

Algorithm Engineering

„Externspeicherplatzsuche“

Stefan Edelkamp

Motivation: Recent Successes in Search

Optimal solutions the **RUBIK'S CUBE**, the (n^2-1) -**PUZZLE**, and the **TOWERS-OF-HANOI** problem, all with state spaces of about or more than a quintillion (a billion times a billion) states.

When processing a million states per second, looking at all states corresponds to **hundreds of thousands** of years.

Even with search heuristics, time and space remain crucial resources: in extreme cases, **weeks of computation time**, **gigabytes of main memory** and **terabytes of hard disk** space have been invested to solve search challenges.

Recent Examples Disk-based Search

- RUBIK'S CUBE:** 43,252,003,274,489,856,000 states,
1997: Solved optimally by a general-propose strategy, which used 110 megabytes of main memory for guiding the search, for the hardest instance the solver generated 1,021,814,815,051 states to find an optimum of 18 moves in 17 days.
- 2007: By performing a breadth-first search over subsets of configurations, starting with the solved one, in 63 hours with the help of 128 processor cores and 7 terabytes of disk space it was shown that 26 moves suffice.

Recent Examples of Disk-based Search

With recent search enhancements, the average solution time for optimally solving the **FIFTEEN-PUZZLE** with over 10^{13} states is about milliseconds, looking at thousands of states.

The state space of the **FIFTEEN-PUZZLE** has been completely generated in 3 weeks using 1.4 terabytes of secondary memory.

Tight bounds on the optimal solutions for the **THIRTY-FIVE-PUZZLE** with over 10^{41} states have been computed in more than one month total time using 16 gigabytes RAM and 3 terabytes hard disk.

Recent Examples of Disk-based Search

The 4-peg 30-disk **TOWERS-OF-HANOI** problem spans a state space of $4^{30} = 1, 152, 921, 504, 606, 846, 976$ states

Optimally solved integrating a significant number of research results consuming about 400 gigabytes hard disk space in 17 days.

Outline

Review of basic graph-search techniques,

- limited-memory graph search,
- including frontier search

Introduction to External-Memory Algorithms and
I/O Complexity Analysis

External-Memory Search Algorithms

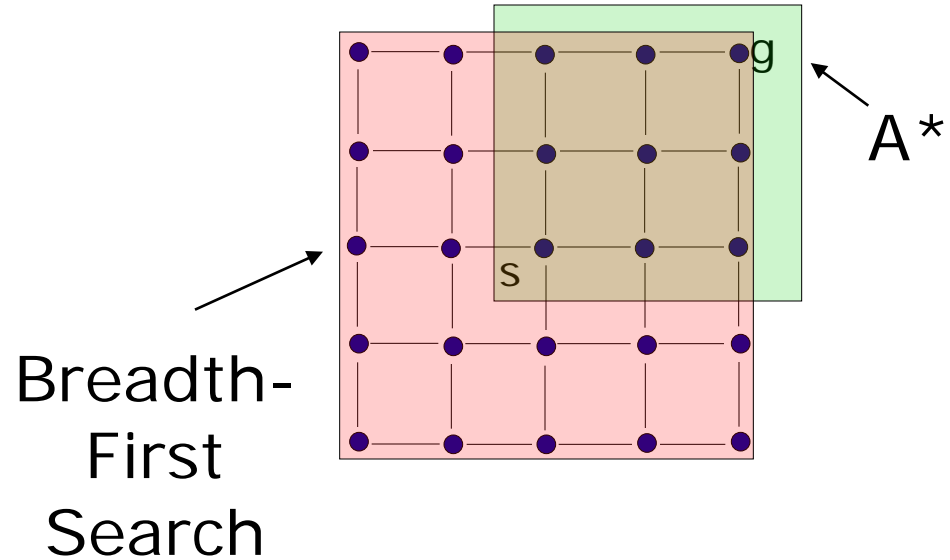
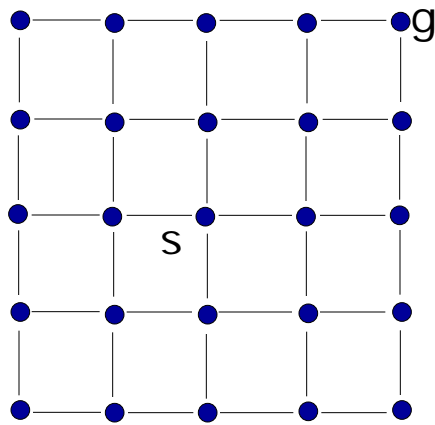
Exploiting Problem Graph Structure

Advanced Topics: Probability, Semi-Externality
Flash-Memory, etc.

Notations

- ▶ **G**: Graph
- ▶ **V**: Set of nodes of G
- ▶ **E**: Set of edges of G
- ▶ **w**: $E \rightarrow \mathbb{R}$: Weight function that assigns a cost to each edge.
- ▶ **δ** : shortest path distance between two nodes.
- ▶ **Open** list: Search frontier – waiting to be expanded.
- ▶ **Closed** list: Expanded nodes.

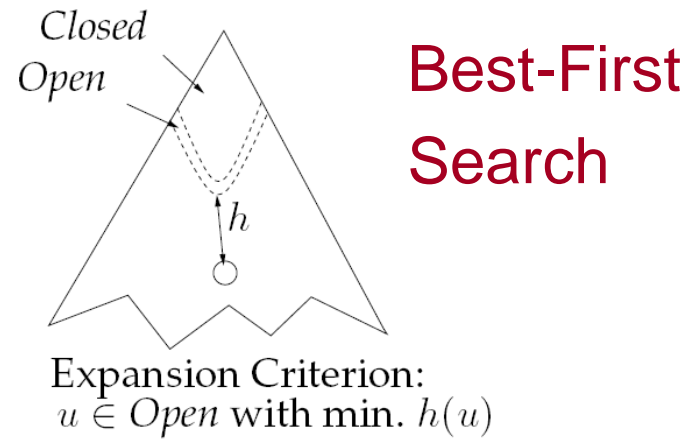
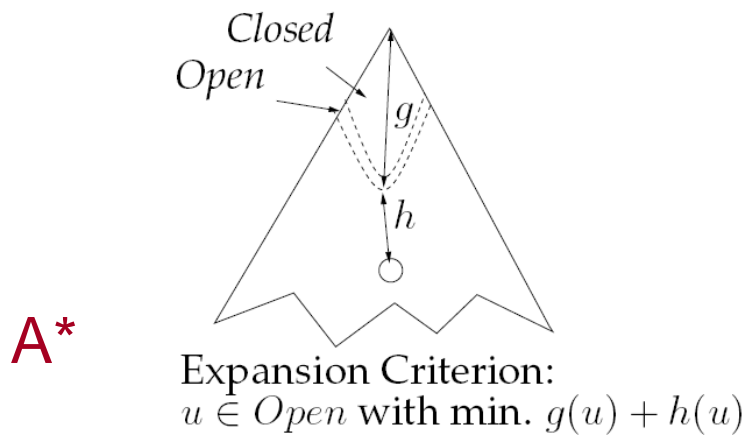
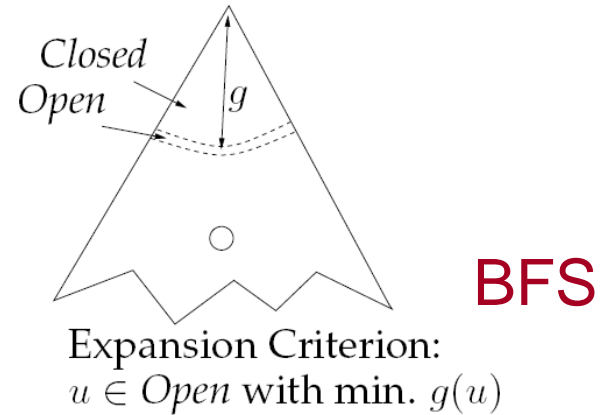
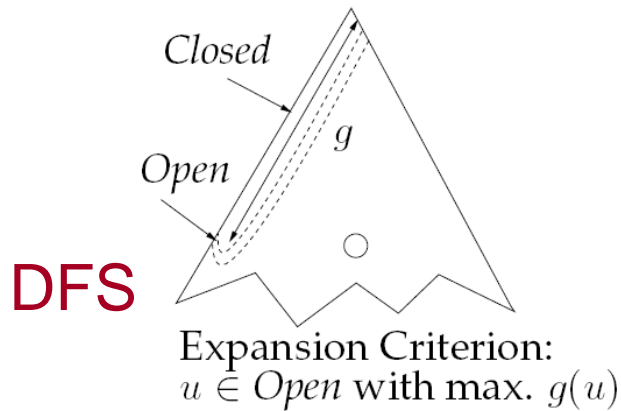
Heuristic Search – A* algorithm



A heuristic estimate is used to guide the search.

- E.g. **Straight line distance** from the current node to the goal in case of a graph with a geometric layout.

Comparison of Search Algorithms



Heuristics

Admissible Heuristics

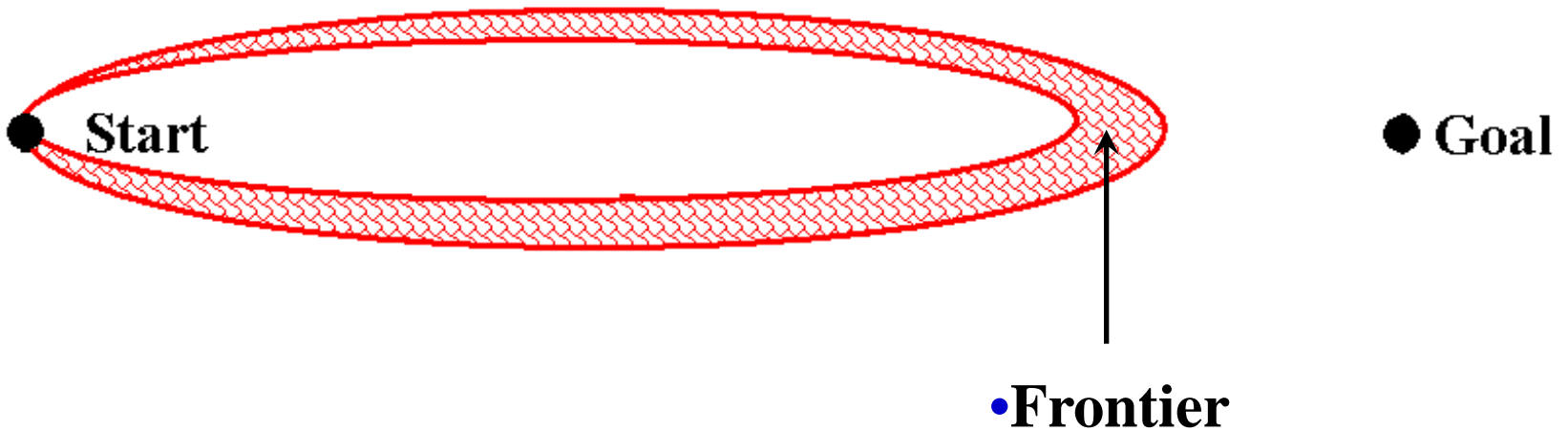
- ▶ Never over-estimates the optimal path.
 - Guarantees the optimal path

Consistent Heuristics

- ▶ Never drops faster than the edge weight.
 - Guarantees the minimum number of expanded nodes.

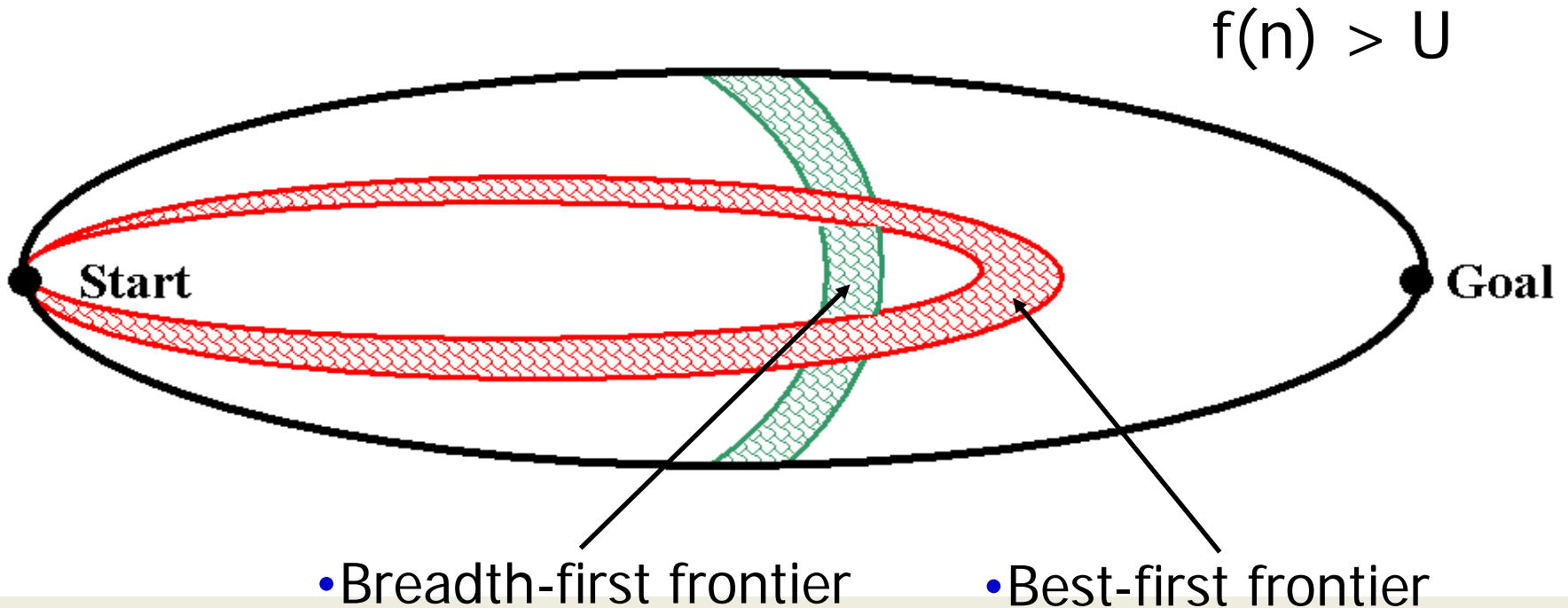
Divide-and-conquer frontier A^* [Korf & Zhang AAAI-00]

- ▶ Stores Open list, but not Closed list
- ▶ Reconstructs solution using divide-and-conquer method



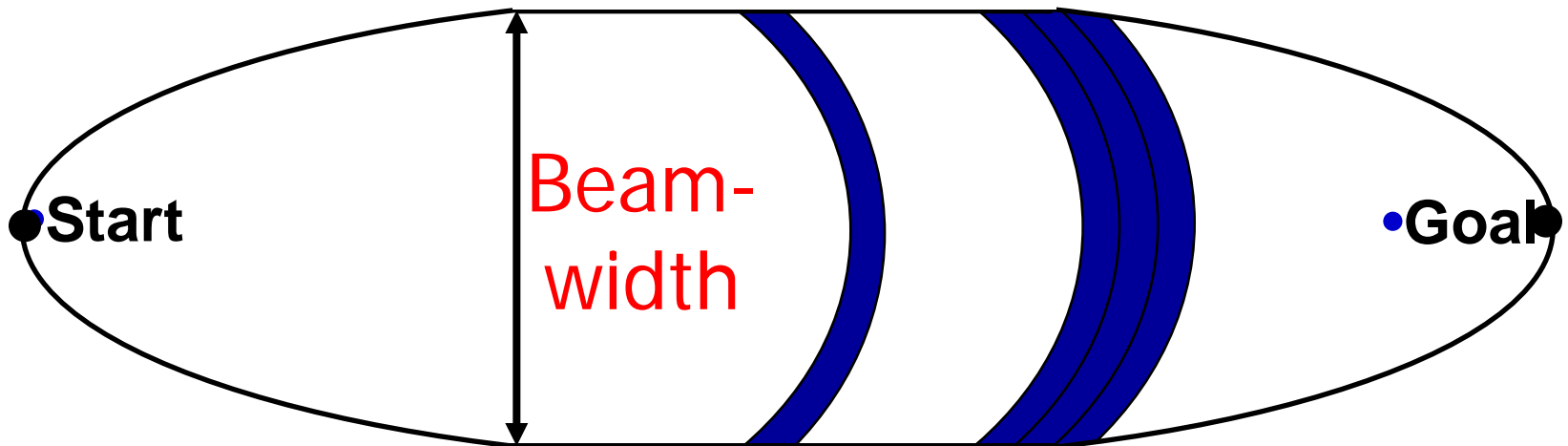
Breadth-first heuristic search

Breadth-first branch-and-bound is more memory-efficient than best-first search



Divide-and-conquer beam search

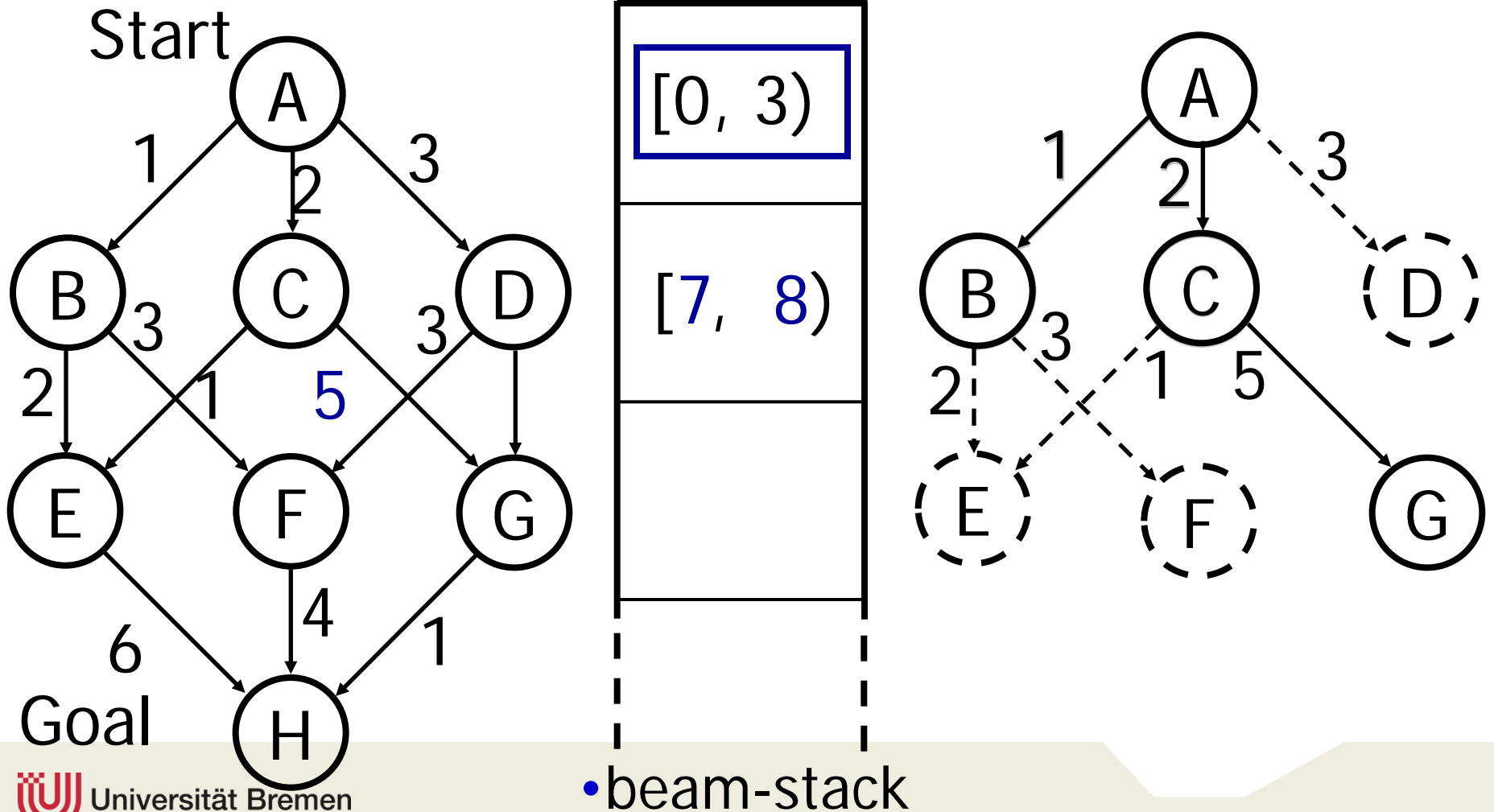
- ▶ Stores 3 layers for duplicate elimination and 1 “middle” layer for solution reconstruction
- ▶ Uses **beam width** to limit size of a layer



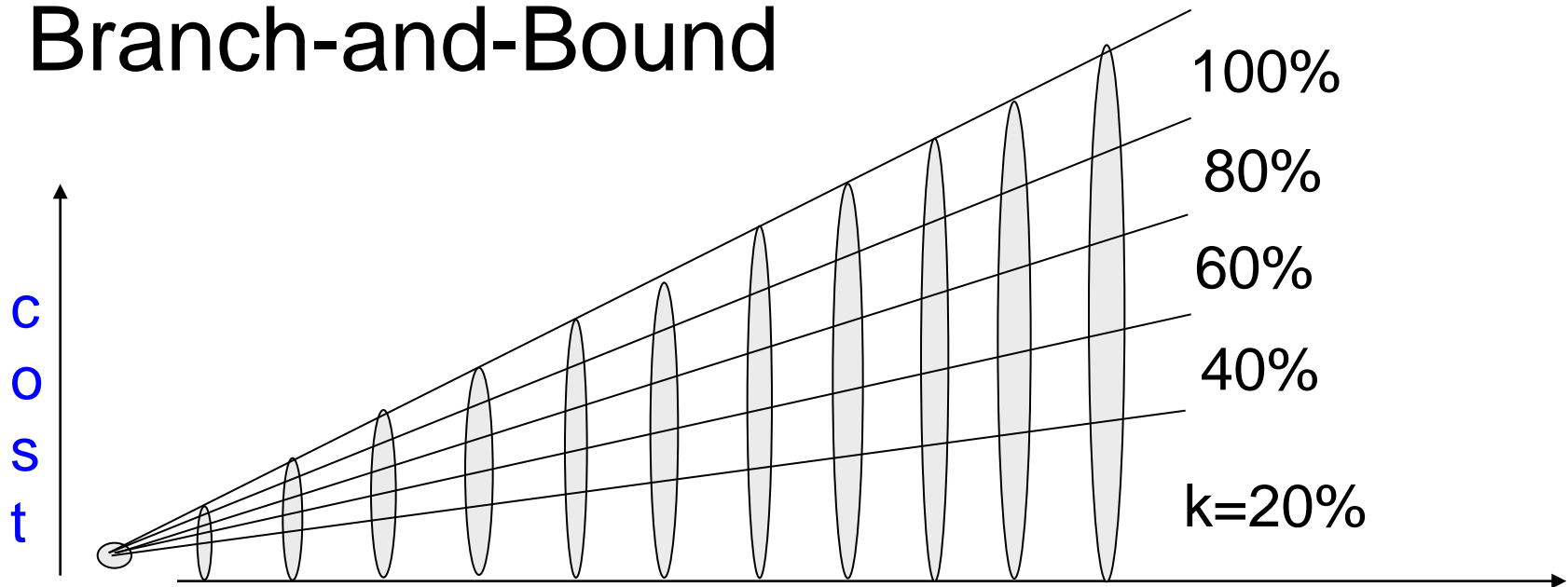
Divide-and-conquer beam-stack search

- ▶ Memory use bounded by $4 \times$ beam width
- ▶ Allows much wider beam width, which reduces backtracking
- ▶ If backtrack to removed layers, use beam stack to recover them

Example: width = 2, U = 8



Iterative Broadening Breadth-First Branch-and-Bound

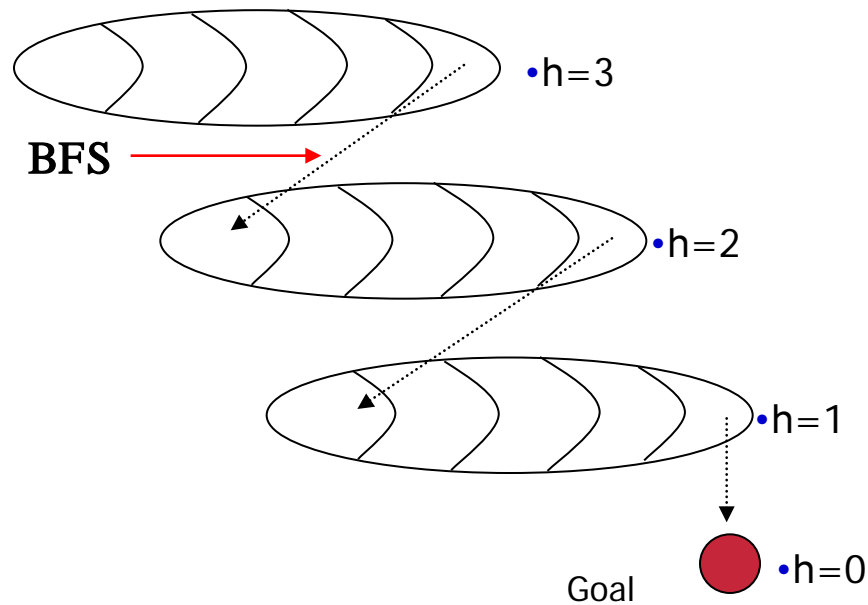


Search
frontier

Only pick best k% nodes for expansion.

Enforced Hill-Climbing

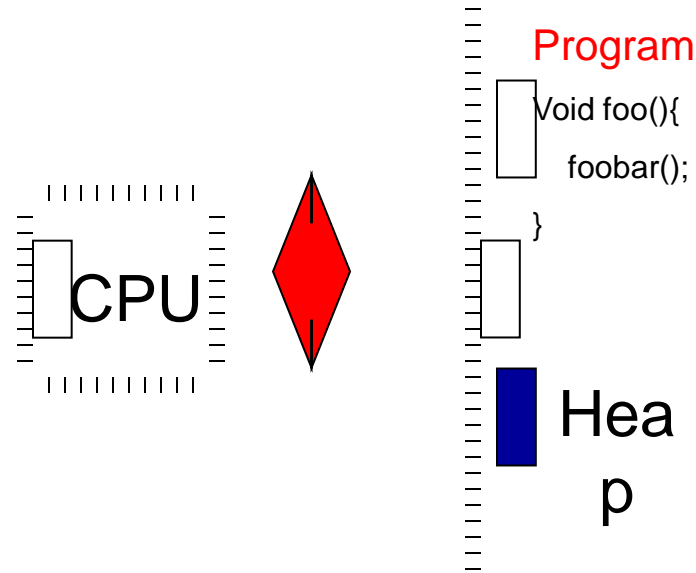
Most successful planning algorithm



Introduction to EM Algorithms

- ▶ Von Neumann RAM Model
- ▶ Virtual Memory
- ▶ External-Memory Model
- ▶ Basic I/O complexity analysis
 - External Scanning
 - External Sorting
- ▶ Breadth-First Search
- ▶ Graphs
 - Explicit Graphs
 - Implicit Graphs

Von Neumann RAM Model?



▶ Main assumptions:

- Program and heap fit into the main memory.
- CPU has a fast constant-time access to the memory contents.

Virtual Memory Management Scheme

- ▶ Address space is divided into **memory pages**.
- ▶ A **large** virtual address space is mapped to a smaller physical address space.
- ▶ If a **required address** is not in the main memory, a **page-fault** is triggered.
 - A memory page is **moved back** from RAM to the hard disk to make space,
 - The required page is **loaded** from hard disk to RAM.

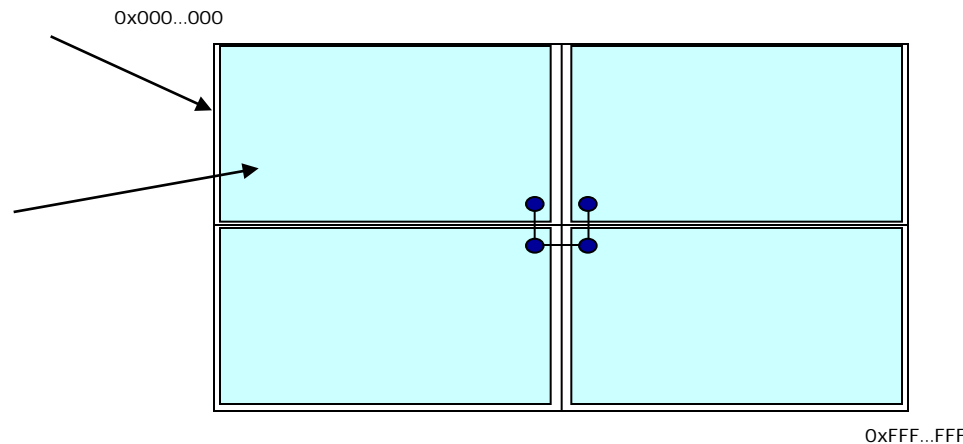
Virtual Memory

- + works well when word processing, spreadsheet, etc. are used.
- does not know any thing about the data accesses in an algorithm.

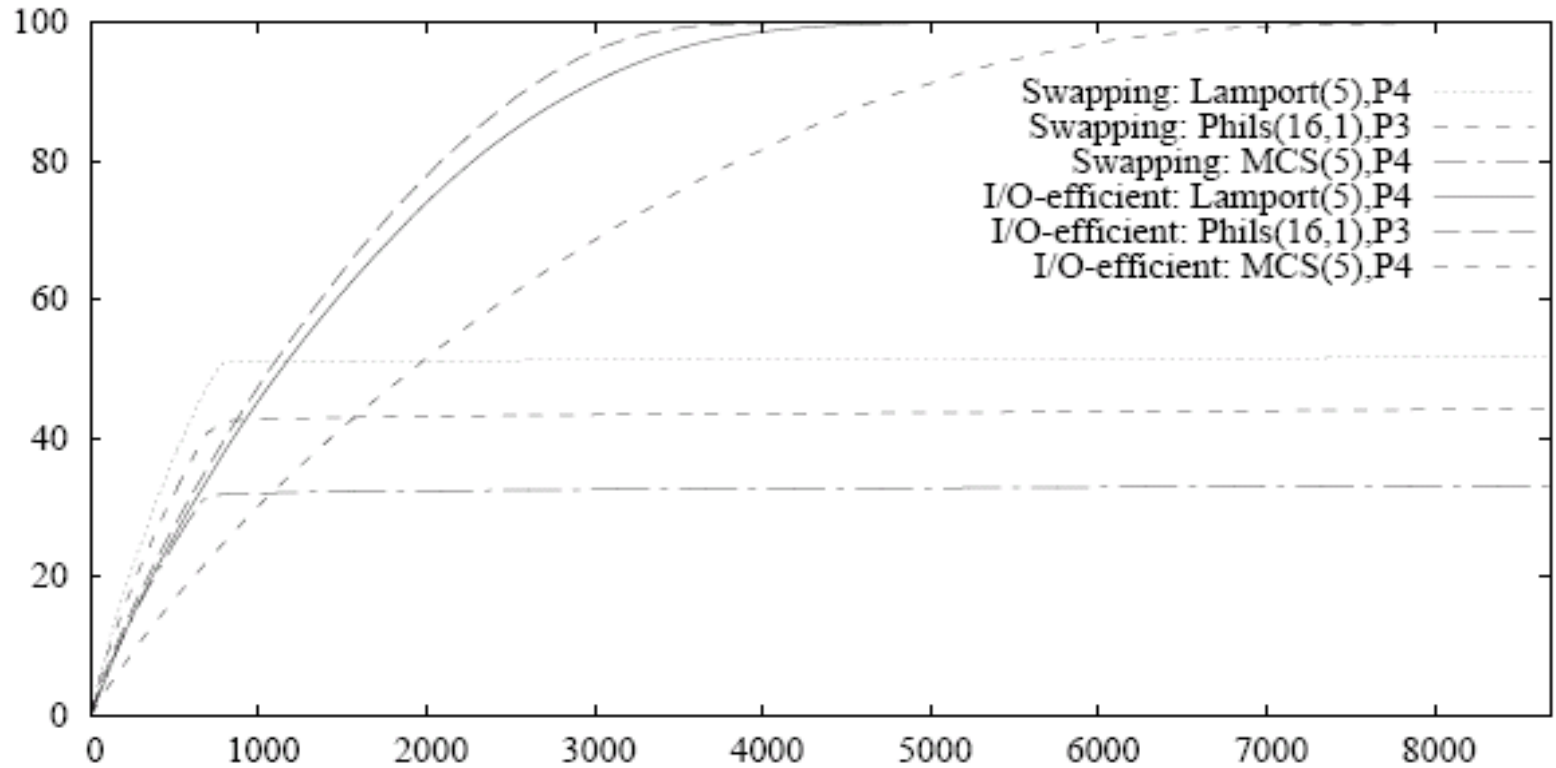
In the worst-case, can result in **one page fault** for every **state access!**

Virtual
Address
Space

Memory
Page



7 I/Os



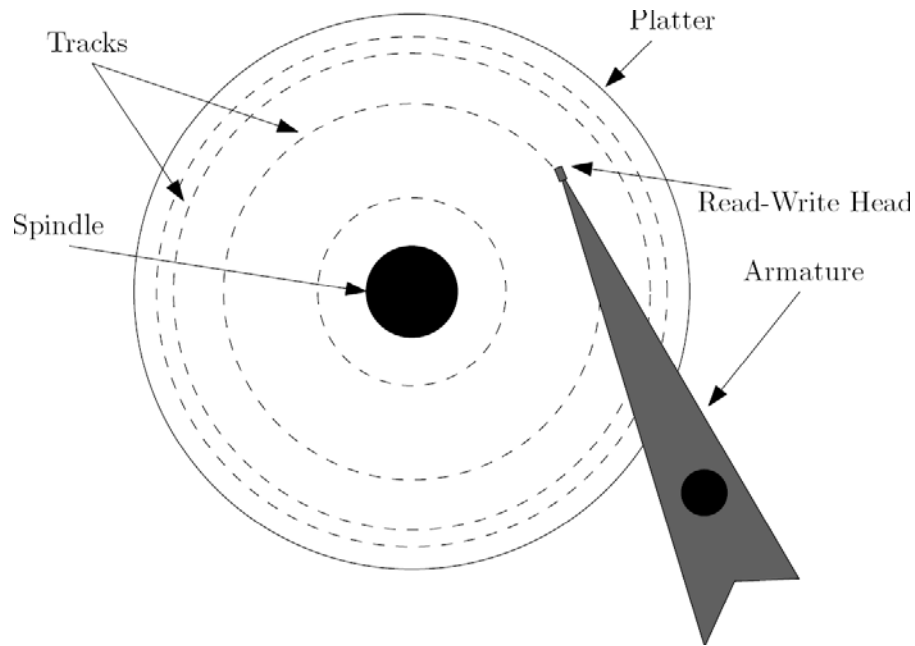
Memory Hierarchy

Latency times

Typical capacity

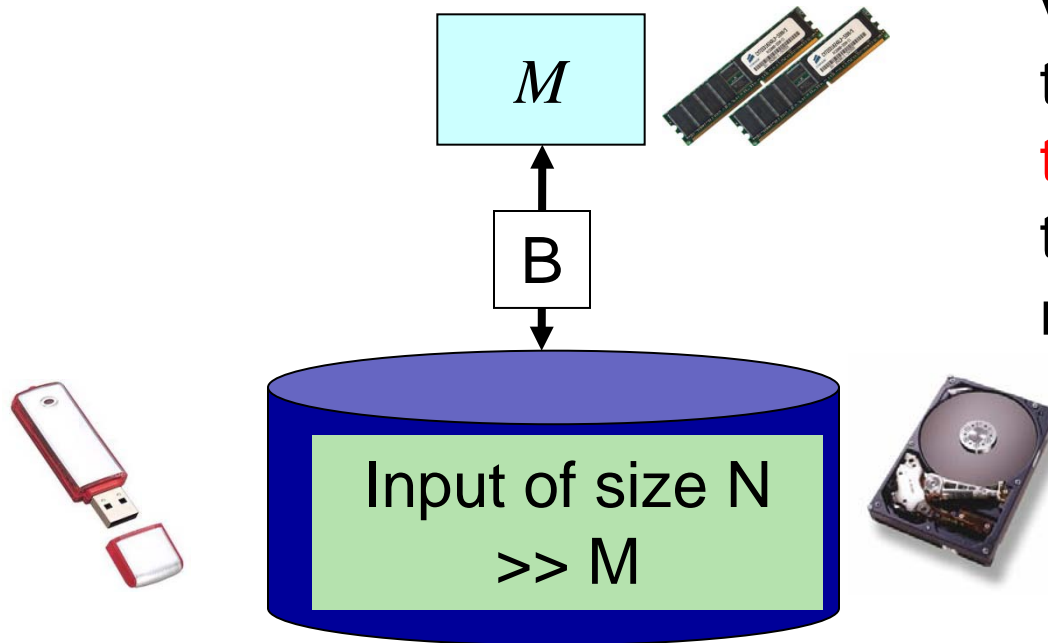
~2 ns	Registers (x86_64)	16 x 64 bits
3.0 ns	L1 Cache	64 KB
17 ns	L2 Cache	512 KB
23 ns	L3 Cache	2 – 4 MB
86 ns	RAM	4 GB
4.2 ms	Hard disk (7200 rpm)	600 GB

Working of a hard disk



- ▶ Data is written on tracks in form of sectors.
- ▶ While reading, armature is moved to the desired track.
- ▶ Platter is rotated to bring the sector directly under the head.
- ▶ A large block of data is read in one rotation.

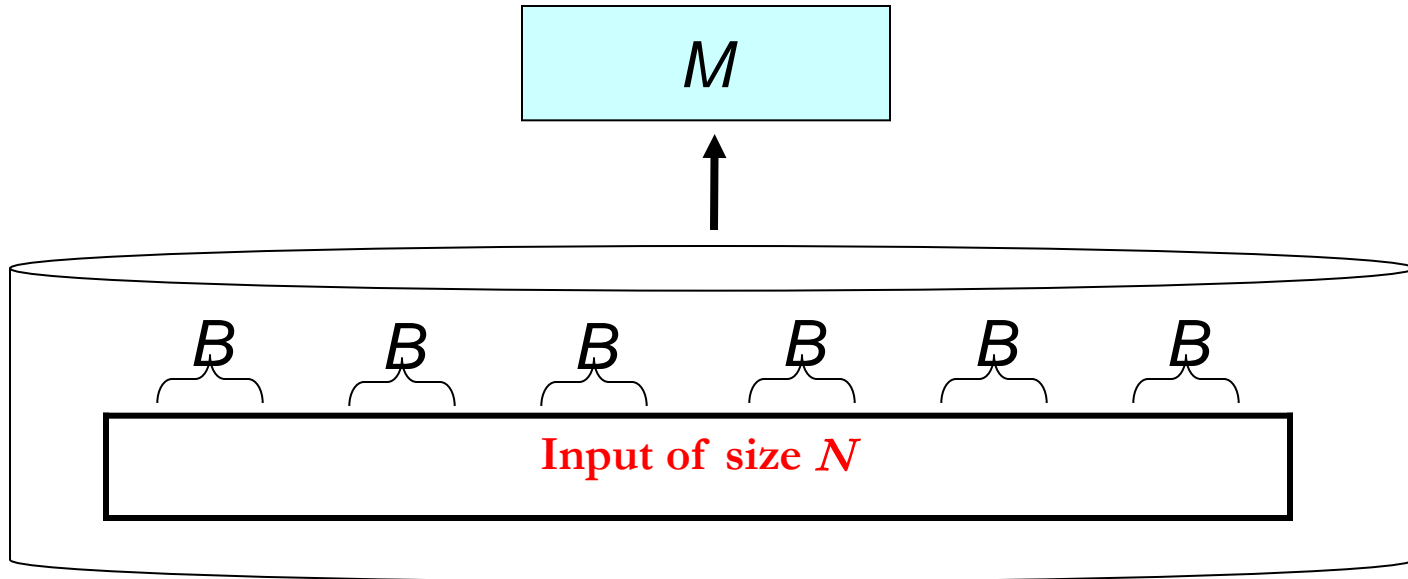
External Memory Model [Aggarwal and Vitter]



If the input size is very large, running time **depends on the I/Os** rather than on the number of instructions.

External Scanning

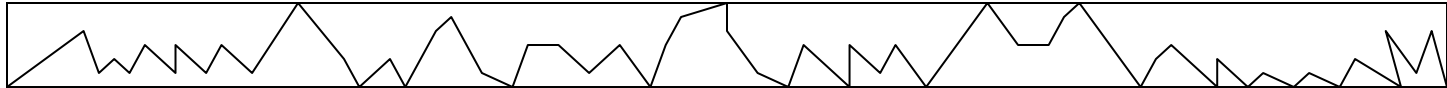
- ▶ Given an input of size N , consecutively read B elements in the RAM.



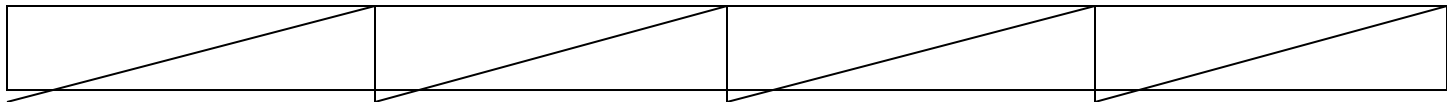
I/O complexity: $scan(N) = \frac{N}{B}$

External Sorting

Unsorted disk file of size N



Read M elements in chunks of B, sort internally and flush



Read M/B sorted buffers and flush a merged and sorted sequence

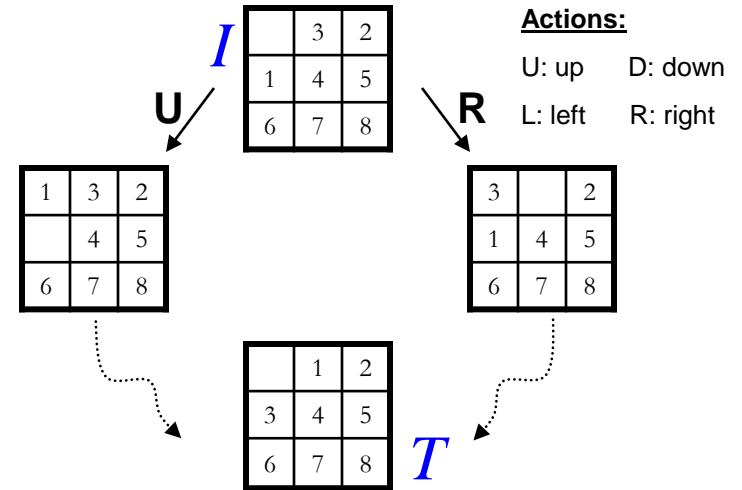
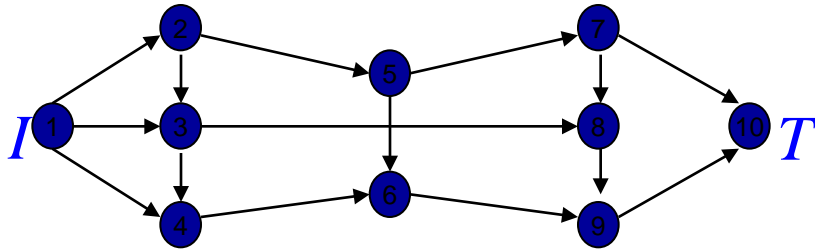


Read final M/B sorted buffers and flush a merged and sorted sequence



► **I/O complexity:**
$$sort(N) = O\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}\right)$$

Explicit vs. Implicit



Path search in **explicit graphs**:

Given a graph, does a path between two nodes I and T exist?

Path search in **implicit graphs**:

Given an initial state(s) I and a set of transformation rules, is a desired state T reachable?

Traverse/Generate the **graph until T is reached**.

Search Algorithms: *DFS, BFS, Dijkstra, A*, etc.*)

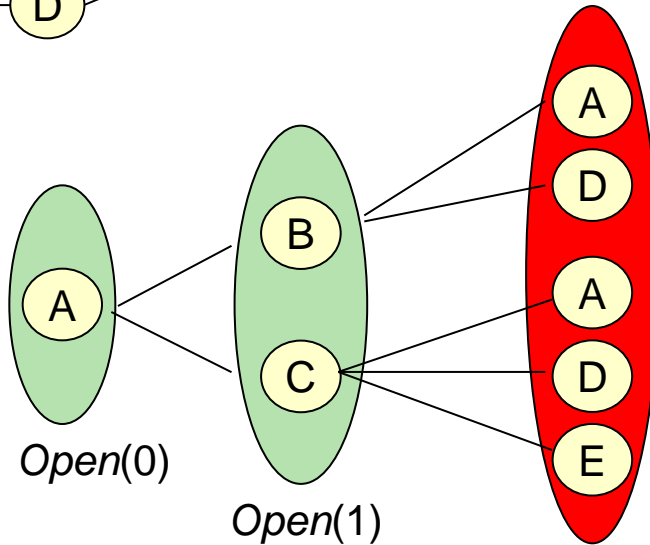
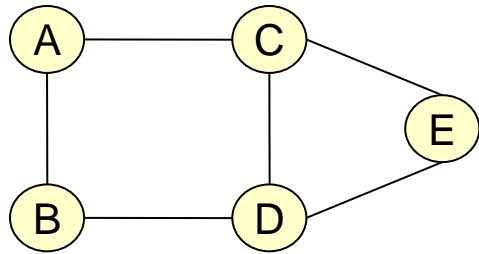
What if the graph is too big to fit in the RAM?

8-puzzle has $9!/2$ states ... 15-puzzle has $16!/2 \approx 10\ 461\ 394\ 900\ 000$ states

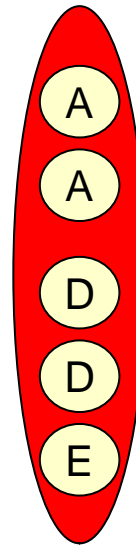
External-Memory Graph Search

- ▶ External BFS
- ▶ Delayed Duplicate Detection
- ▶ Locality
- ▶ External A*
 - Bucket Data Structure
 - I/O Complexity Analysis

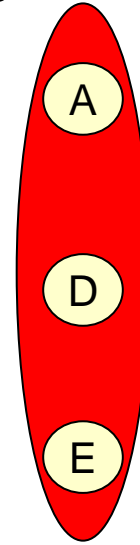
External Breadth-First Search



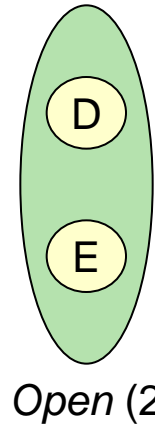
External Sort Open (2)



Compact Open (2)



Remove Duplicates w.r.t 2 previous layers



I/O Complexity Analysis of EM-BFS for Explicit Graphs

▶ Expansion:

- Sorting the adjacency lists: $O(\text{Sort}(|V|))$
- Reading the adjacency list of all the nodes: $O(|V|)$

▶ Duplicates Removal:

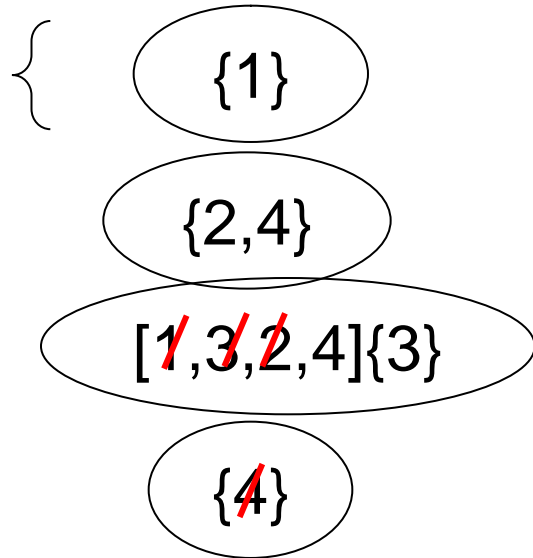
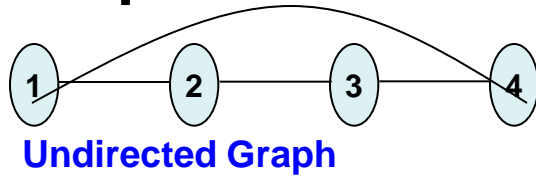
- Phase I: External sorting followed by scanning. $O(\text{Sort}(|E|) + \text{Scan}(|E|))$
- Phase II: Subtraction of previous two layers: $O(\text{Scan}(|E|) + \text{Scan}(|V|))$

▶ Total: $O(|V| + \text{Sort}(|E| + |V|))$ I/Os

Delayed Duplicate Detection (Korf 2003)

- ▶ Essentially idea of Munagala and Ranade applied to implicit graphs ...
- ▶ **Complexity:**
 - Phase I: External sorting followed by scanning. $O(\text{Sort}(|E|) + \text{Scan}(|E|))$
 - Phase II: Subtraction of previous two layers: $O(\text{Scan}(|E|) + \text{Scan}(|V|))$
- ▶ **Total: $O(\text{Sort}(|E|) + \text{Scan}(|V|))$ I/Os**

Duplicate Detection Scope



BFS Layer

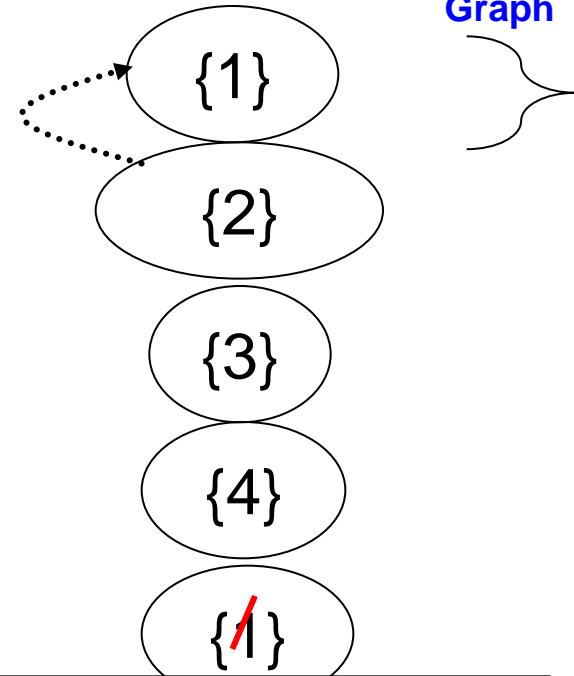
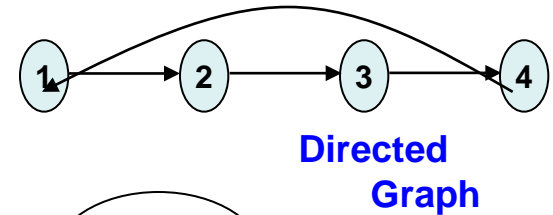
Layer-0

Layer-1

Layer-2

Layer-4

Layer-5



Longest Back-
edge:

$$locality = \max_{\forall u, v \in S | v \in Succ(u)} \{ \delta(I, u) - \delta(I, v) \} + 1$$

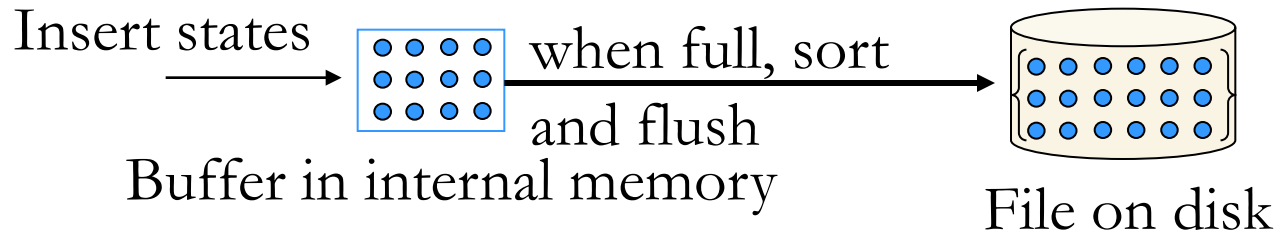
[Zhou & Hansen 05]

Problems with A* Algorithm

- ▶ A* **needs to store** all the states during exploration.
- ▶ A* generates **large amount of duplicates** that can be removed using an internal hash table – **only if it can fit in the main memory.**
- ▶ A* **do not exhibit any locality of expansion.** For large state spaces, standard virtual memory management can
 - **Can we follow the strict order of expanding with respect to the minimum $g+h$ value? - Without compromising the optimality?**

Data Structure: Bucket

- ▶ A Bucket is a set of states, residing on the disk, **having the same (g, h) value**, where:
 - $g =$ **number of transitions** needed to transform the initial state to the states of the bucket,
 - and $h =$ **Estimated distance** of the bucket's state to the goal
- ▶ No state is inserted again in a bucket that is expanded.
- ▶ If **Active** (being **read** or **written**), represented internally by a small buffer.



External A*

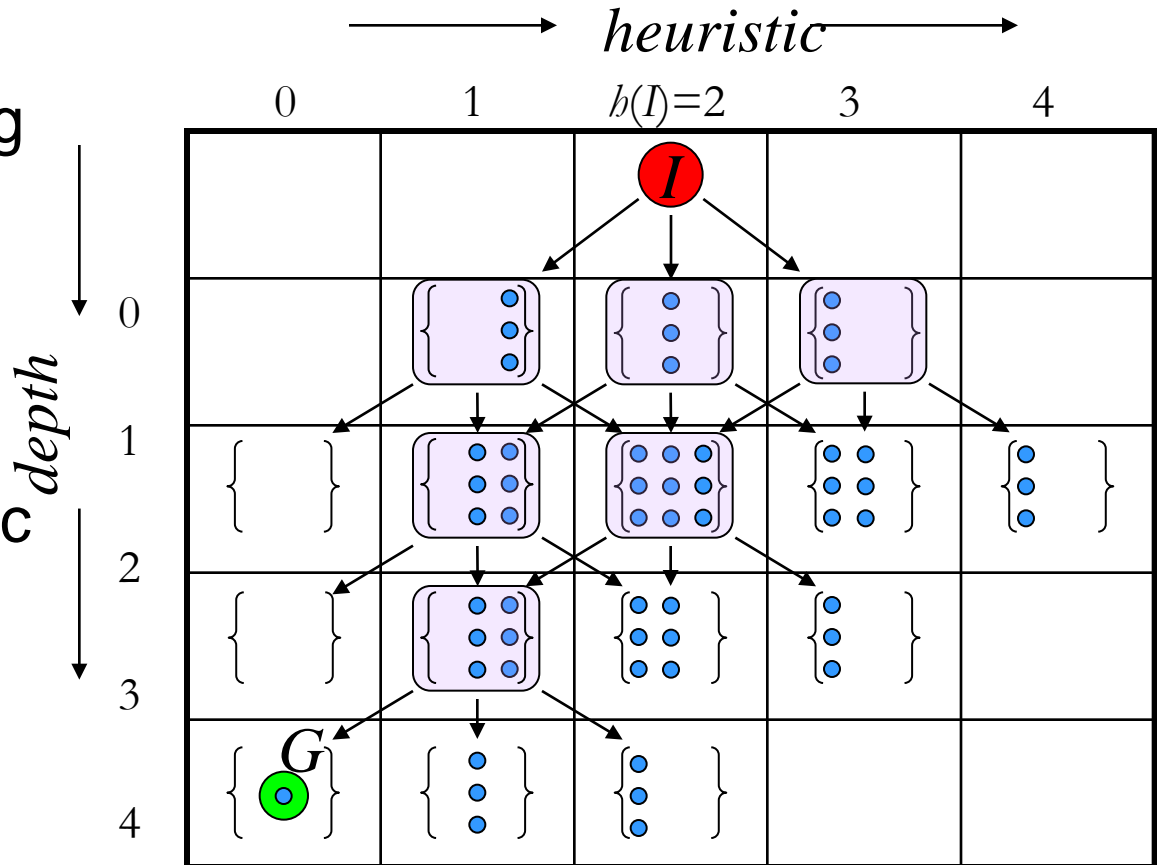
Simulates a priority queue by exploiting the properties of the heuristic function:

- ▶ h is a total function!!
- ▶ Consistent heuristic estimates.

$$\Rightarrow \Delta h = \{-1, 0, 1, \dots\}$$

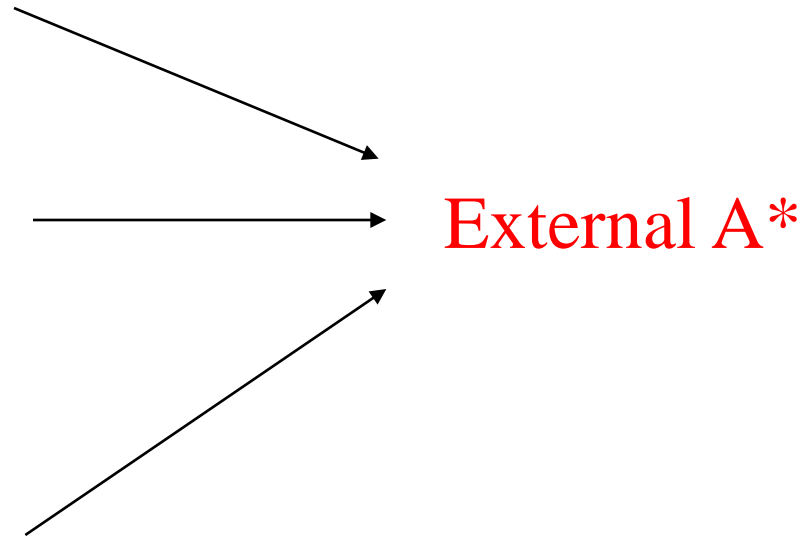
$$w'(u, v) = w(u, v) - h(u) + h(v)$$

$$\Rightarrow w'(u, v) = 1 + \{-1, 0, 1\}$$



External A*

- ▶ Buckets represent *temporal locality* – cache efficient order of expansion.
- ▶ If we store the states in the same bucket together we can exploit the *spatial locality*.
- ▶ Munagala and Ranade's BFS and Korf's delayed duplicate detection for implicit graphs.



Procedure *External A**

$Bucket(0, h(I)) \leftarrow \{I\}$

$f_{min} \leftarrow h(I)$

while ($f_{min} \neq \infty$)

$g \leftarrow \min\{i \mid Bucket(i, f_{min} - i) \neq \phi\}$

while ($g_{min} \leq f_{min}$)

$h \leftarrow f_{min} - g$

$Bucket(g, h) \leftarrow$ *remove duplicates from* $Bucket(g, h)$

$Bucket(g, h) \leftarrow Bucket(g, h) \setminus$

$(Bucket(g - 1, h) \cup Bucket(g - 2, h)) //$

Subtraction

$A(f_{min}), A(f_{min} + 1), A(f_{min} + 2) \leftarrow N(Bucket(g, h)) //$ *Generate Neighbours*

$Bucket(g + 1, h + 1) \leftarrow A(f_{min} + 2)$

$Bucket(g + 1, h) \leftarrow A(f_{min} + 1) \cup Bucket(g + 1, h)$

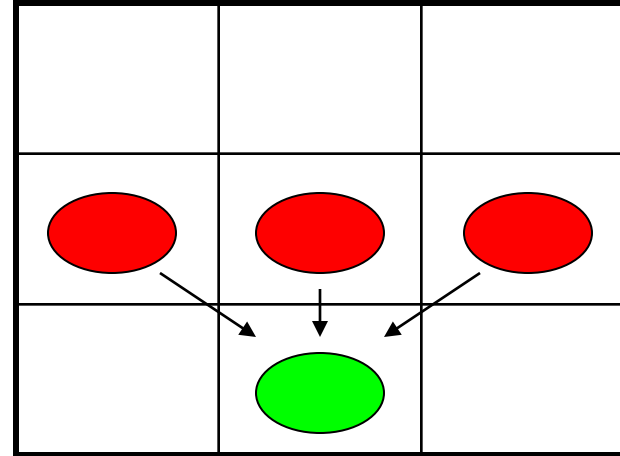
$Bucket(g + 1, h - 1) \leftarrow A(f_{min}) \cup Bucket(g + 1, h - 1)$

$g \leftarrow g + 1$

$f_{min} \leftarrow \min\{i + j > f_{min} \mid Bucket(i, j) \neq \phi\} \cup \{\infty\}$

I/O Complexity Analysis

- ▶ Internal A^* \Rightarrow Each edge is looked at most once.
- ▶ Duplicates Removal:
 - Sorting the **green** bucket having **one state for every edge** from the 3 red buckets.
 - Scanning and compaction.
 - $O(\text{sort}(|E|))$



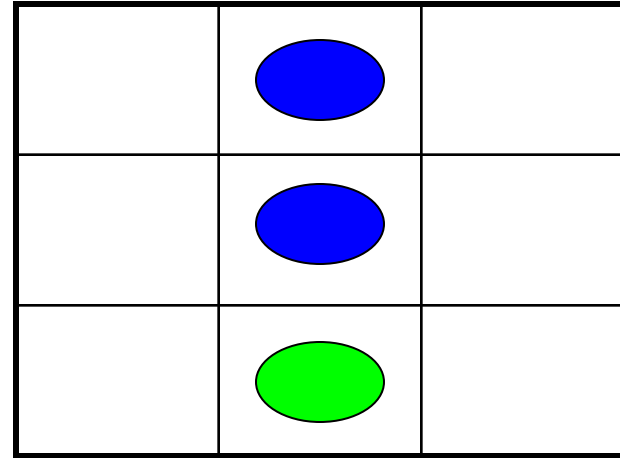
Total I/O complexity:

$$\theta(\text{sort}(|E|) + \text{scan}(|V|)) \text{ I/Os}$$

Cache-Efficient at all levels!!!

Complexity Analysis

- ▶ Subtraction:
 - Removing states of blue buckets (duplicates free) from the green one.
 - $O(\text{scan}(|V|) + \text{scan}(|E|))$



Total I/O complexity:

$$\theta(\text{sort}(|E|) + \text{scan}(|V|)) \text{ I/Os}$$

Cache-Efficient at all levels!!!

I/O Performance of External A*

Theorem: The complexity of **External A*** in an implicit **unweighted** and **undirected** graph with a **consistent** heuristic estimate is bounded by

$$O(\text{sort}(|E|) + \text{scan}(|V|)) \text{ I/Os.}$$

Test Run – Generated states

<i>g/h</i>	1	2	3	4	5	6	7	8	9	10	11
0	-	-	-	1+0	-	-	-	-	-	-	-
1	-	-	-	-	2+0	-	-	-	-	-	-
2	-	-	-	0+4	-	2+0	-	-	-	-	-
3	-	-	-	-	7+3	-	4+0	-	-	-	-
4	-	-	-	0+7	-	13+4	-	10+0	-	-	-
5	-	-	-	-	5+15	-	24+10	-	24+0	-	-
6	-	-	-	0+6	-	12+26	-	46+28	-	44+0	-
7	-	-	-	-	9+10	-	20+51	-	99+57	-	76+0
8	-	-	-	0+8	-	15+25	-	48+137	-	195+0	-
9	-	-	-	-	4+17	-	45+52	-	203+0	-	-
10	-	-	-	0+3	-	13+49	-	92+0	-	-	-
11	-	-	-	-	2+19	-	46+0	-	-	-	-
12	-	-	-	0+5	-	31+0	-	-	-	-	-
13	-	-	0+2	-	10+0	-	-	-	-	-	-
14	-	0+2	-	5+0	-	-	-	-	-	-	-
15	0+2	-	5+0	-	-	-	-	-	-	-	-

Test	g/h	0	1	2	3	4	5	6	7	8	9	10	11
	0	-	-	-	-	-	-	-	-	-	-	-	-
	1	-	-	-	-	-	-	-	-	-	-	-	-
	2	-	-	-	-	1+1	-	-	-	-	-	-	-
	3	-	-	-	-	-	2+2	-	-	-	-	-	-
	4	-	-	-	-	3+2	-	3+2	-	-	-	-	-
	5	-	-	-	-	-	8+6	-	6+4	-	-	-	-
	6	-	-	-	-	1+2	-	16+12	-	14+10	-	-	-
	7	-	-	-	-	-	6+6	-	24+24	-	26+24	-	-
	8	-	-	-	-	3+10	-	10+10	-	52+50	-	-	-
	9	-	-	-	-	-	9+7	-	29+23	-	-	-	-
	10	-	-	-	-	0+2	-	21+20	-	-	-	-	-
	11	-	-	-	-	-	6+5	-	-	-	-	-	-
	12	-	-	-	-	0+1	-	-	-	-	-	-	-
	13	-	-	-	-	-	-	-	-	-	-	-	-
	14	-	-	-	-	-	-	-	-	-	-	-	-
	15	-	1+0	-	-	-	-	-	-	-	-	-	-

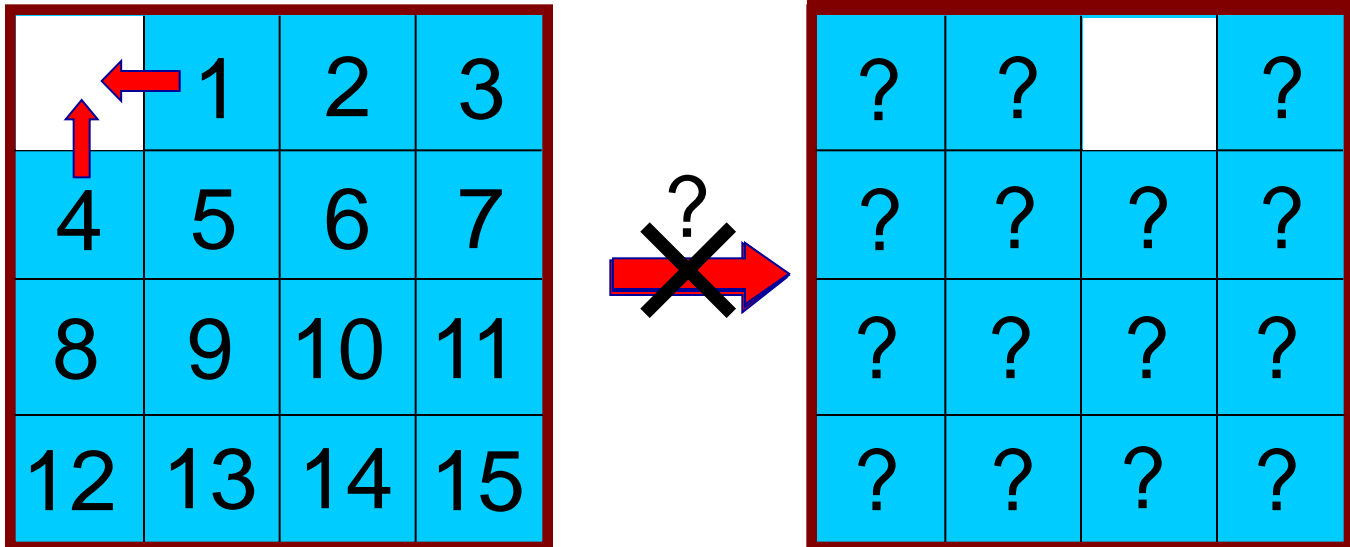
Exploiting Problem Graph Structure

Structured Duplicate Detection

- ▶ Basic Principles
- ▶ Manual and automated Partitioning
- ▶ Edge Partitioning
- ▶ External Memory Pattern Databases

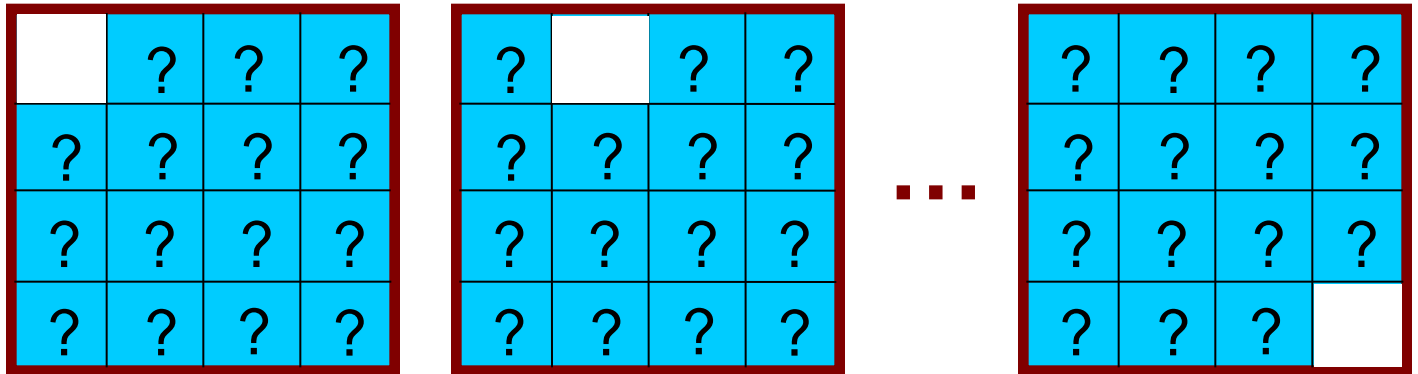
Structured duplicate detection

- ▶ Idea: localize memory references in duplicate detection by exploiting graph structure
- ▶ Example: Fifteen-puzzle



State-space projection function

- ▶ Many-to-one mapping from original state space to abstract state space
- ▶ Created by ignoring some state variables
- ▶ Example



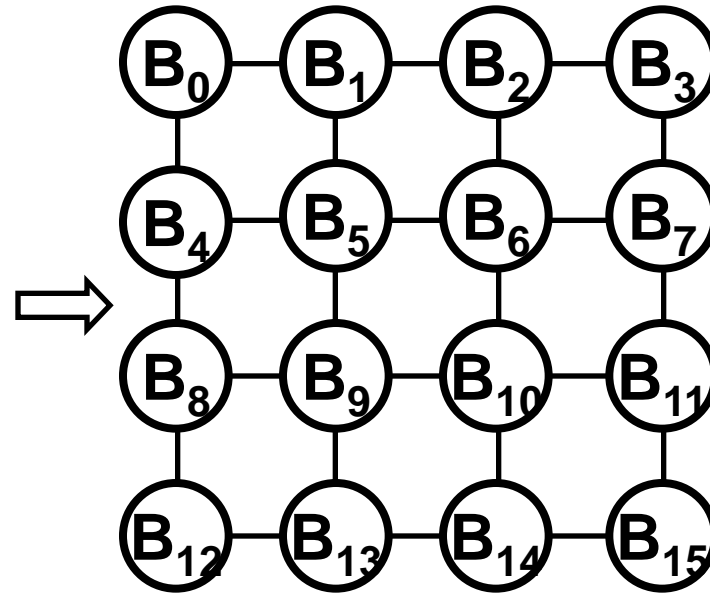
blank pos. = 0 1 ... 15

Abstract state-space graph

- Created by state-space projection function
- Example

	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

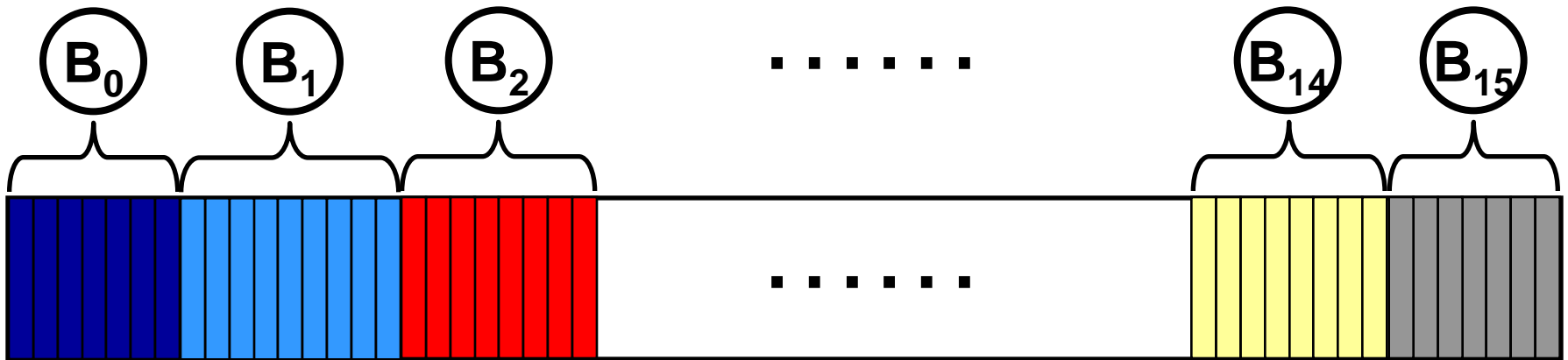
> 10 trillion states



16 abstract states

Partition stored nodes

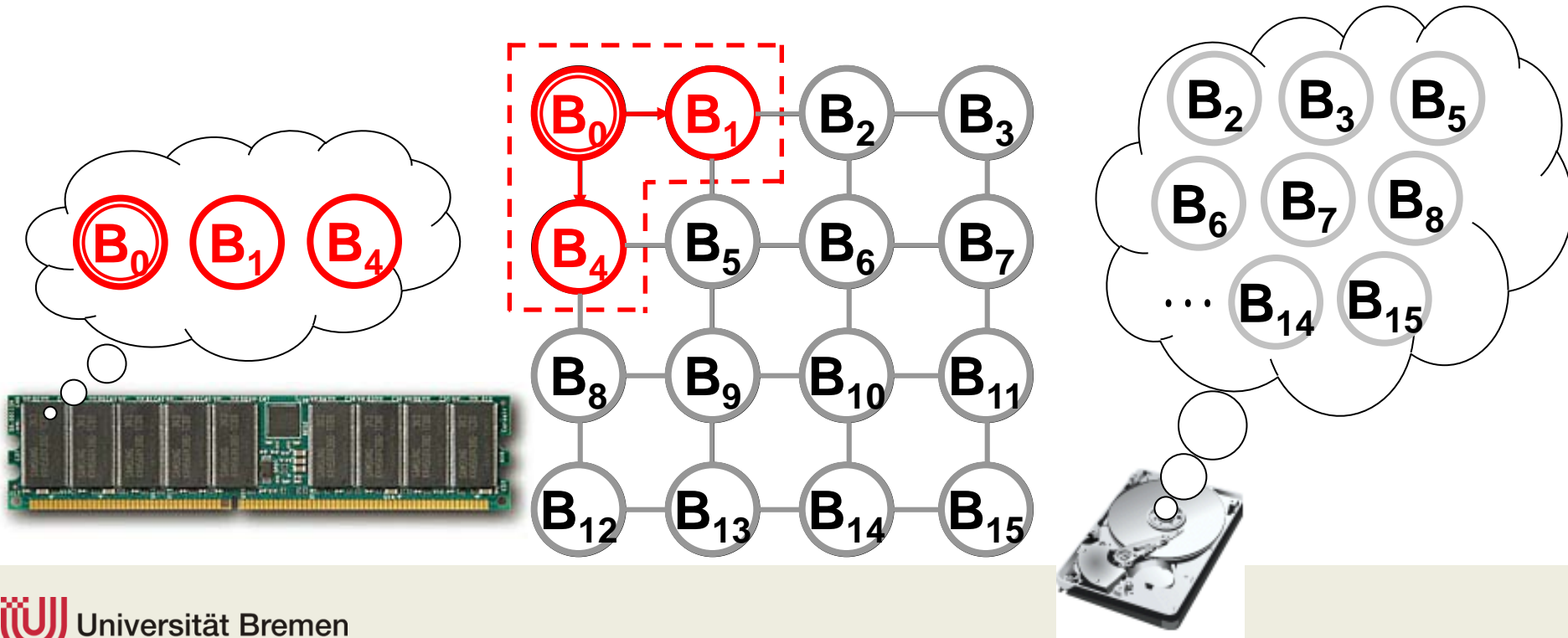
Open and Closed lists are partitioned into blocks of nodes, with one block for each abstract node in abstract state-space graph



Logical memory

Duplicate-detection scope

A set of blocks (of stored nodes) that is guaranteed to contain all stored successor nodes of the currently-expanding node



When is disk I/O needed?

- ▶ If internal memory is full, write blocks outside current duplicate-detection scope to disk
- ▶ If any blocks in current duplicate-detection scope are not in memory, read missing blocks from disk

How to minimize disk I/O?

Given a set of nodes on search frontier, expand nodes in an order such that

- Nodes in the same duplicate-detection scope are expanded together
- Nodes in duplicate-detection scopes with overlapping abstract nodes, are expanded near each other

Locality-preserving abstraction

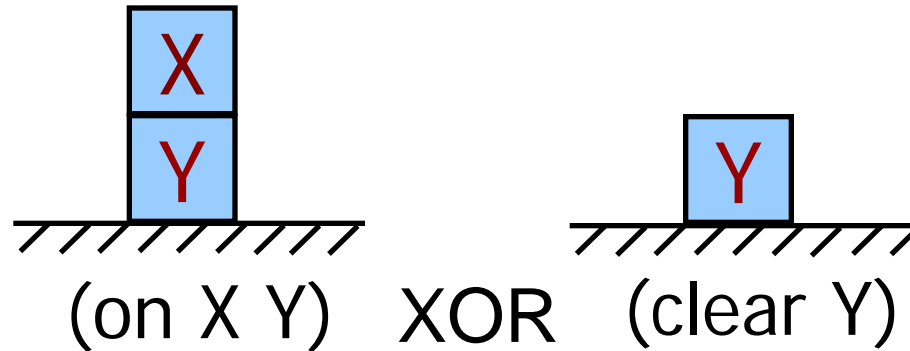
- ▶ Max. duplicate-detection scope ratio δ

$$\delta = \frac{\text{max \# of successors of an abstract state}}{\text{size of abstract graph}}$$

- Measures degree of graph local structure
 - \approx % of nodes that must be stored in RAM
 - Smaller $\delta \rightarrow$ Less RAM needed
- ▶ Search for abstraction that minimizes δ

Exploiting state constraints

XOR group: a group of atoms s.t. exactly one must be true at any time



Advantage: reduce size of abstract graph

Example: 2 XOR groups of 5 atoms each
size of abstract graph =

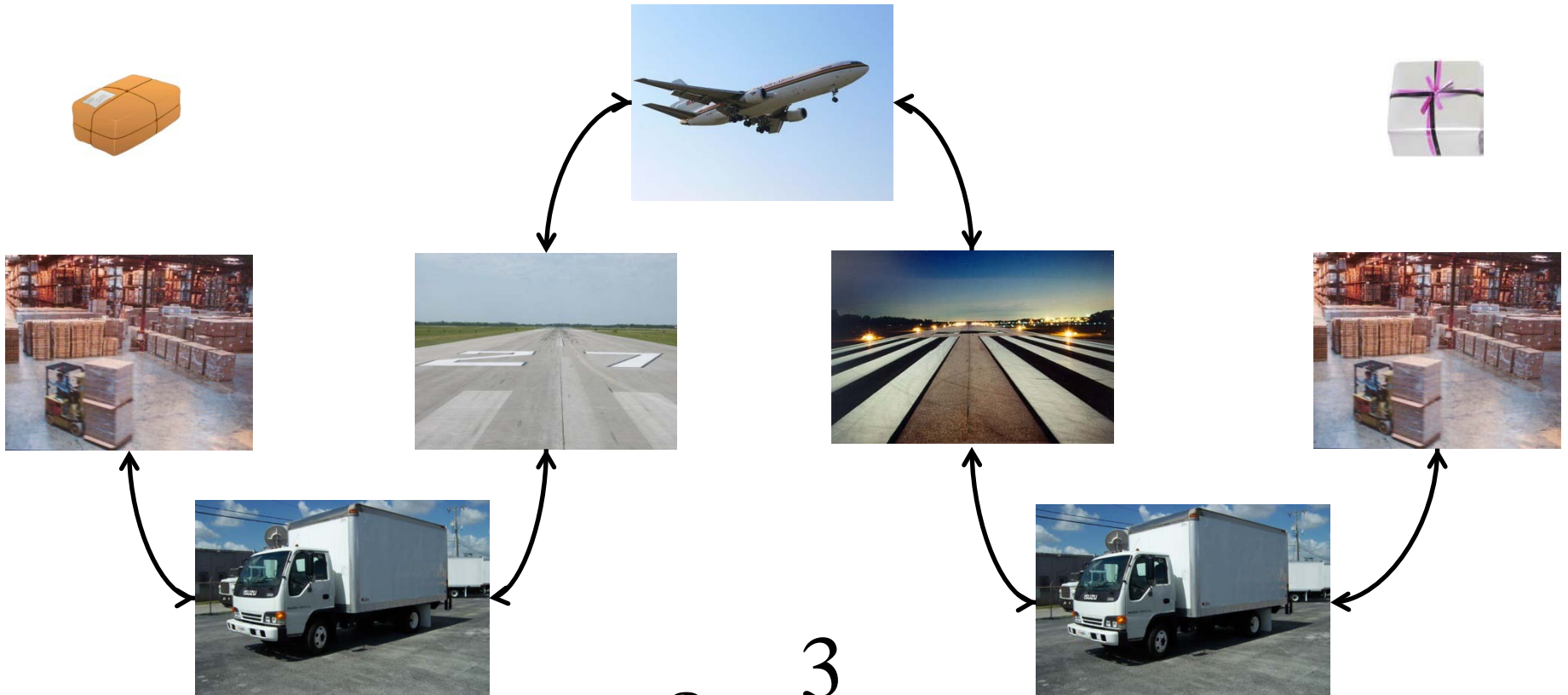
$$2^{5+5} = 1024 \quad \leftarrow \text{without XOR constraints}$$

$$5 \times 5 = 25 \quad \leftarrow \text{with XOR constraints}$$

Greedy abstraction algorithm

- ▶ Starts with empty set of abstraction atoms
- ▶ Mark all XOR groups as unselected
- ▶ While (size of abstract graph $\leq M$)
 - Find an unselected XOR group P_i s.t. union of abstraction atoms and P_i creates abstract graph with minimum δ
 - Add P_i into set of abstraction atoms
 - Mark P_i as selected

Example: Logistics



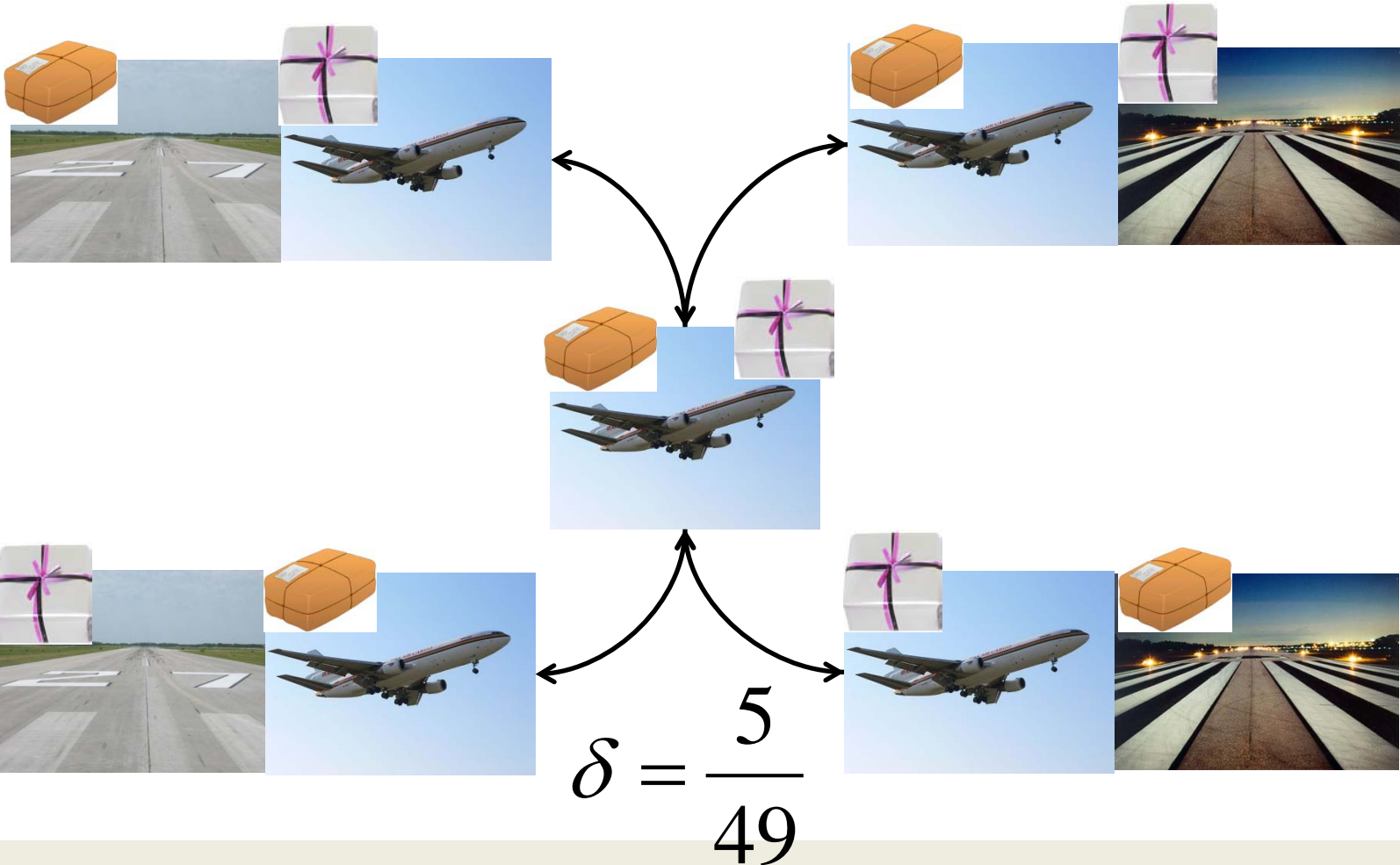
$$\delta = \frac{3}{7}$$

Abstraction based on truck locations



$$\delta = 1$$

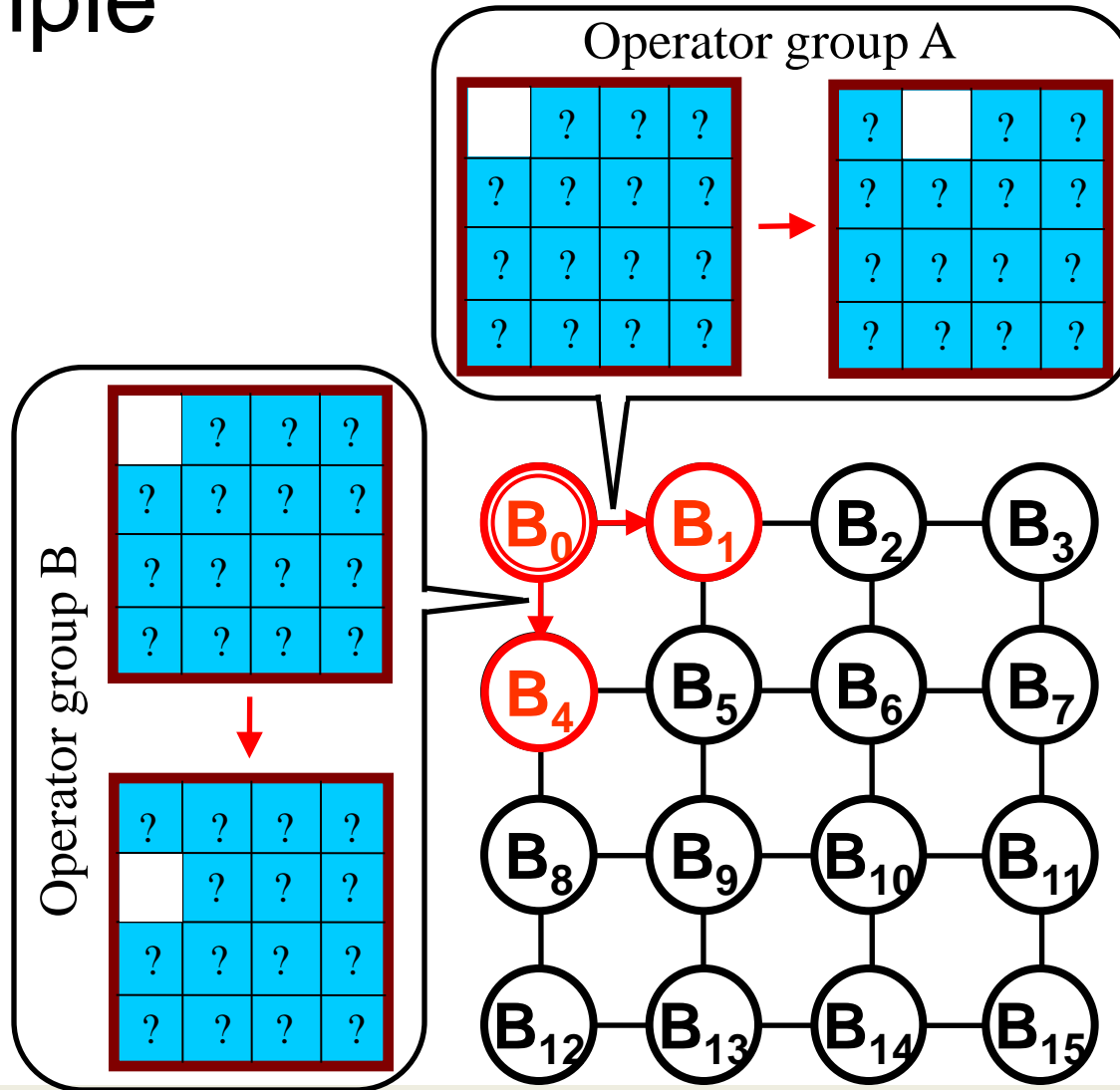
Largest duplicate-detection scope based on locations of 2 packages



Operator grouping

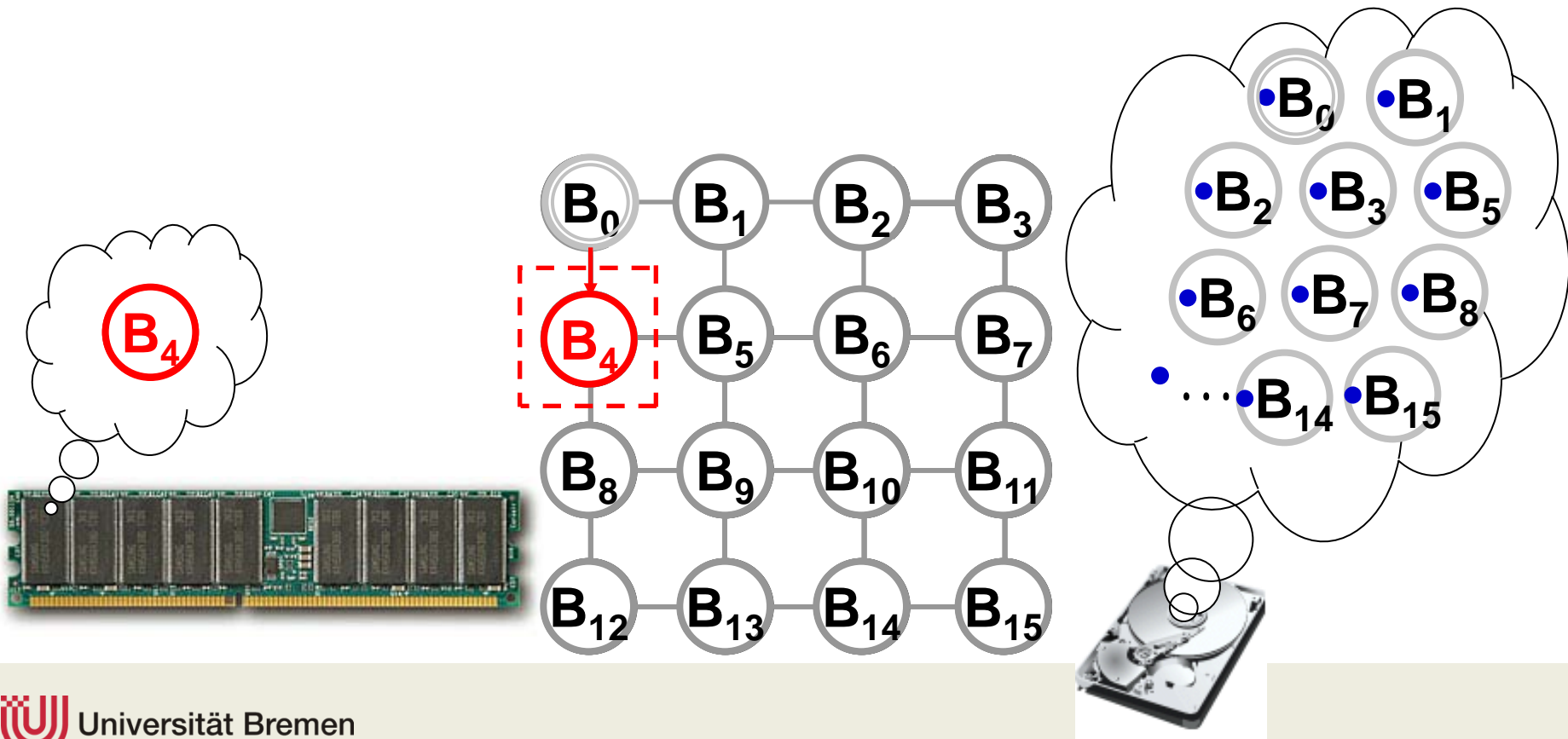
- ▶ Exploits structure in operator space
- ▶ Divides operators into operator groups for each abstract state
- ▶ Operators belong to the same group if they
 - are applicable to the same abstract state
 - lead to the same successor abstract state

Example



Edge Partitioning

Reduces duplicate-detection scope to **one** block of stored nodes – Guaranteed!



External-memory pattern database

- ▶ Creating an external-memory PDB
 - Breadth-first search in pattern space using delayed or structured duplicate detection
- ▶ Two ways of using an external-memory PDB
 - Compress PDB to fit in RAM
 - Use **structured** duplicate detection to localize references to PDB, so only a small fraction of PDB needs to be stored in RAM at a time

Compatible state-space abstraction

	?	2	?
4	?	?	?
?	?	?	?
?	?	?	?

Abstract state space

Abstract pattern space

Original state space

Pattern space

	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

	1	2	?
?	?	?	?
?	?	?	?
?	?	?	?

	?	2	?
?	?	?	?
?	?	?	?
?	?	?	?

Parallel External-Memory Graph Search

- ▶ Motivation Shared and Distributed Environments
- ▶ Parallel Delayed Duplicate Detection
 - Parallel Expansion
 - Distributed Sorting
- ▶ Parallel Structured Duplicate Detection
 - Finding Disjoint Duplicate Detection Scopes
 - Locking

Advanced Topics

- ▶ External Value Iteration
- ▶ Semi-External-Memory Graph Search
 - (Minimal) Perfect Hash Functions
 - c-bit Semi-Externality
- ▶ Flash Memory (Solid State Disk)
 - Immediate Duplicate Detection
 - Hashing with Fore- and Background Memory

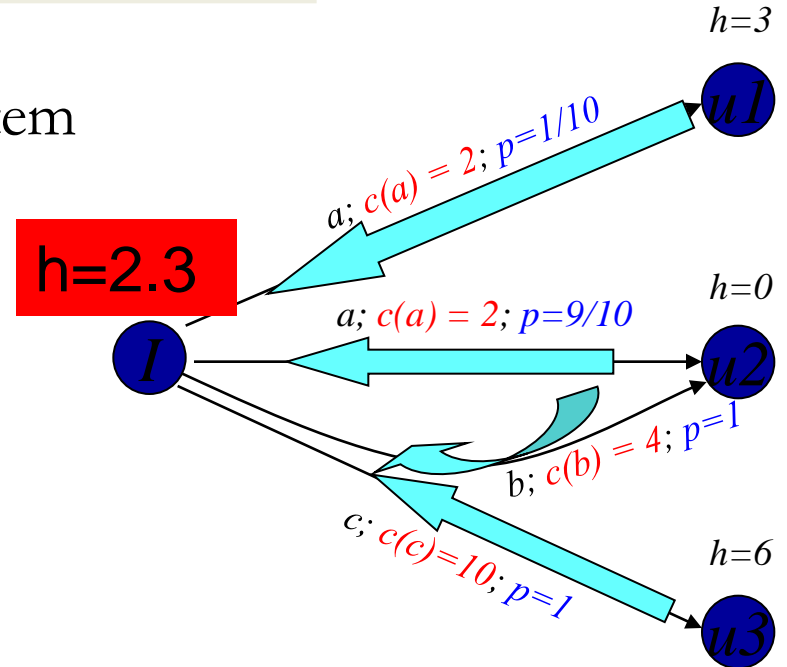
Given: Finite State-Transition System

Probabilistic + Non-deterministic

Action a: $2 + 1/10 \times 3 + 9/10 \times 0 = 2.3$

b: $4 + 1 \times 0 = 4$

c: $10 + 1 \times 6 = 16$



$$h(u) = \begin{cases} c_{\mathcal{T}}(u) & \text{if } u \in \mathcal{T} \\ \min_{a \in A(u)} \left\{ c(a, u) + \sum_{v \in \Gamma(u, a)} P_a(v | u) \cdot h(v) \right\} & \text{otherwise} \end{cases}$$

Find: Optimal h-value assignment

Uniform Search Model:

Deterministic

$$h(u) = \begin{cases} 0 & \text{if } u \in \mathcal{T} \\ \min_{v \in \Gamma(u)} \{c(a, u) + h(v)\} & \text{otherwise} \end{cases}$$

Non-Deterministic

$$h(u) = \begin{cases} 0 & \text{if } u \in \mathcal{T} \\ \min_{a \in A(u)} \left\{ c(a, u) + \sum_{v \in \Gamma(u, a)} h(v) \right\} & \text{otherwise} \end{cases}$$

Probabilistic

$$h(u) = \begin{cases} c_{\mathcal{T}}(u) & \text{if } u \in \mathcal{T} \\ \min_{a \in A(u)} \left\{ c(a, u) + \sum_{v \in \Gamma(u, a)} P_a(v | u) \cdot h(v) \right\} & \text{otherwise} \end{cases}$$

Internal Memory Value Iteration

Algorithm 1 Value Iteration

Input: State space \mathcal{S} ; initial value func. h ; tolerance $\epsilon \geq 0$

Output: Optimal value function h^*

```
1: for all  $u \in \mathcal{S}$  do
2:    $h_0(u) \leftarrow h(u)$ 
3: end for
4:  $t \leftarrow 0$ ;  $Res \leftarrow +\infty$ 
5: while  $t < t_{\max} \wedge Res > \epsilon$  do
6:    $Res \leftarrow 0$ 
7:   for all  $u \in \mathcal{S}$  do
8:     Apply update rule for  $h_{t+1}(u)$  based on the model.
9:      $Res \leftarrow \max\{|h_{t+1}(u) - h_t(u)|, Res\}$ 
10:  end for
11:   $t \leftarrow t + 1$ 
12: end while
13: return  $h_{t-1}$ 
```

- **ϵ -Optimal** for solving MDPs, AND/OR trees...

- **Problem:**

- Needs to have the whole state space in the main memory.

External-Memory Algorithm for Value Iteration

- ▶ What makes value iteration **different** from the usual external-memory **search algorithms**?
- ▶ **Answer:**
 - Propagation of information from states to predecessors!
- ➔ Edges are more important than the states.

Ext-VI works on Edges:

$$(u, v, h(v), a), \text{ where } u \xrightarrow{a} v$$

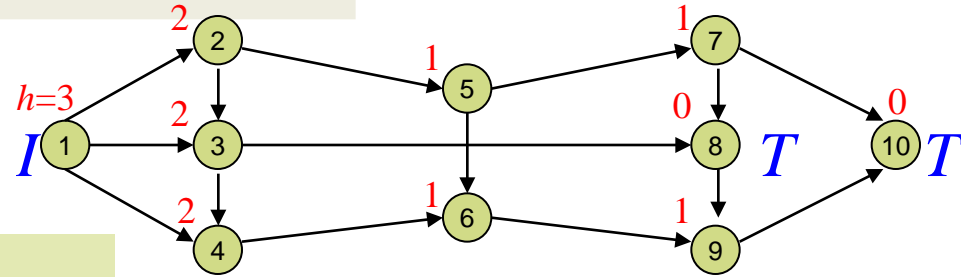
External Memory Value Iteration

- ▶ **Phase I:** Generate the **edge space** by **External BFS**.
- ▶ $Open(0) = Init, i = -1$
- ▶ **while** ($Open(i-1) \neq \text{empty}$)
 - $Open(i) = Succ(Open(i-1))$
 - Externally-Sort-and-Remove-Duplicates($Open(i)$)
 - **for** $loc = 1$ **to** $Locality(Graph)$
 - $Open(i) = Open(i) \setminus Open(i - loc)$
 - $i++$
- ▶ **endwhile**

Remove previous layers

- **Merge all BFS layers into one edge list on disk!**
- $Open_t = Open(0) \cup Open(1) \cup \dots \cup Open(DIAM)$
- $Temp = Open_t$
- Sort $Open_t$ wrt. the successors; Sort $Temp$ wrt. the predecessors*

Working of Ext-VI Phase-II



Temp: Edge List on Disk – Sorted on Predecessors

h
=

	3	2	2	2	2	1	2	0	1	1	1	1	0	0	0	0
	{(∅, 1), (1,2), (1,3), (1,4), (2,3), (2,5), (3,4), (3,8), (4,6), (5,6), (5,7), (6,9), (7,8), (7,10), (9,8), (9,10)}															

{(∅,1), (1,2), (1,3), (2,3), (1,4), (3,4), (2,5), (4,6), (5,6), (5,7), (3,8), (7,8), (9,8), (6,9), (7,10), (9,10)}

h
=

	3	2	2	2	2	2	1	1	1	1	0	0	0	1	0	0
h'	3	2	<u>1</u>	<u>1</u>	2	2	<u>2</u>	<u>2</u>	<u>2</u>	1	0	0	0	1	0	0

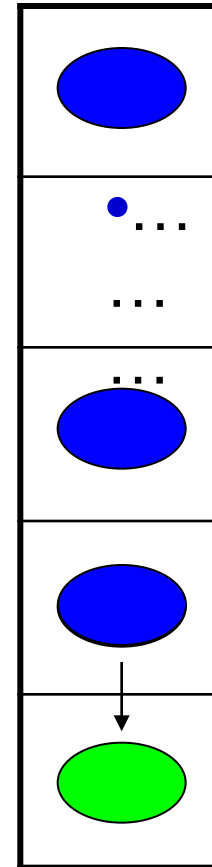
Open_t: Edge List on Disk – Sorted on Successors

$$h'(u) = 1 + \min_{v \in Succ(u)} \{h(v)\}$$

Alternate sorting and update until residual < epsilon

Complexity Analysis

- ▶ **Phase-I: External Memory Breadth-First Search.**
- ▶ **Expansion:**
 - Scanning the red bucket: $O(\text{scan}(|E|))$
- ▶ **Duplicates Removal:**
 - Sorting the green bucket having **one state for every edge** from the red bucket.
 - Scanning and compaction: $O(\text{sort}(|E|))$
- ▶ **Subtraction:**
 - Removing states of blue buckets (duplicates free) from the green one: $O(l \times \text{scan}(|E|))$



Complexity of Phase-I:

$$O(l \times \text{scan}(|E|) + \text{sort}(|E|))$$

I/Os

Complexity Analysis

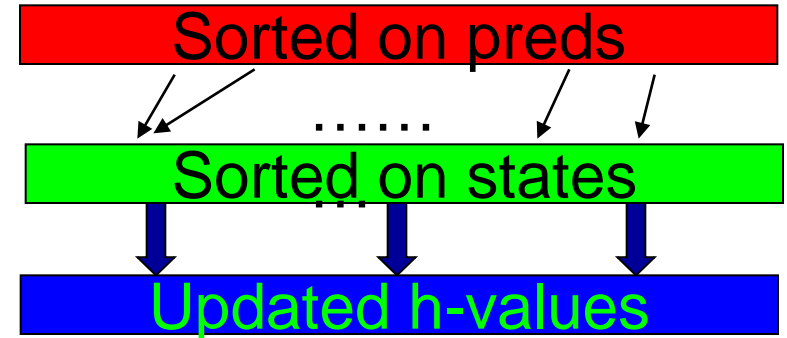
▶ Phase-II: Backward Update

▶ Update:

- Simple block-wise scanning.
- Scanning time for red and green files: $O(\text{scan}(|E|))$ I/Os

▶ External Sort:

- Sorting the blue file with the updated values to be used as red file later: $O(\text{sort}(|E|))$ I/Os



Total Complexity of Phase-II: For t_{max} iterations,

$$O(t_{max} \times \text{sort}(|E|)) \text{ I/Os}$$

Semi-External EM Search

- ▶ generate state space with **external BFS**
- ▶ construct **perfect hash function** (MPHF) from disk
- ▶ use **bit-state hash table** $\text{Visited}[h(u)]$ in RAM and **stack** on disk to perform cycle detection **DFS**
- ➔ I/Os for Ex-BFS + **const. scans and sorts**

Optimal counter-examples:

- ➔ I/Os for Ex-BFS + $|F|$ **scans**

On-the-fly by iterative deepening (bounded MC)

- ➔ I/Os for Ex-BFS + **max-BFS-depth scans**



Semi-Externality

Graph search algorithm A is **c-bit semi-external** if for each implicit graph $G = (V, E)$ RAM requirements are at most $O(v_{\max}) + c \cdot |V|$ bits.

$O(v_{\max})$ covers the RAM needed for program code, auxiliary variables, and storage of a constant amount of vertices.

Lower bound

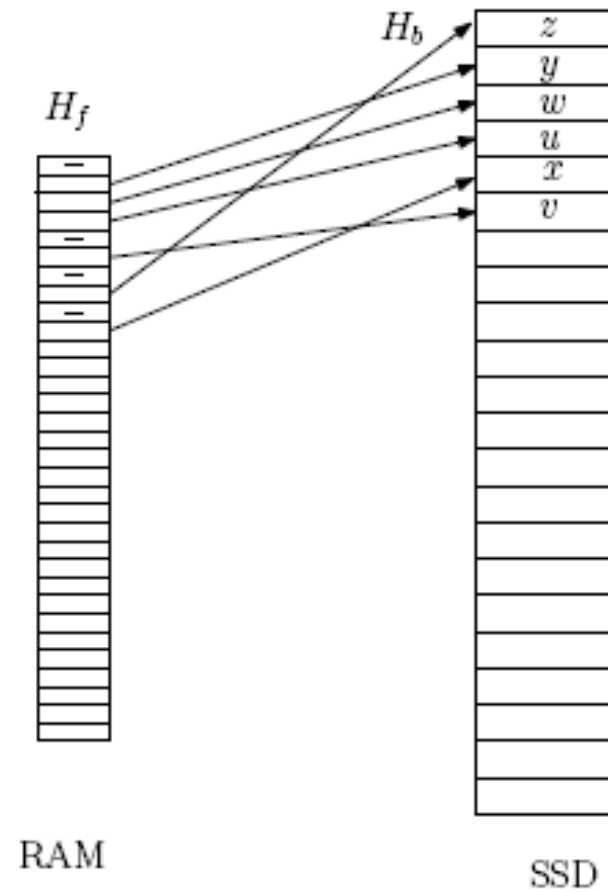
$\log \log |U| + (\log |E|) |V| + O(\log |V|)$ bits

Model	Number of Vertices	v_{max}	ϵ_s	MPHF Size (bits/vertex)
Elev.2(16),P4	173,916,122	30 bytes	94	4.941
Lamport(5),P4	74,413,141	24 bytes	99	4.941
MCS(5),P4	119,663,657	28 bytes	91	4.941
Peterson(5),P4	284,942,015	32 bytes	177	4.941
Phils(16,1),P3	61,230,206	50 bytes	47	4.941
Ret.(16,8,4),P2	31,087,573	91 bytes	553	4.941
Szyman.(5),P4	419,183,762	32 bytes	223	4.941

Flash-Memory Graph Search

- ▶ **Solid State Disk** operate as trade-off between **RAM** and **Hard Disk**
- ▶ On NAND technology, **random reads** are **fast**, **random writes** are **slow**
- ▶ With refined hashing, **immediate duplicate detection** becomes feasible for external memory graph search (CPU usage > 70%)
- ▶ Beats DDD in large search depth

Compression Strategy



Conclusion

- ▶ Disk-based algorithms with I/O complexity analysis.
- ▶ Can **pause-and-resume** execution to **add** more hard disks.
 - **Error trace:**
 - **No** predecessor pointers!
 - **Save the predecessor** with each state.
 - **Trace back** from the goal state to the start state breadth-wise.
 - **Disk space eaten by duplicate states:**
 - Start **“Early” Delayed Duplicate Detection**

Applications & Future Extensions

Applications:

- ▶ Sequence Alignment Problem
- ▶ Parallel External C++ Model Checking

In Implementation:

- ▶ Partial-Order Reduction
- ▶ Pipelined I/Os – keep block in the memory as long as possible