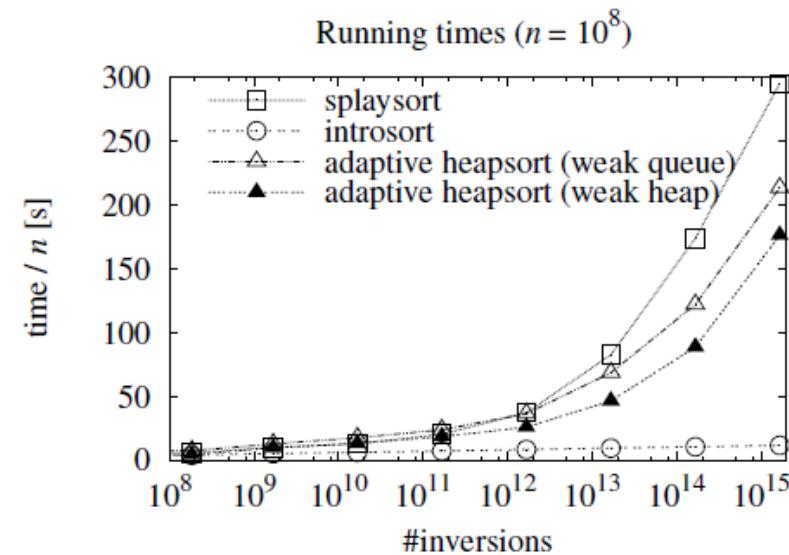
**Idea** Temporarily store inserted elements in a buffer and, once it is full, move elements to the main structure using **bulk-insertion**



## 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 ~ In terms of Comparisons and Practical Runtime

Basic ~ no handles, no support of decrease and delete

# Basic Heap Priority Queues

---

Lower bound  $\Omega(\lg \lg n)$  for insert if  $\lg n$  for delete-min

Bulk-Insertion in Weak Heaps

$O(1)$  amortized for insert,  $\lg n$  amortized for delete-min

~ Engineered Weak Heaps

$O(1)$  for insert,  $\lg n$  for delete-min, memory  $n/w + O(1)$

Bulk-Insertion in Heaps

$O(1)$  amort. for insert,  $\lg n$  amort. for delete-min

~ Optimal In-Place Heaps

$O(1)$  for insert,  $\lg n$  for delete-min, memory  $O(1)$  [E., Elmasry, Katajainen, TCS 2017]

# Complexity of Some Priority Queues

---

| Data structure            | Space        | <i>insert</i>      | <i>extract-min</i>          |
|---------------------------|--------------|--------------------|-----------------------------|
| binary heaps [Wil64]      | $O(1)$       | $\lg n + O(1)$     | $2 \lg n + O(1)$            |
| bin. queues [Bro78,Vui78] | $O(n)$       | $O(1)$             | $2 \lg n + O(1)$            |
| heaps on heaps [GM86]     | $O(1)$       | $\lg \lg n + O(1)$ | $\lg n + \log^* n + O(1)$   |
| queue of pennants [CMP88] | $O(1)$       | $O(1)$             | $3 \lg n + \log^* n + O(1)$ |
| multipartite PQs [EJK08]  | $O(n)$       | $O(1)$             | $\lg n + O(1)$              |
| engin. weak heaps [EEK12] | $n/w + O(1)$ | $O(1)$             | $\lg n + O(1)$              |
| optimal in-place heaps    | $O(1)$       | $O(1)$             | $\lg n + O(1)$              |

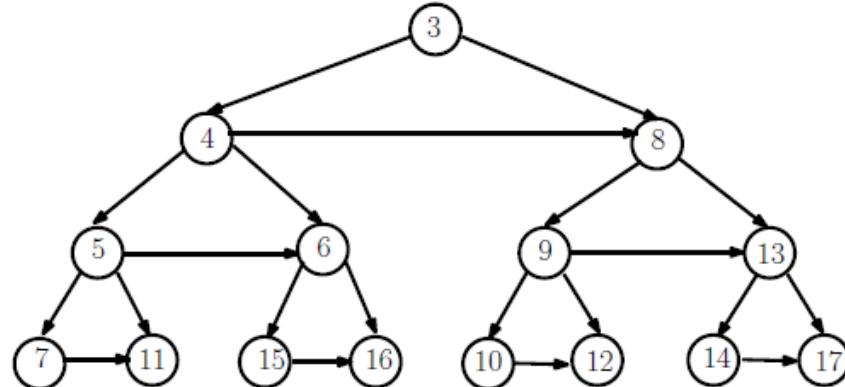
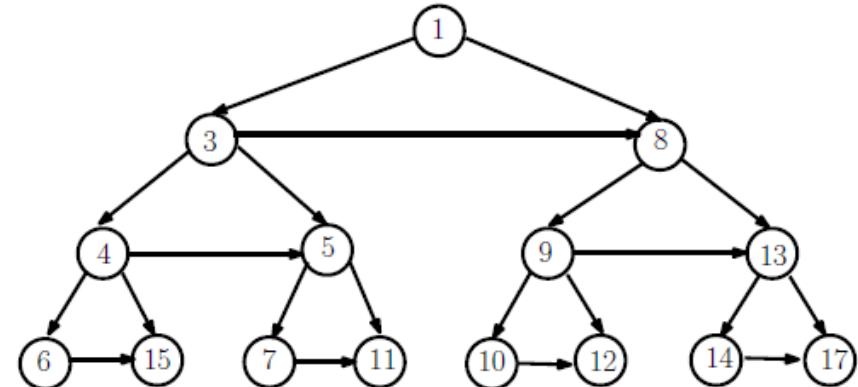
All data structures support *construct* in  $O(n)$  and *minimum* in  $O(1)$  worst-case time

# Strong Heaps

---

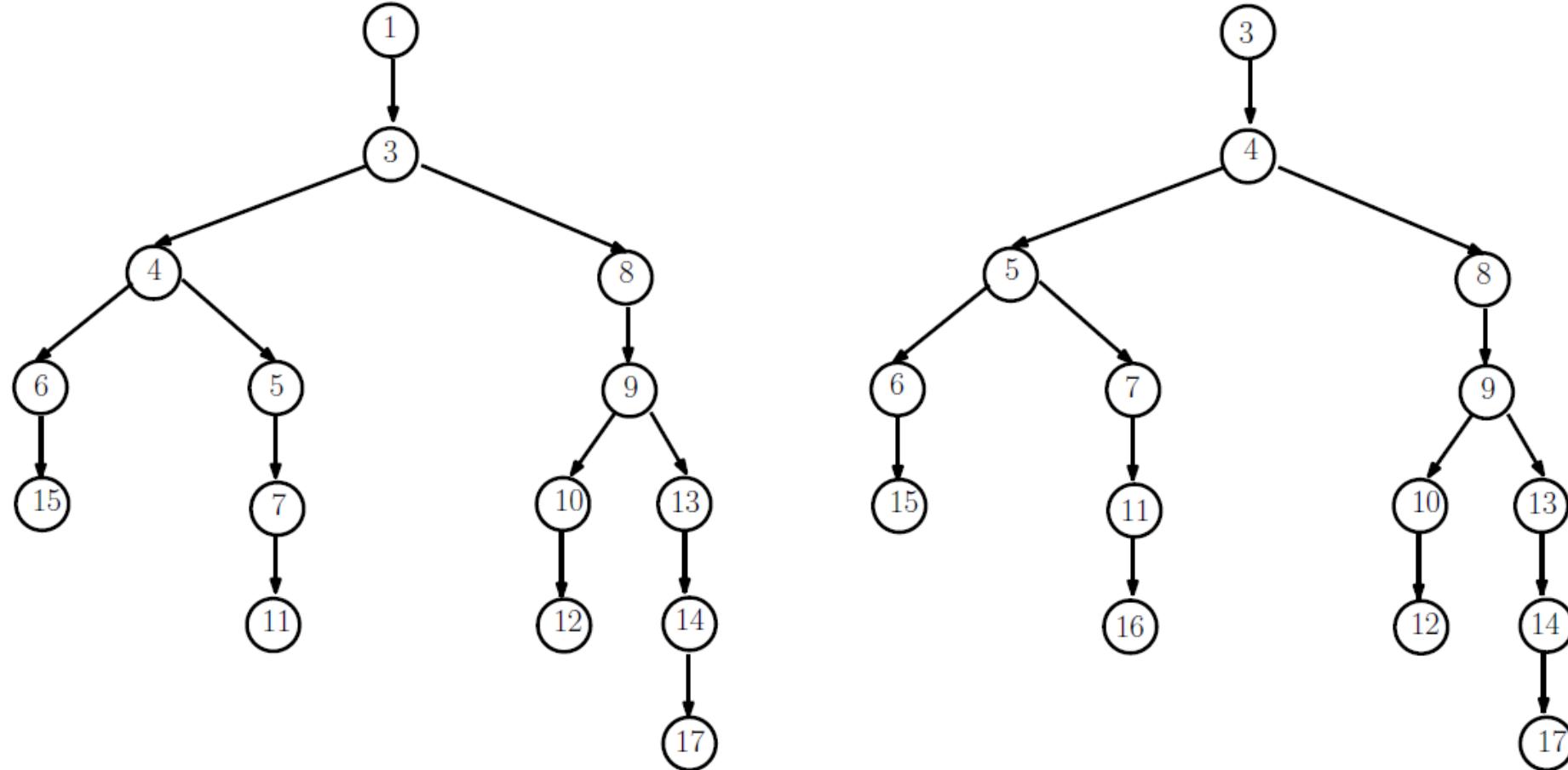
A **strong heap** is a heap together where nodes dominate their right siblings

Rotating Sift-Down:



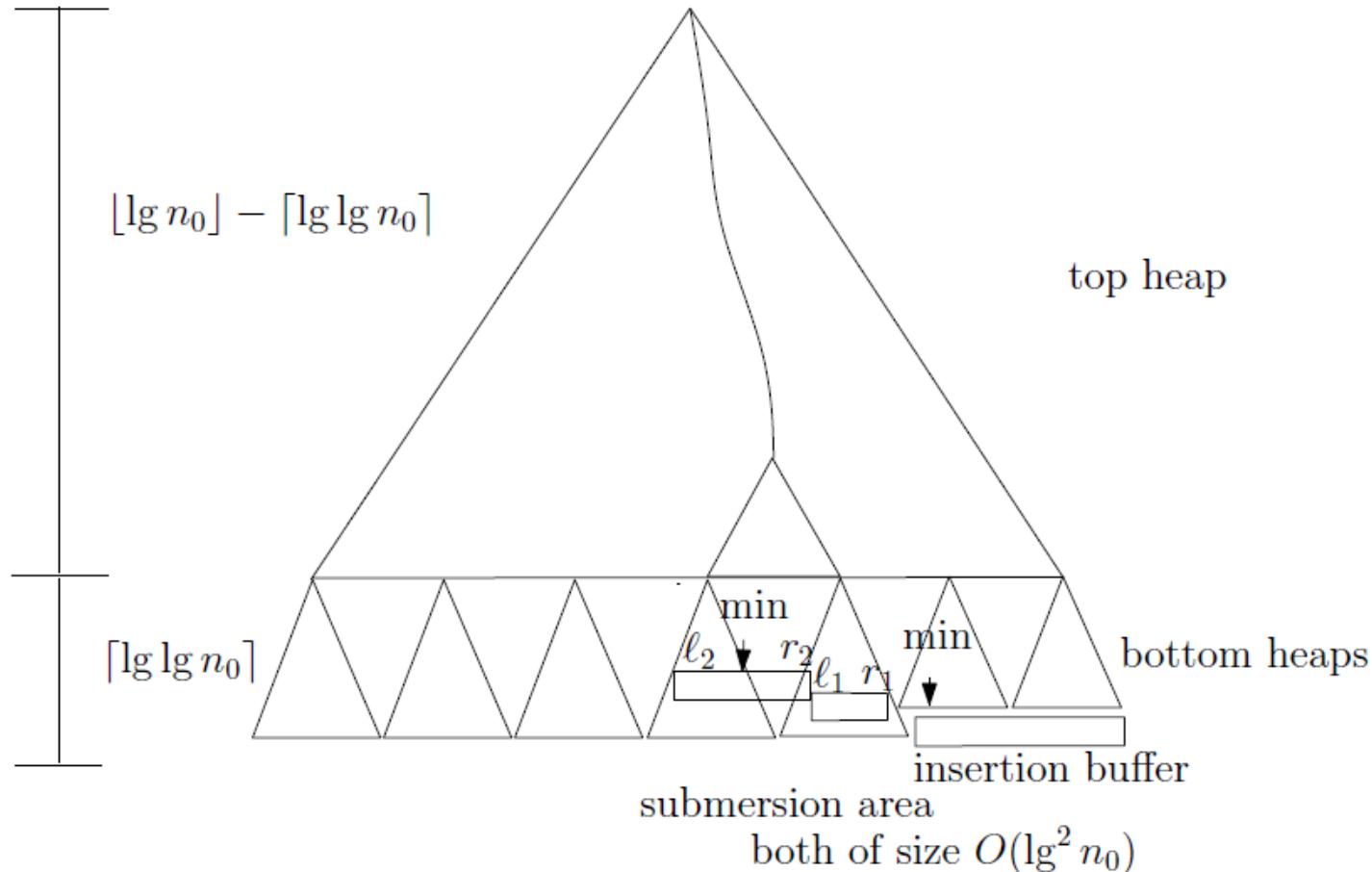
## Strong Heaps - Strong Sift-Down

---



# In-Place Heaps

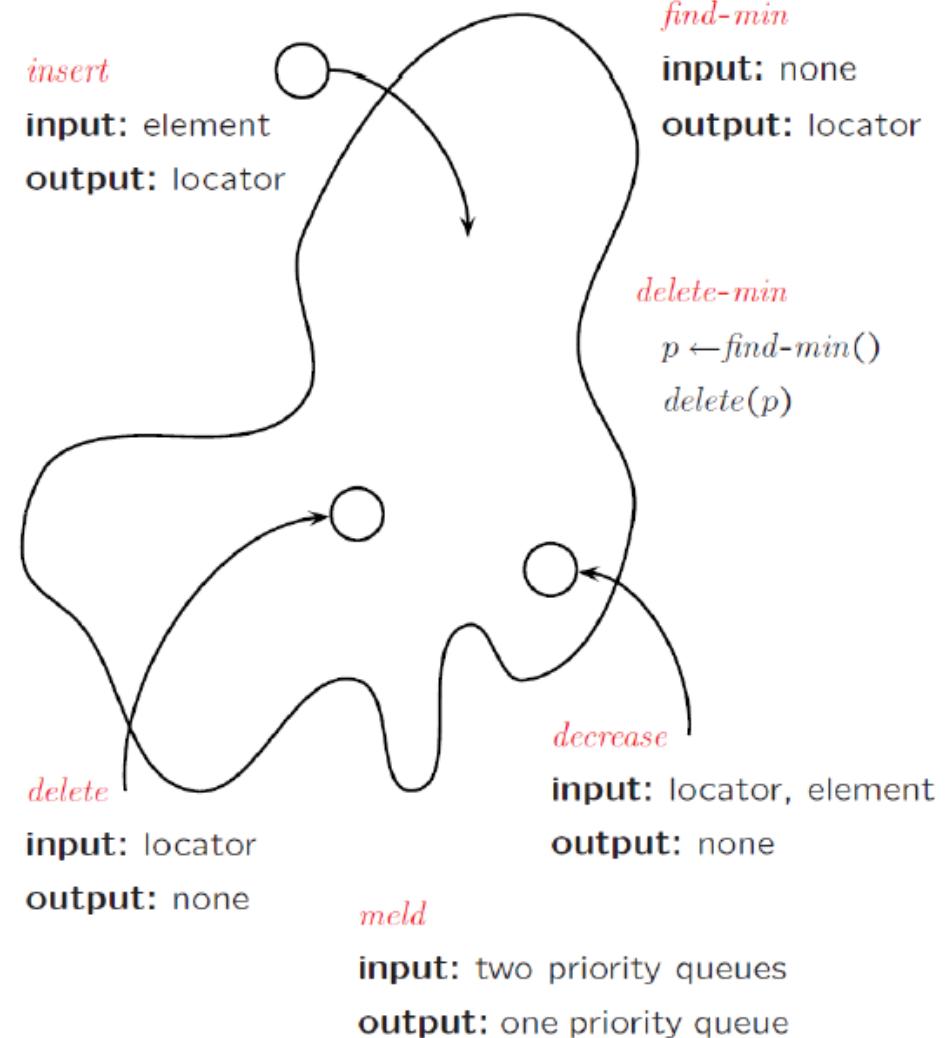
---



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



# Market Analysis

---

| efficiency \ method | binary heap worst case       | binomial queue worst case      | Fibonacci heap amortized | run-relaxed heap worst case    |
|---------------------|------------------------------|--------------------------------|--------------------------|--------------------------------|
| find-min            | $\Theta(1)$                  | $\Theta(1)$                    | $\Theta(1)$              | $\Theta(1)$                    |
| insert              | $\Theta(\lg n)$              | $\Theta(1)$                    | $\Theta(1)$              | $\Theta(1)$                    |
| decrease            | $\Theta(\lg n)$              | $\Theta(\lg n)$                | $\Theta(1)$              | $\Theta(1)$                    |
| delete              | $\Theta(\lg n)$              | $\Theta(\lg n)$                | $\Theta(\lg n)$          | $\Theta(\lg n)$                |
| meld                | $\Theta(\lg m \times \lg n)$ | $\Theta(\min\{\lg m, \lg n\})$ | $\Theta(1)$              | $\Theta(\min\{\lg m, \lg n\})$ |

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

# Result

---

Rank-relaxed weak heaps are **better** than Fibonacci heaps!

| Data structure         | # element comparisons |
|------------------------|-----------------------|
| Fibonacci heap         | $2m + 2.89n \lg n$    |
| Rank-relaxed weak heap | $2m + 1.5n \lg n$     |

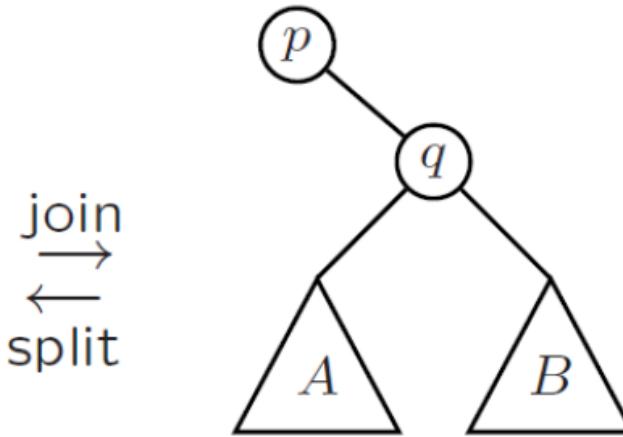
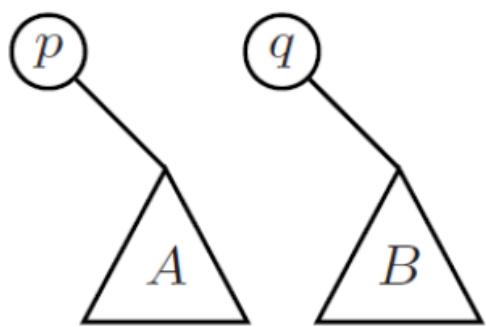
But they are not **simpler**!

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

## Joins

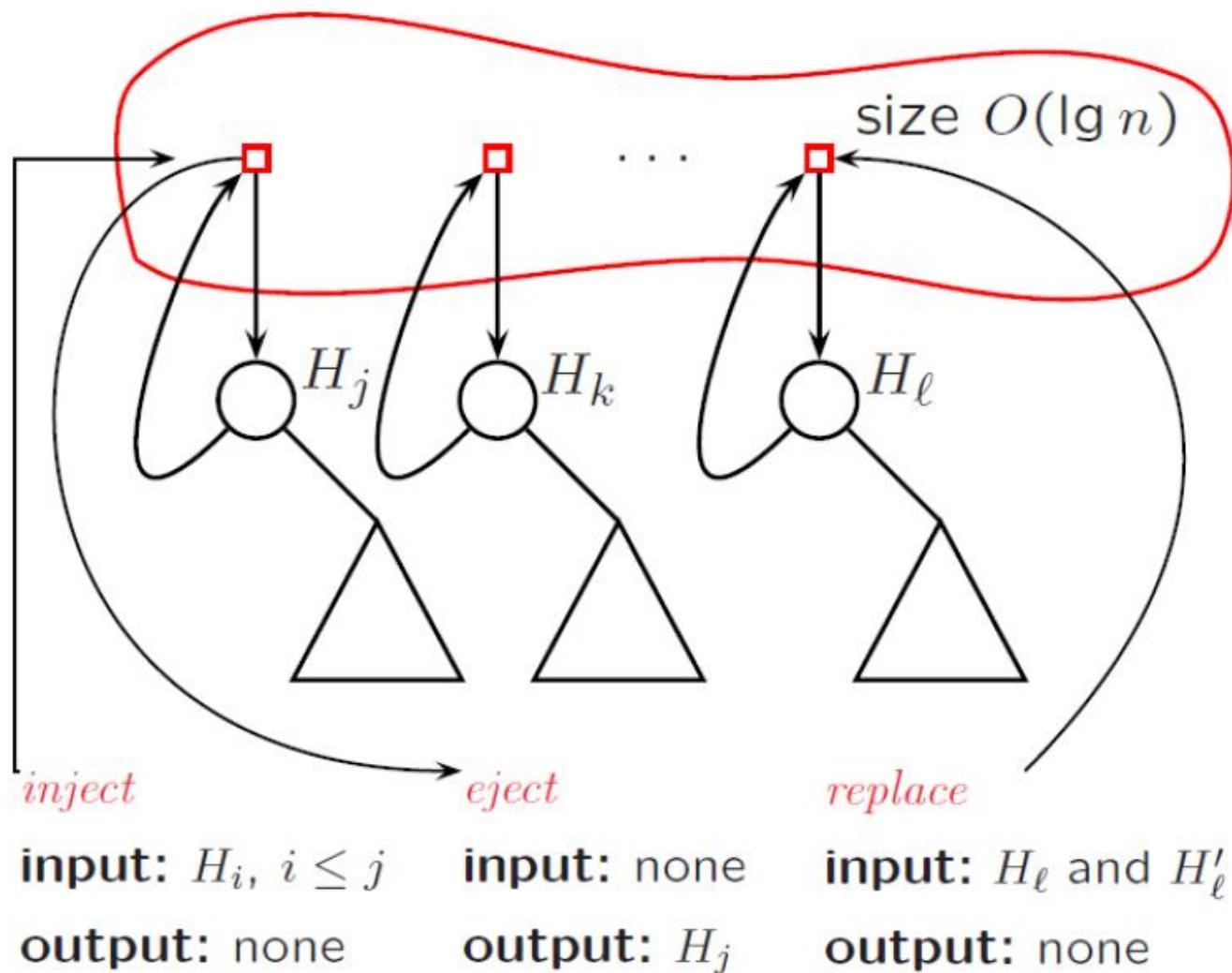
---

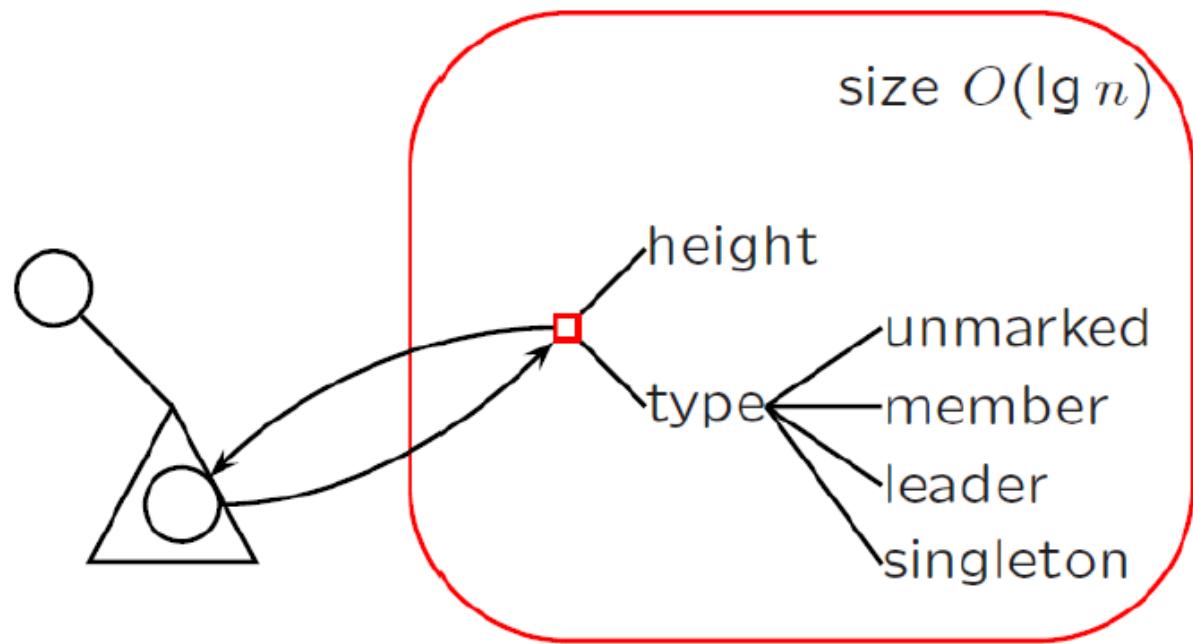
Joining and splitting two perfect weak heaps of the same size:



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

# Heap Registry





*mark*

**input:** a node

**output:** none

*unmark*

**input:** a node

**output:** none

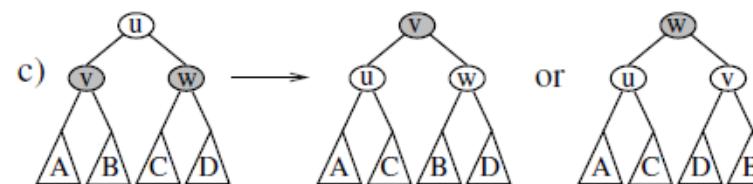
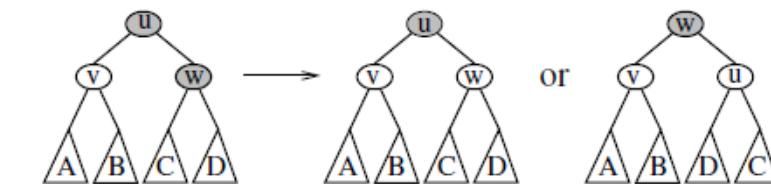
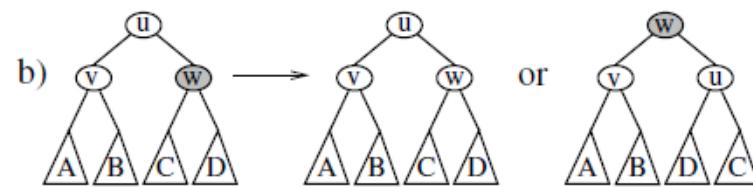
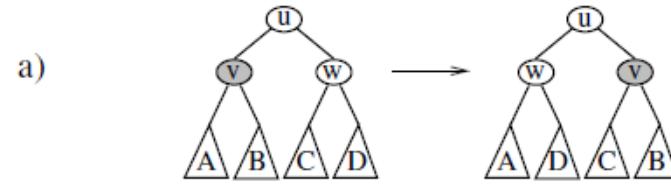
*reduce*

**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.

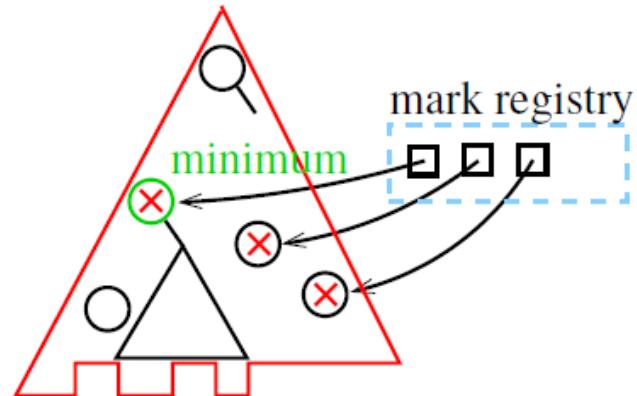
Gray nodes are marked.

- $\lambda \leq \lfloor \lg n \rfloor - 1$  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.

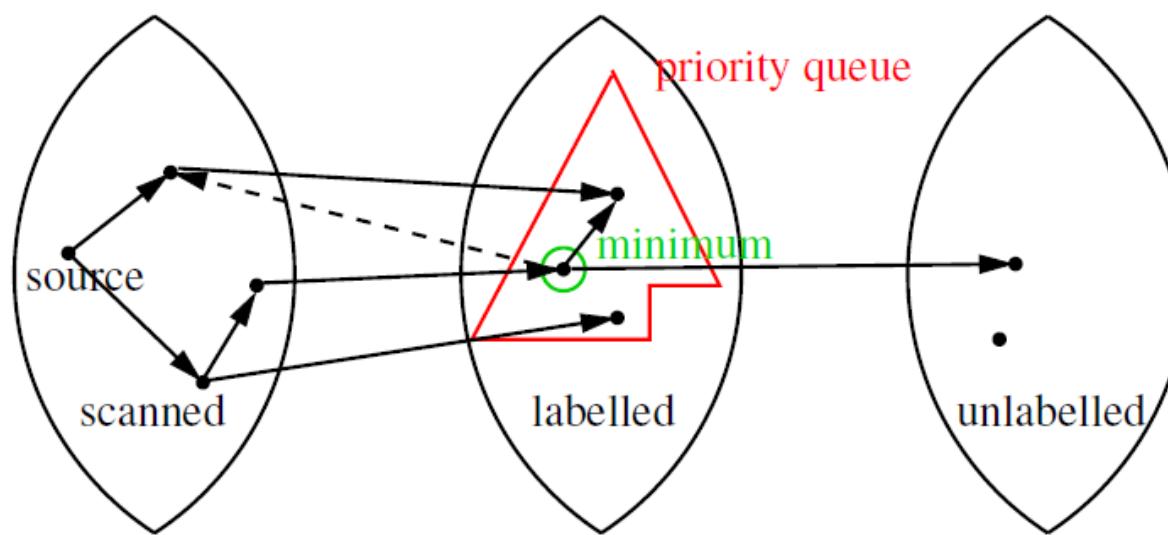


**Improvement in *extract-min*:** If the mark registry is more than half full before the minimum finding, empty it.

## 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.

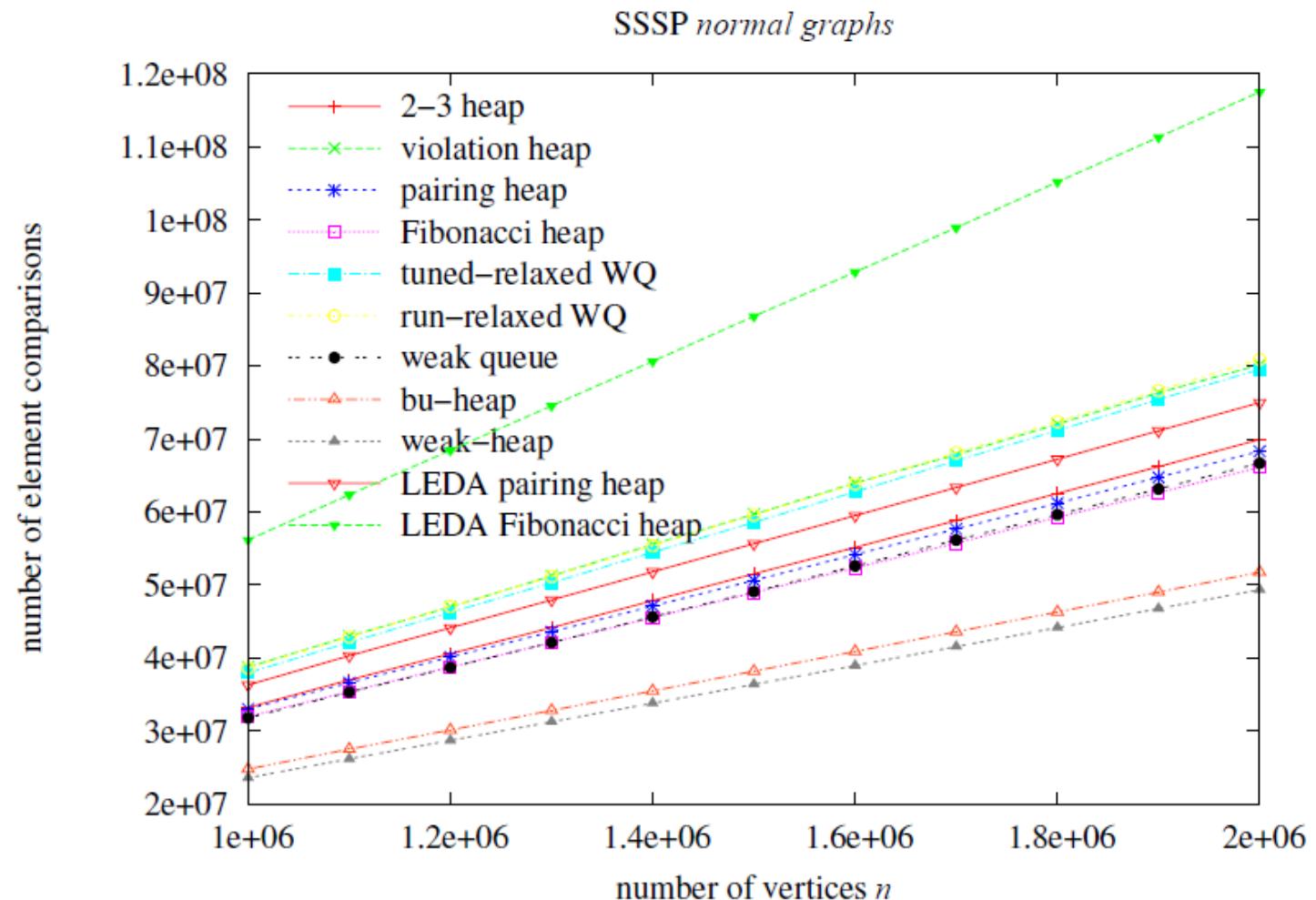
- Which algorithm
- Which graph representation
- Which priority queue
- Which tuning level

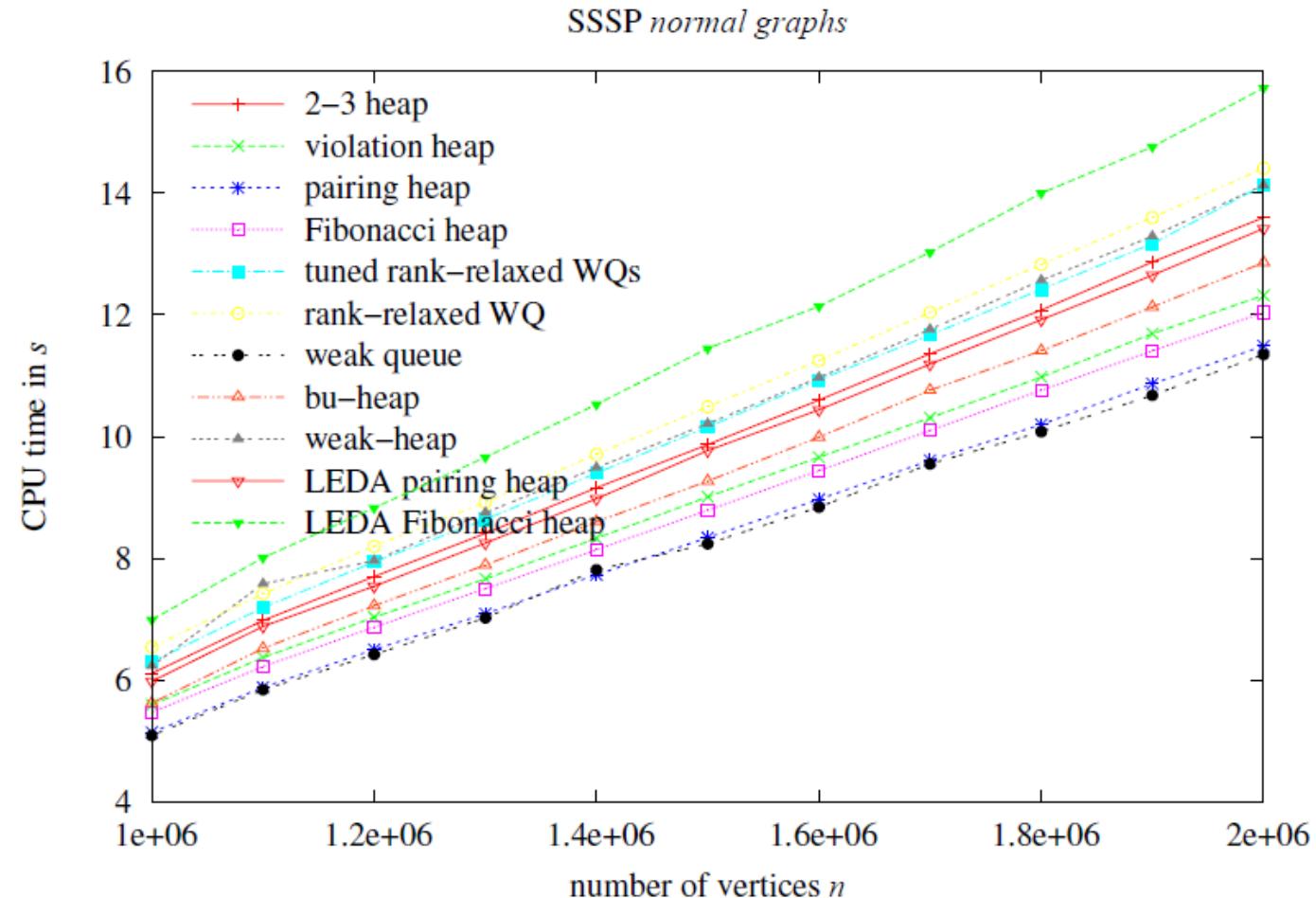


a factor of two speed-up

# Policy-Based Benchmarking - Comparisons

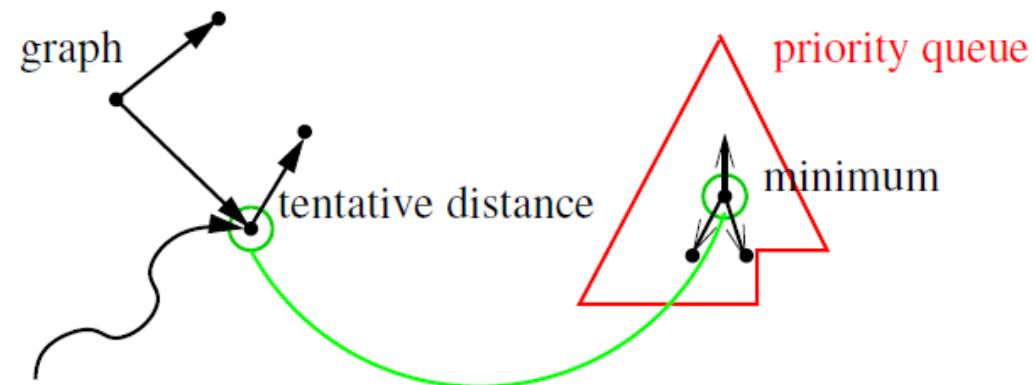
---





# 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 $n$ [ $\mu\text{s}$ ]

| Structure           | CPH STL        | LEDA 6.2       |
|---------------------|----------------|----------------|
| Operation           | Fibonacci heap | Fibonacci heap |
| <i>insert</i>       |                |                |
| $n: 10\,000$        | 0.10           | 0.18           |
| $n: 100\,000$       | 0.09           | 0.15           |
| $n: 1\,000\,000$    | 0.09           | 0.15           |
| <i>decrease-key</i> |                |                |
| $n: 10\,000$        | 0.03           | 0.06           |
| $n: 100\,000$       | 0.05           | 0.22           |
| $n: 1\,000\,000$    | 0.06           | 0.31           |
| <i>extract-min</i>  |                |                |
| $n: 10\,000$        | 0.7            | 1.2            |
| $n: 100\,000$       | 1.4            | 2.7            |
| $n: 1\,000\,000$    | 2.8            | 4.5            |

## Element comparisons per $n$

| Structure           | CPH STL        | LEDA 6.2       |
|---------------------|----------------|----------------|
| Operation           | Fibonacci heap | Fibonacci heap |
| <i>insert</i>       |                |                |
| $n: 10\,000$        | 0              | 1              |
| $n: 100\,000$       | 0              | 1              |
| $n: 1\,000\,000$    | 0              | 1              |
| <i>decrease-key</i> |                |                |
| $n: 10\,000$        | 0              | 2              |
| $n: 100\,000$       | 0              | 2              |
| $n: 1\,000\,000$    | 0              | 2              |
| <i>extract-min</i>  |                |                |
| $n: 10\,000$        | 16.2           | 29.9           |
| $n: 100\,000$       | 21.2           | 38.3           |
| $n: 1\,000\,000$    | 26.2           | 46.5           |

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

a factor of two speed-up

# What is the Best?

---

## Our reference sequence

**Theory:** rank-relaxed weak heap

**Dijkstra—time:** binary heap  
[Williams 1964]

**Dijkstra—comps:** weak heap  
[Dutton 1993]

## Worst case per operation

*insert*—**time:** Fibonacci heap  
[Fredman & Tarjan 1987]

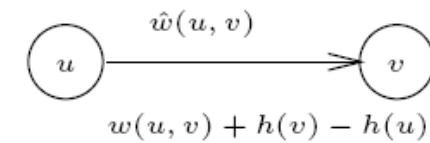
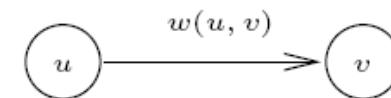
*insert*—**comps:** Fibonacci heap  
*decrease-key*—**time:** Fibonacci  
heap

*decrease-key*—**comps:** Fibonacci heap

*extract-min*—**time:** weak queue  
[Vuillemin 1978]

*extract-min*—**comps:** weak heap

# A\* = Dijkstra + Reweighting



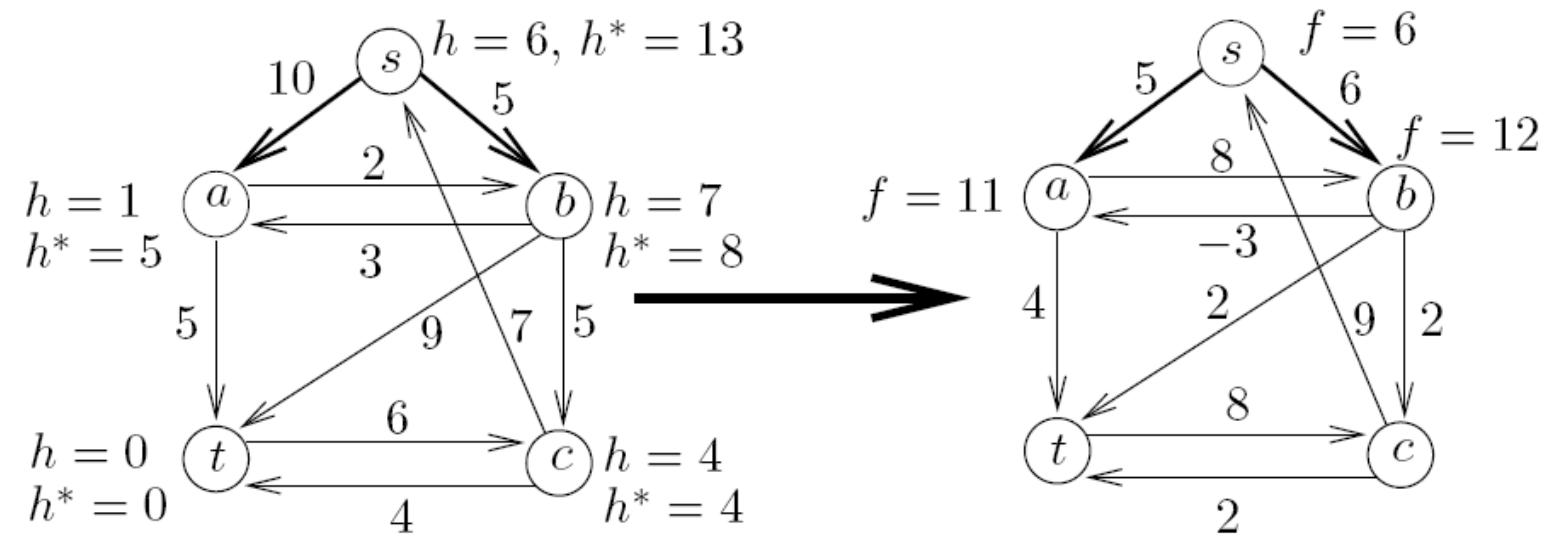
$h$  **consistent**  $0 \leq w(u,v) + h(v) - h(u)$

**Initialisierung:**  $f(s) = h(s)$ ,  $f(u) = \text{infinity}$ , if  $u \neq s$

**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



# Set-Based Dijkstra

$\text{open}_0 \leftarrow I, \text{closed} \leftarrow \perp, g \leftarrow 0$

**repeat**

**if** ( $\text{open}_g \wedge G \neq \perp$ ) **STOP**

$\text{open}_g \leftarrow \text{open}_g \wedge \neg \text{closed}$

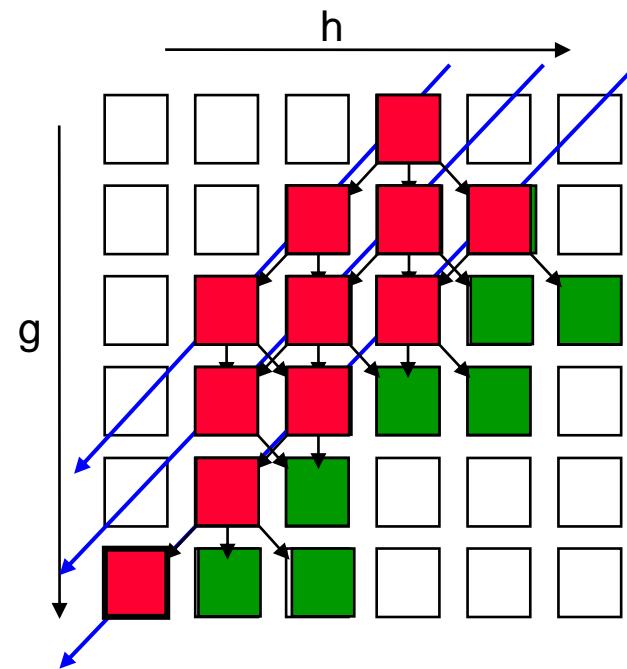
für  $c \leftarrow 1, \dots, C$

$\text{open}_{g+c} \leftarrow \text{open}_{g+c} \vee \text{image}_c(\text{open}_g)$

$\text{closed} \leftarrow \text{closed} \vee \text{open}_g$

$g \leftarrow g + 1$

# Set A\*



# Overview

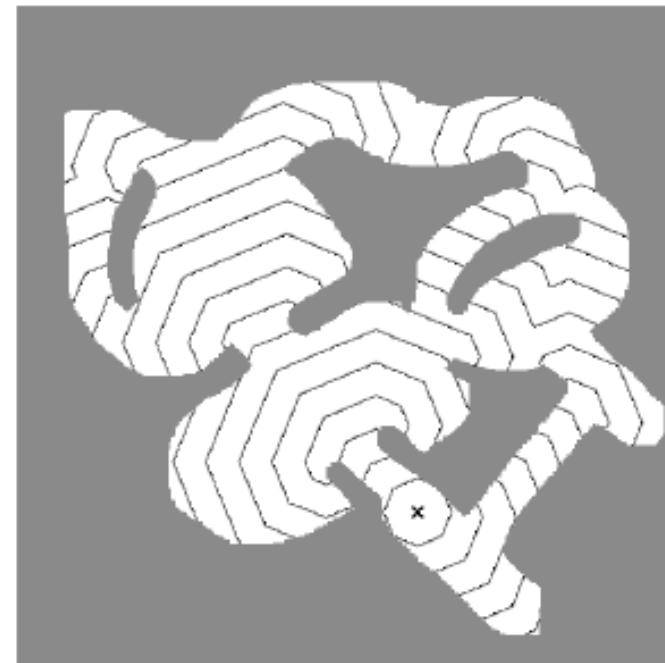
- AE 4 Sorting
- AE 4 Searching
- Application Area

# 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

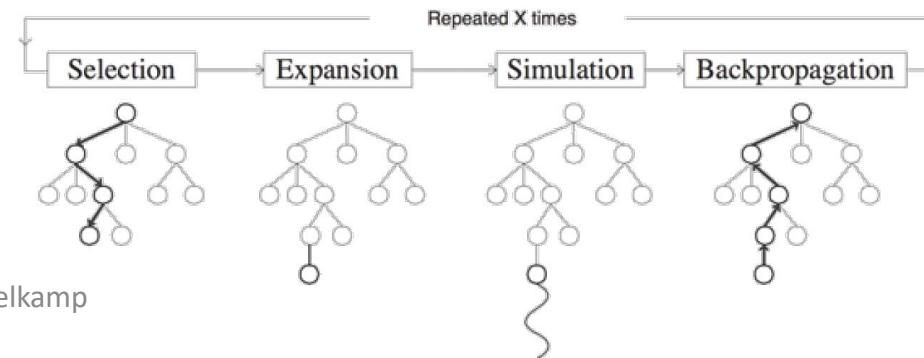
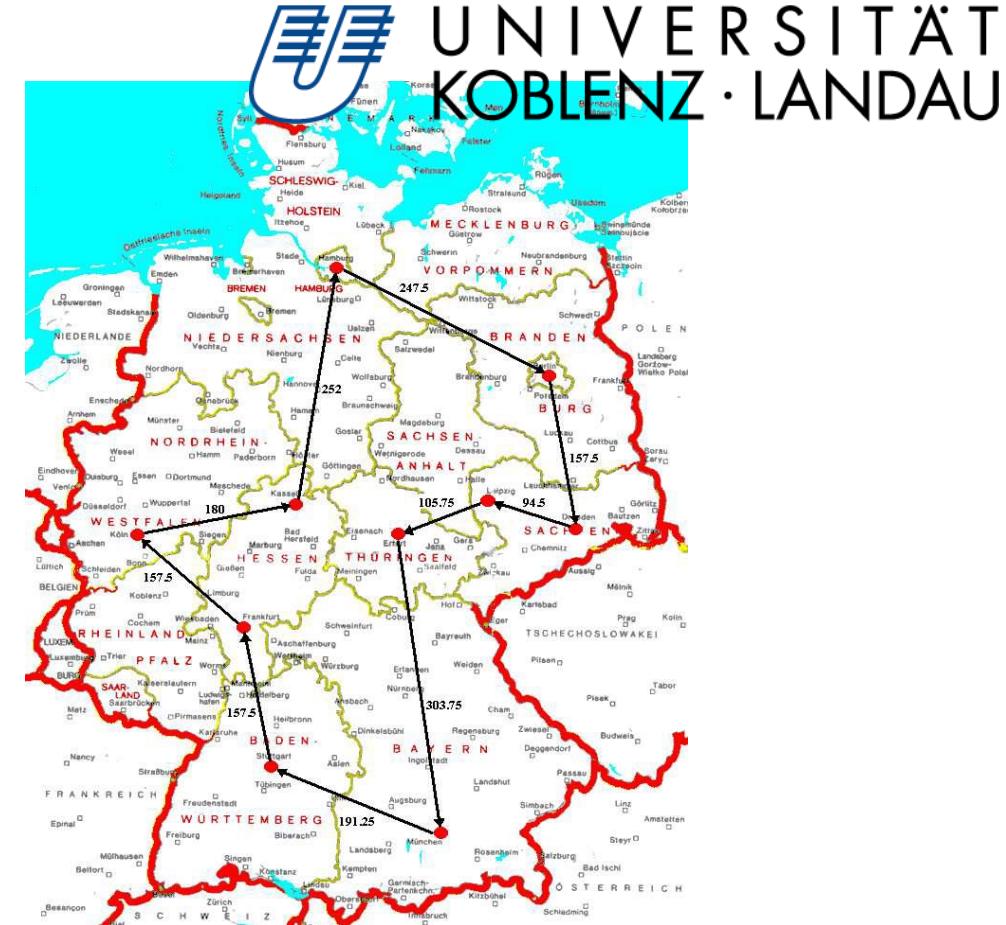
# Idea for PTSP

|  |  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|--|--|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|  |  |   |   | q |   |   | q |   |   |   |   |   |   |   |   |
|  |  | f | f | f | f | f | A | f | f | f | f | f | f | f | f |
|  |  |   |   | q |   |   |   | q |   |   | q |   | q |   |   |



# Traveling Salesman Tours

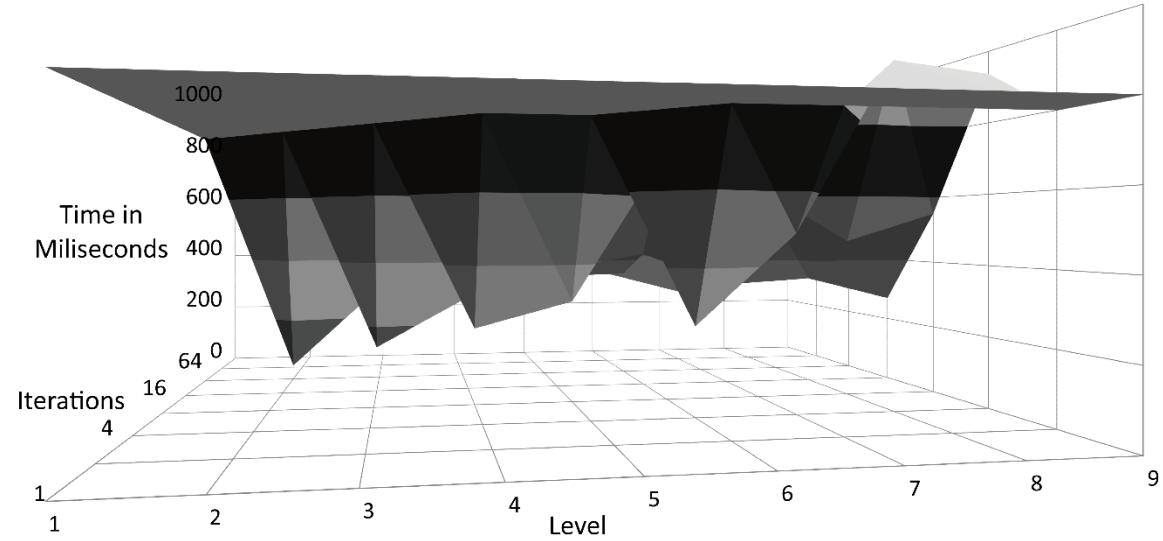
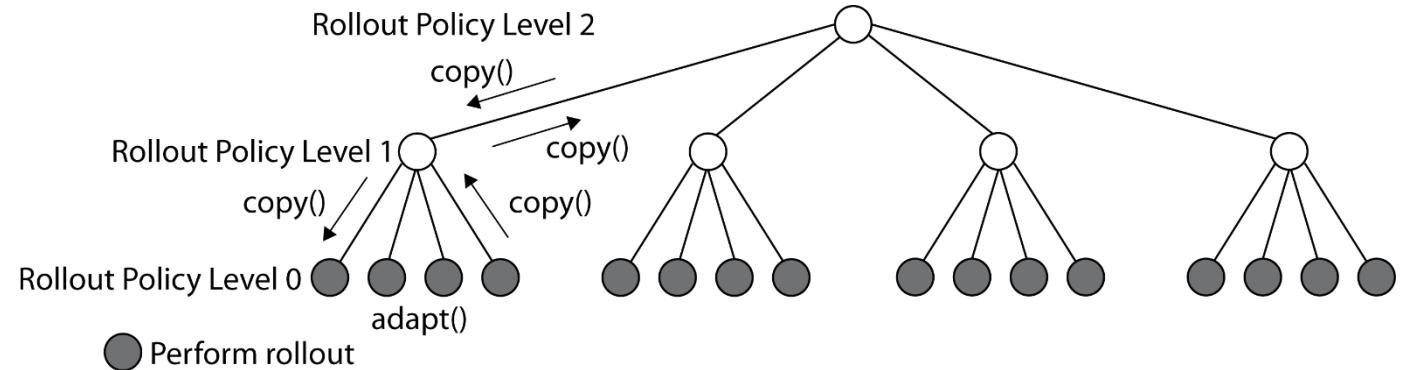
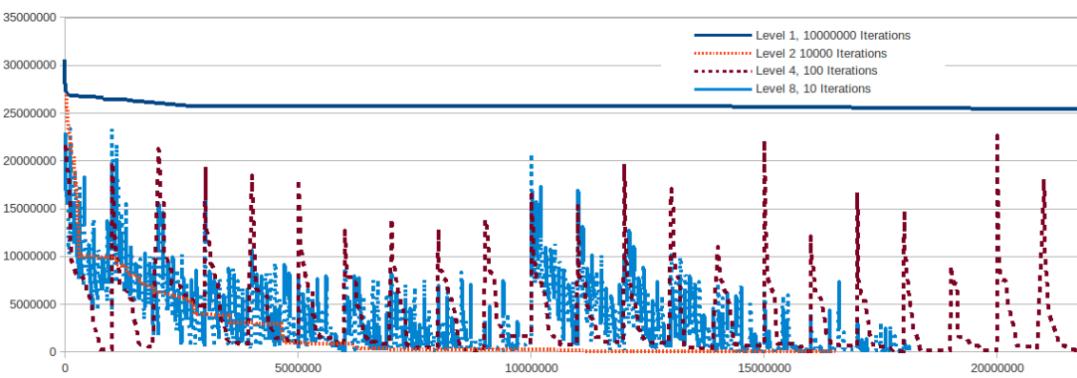
- Given a map, compute a minimum-cost round trip visiting certain cities
- Shortest paths graph reduction: precompute all-pairs-shortest-paths with
- Dijkstra's algorithm (be smart: employ radix heaps)
- Model problem as an IP and call solver (CPLEX, IPSolve, ...)
- Neighborhood search (xOPT: SA; GA; AA; PSO; LNS, ...)
- Depth-First Branch and Bound** with
- DFBnB<sub>0</sub>** No Heuristic – incremental O(1) time
- DFBnB<sub>1</sub>** Column/Row Minima – incremental O(1) time
- DFBnB<sub>2</sub>** Assignment Problem – incremental O( $n^2$ ) time
- ... New in the arena: Monte-Carlo Search



# Nested Rollout Policy Adaptation

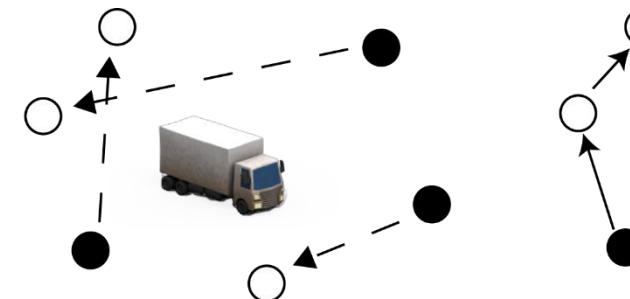
**Input:** Iteration width (exploitation),  
nestedness (exploration)

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

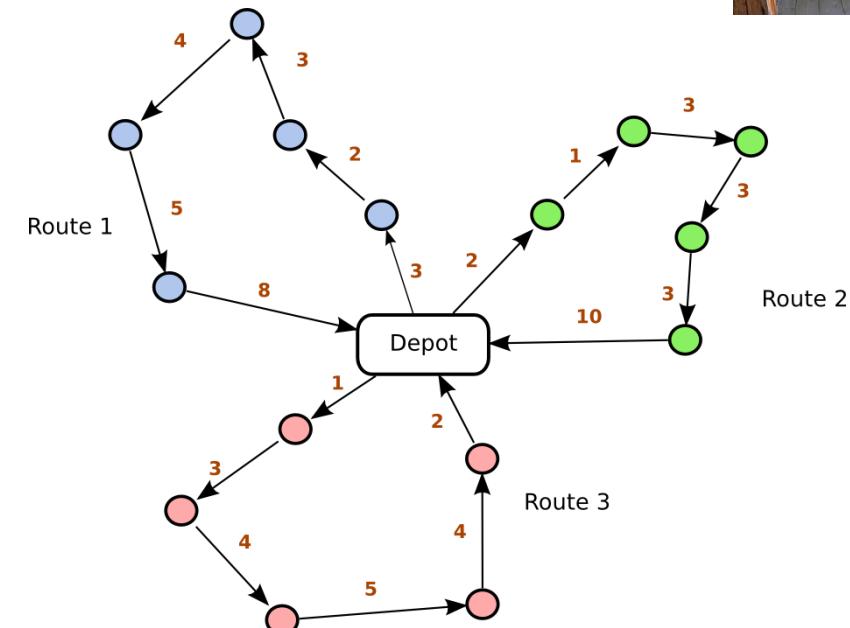
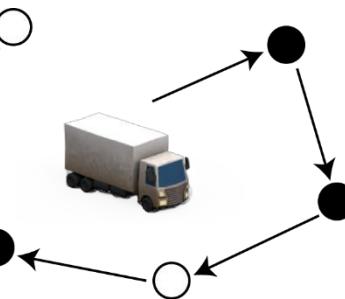


# Constraints

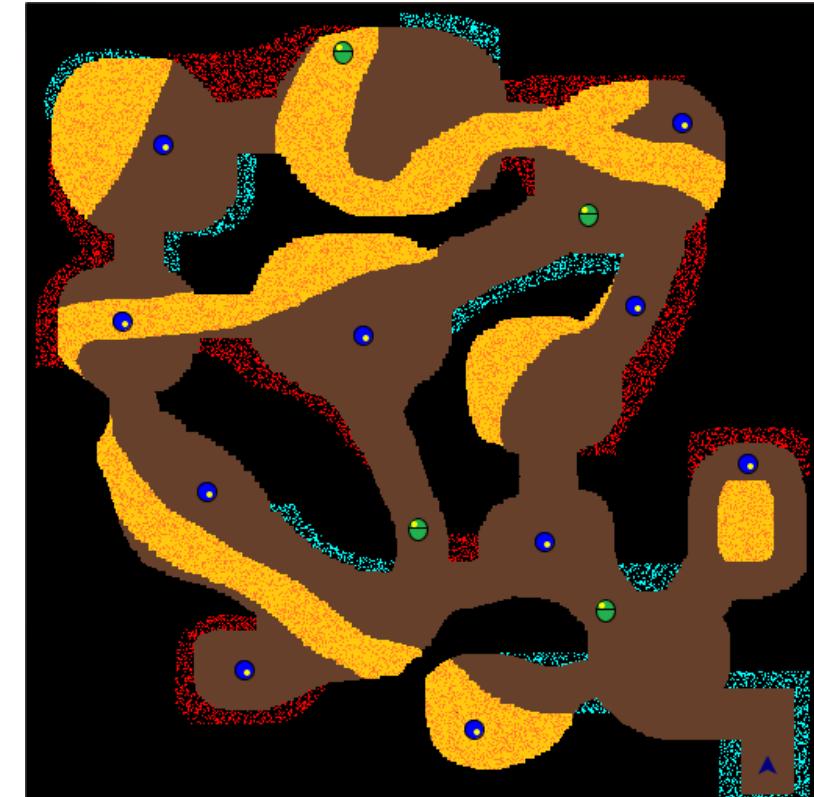
- Time Windows, Capacities, Premium Services, Pickup and Deliveries
- TSP+TW: Restricted time intervals / service times
- C+TSP: Limited vehicle load
- TSP+PD: Pickup and deliveries (PDP)
- TSP\*: Premium service – same-day delivery preferred
- VRP: Vehicle routing – several vehicles



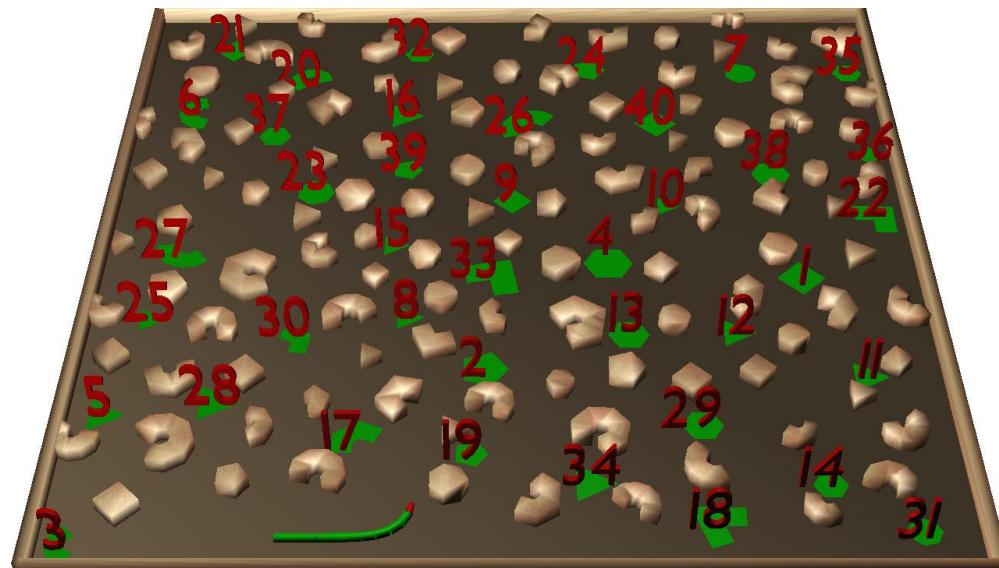
● pickup stop  
 ○ delivery stop



# Physical TSP

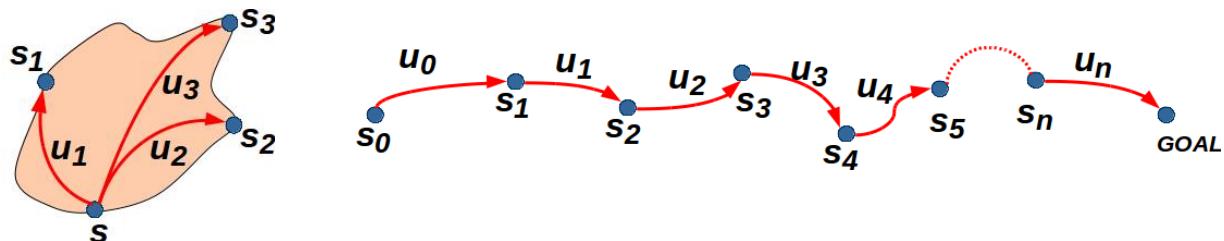


# Multi-Goal Motion Planning with Dynamics



# Dynamics

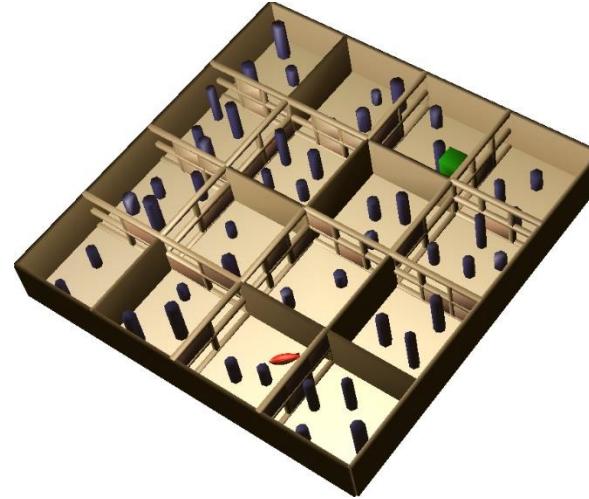
- Express relation between input controls and resulting motions
- Necessary to plan motions that can be executed
- Impose significant challenges
  - Constrain the feasible motions
  - Often are nonlinear and high-dimensional
  - Give rise to nonholonomic systems
  - State and control spaces are continuous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

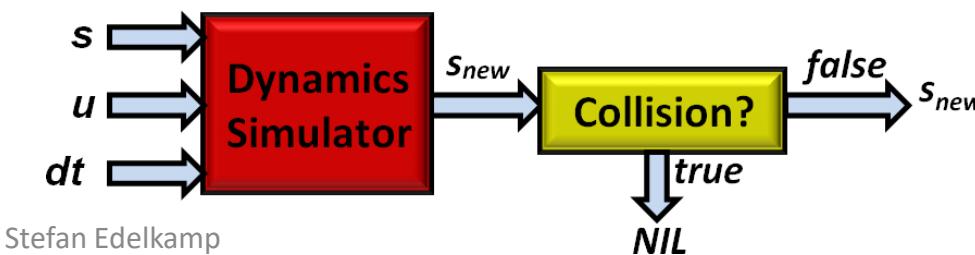
ROS/Gazebo, ODE, Bullet, PhysX  
general rigid-body dynamics  
friction and gravity

$$\dot{s} = f(s, u)$$

$$s = (x, y, \theta_0, v, \psi, \theta_1, \dots, \theta_n)$$

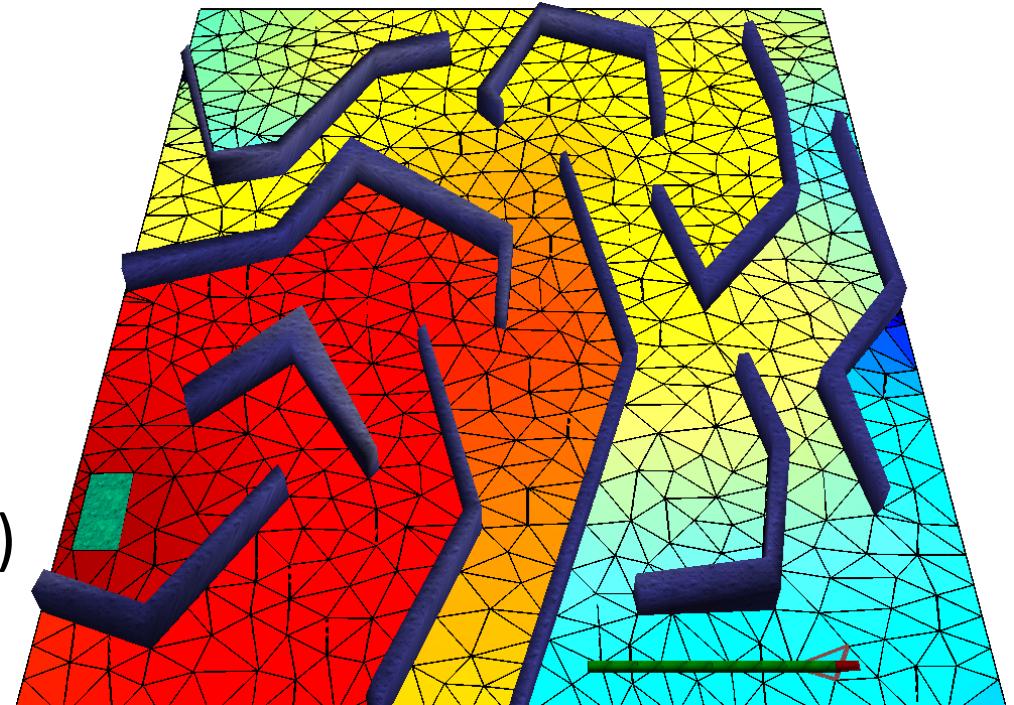
$$\dot{x} = v \cos(\theta_0) \quad \dot{y} = v \sin(\theta_0) \quad \dot{\theta}_0 = v \tan(\psi) \quad \dot{v} = a \quad \dot{\psi} = \omega$$


$$\dot{\theta}_i = \frac{v}{d} \left( \prod_{j=1}^{i-1} \cos(\theta_{j-1} - \theta_j) \right) (\sin(\theta_{i-1}) - \sin(\theta_i))$$

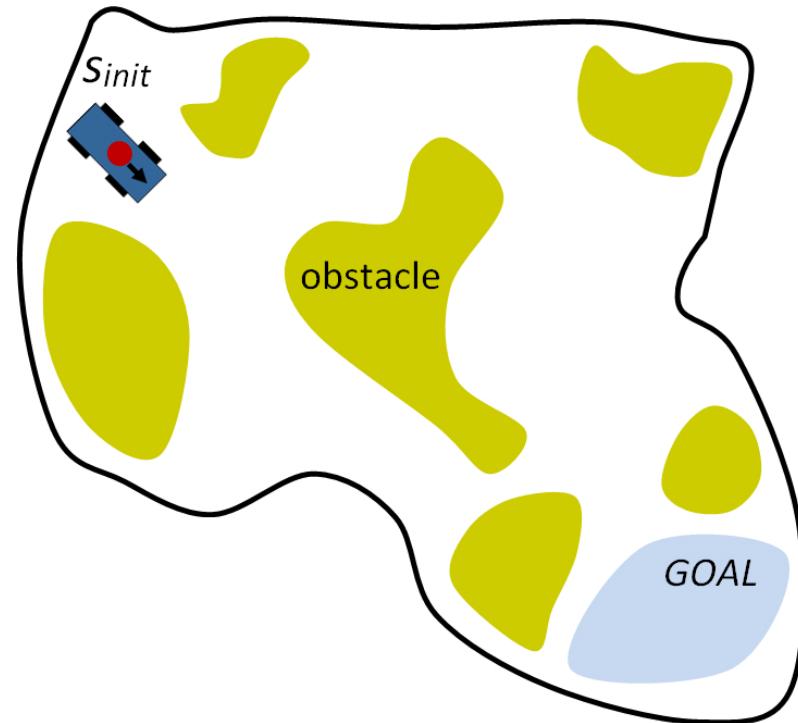


# Introduce Discrete Layer to Guide the Search

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

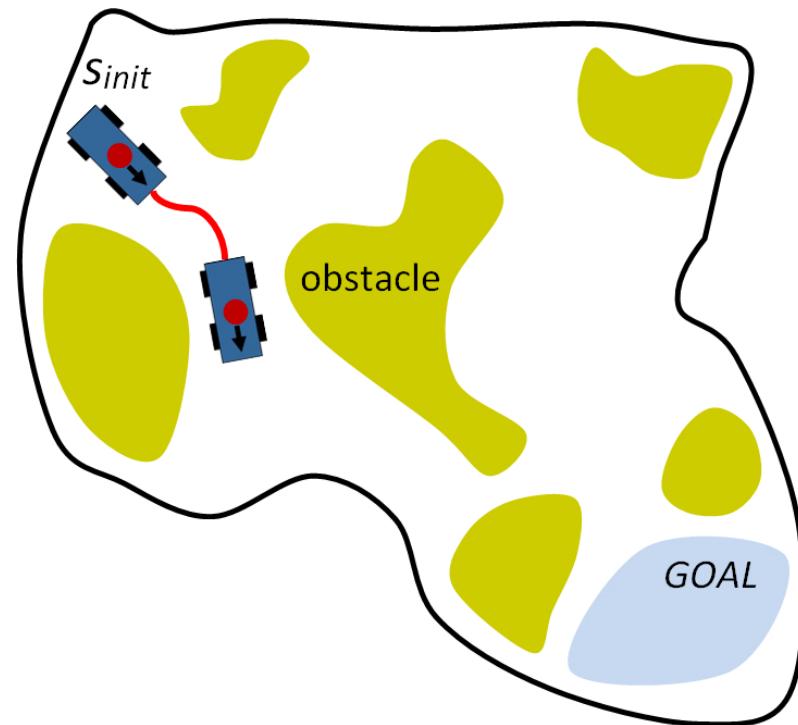


# Sampling Based Motion Planning



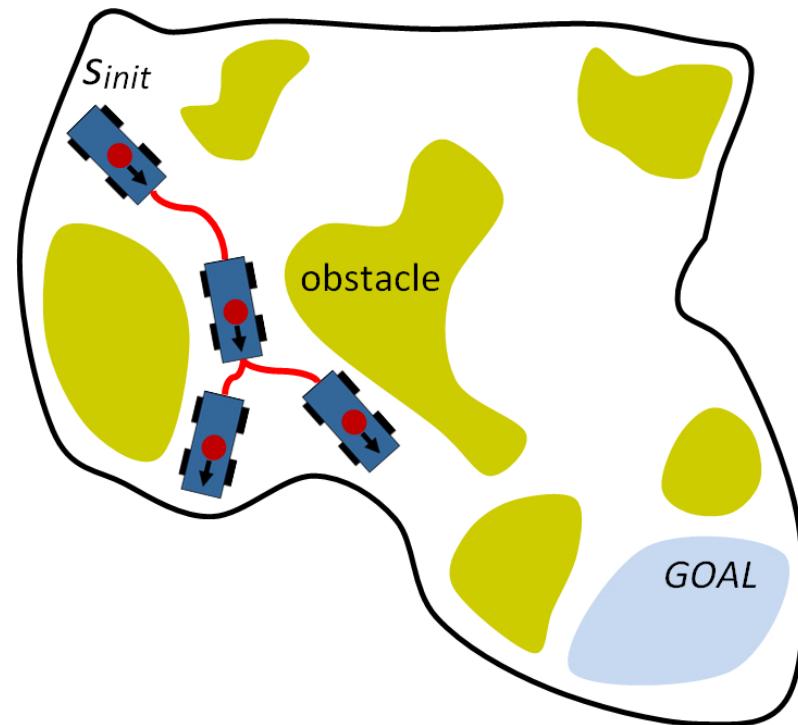
- Expand a tree  $T$  of collision-free and
- dynamically-feasible motions
  - select a state  $s$  from which to expand the tree
  - sample control input  $u$
  - generate new trajectory by
  - applying  $u$  to  $s$

# 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

selecting an equivalence class  
from which to expand motion tree T

- **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

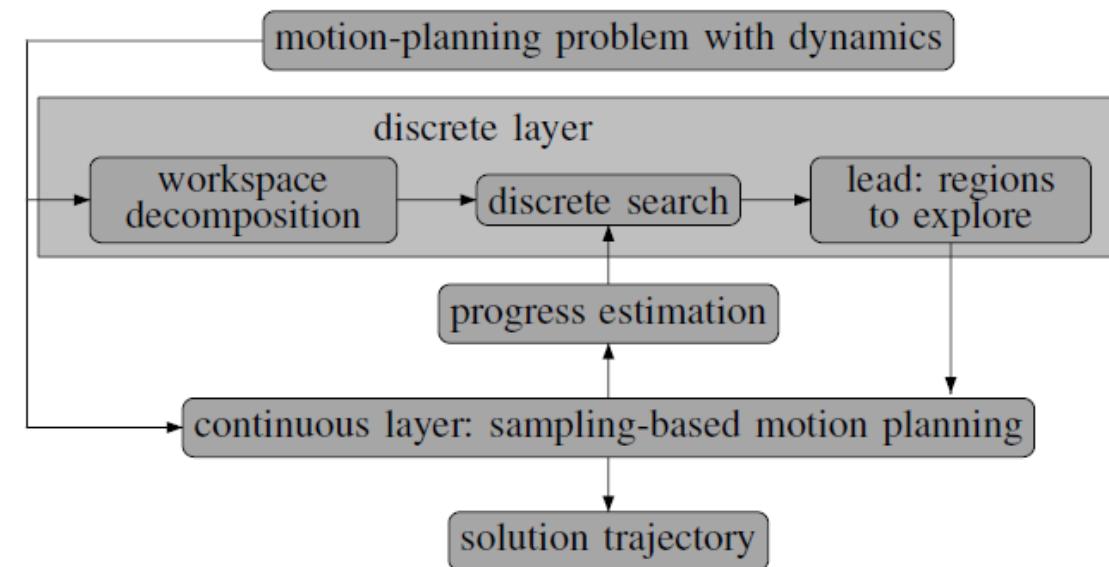
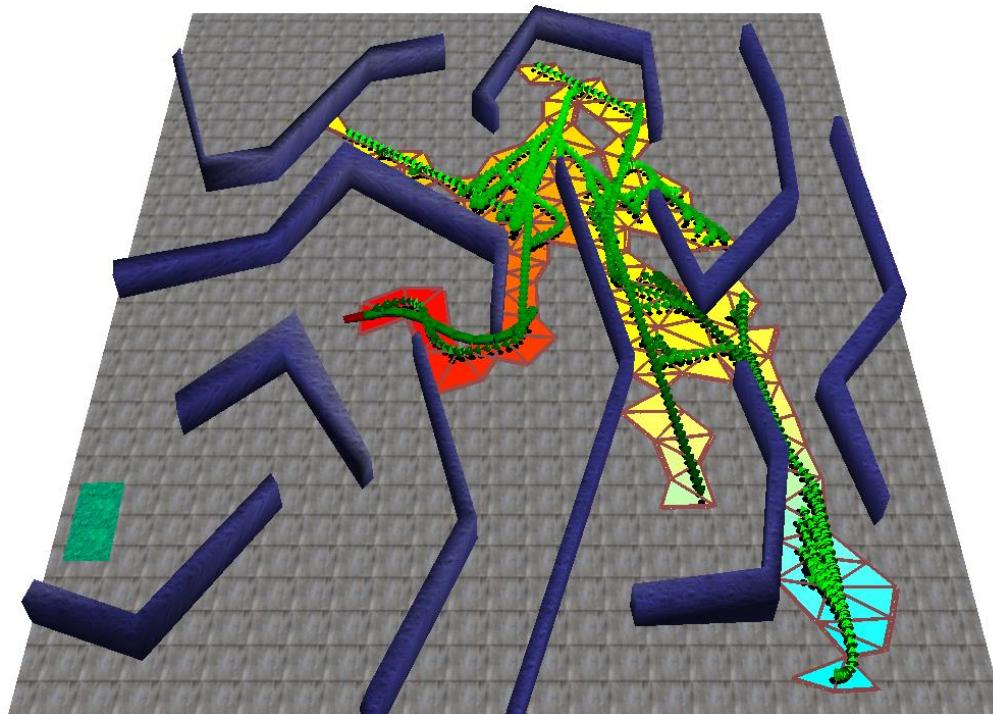
- **Formal guarantees**

- probabilistic completeness

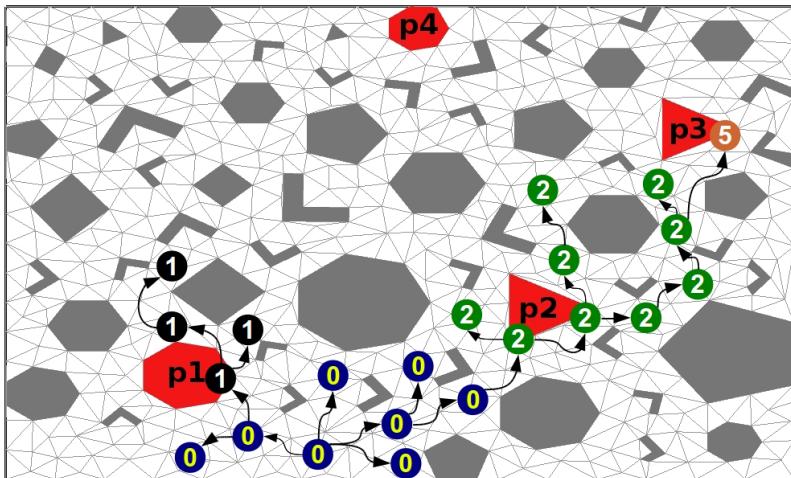


sampling-based motion planning to expand T

# Architecture



# Abstraction



Used to induce partition of motion tree into equivalence classes

$$v_i = v_j \text{ iff}$$

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

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

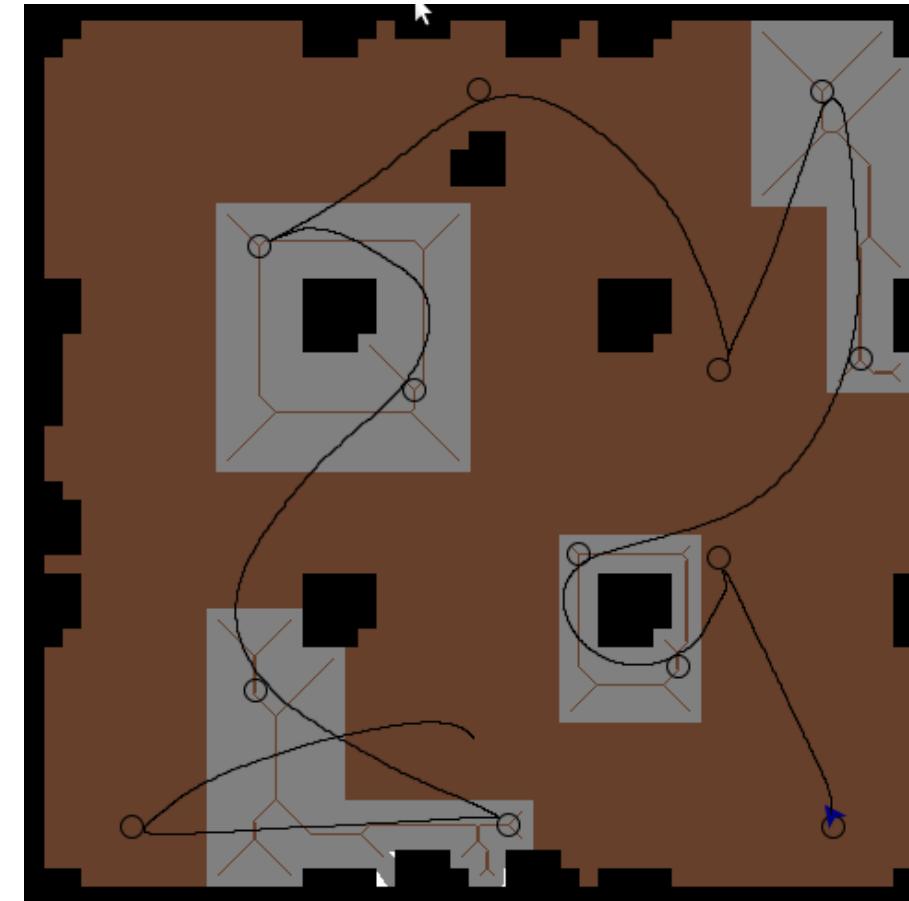
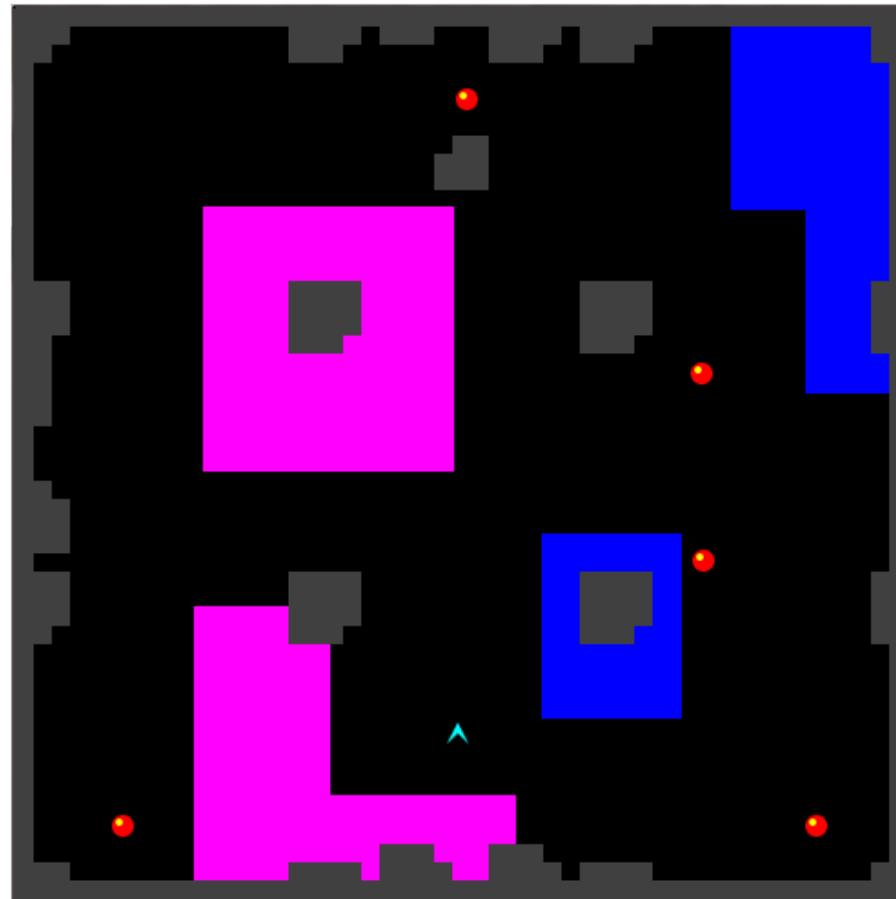
→ equivalence class corresponding to abstract state  $\langle r \rangle$

$$\Gamma_{\langle r \rangle} = \{v \mid v \text{ in } T \text{ and } \text{region}(v) = r\}$$

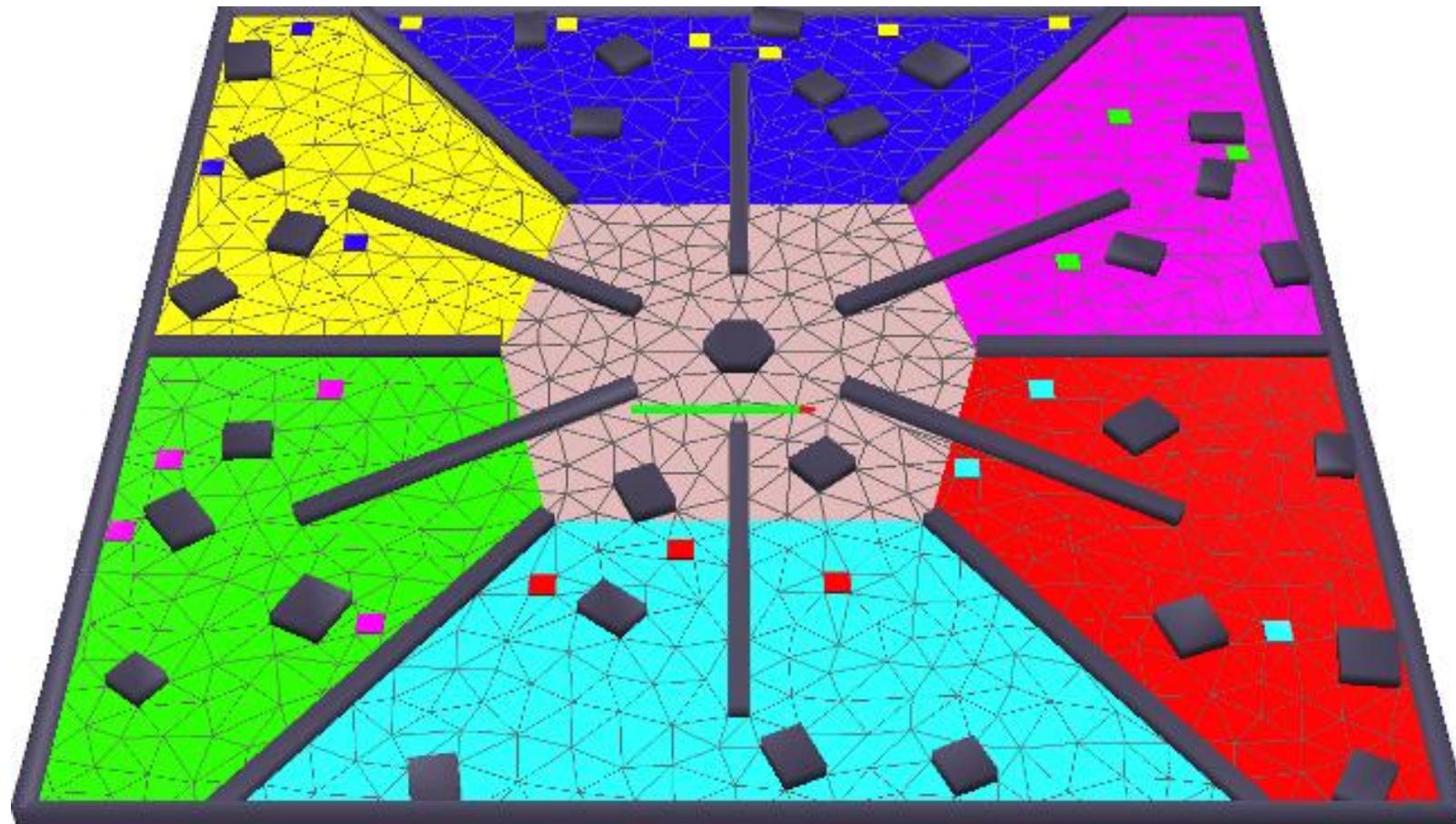
→ partition of motion tree  $T$  into equivalence classes

$$\Gamma = \{\Gamma_{\langle r \rangle} : \Gamma_{\langle r \rangle} > 0\}$$

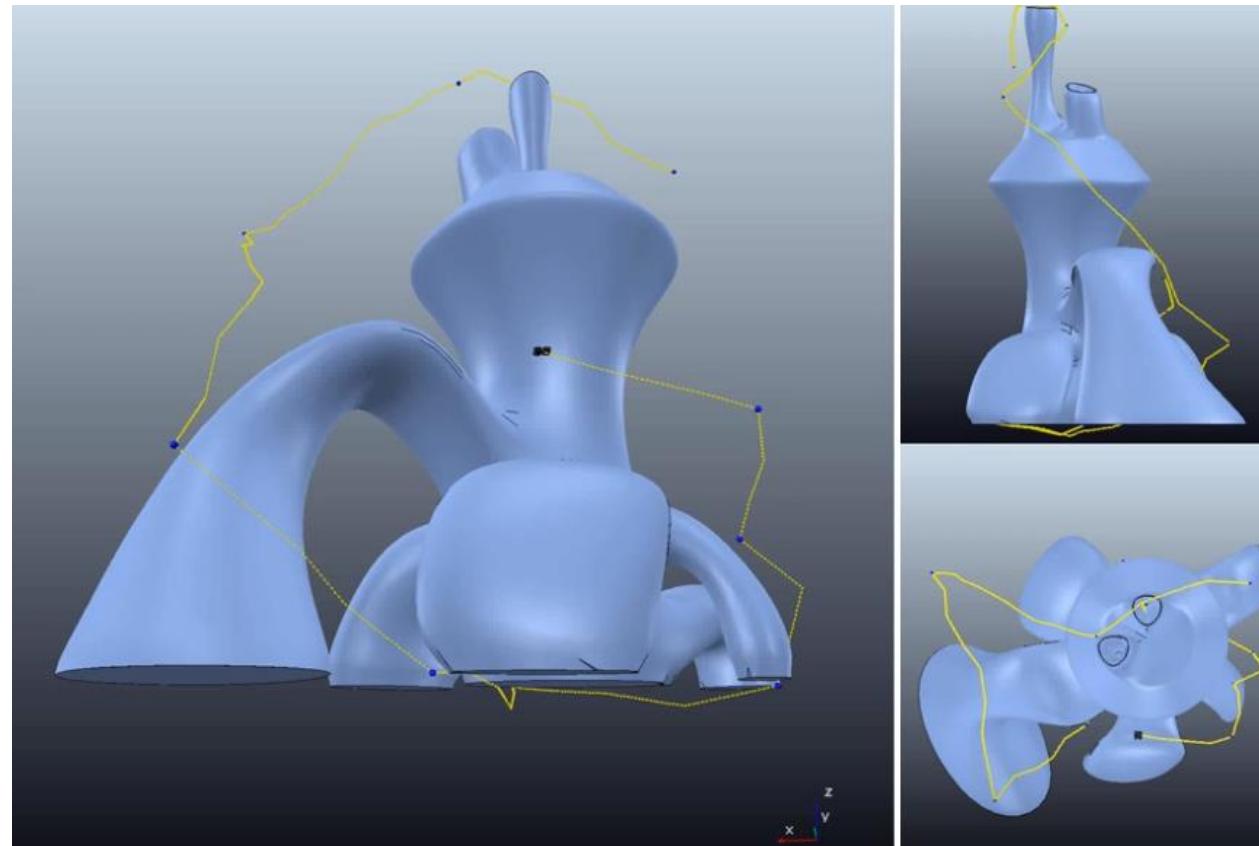
# Coloring and Inspection



# Inspection Problem



# 3D Inside / Outside Inspection



# Questions

---

