AAMAS 2019 Tutorial on Heuristic Search

Nathan Sturtevant, Daniel Harabor,
Sven Koenig

USC University of Southern California

MONASH University

UNIVERSITY OF ALBERTA

Part III
Preprocessing & Constraints

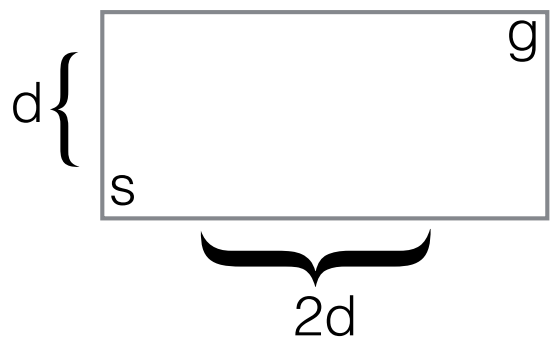
Overview

- Part I problem definition
  - Implicit state space
  - Only given heuristic function
- Part III
  - More than just heuristics

Canonical Ordering Jump Point Search

How many duplicates?

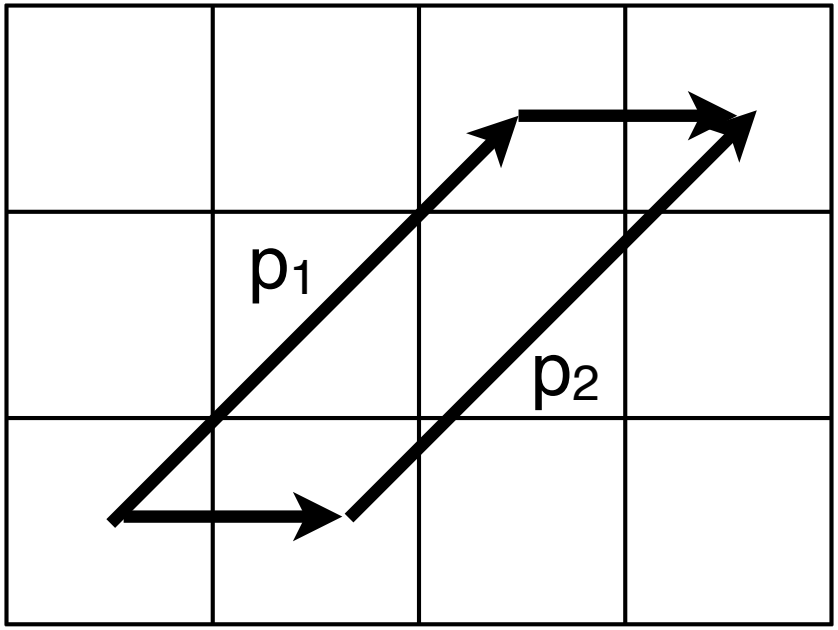
- How many unique paths between two states in an octile map?
  - Assume a 2d x d rectangle with start/goal in the corners



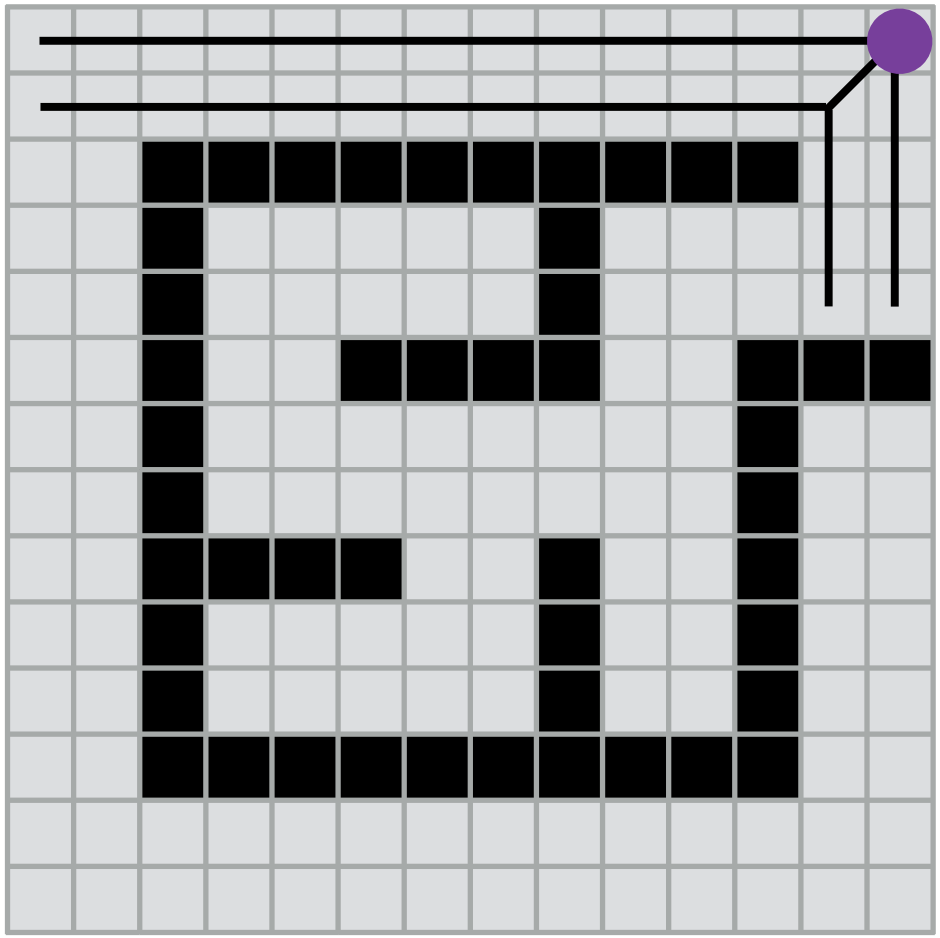
$$\binom{2d}{d} = \frac{(2d)!}{2(d!)}$$

Canonical Ordering of Paths

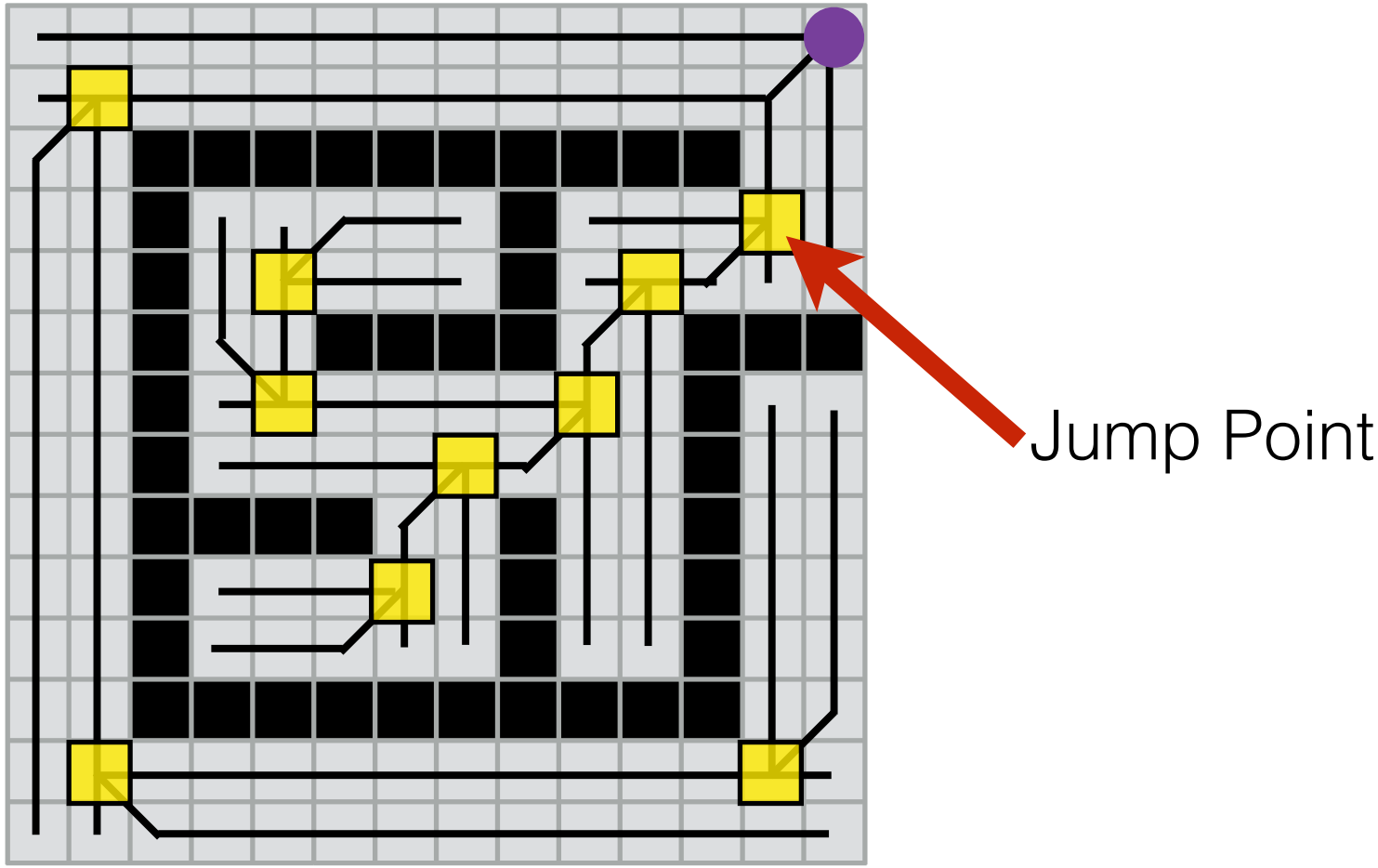
- Order all optimal paths:
  - Path p_1 is preferred over path p_2 if
    - p_1 has diagonal actions prior to cardinal actions



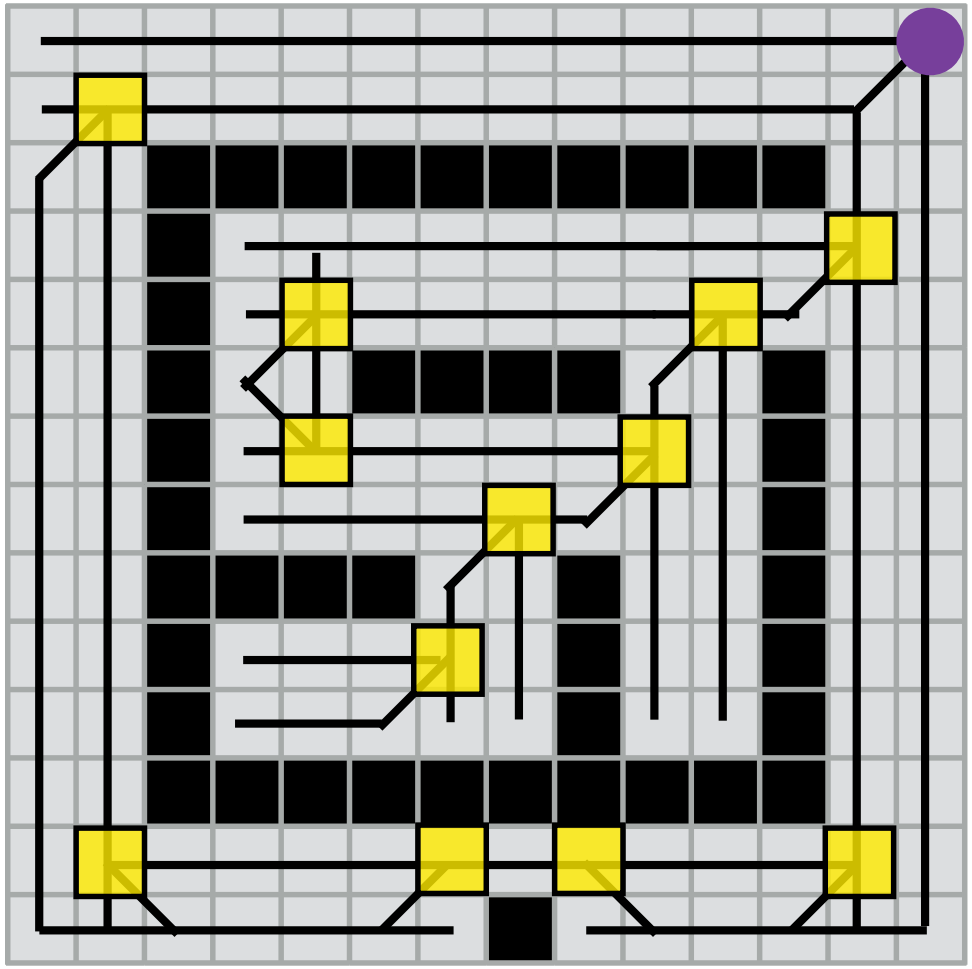
Basic Canonical Ordering



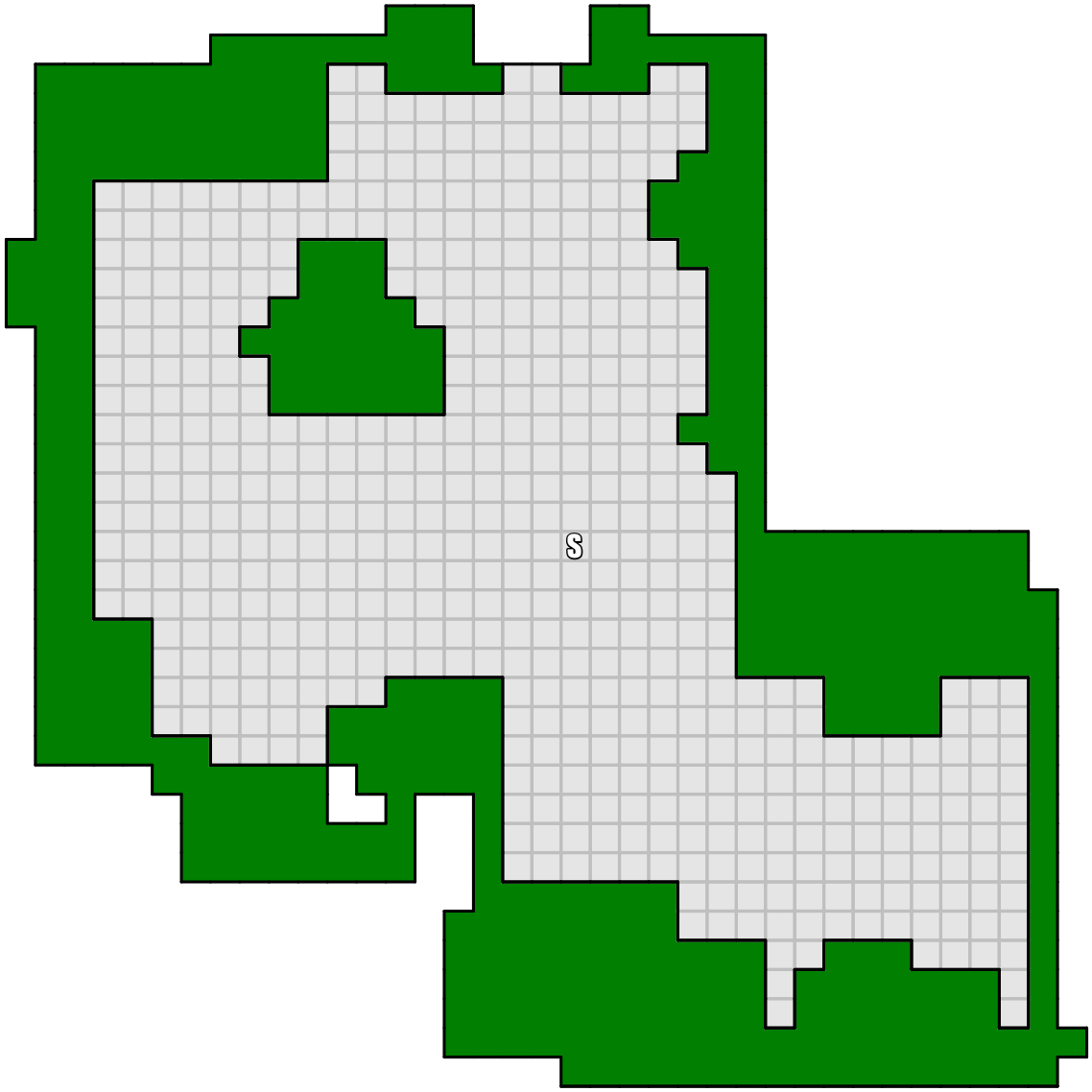
Full Canonical Ordering (Tree)



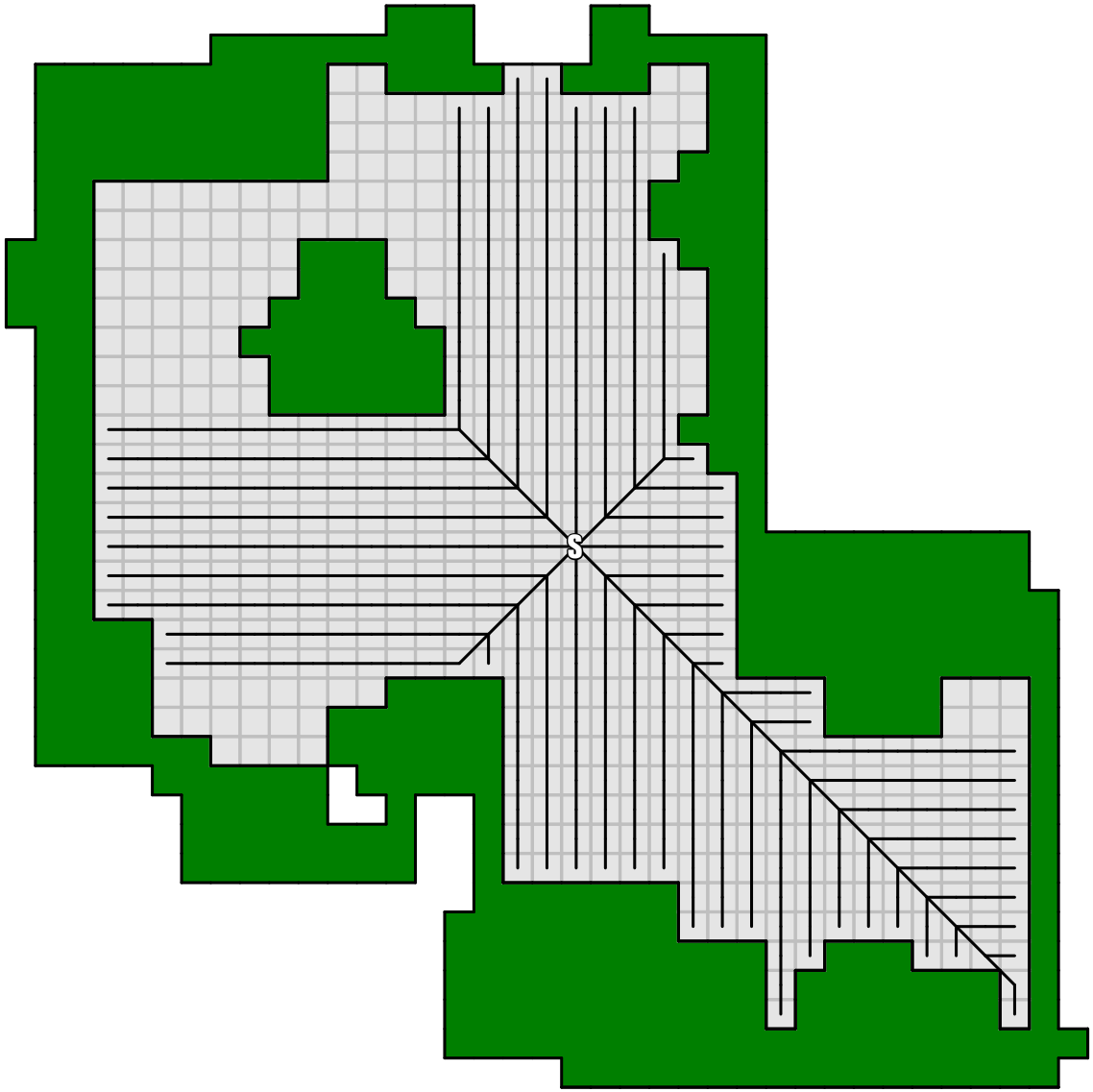
Full Canonical Ordering (graph)



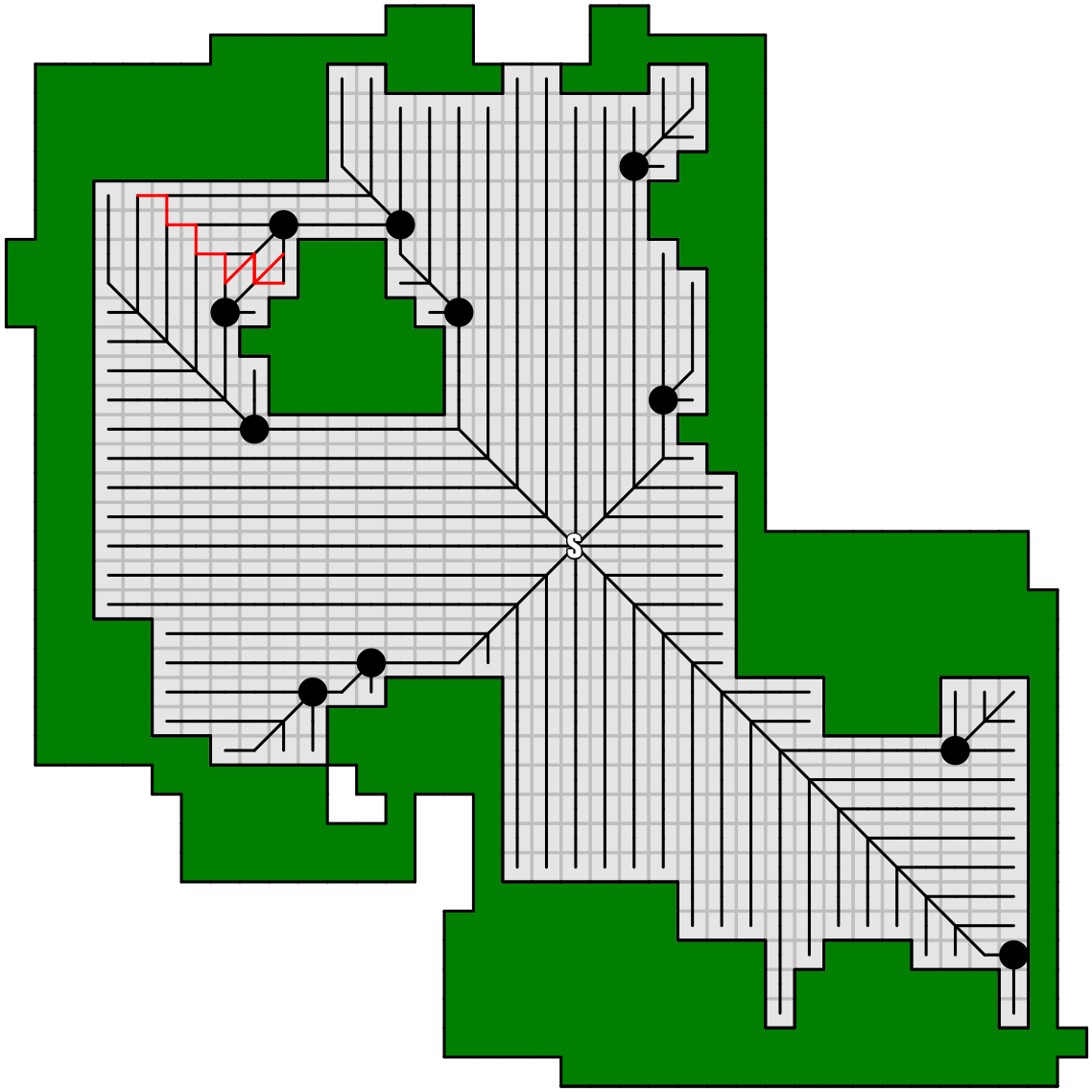
Basic Map



Simple Canonical Ordering



Canonical Ordering



Algorithms

- Canonical A* (Sturtevant & Rabin, 2016)
  - Search using canonical ordering
- Jump Point Search (JPS) (Harabor & Grastien, 2011)
  - Following canonical ordering to jump point, and only point jump points on open list
- Bounded JPS (Sturtevant & Rabin, 2016)
  - Following canonical ordering depth k
  - Could use other policies

Algorithms

- Canonical A* BJPS(0)
  - Search using canonical ordering
- Jump Point Search (JPS) BJPS(∞)
  - Following canonical ordering to jump point, and only point jump points on open list
- Bounded JPS BJPS(k)
  - Following canonical ordering depth k
  - Could use other policies

Algorithms

- Canonical A* BJPS(0)
  - Search using canonical ordering
- Jump Point Search (JPS) BJPS(∞)
  - Following canonical ordering to jump point, and only point jump points on open list
- Bounded JPS BJPS(k)
  - Following canonical ordering depth k
- Canonical Dijkstra
  - 4x faster single-source shortest path

Demo

Demo Tasks

- Print grid map and ask students to:
  - Draw canonical ordering
  - Label jump points
- Does JPS put a state on OPEN that A* would not?
- How does JPS generate a node on different paths?

Distance Constraints (Reach, Bounding Boxes)

Constraints

- Heuristic knowledge is contained in distance function
- Some information is not easily represented in distances
- Given start, goal, current state, etc
  - Should we generate a particular successor?
  - Only if the constraints allow us to

Reach

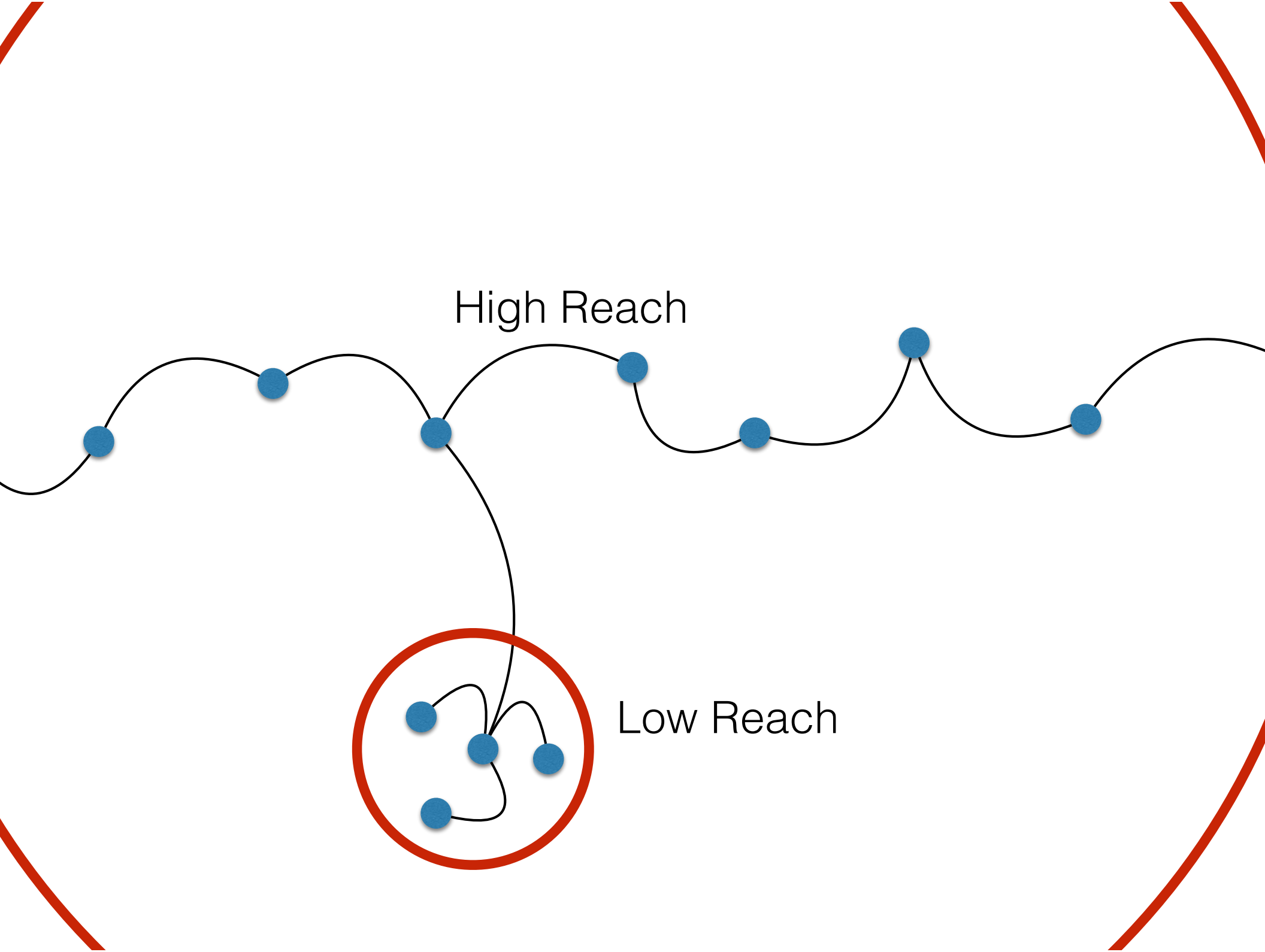
- Given a path P from *s* to *t* and a vertex *v* on P, the reach of *v* with respect to P is the minimum of the length of the prefix of P (the subpath from *s* to *v*) and the length of the suffix of P (the subpath from *v* to *t*).
- The reach of *v*, r(*v*), is the maximum, over all shortest paths P through *v*, of the reach of *v* with respect to P.
- (Goldberg, Kaplan & Werneck, 2005)

(Unidirected) Reach

- Each node has reach *r*
- For an optimal path to go through node n
  - dist(start, n) ≤ r

    or
  - dist(n, goal) ≤ r

- (Goldberg, Kaplan & Werneck, 2005)


High Reach
Low Reach

Demo

Demo Tasks

- What states have high reach?
- What states have low reach?

Bounding Boxes

- Assume you are at state *n* and and take action *a*
- What states can we reach on an optimal path?
  - Instead of storing states, represent with bounding box

- Geometric containers - Wagner et al (2005)
  - Only tested with Dijkstra's algorithm
- Bounding Boxes - Rabin and Sturtevant (2016)

Demo

Demo Tasks

- What is the bounding box for a particular state/ action?
- Why does the canonical ordering not improve result for diagonal actions?
- In what sort of maps will bounding boxes be ineffective?

Compressed Path Databases

Compressed Path Databases
Part II: Preprocessing

Daniel D. Harabor
Monash University
http://harabor.net/daniel

Joint work with:
Adi Botea, Mattia Chiari, Alfonso Gerevini, Alessandro Saeti,
Matteo Salvetti, Ben Strasser, Peter Stuckey and Shizhe Zhao

AAMAS Tutorial
2019-05-14

Working with Compressed Path Databases

Compressed Path Databases [Botea, 2011; Botea and Harabor, 2013]

CPDs store optimal *first-move* data for any pair of nodes.

Pro Perfect information, for any source-target pair.

Pro State-of-the-art performance on graphs with millions of nodes.

Con Substantial preprocessing cost — $\mathbf{O(n^2)}$ time and space.

Reducing the time cost

- CPDs target problems where preprocessing is entirely offline;
- i.e. no need to amortise costs, so time is usually not an issue.

Reducing the space cost

- CPDs are most effective when first-move data fits into main memory.
- Reducing the space is considered very important.
- Many effective techniques exist.

Building a CPD

Preprocessing

1. Run Dijkstra repeatedly, once for each node
2. After each Dijkstra search, store all the first-move data

Building a CPD

Preprocessing

1. Run Dijkstra repeatedly, once for each node
2. After each Dijkstra search, store all the first-move data

Storage

A naive encoding of APSP data is prohibitively large (often more than the size of available memory). We need to reduce the size, but:

- Without losing any information (i.e. lossless compression) and,
- While maintaining fast lookup performance

Building a CPD

Preprocessing

1. Run Dijkstra repeatedly, once for each node
2. After each Dijkstra search, store all the first-move data

Storage

A naive encoding of APSP data is prohibitively large (often more than the size of available memory). We need to reduce the size, but:

- Without losing any information (i.e. lossless compression) and,
- While maintaining fast lookup performance

Compression Schemes

Some approaches that appear in the literature:

- Quad-tree decomposition [Sankaranarayanan *et al.*, 2005]
- Rectangle decomposition [Botea and Harabor, 2013]
- **Run-length encoding** [Strasser *et al.*, 2015]

Compression using Run-length Encoding

Algorithmic Sketch

- Input: A weighted graph

Compression using Run-length Encoding

Algorithmic Sketch

- Input: A weighted graph
- Compute a first-move matrix

Compression using Run-length Encoding

Algorithmic Sketch

- Input: A weighted graph
- Compute a first-move matrix
- Run-length encode every row

Compression using Run-length Encoding

Algorithmic Sketch

- Input: A weighted graph
- Compute a first-move matrix
- Run-length encode every row
- First-move queries answered using a binary search

Compression using Run-length Encoding

Algorithmic Sketch

- Input: A weighted graph
- Compute a first-move matrix
- Run-length encode every row
- First-move queries answered using a binary search

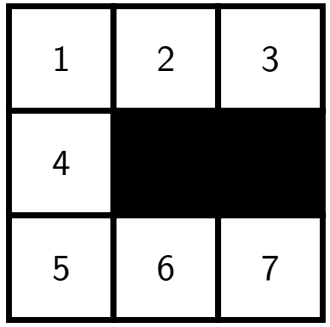
SRC, a system based on these ideas, was the fastest optimal solver in the 2014 Grid-Based Path Planning Competition [Sturtevant *et al.*, 2015].

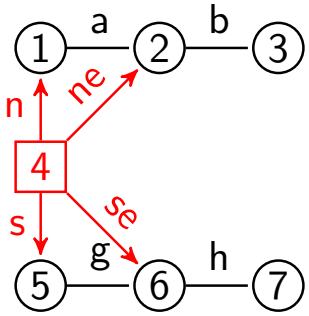
Full technical details in [Strasser *et al.*, 2014; Strasser *et al.*, 2015].

First-move matrix

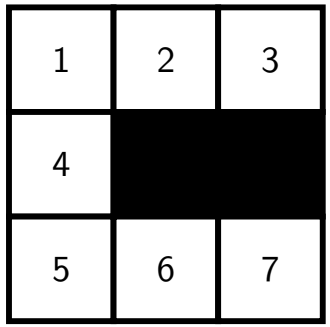
1	2	3
4		
5	6	7

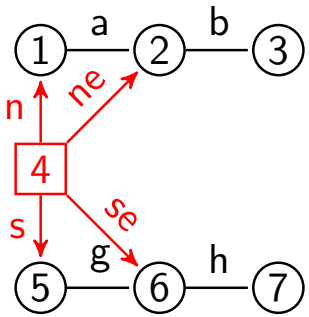
First-move matrix





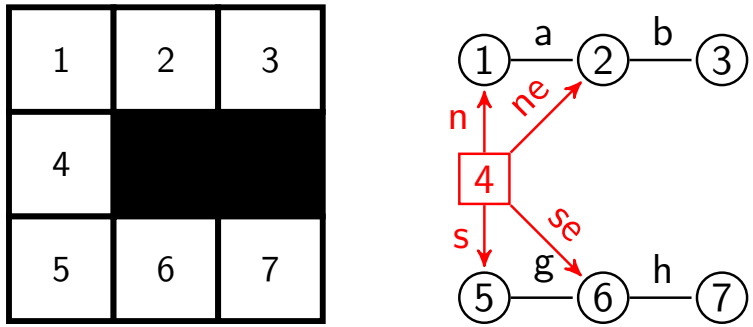
First-move matrix





s \ t	1	2	3	4	5	6	7
1							
2							
3							
4	n	ne	ne	*	s	se	se
5							
6							
7							

First-move matrix



s \ t	1	2	3	4	5	6	7
1	*	e	e	s	s	s	s
2	w	*	e	sw	sw	sw	sw
3	w	w	*	w	w	w	w
4	n	ne	ne	*	s	se	se
5	n	n	n	n	*	e	e
6	nw	nw	nw	nw	w	*	e
7	w	w	w	w	w	w	*

Compressing first-matrix rows

- First-matrix rows are compressed with run-length encoding (RLE)
- *Runs* are repetitions of the same token
  - E.g., the string **ssnnsss** has three runs: **ss**; **nn**; **sss**

Compressing first-matrix rows

- First-matrix rows are compressed with run-length encoding (RLE)
- *Runs* are repetitions of the same token
  - E.g., the string **ssnnsss** has three runs: **ss**; **nn**; **sss**

s \ t	1	2	3	4	5	6	7
1	*	e	e	s	s	s	s
2	w	*	e	sw	sw	sw	sw
3	w	w	*	w	w	w	w
4	n	ne	ne	*	s	se	se
5	n	n	n	n	*	e	e
6	nw	nw	nw	nw	w	*	e
7	w	w	w	w	w	w	*

Uncompressed matrix has 49 tokens

Compressing first-matrix rows

- First-matrix rows are compressed with run-length encoding (RLE)
- *Runs* are repetitions of the same token
  - E.g., the string **ssnnsss** has three runs: **ss**; **nn**; **sss**

s \ t	1	2	3	4	5	6	7
1	*	e	e	s	s	s	s
2	w	*	e	sw	sw	sw	sw
3	w	w	*	w	w	w	w
4	n	ne	ne	*	s	se	se
5	n	n	n	n	*	e	e
6	nw	nw	nw	nw	w	*	e
7	w	w	w	w	w	w	*

Uncompressed matrix has 49 tokens

1	1/e	4/s		
2	1/w	3/e	4/sw	
3	1/w			
4	1/n	2/ne	5/s	6/se
5	1/n	6/e		
6	1/nw	5/w	7/e	
7	1/w			

CPD has 16 runs

Non-Unique Shortest Paths?

Roads

- On roads shortest paths are very often unique.

Game Maps

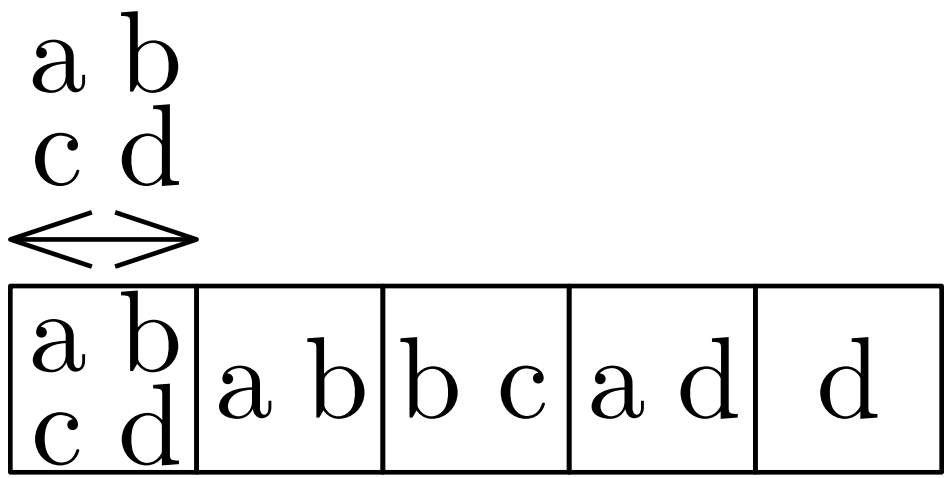
- Game maps often contain unit grids with highly non-unique shortest paths.
- **Idea**: Tie-break paths such that compression is maximised

Tie-Breaking

a b c d	a b	b c	a d	d

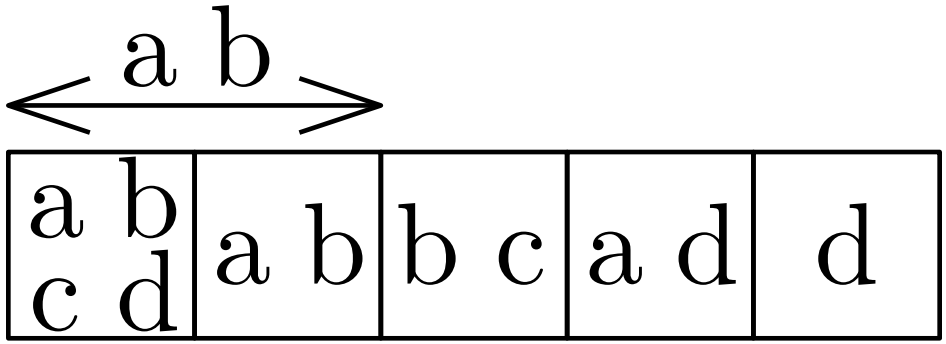
For every row compute all first moves.
(Simple extension of Dijkstra's algorithm)

Tie-Breaking



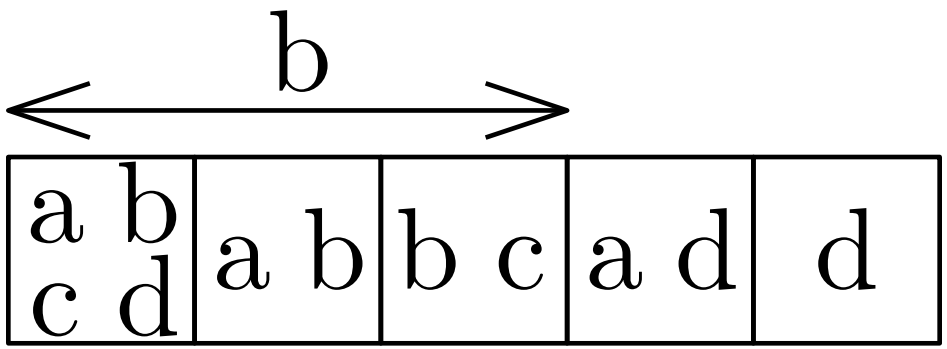
Greedily grow runs from the left to right.

Tie-Breaking



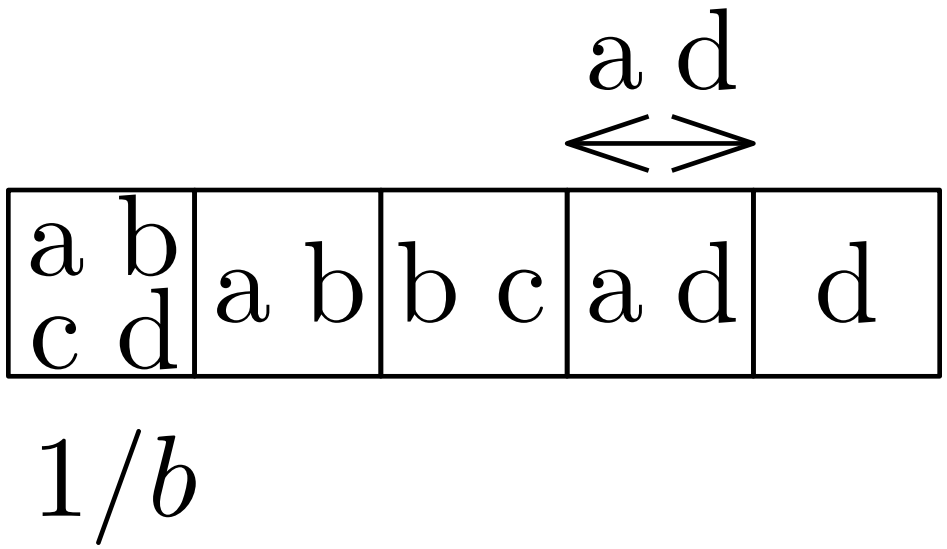
Greedily grow runs from the left to right.

Tie-Breaking



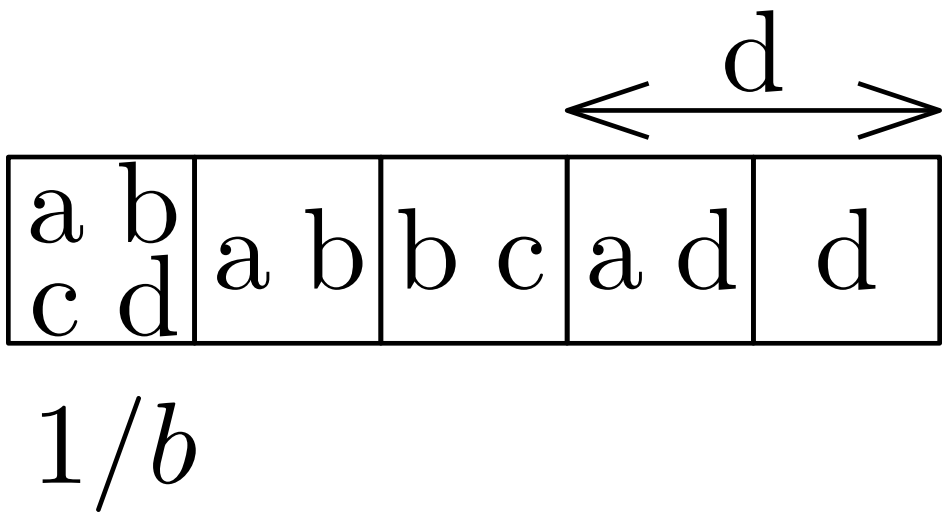
Greedily grow runs from the left to right.

Tie-Breaking



Greedily grow runs from the left to right.

Tie-Breaking



Greedily grow runs from the left to right.

Tie-Breaking

a b c d	a b	b c	a d	d

$1/b\,3/d$

This algorithm produces a minimum number of runs.

The column ordering matters

s \ t	1	2	3	4	5	6	7
1	*	e	e	s	s	s	s
2	w	*	e	sw	sw	sw	sw
3	w	w	*	w	w	w	w
4	n	ne	ne	*	s	se	se
5	n	n	n	n	*	e	e
6	nw	nw	nw	nw	w	*	e
7	w	w	w	w	w	w	*

Column ordering: 1, 2, 3, 4, 5, 6, 7

1	1/e	4/s		
2	1/w	3/e	4/sw	
3	1/w			
4	1/n	2/ne	5/s	6/se
5	1/n	6/e		
6	1/nw	5/w	7/e	
7	1/w			

CPD has 16 runs

The column ordering matters

s \ t	1	2	3	4	5	6	7
1	*	e	e	s	s	s	s
2	w	*	e	sw	sw	sw	sw
3	w	w	*	w	w	w	w
4	n	ne	ne	*	s	se	se
5	n	n	n	n	*	e	e
6	nw	nw	nw	nw	w	*	e
7	w	w	w	w	w	w	*

1	1/e	4/s		
2	1/w	3/e	4/sw	
3	1/w			
4	1/n	2/ne	5/s	6/se
5	1/n	6/e		
6	1/nw	5/w	7/e	
7	1/w			

Column ordering: 1, 2, 3, 4, 5, 6, 7

CPD has 16 runs

s \ t	1	2	4	5	6	7	3
1	*	e	s	s	s	s	e
2	w	*	sw	sw	sw	sw	e
3	w	w	w	w	w	w	*
4	n	ne	*	s	se	se	ne
5	n	n	n	*	e	e	n
6	nw	nw	nw	w	*	e	nw
7	w	w	w	w	w	*	w

1	1/e	3/s	7/e		
2	1/w	3/sw	7/e		
3	1/w				
4	1/n	2/ne	4/s	5/se	7/e
5	1/n	5/e	7/n		
6	1/nw	4/w	6/e	7/nw	
7	1/w				

Column ordering: 1, 2, 4, 5, 6, 7, 3

CPD has 20 runs

Computing a column ordering

The bad news

Finding an optimal column ordering is NP-complete in general [Strasser *et al.*, 2015].

The good news

A variety of effective heuristic orderings exist [Strasser *et al.*, 2014]

- DFS Order
  - **Idea:** Label nodes with ids using a DFS pre-order traversal from a random starting node.
- CUT Order (Nested-Edge-Cut-Order)
  - **Idea:** find a small edge cut which divides the graph into disjoint sets. Two nodes in the same set have a small difference in their index.

Even Stronger Compression

The following methods (alone or in conjunction) significantly improve RLE compression. All without impacting runtime performance.

Redundant Symbols [Chiari *et al.*, 2019]

Compress together targets for which the optimal first-move from the source node is the same as the *default move* — i.e. the move that would be taken when following a freespace heuristic.

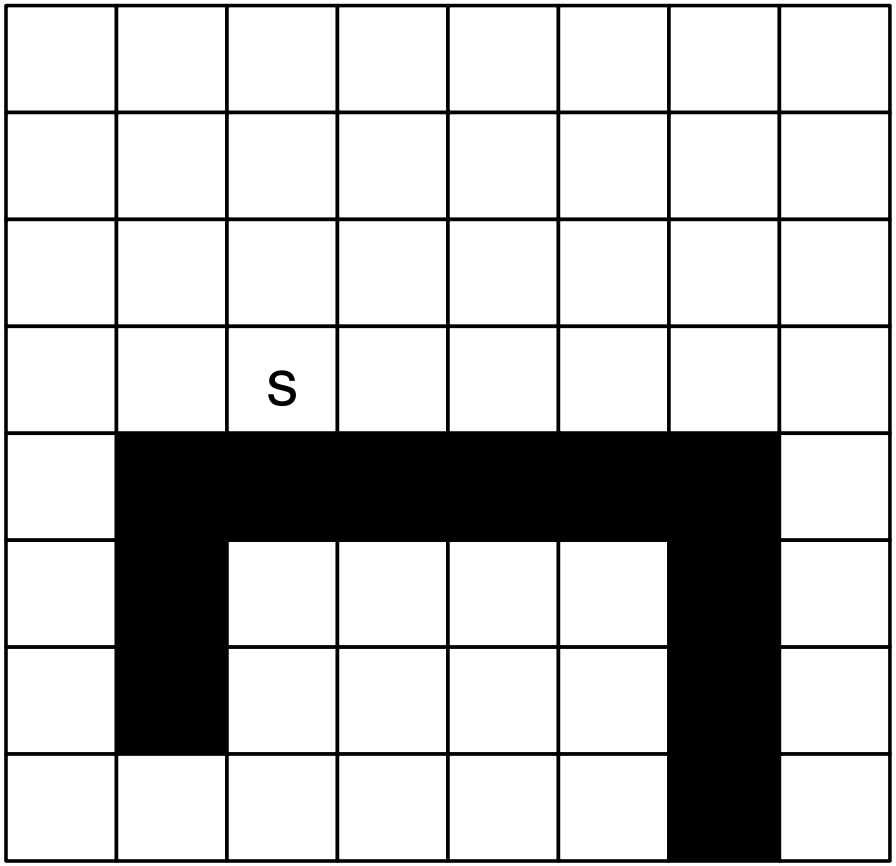
Proximity Wildcards [Chiari *et al.*, 2019]

Construct a bounding box that contains only nodes where the default move coincides with the optimal move. For all targets in this area store a wildcard (don't care) symbol.

Wildcard Substitutions [Salvetti *et al.*, 2017]

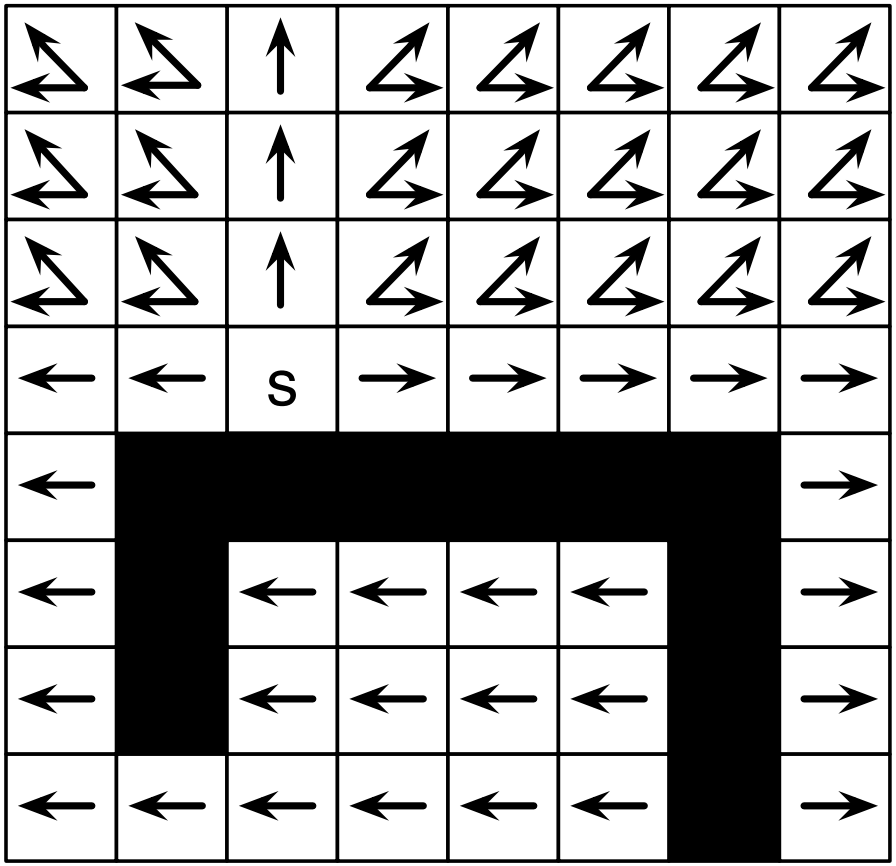
In undirected graphs the optimal path is the same forwards as backwards. This can be exploited to store moves in only one direction.

Proximity Wildcards [Chiari *et al.*, 2019]



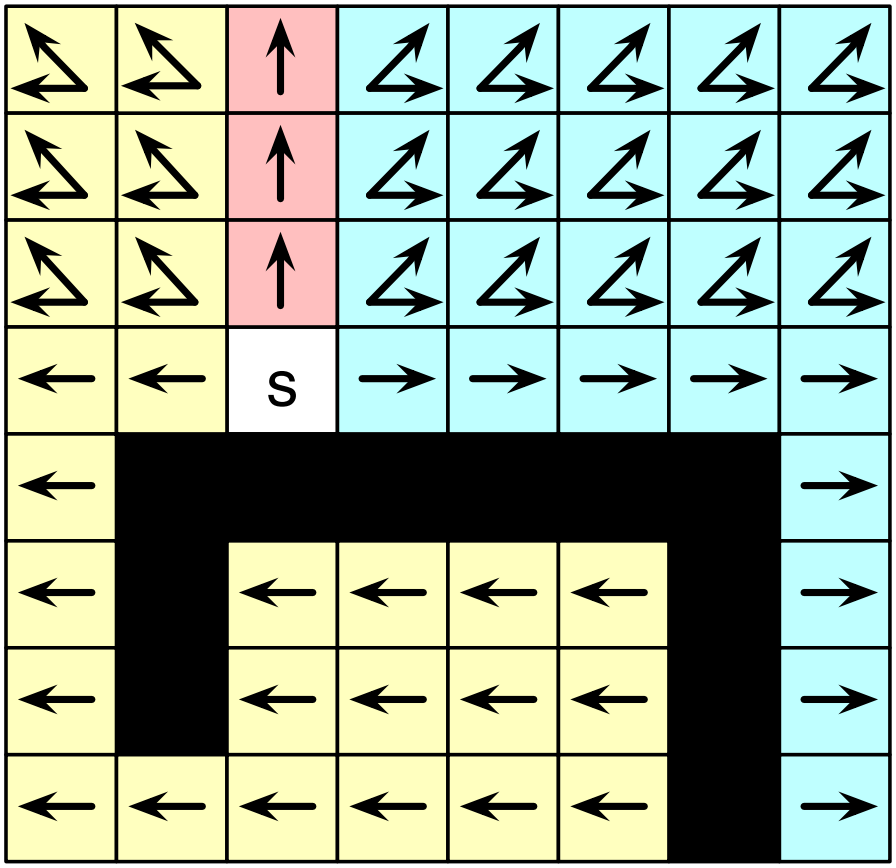
We compress the first-move table for the source node **s**. The column ordering proceeds left-to-right and top-to-bottom.

Proximity Wildcards [Chiari *et al.*, 2019]



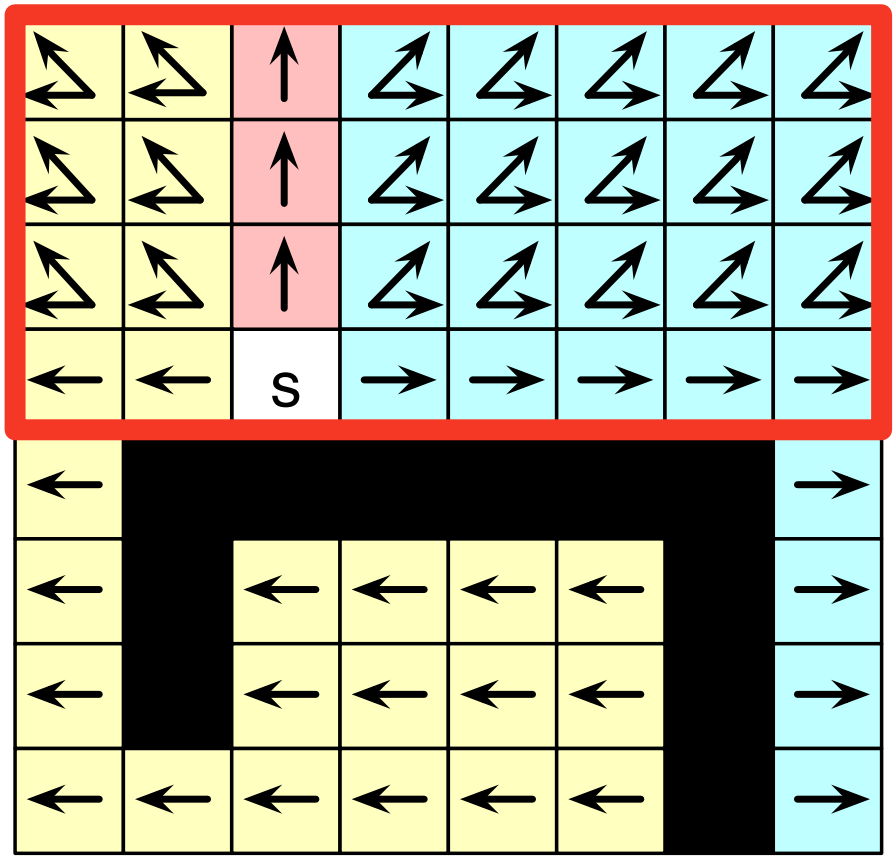
All optimal first moves computed by Dijkstra's algorithm.

Proximity Wildcards [Chiari *et al.*, 2019]



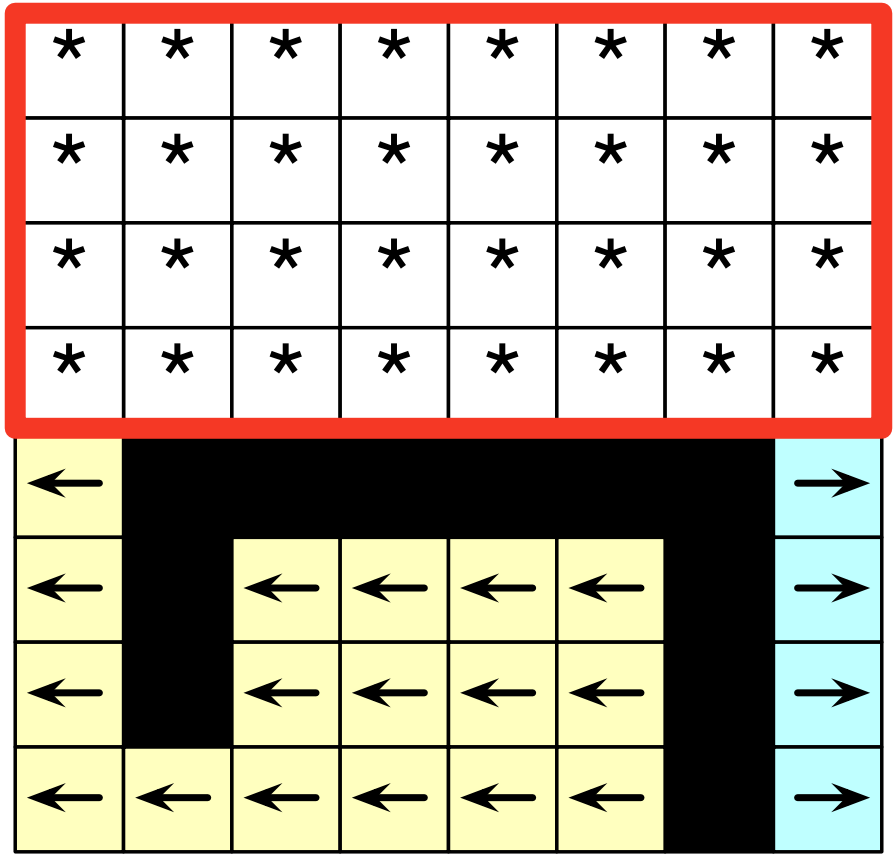
SRC compresses this table into 19 runs. Assume 5 bytes per run (4 bytes for the start index, 1 byte for the label).:

Proximity Wildcards [Chiari *et al.*, 2019]



Notice that all nodes in the rectangle can be optimally reached by following the octile heuristic.

Proximity Wildcards [Chiari *et al.*, 2019]



We throw away all first-move data for all nodes in the rectangle. Instead we store together with **s** the position and size of the rectangle. At query time, we check if the target is in the rectangle. Follow the octile heuristic if yes, otherwise extract a move from the CPD.

Compression results

Setup

We compute CPDs for a variety of standard uniform-cost game maps. The games are: Baldur's Gate II, Dragon Age and StarCraft.

		The size of First Move data in MB				
Maps	#cells	Uncompressed	*SRC*	*h*	*w*	*hw*
AR0044SR	231,469	51,096	507.1	14.3	34.8	**9.1**
AR0605SR	140,922	18,958	179.4	25.4	23.0	**14.2**
AR0700SR	131,852	16,580	69.0	37.8	23.3	**20.3**
Aftershock	166,076	26,303	88.0	8.9	14.7	**7.6**
DarkContinent	285,669	77,826	213.8	70.5	50.6	**40.4**
TheatreofWar	220,816	46,501	170.1	53.9	42.2	**36.6**
hrt000d	106,608	10,839	60.8	11.0	11.4	**6.8**
ost000a	130,478	16,313	44.1	15.9	13.4	**9.7**
ost000t	105,707	10,656	38.0	13.4	11.2	**8.0**

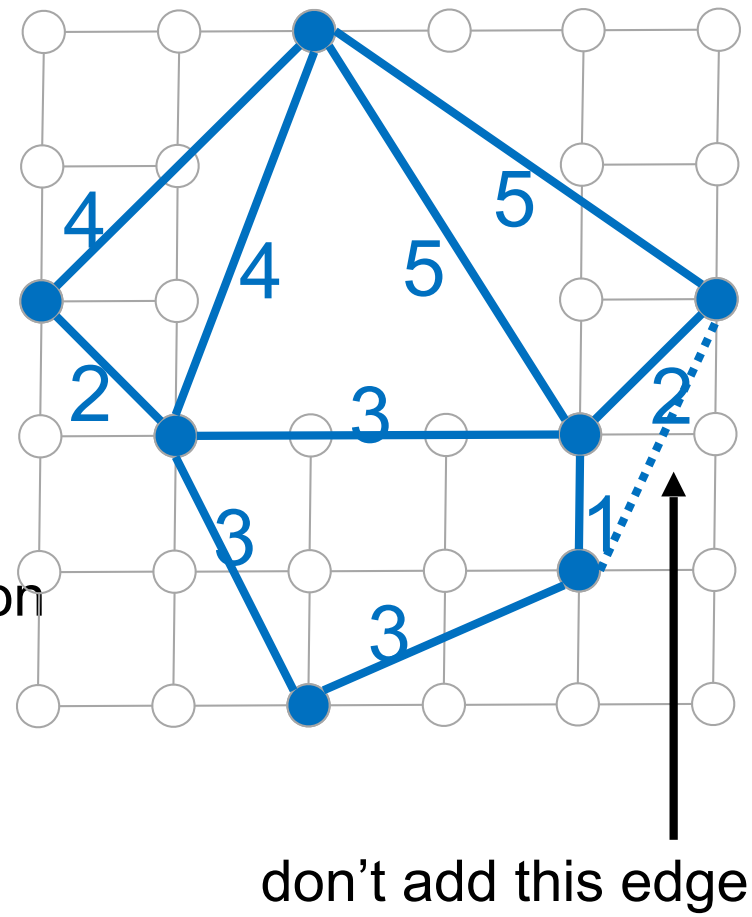
h = Redundant Symbols — w = Wildcards — hw = Redundant Symbols + Wildcards

For more details see [Chiari *et al.*, 2019].

Subgoals

Subgoal Graphs

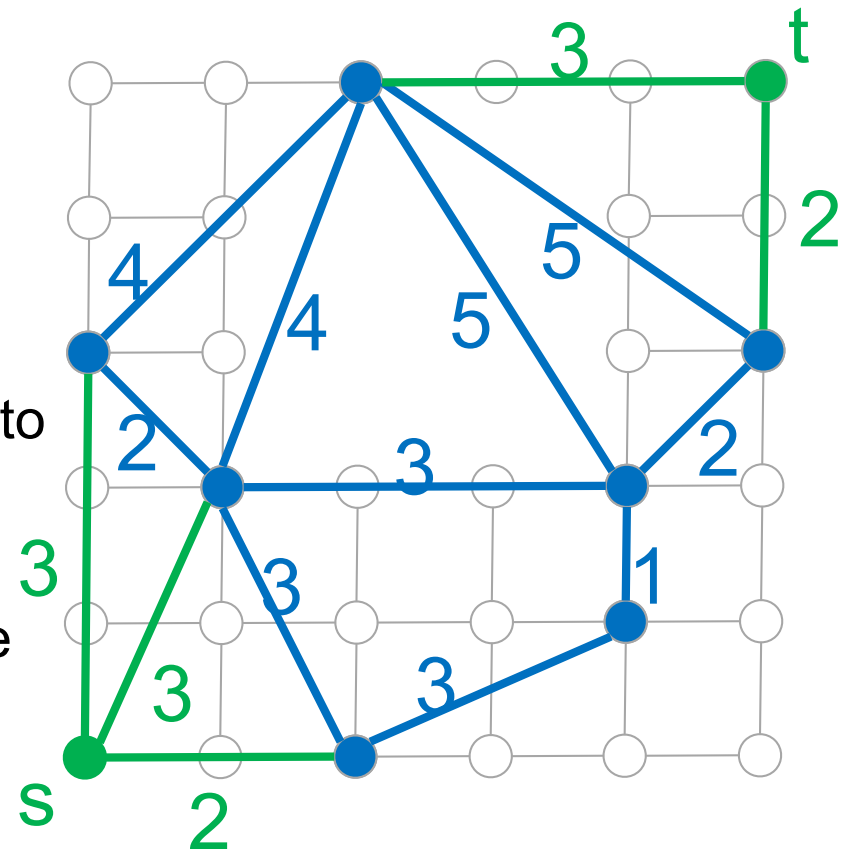
- Idea: Search the blue overlay graph (= subgoal graph) rather than the black original graph
- During preprocessing
  - Pick a set of subgoals S ⊆ V
  - Add edges between direct-reachable subgoals (= no shortest path passes through a subgoal) so that the distances on the subgoal graph equal the distances on the original graph



- + search ignores non-subgoal nodes

Subgoal Graphs

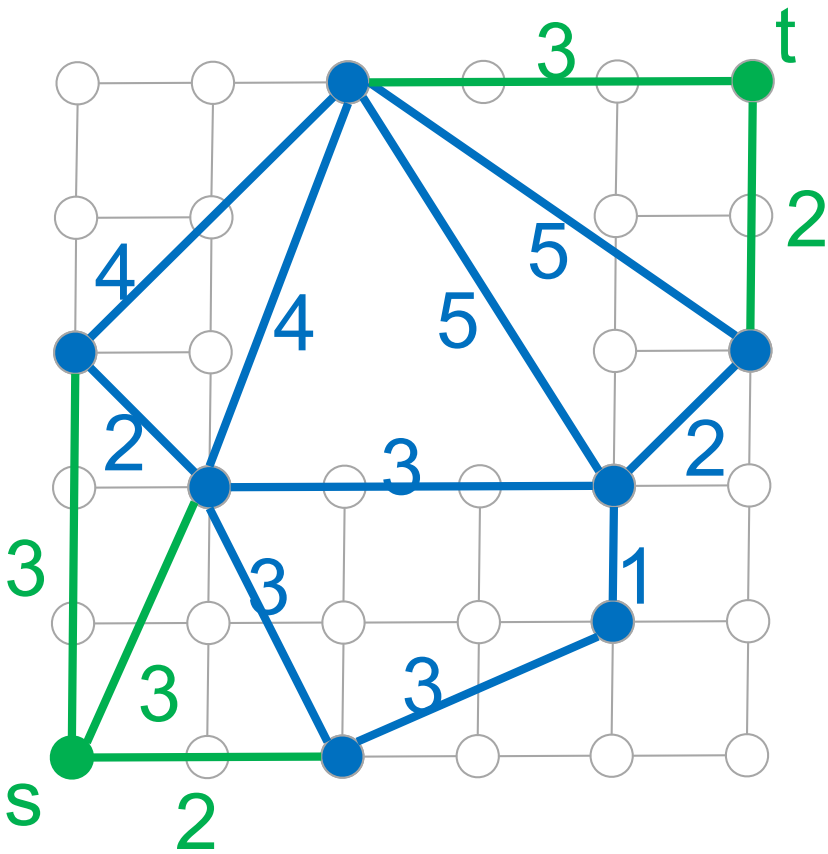
- Idea: Search an overlay graph (= subgoal graph) rather than the original graph
- During (shortest path) query
  - Connect the start node s and goal node t to the subgoal graph
  - Search the query subgoal graph
  - Refine the subgoal path into a path on the original graph



- - one has to connect and refine

Subgoal Graphs

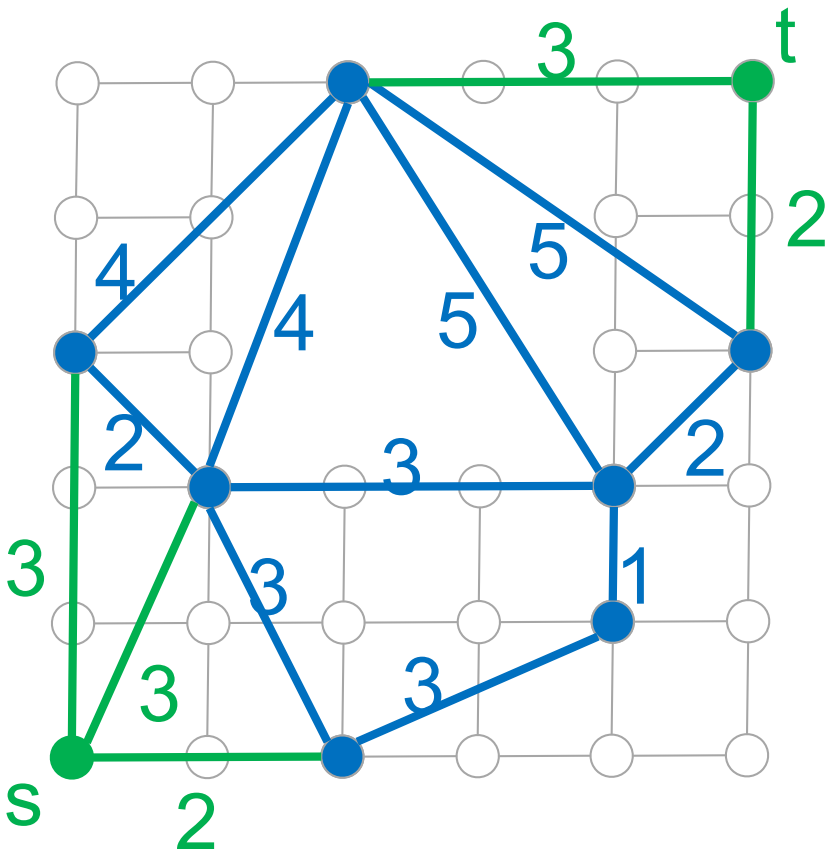
- Idea 2: Use a reachability relation $R \subseteq V \times V$ to determine R-reachable subgoals and implement R-connect and R-refine operations



- + R can decrease the connect and refine times

Subgoal Graphs

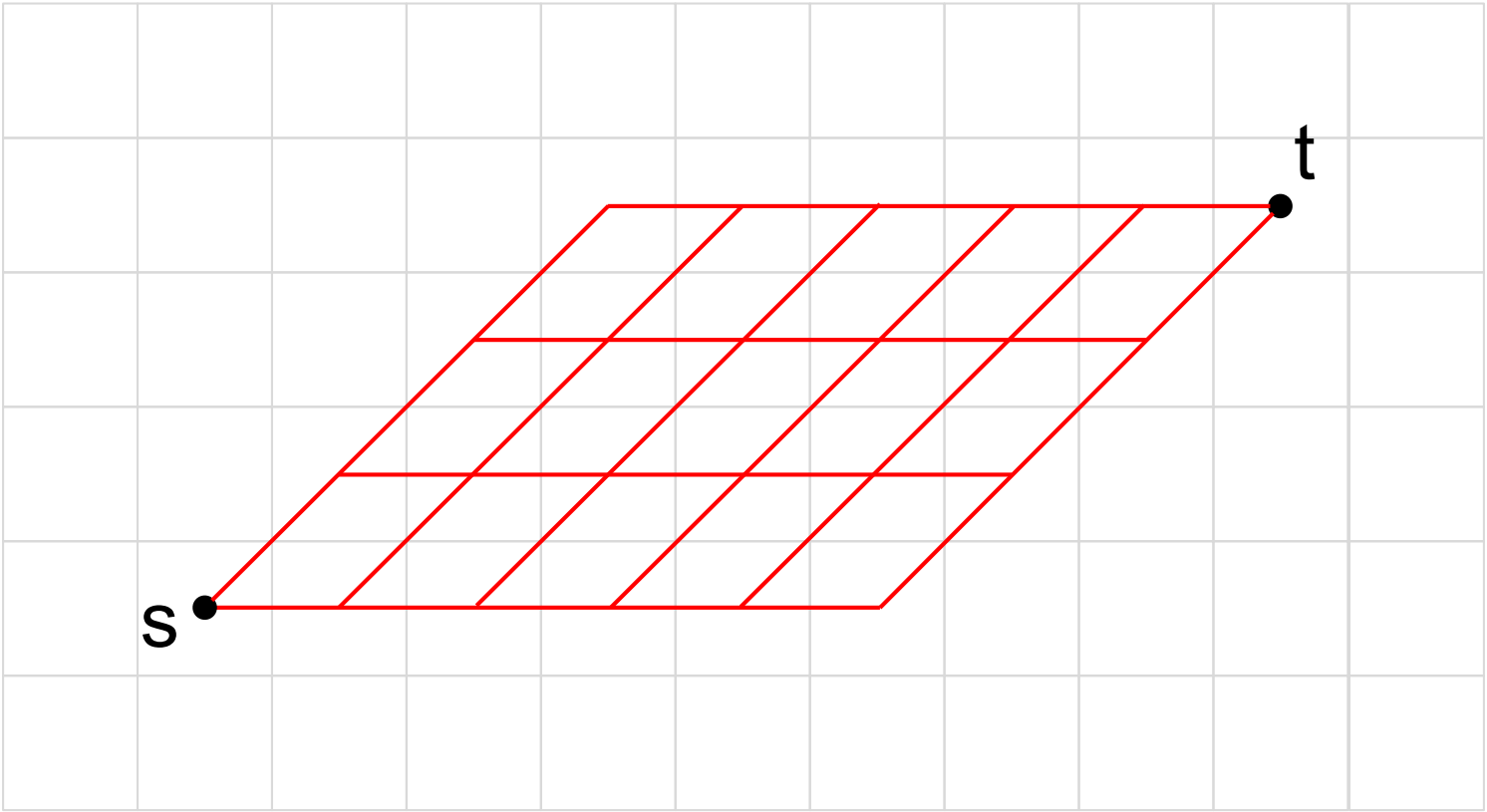
- Idea 2: Use a reachability relation $R \subseteq V \times V$ to determine R-reachable subgoals and implement R-connect and R-refine operations
- Restriction: If a node t is not R-reachable from a node s, then at least one shortest path from s to t has to pass through a subgoal.



- - R limits the construction of the subgoal graph
- + R can decrease the connect and refine times

Subgoal Graphs on Grids

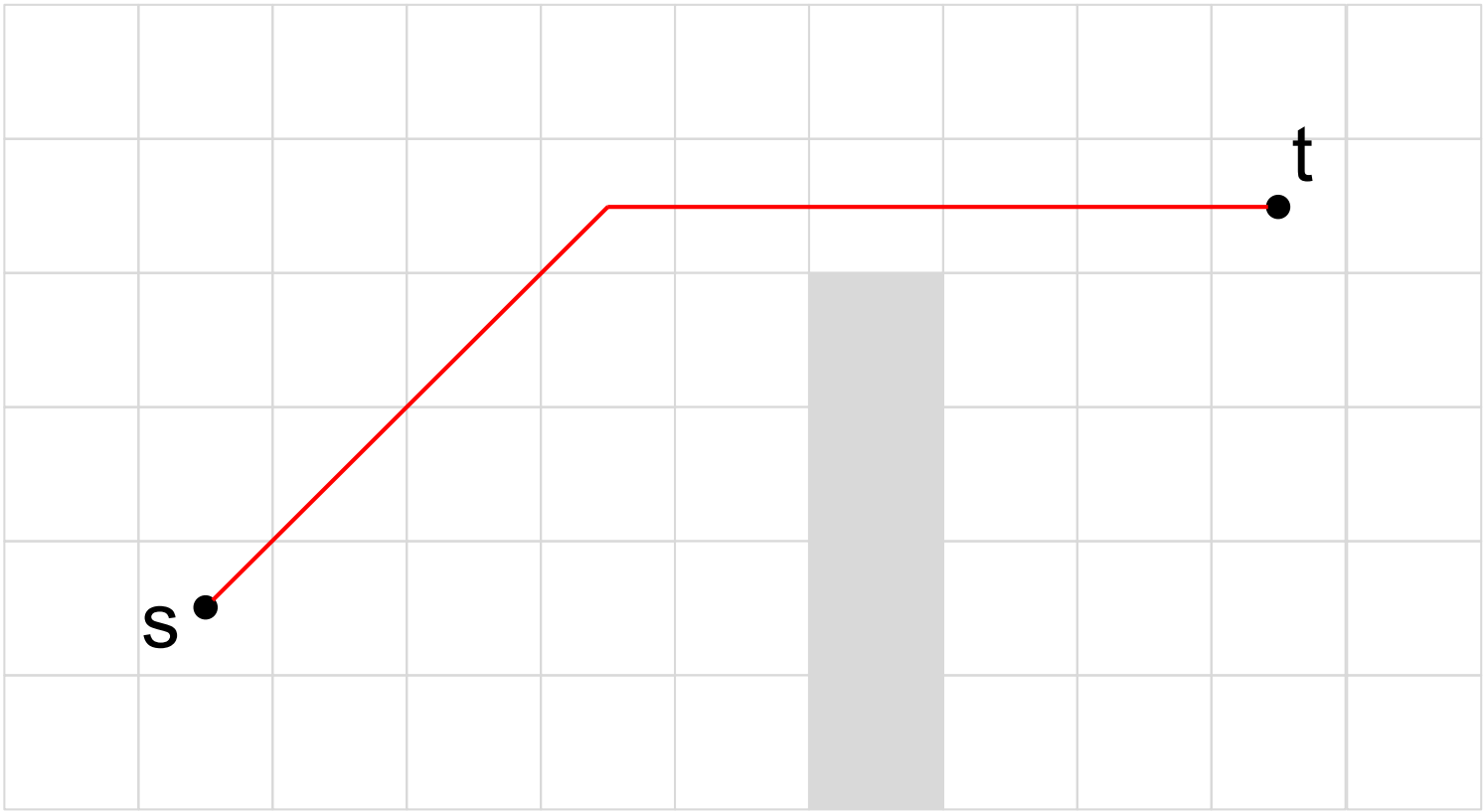
- We exploit the structure of grids on the following slides
- For example, shortest paths in freespace (= no obstacles) travel in at most two directions



8-neighbor grid

Reachability Relation

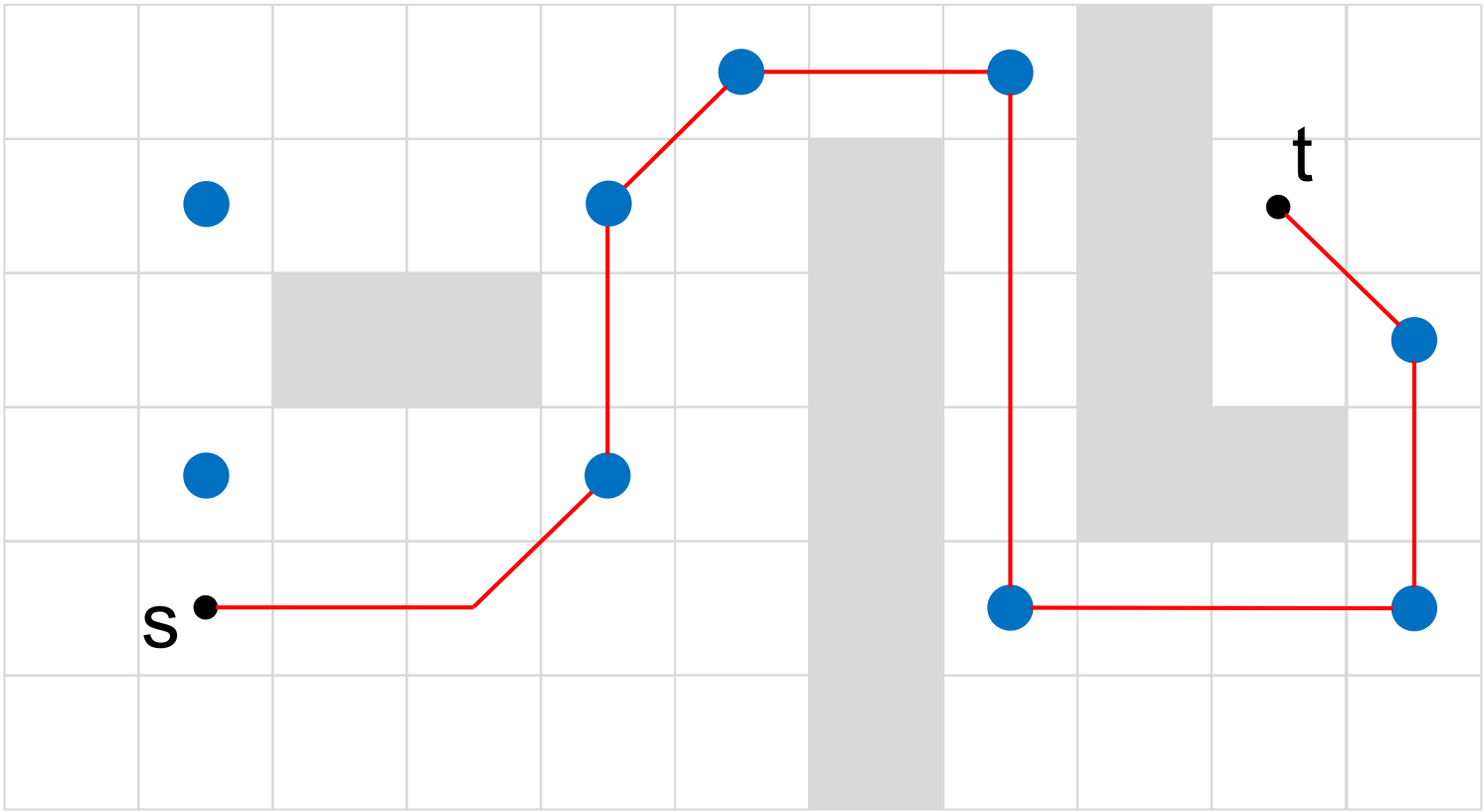
- Freespace reachability
  = at least one freespace path is unblocked



8-neighbor grid

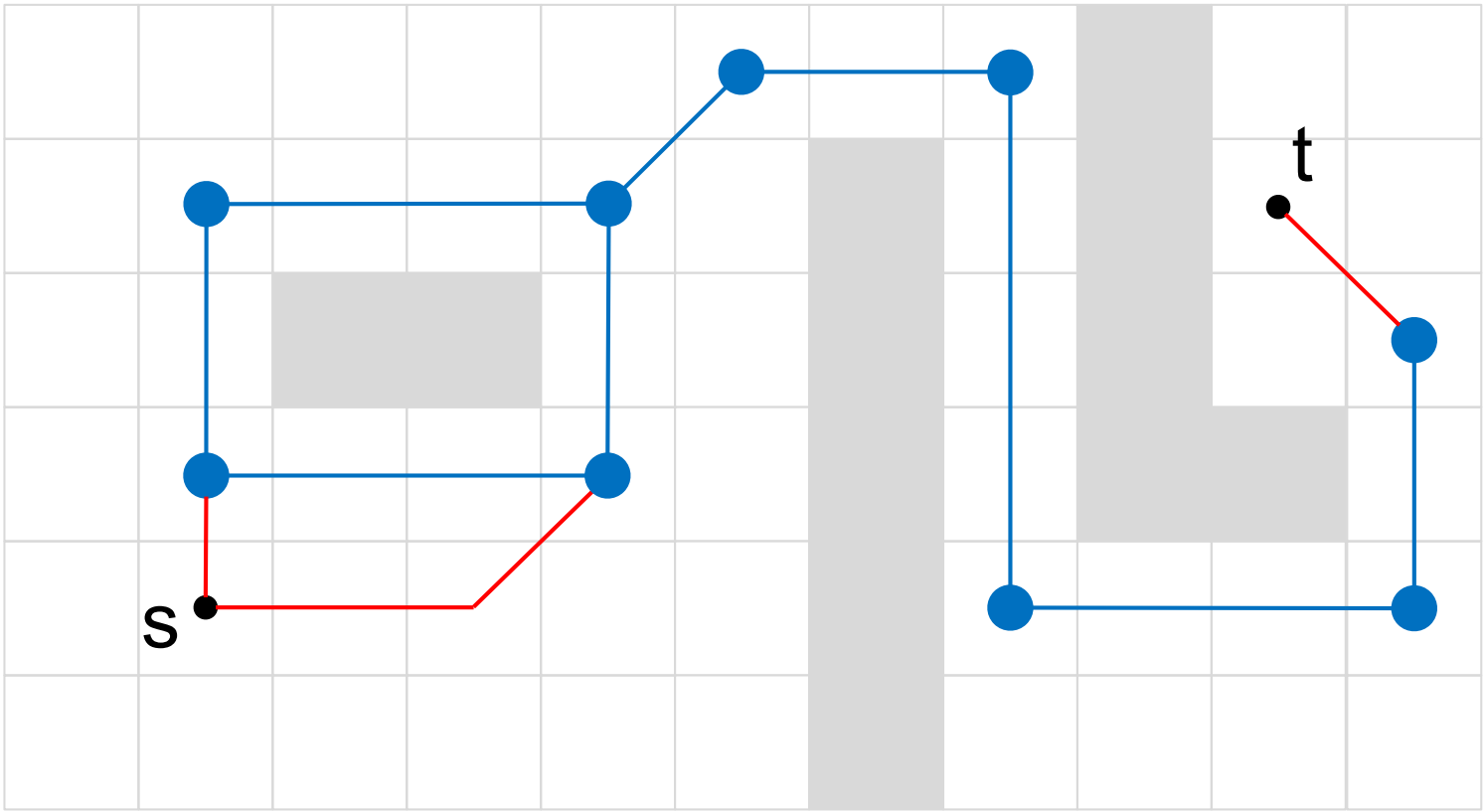
Subgoals

- Place subgoals at the “convex corners” of obstacles



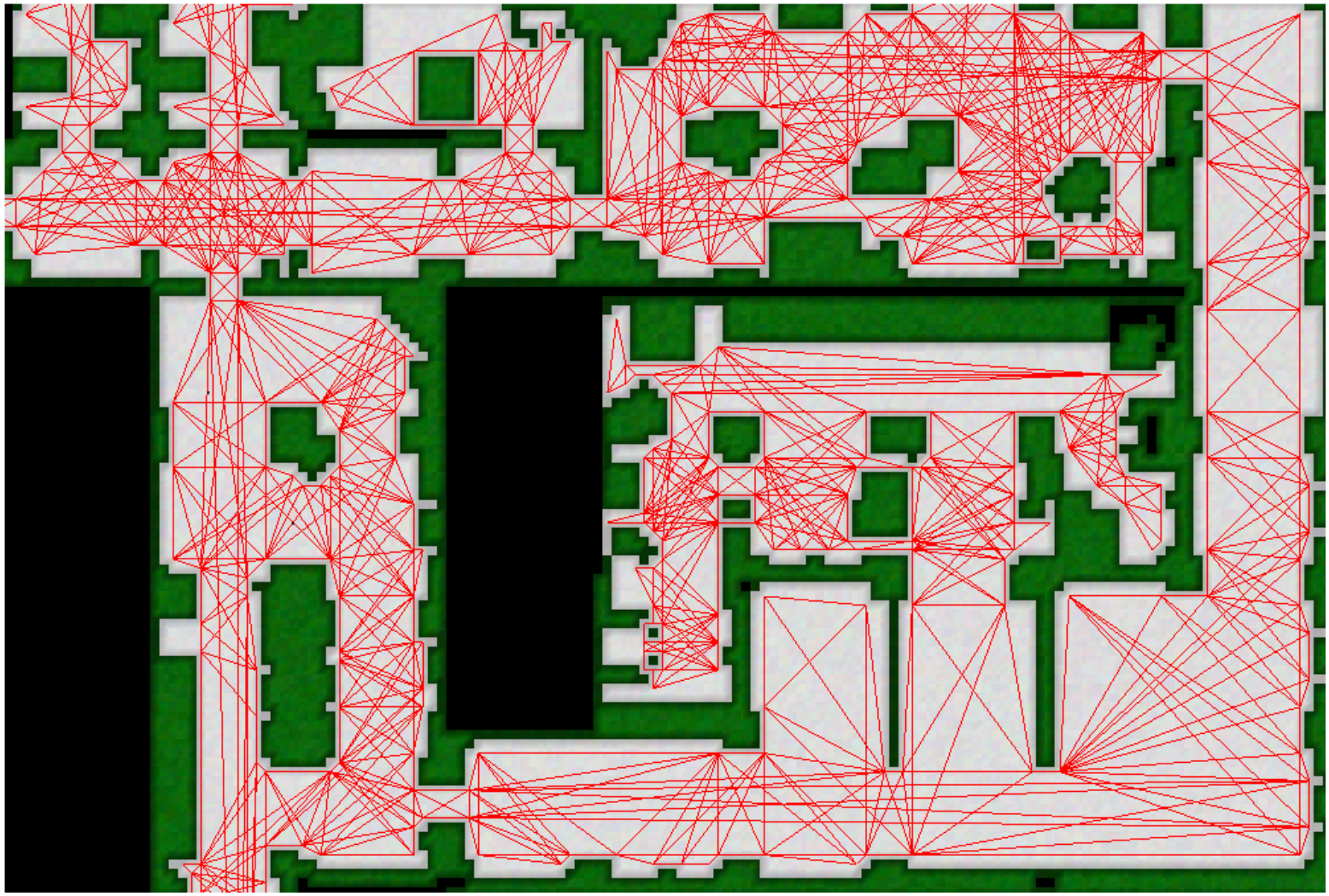
8-neighbor grid

Subgoals



8-neighbor grid

Subgoal Graphs

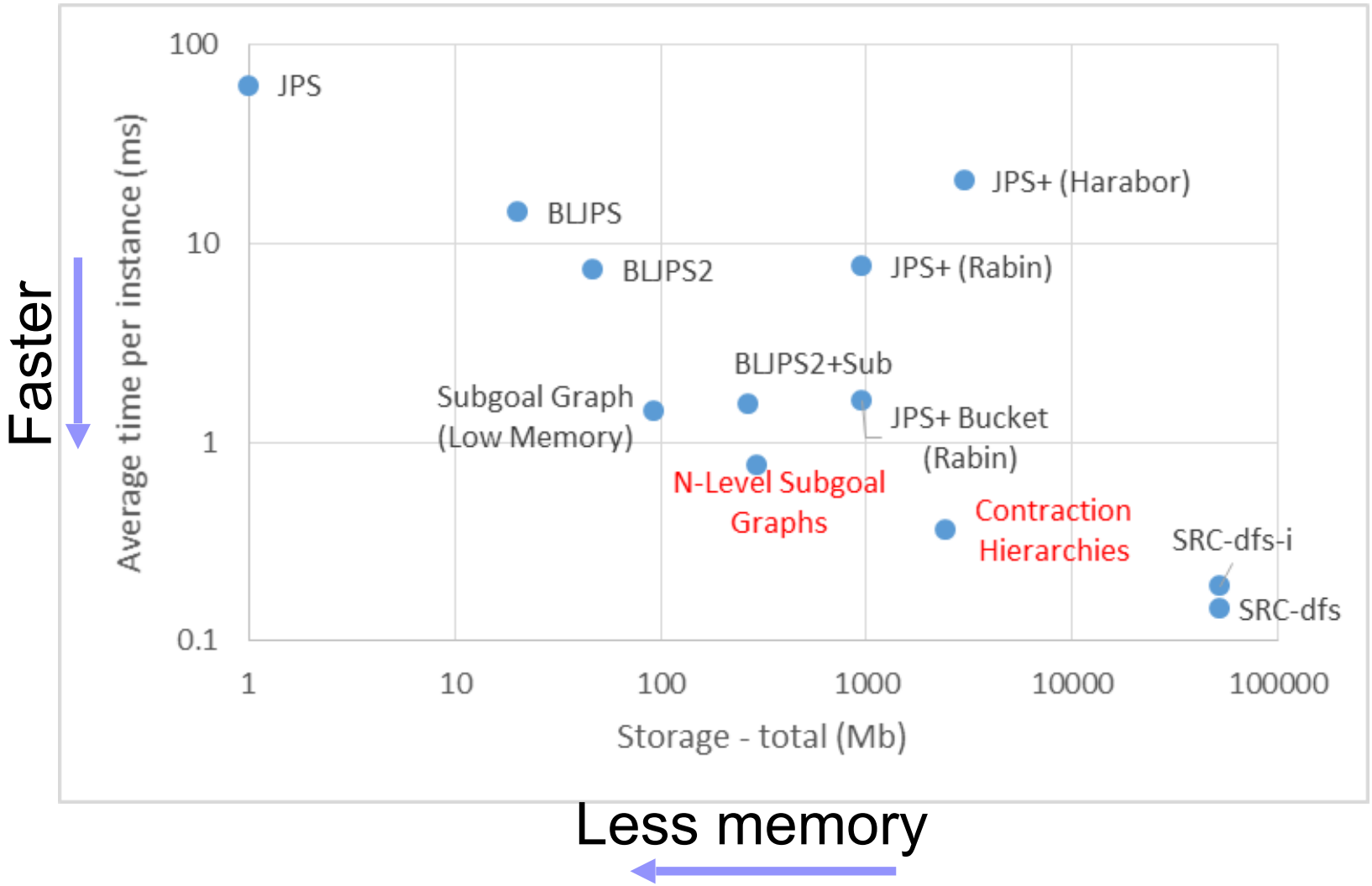


Experimental Results

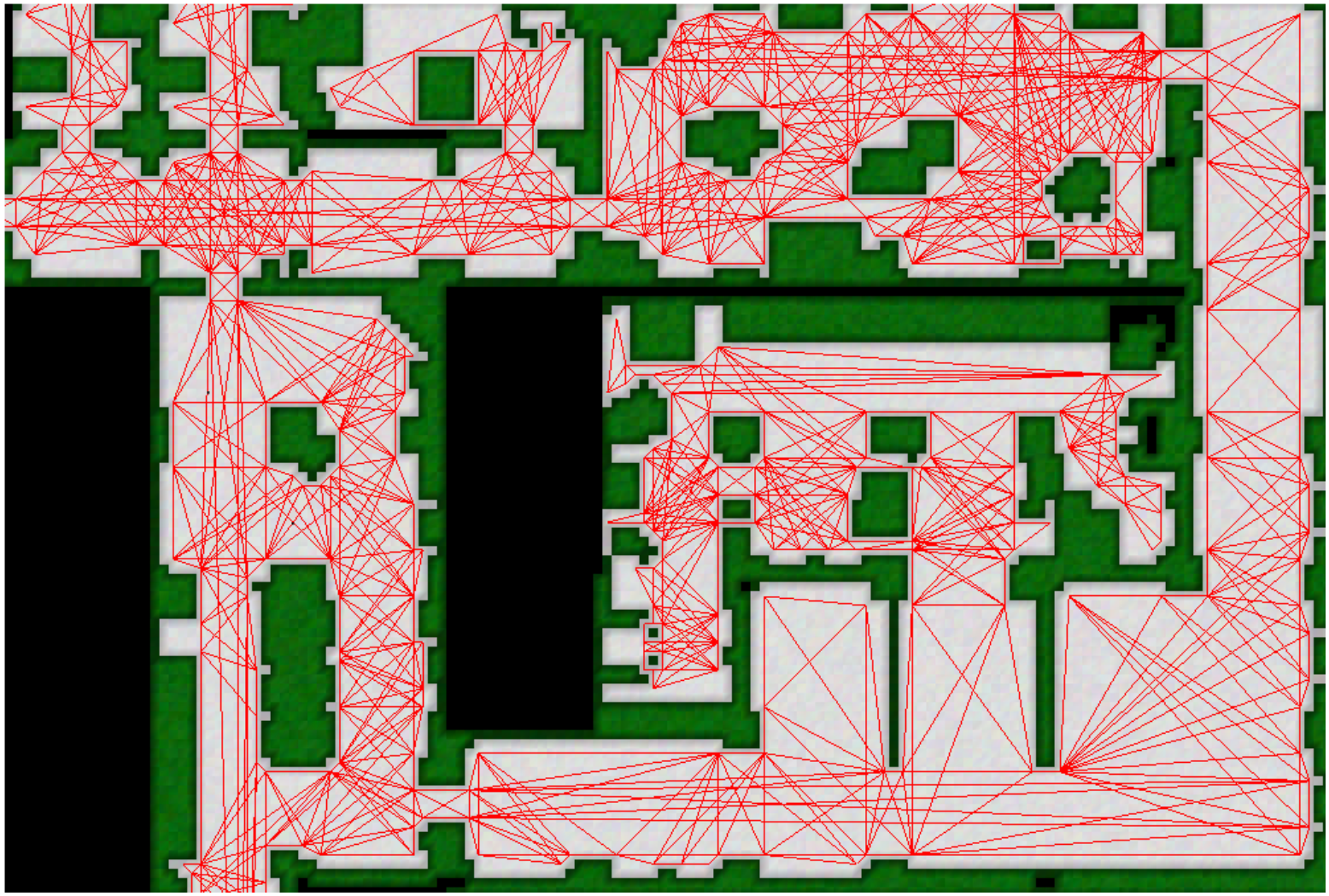
	Preprocessing time (ms)	Storage (MB)	Query time (µs)			Speed up over A*
			Connect	Search	Refine	
game	8.91	1.24	5.20	82.13	2.10	36.63
maze	11.62	1.11	2.24	518.06	12.31	29.91
room	7.30	1.06	2.55	83.36	2.83	78.46
random	64.64	2.27	1.85	1635.98	7.25	2.64

Experimental Results: GPPC 2014

- Optimal algorithms

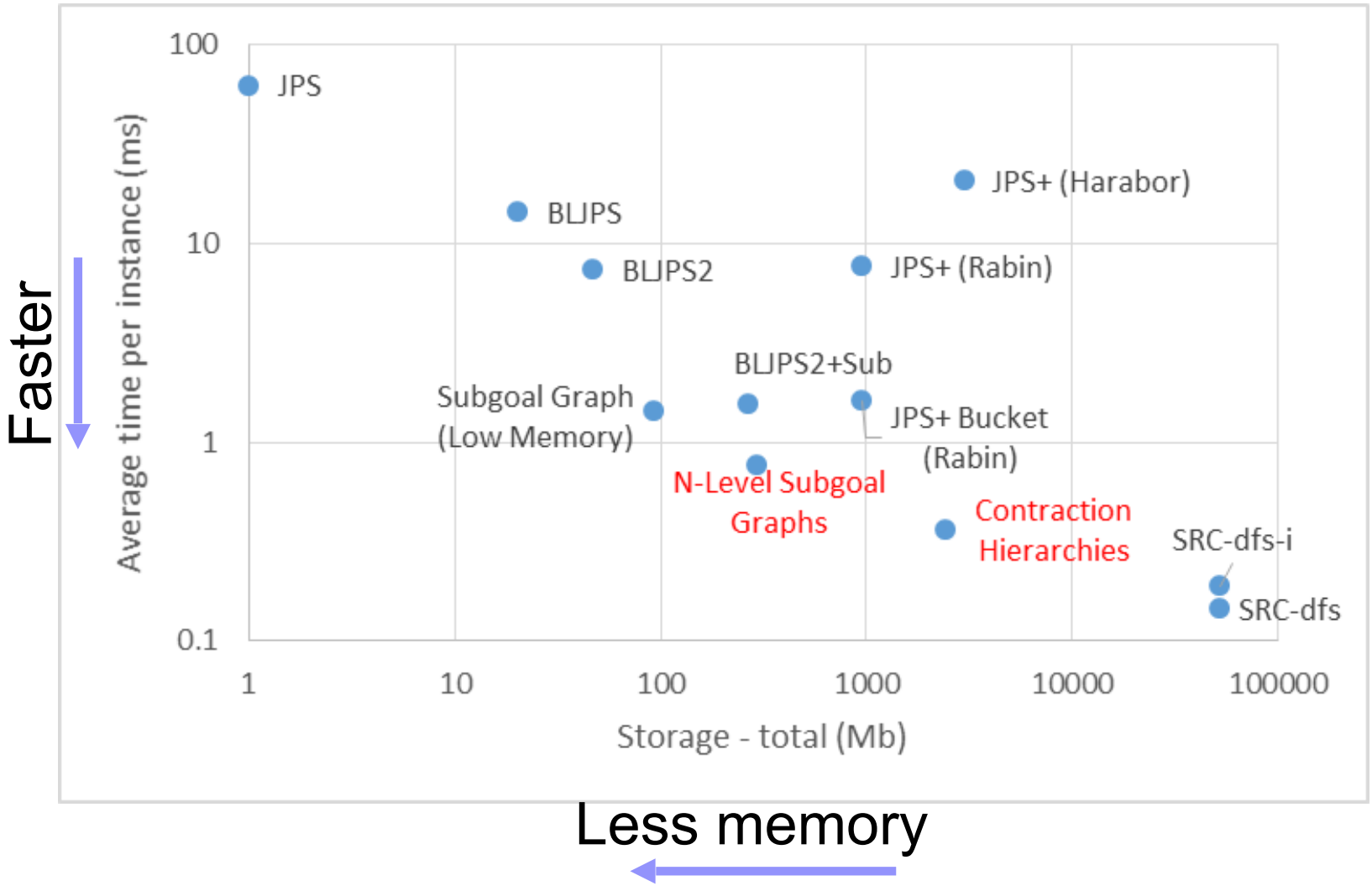


Subgoal Graphs



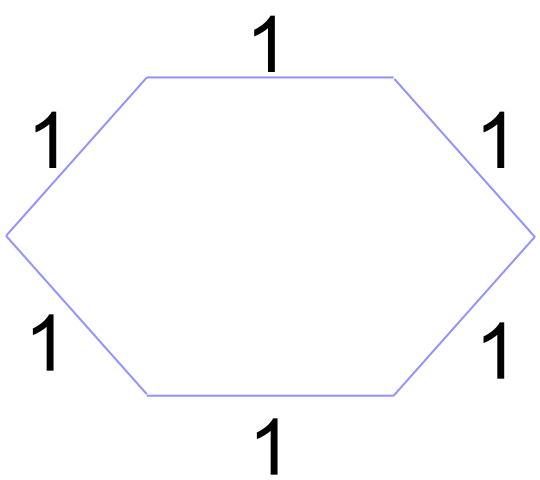
Experimental Results: GPPC 2014

- Optimal algorithms



Contraction Hierarchies

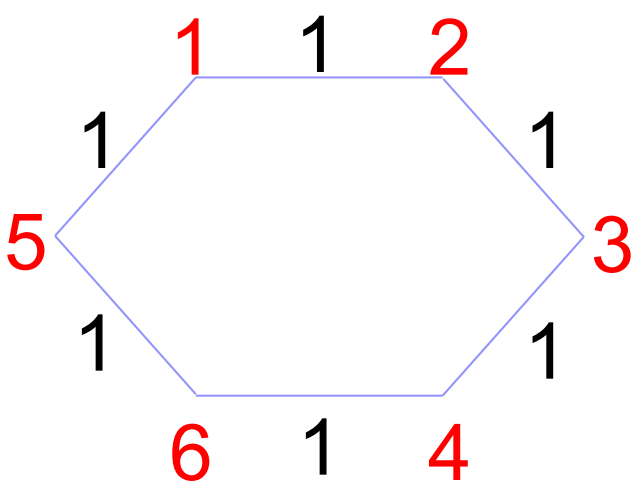
- Offline phase:
  - Transform the graph into a contraction hierarchy



- Geisberger, Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks; Diploma Thesis; U of Karlsruhe; 2008

Contraction Hierarchies

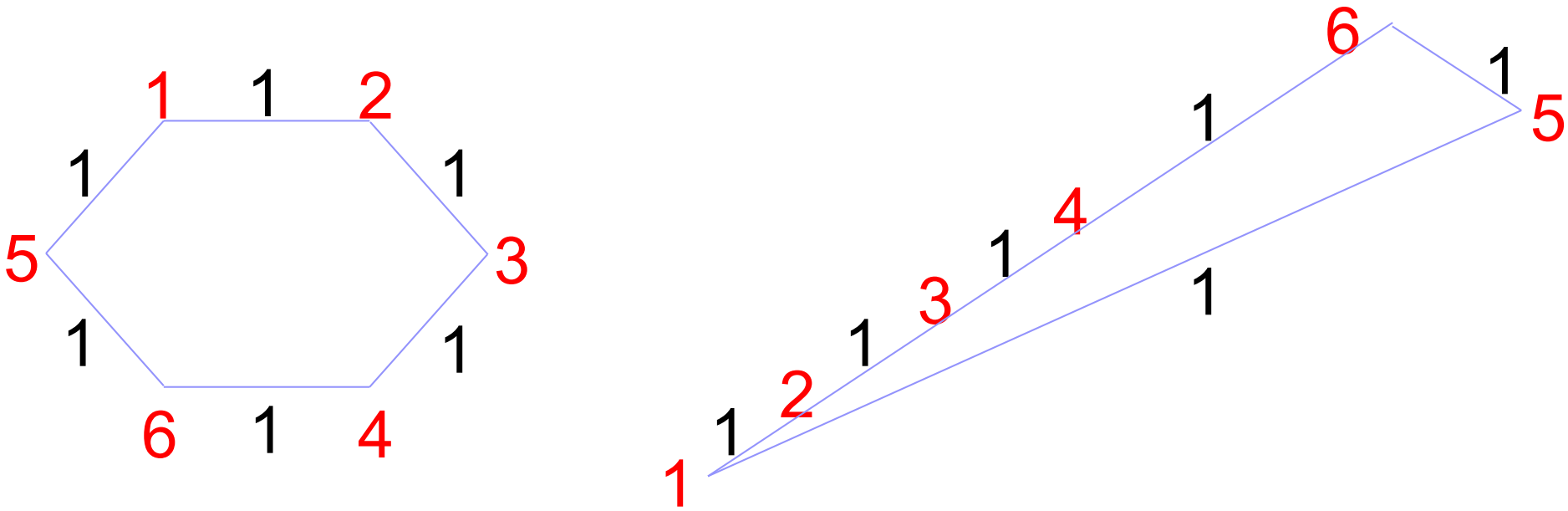
- Offline phase:
  - Transform the graph into a contraction hierarchy



- Geisberger, Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks; Diploma Thesis; U of Karlsruhe; 2008

Contraction Hierarchies

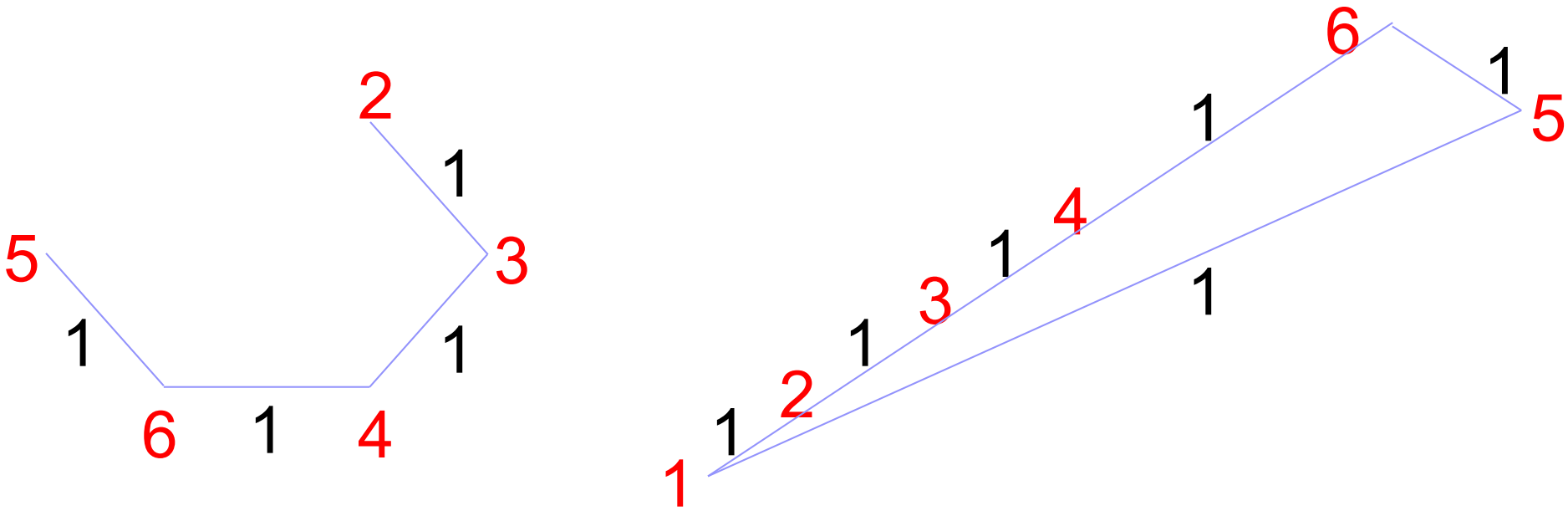
- Offline phase:
  - Transform the graph into a contraction hierarchy



- Geisberger, Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks; Diploma Thesis; U of Karlsruhe; 2008

Contraction Hierarchies

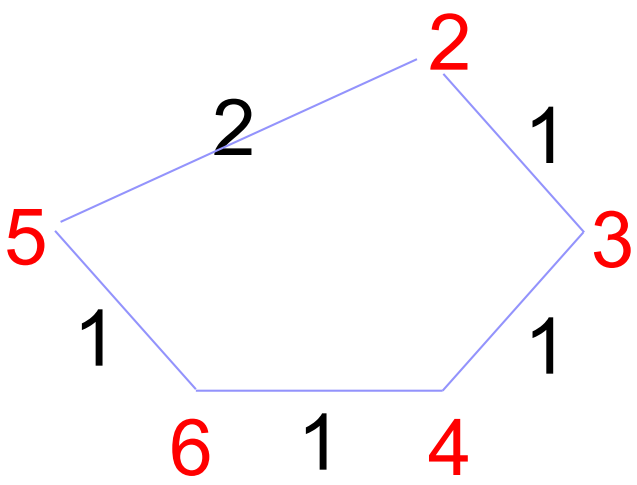
- Offline phase:
  - Transform the graph into a contraction hierarchy

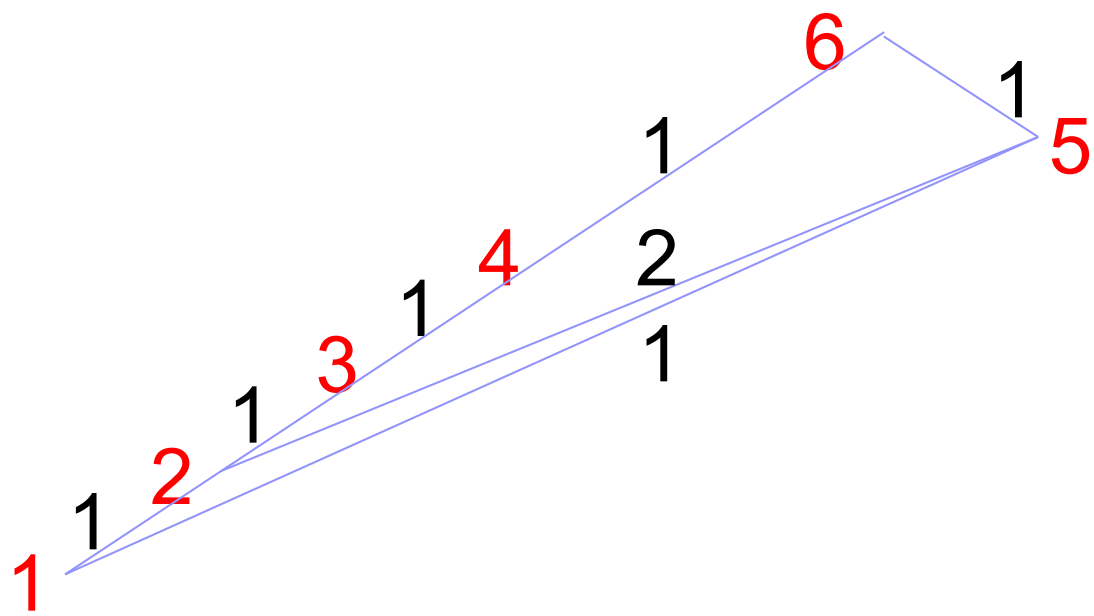


- Geisberger, Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks; Diploma Thesis; U of Karlsruhe; 2008

Contraction Hierarchies

- Offline phase:
  - Transform the graph into a contraction hierarchy

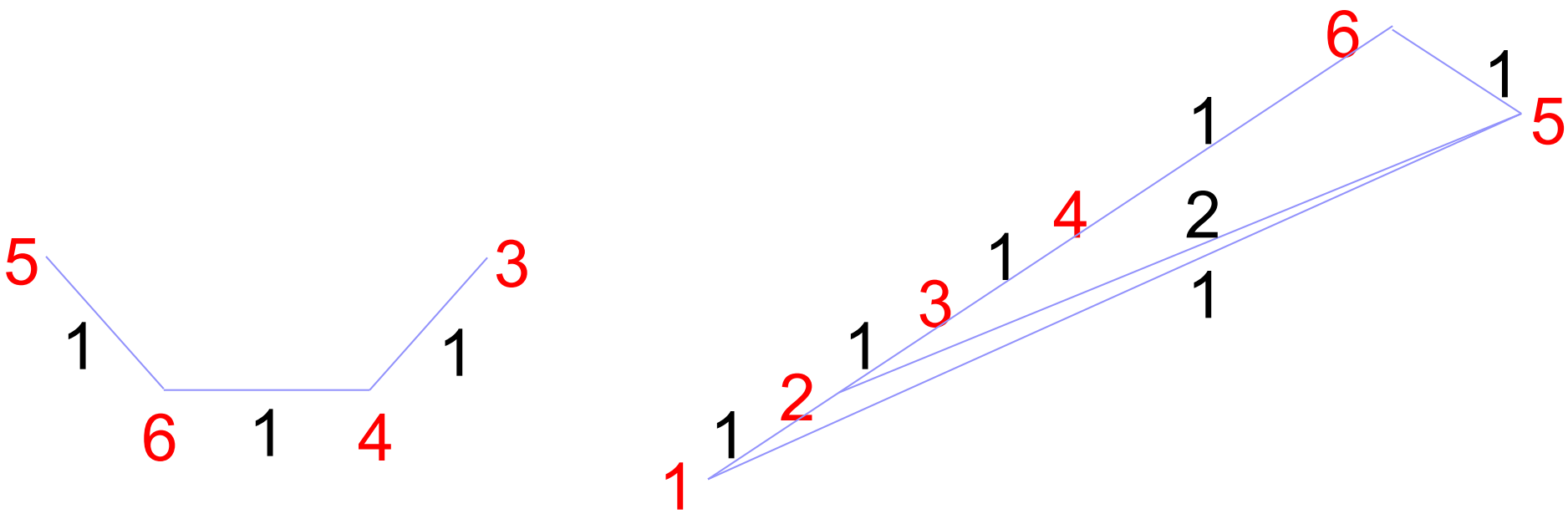




- Geisberger, Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks; Diploma Thesis; U of Karlsruhe; 2008

Contraction Hierarchies

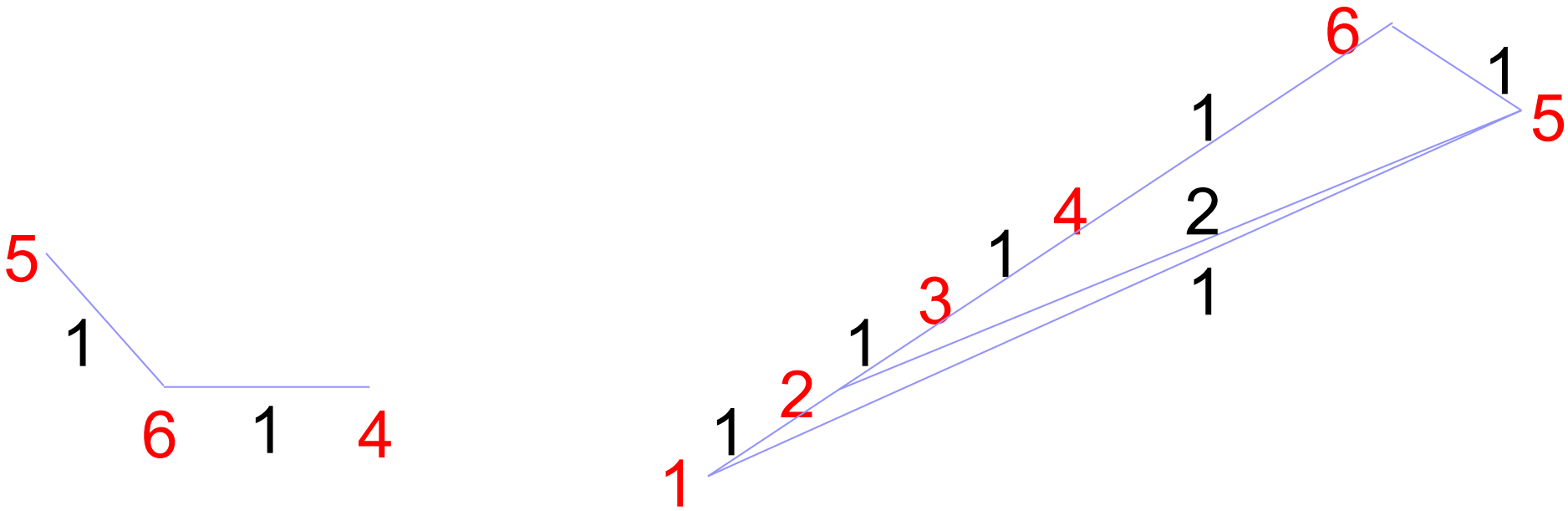
- Offline phase:
  - Transform the graph into a contraction hierarchy



- Geisberger, Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks; Diploma Thesis; U of Karlsruhe; 2008

Contraction Hierarchies

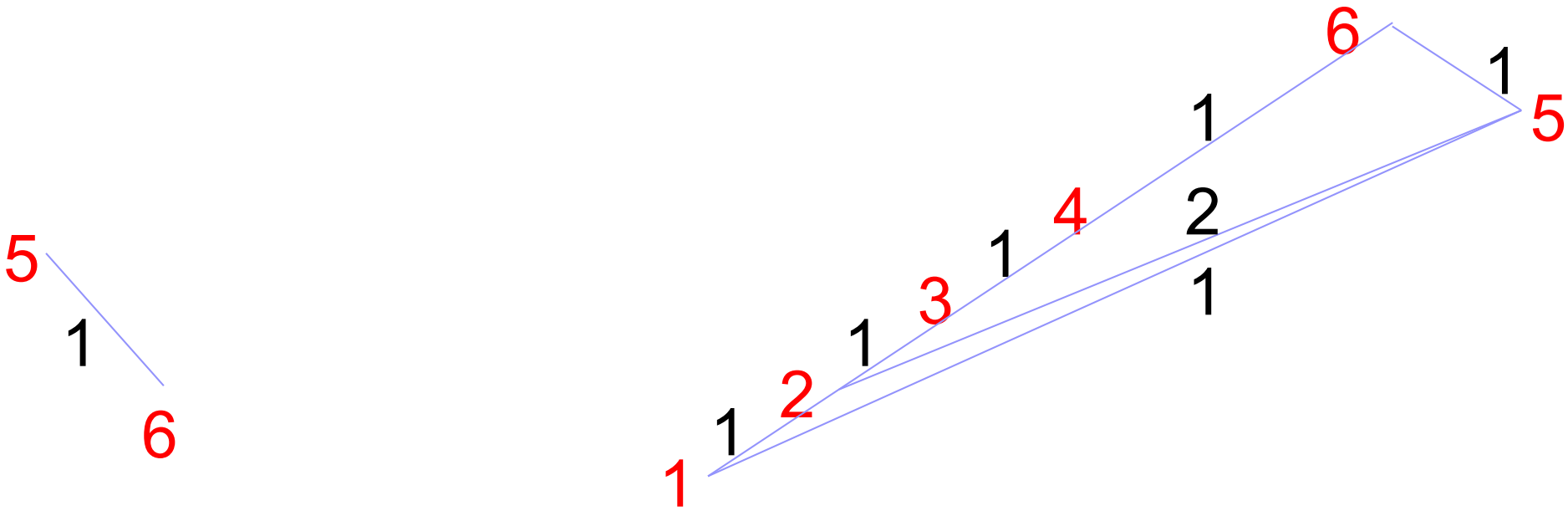
- Offline phase:
  - Transform the graph into a contraction hierarchy



- Geisberger, Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks; Diploma Thesis; U of Karlsruhe; 2008

Contraction Hierarchies

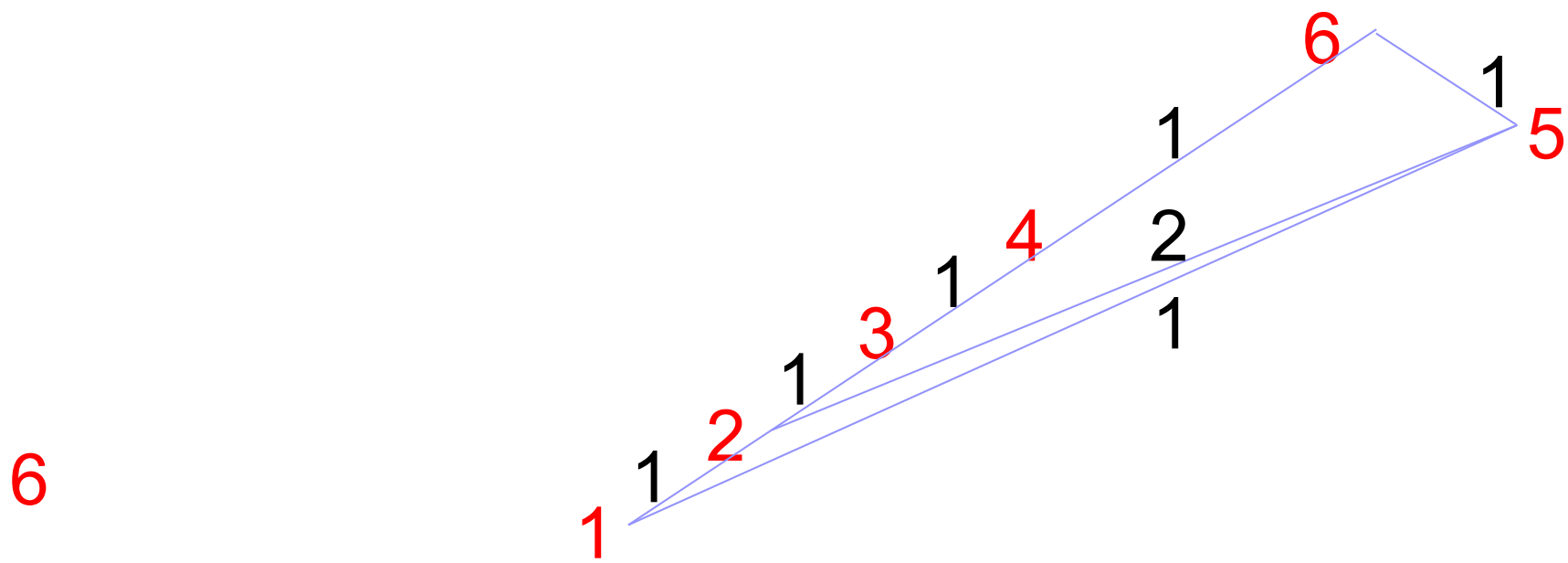
- Offline phase:
  - Transform the graph into a contraction hierarchy



- Geisberger, Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks; Diploma Thesis; U of Karlsruhe; 2008

Contraction Hierarchies

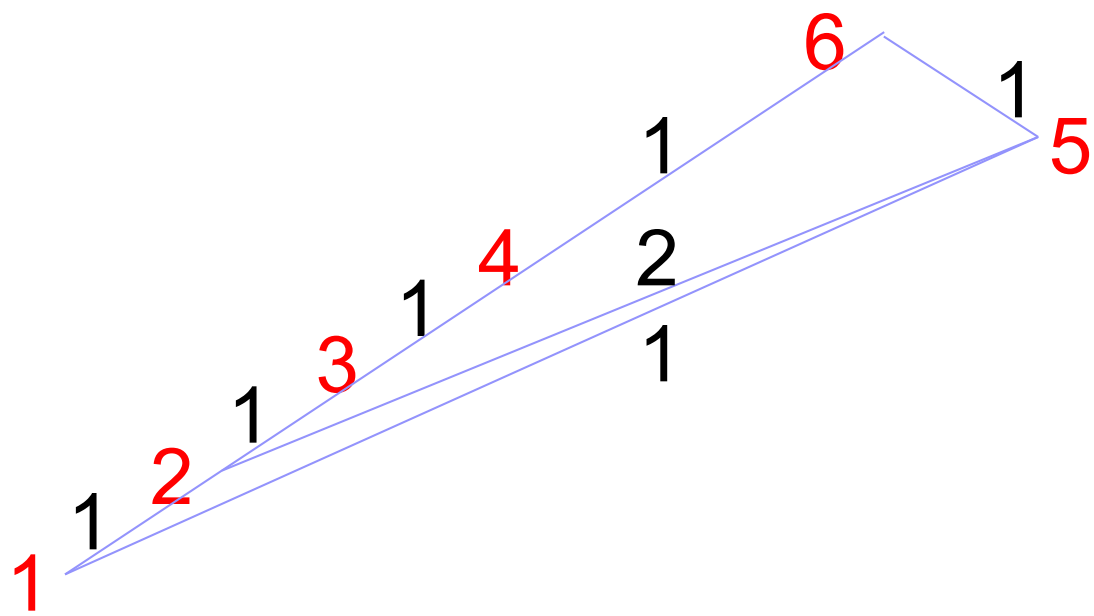
- Offline phase:
  - Transform the graph into a contraction hierarchy



- Geisberger, Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks; Diploma Thesis; U of Karlsruhe; 2008

Contraction Hierarchies

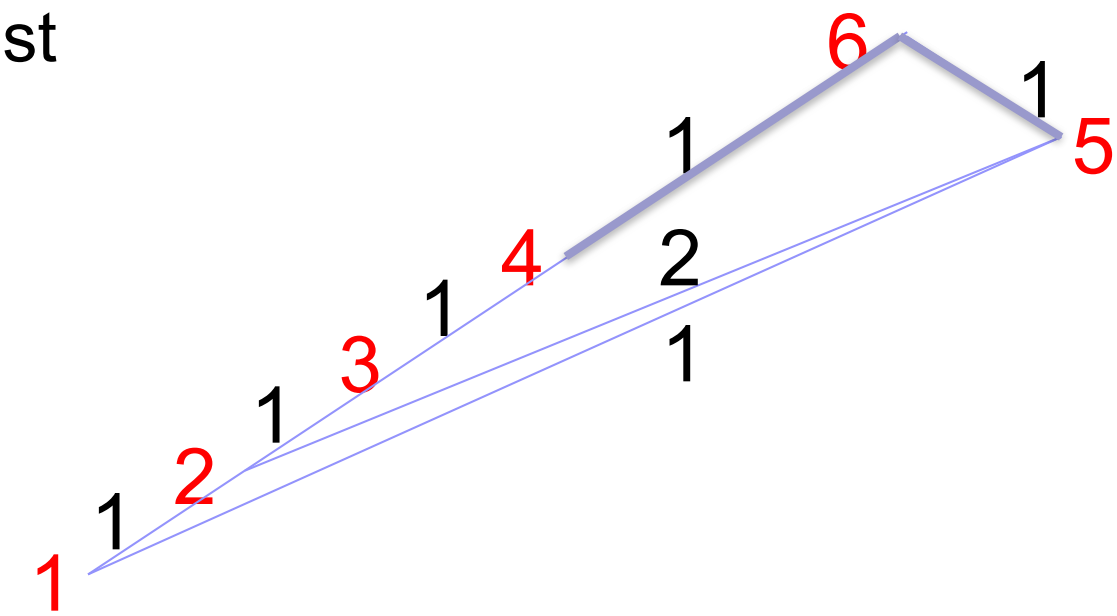
- Offline phase:
  - Transform the graph into a contraction hierarchy



- Geisberger, Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks; Diploma Thesis; U of Karlsruhe; 2008

Contraction Hierarchies

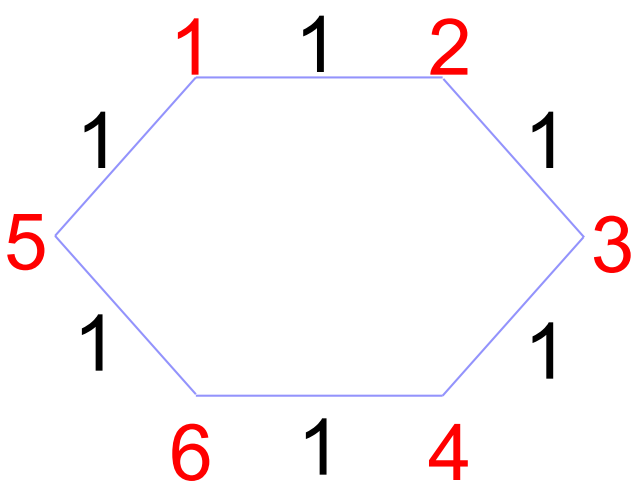
- Online phase: Find a shortest path from 4 to 5
  - One only needs to consider all shortest up-down paths and thus can ignore most of the graph, which makes the search very fast

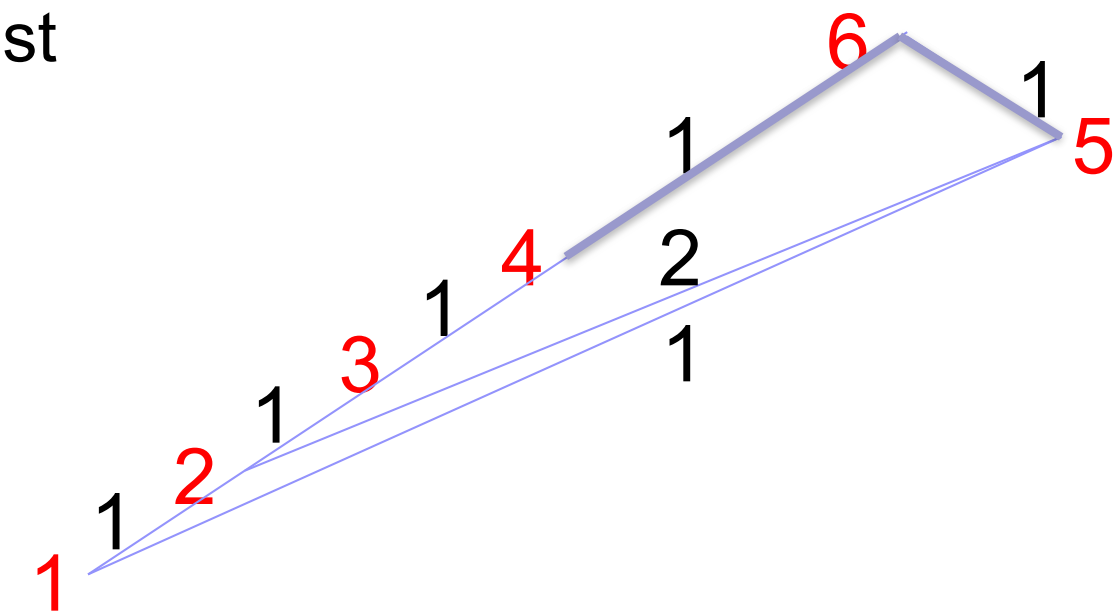


- Geisberger, Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks; Diploma Thesis; U of Karlsruhe; 2008

Contraction Hierarchies

- Online phase: Find a shortest path from 4 to 5
  - One only needs to consider all shortest up-down paths and thus can ignore most of the graph, which makes the search very fast

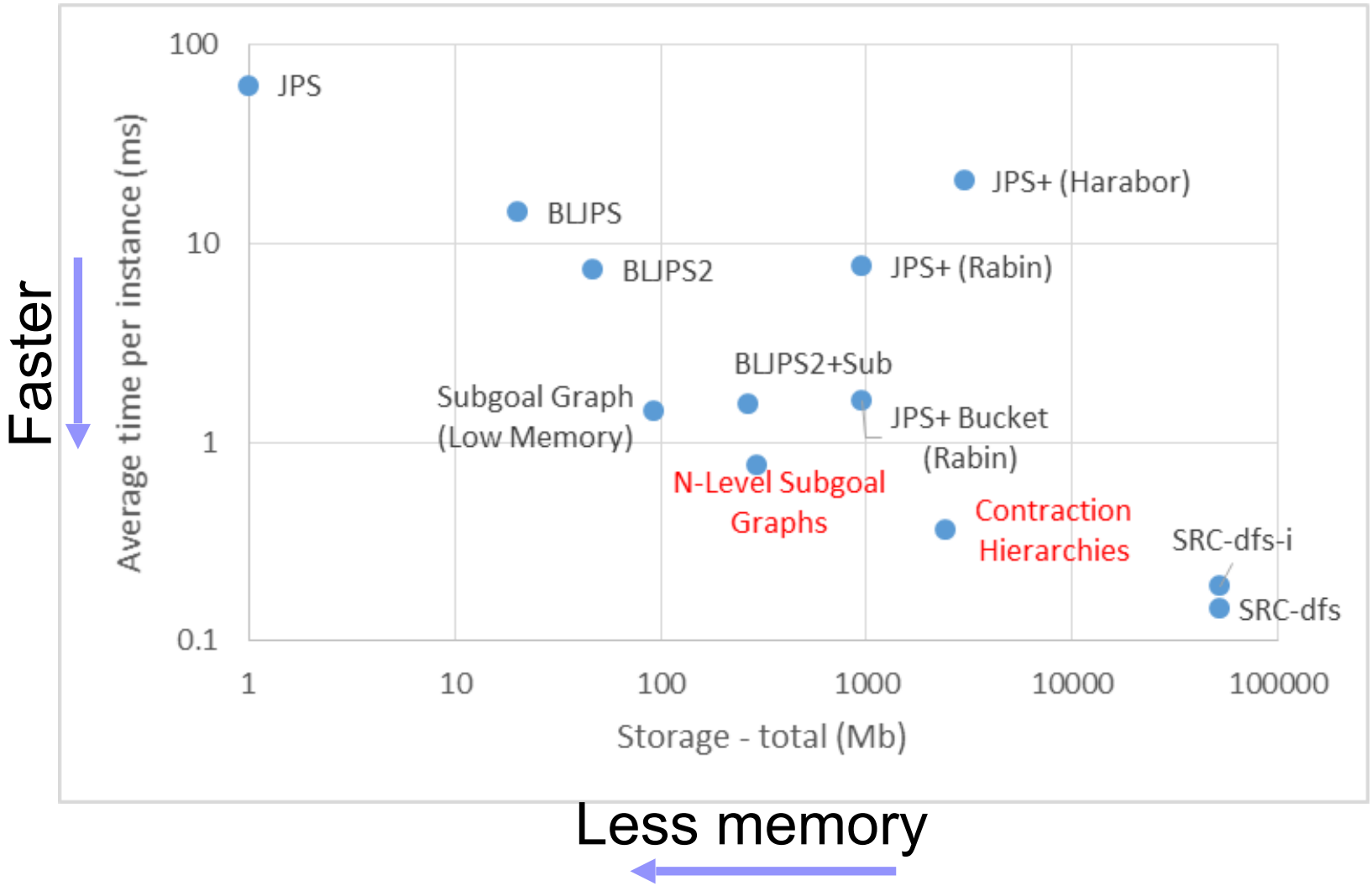




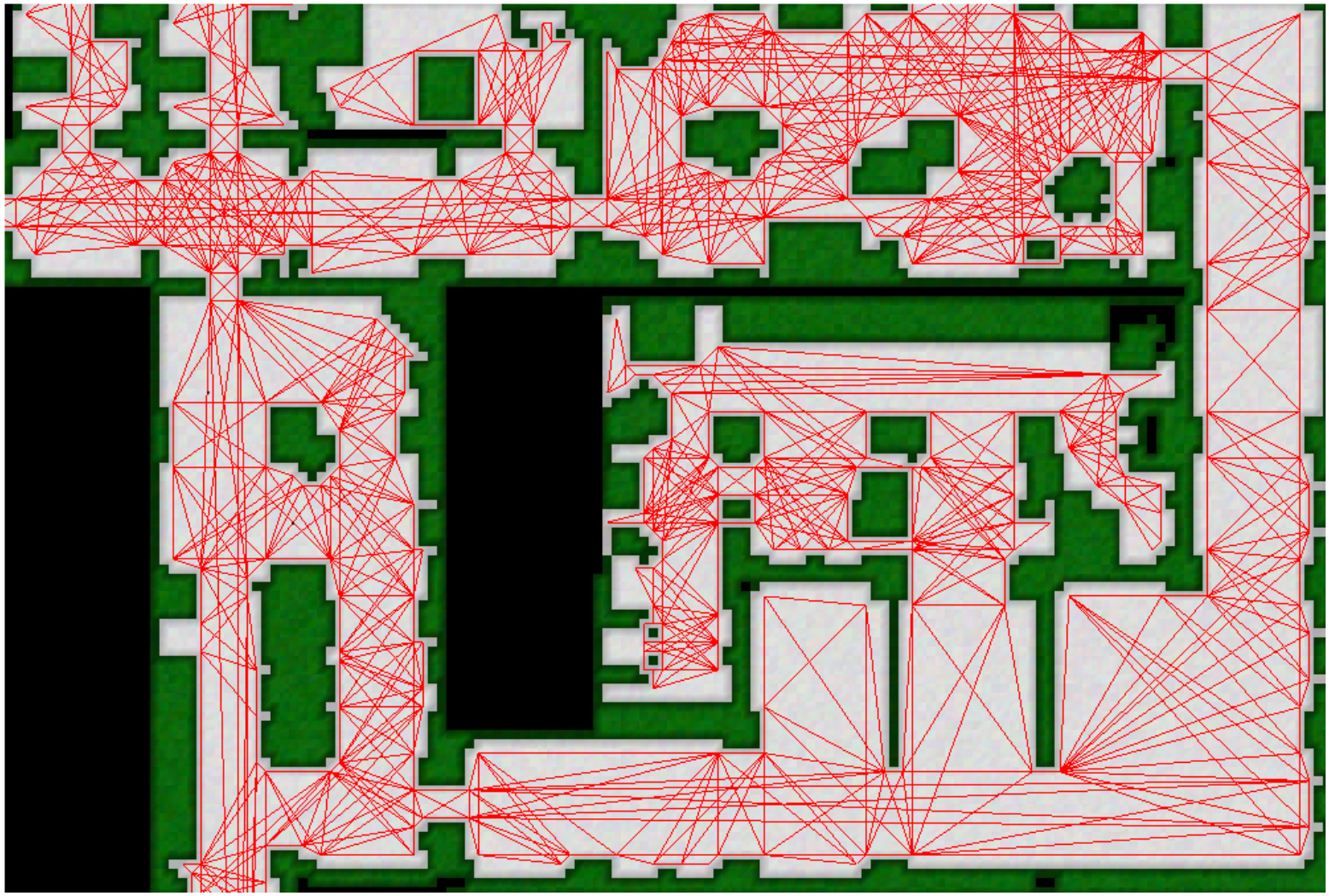
- Geisberger, Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks; Diploma Thesis; U of Karlsruhe; 2008

Experimental Results: GPPC 2014

- Optimal algorithms

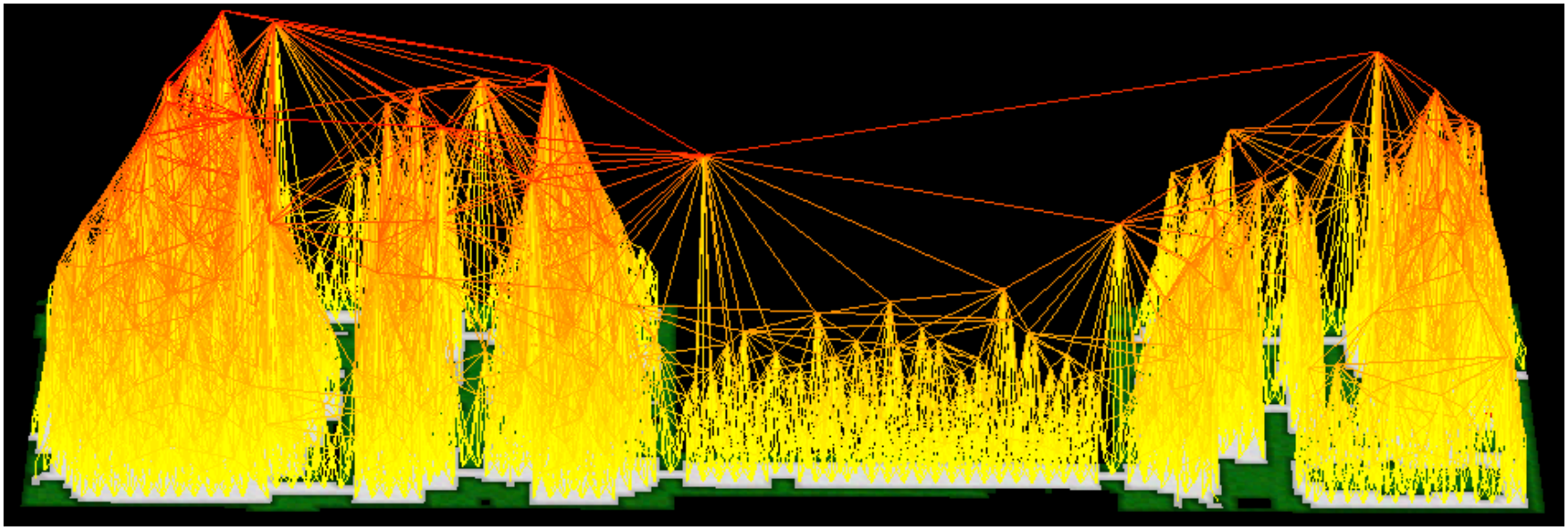


Subgoal Graphs

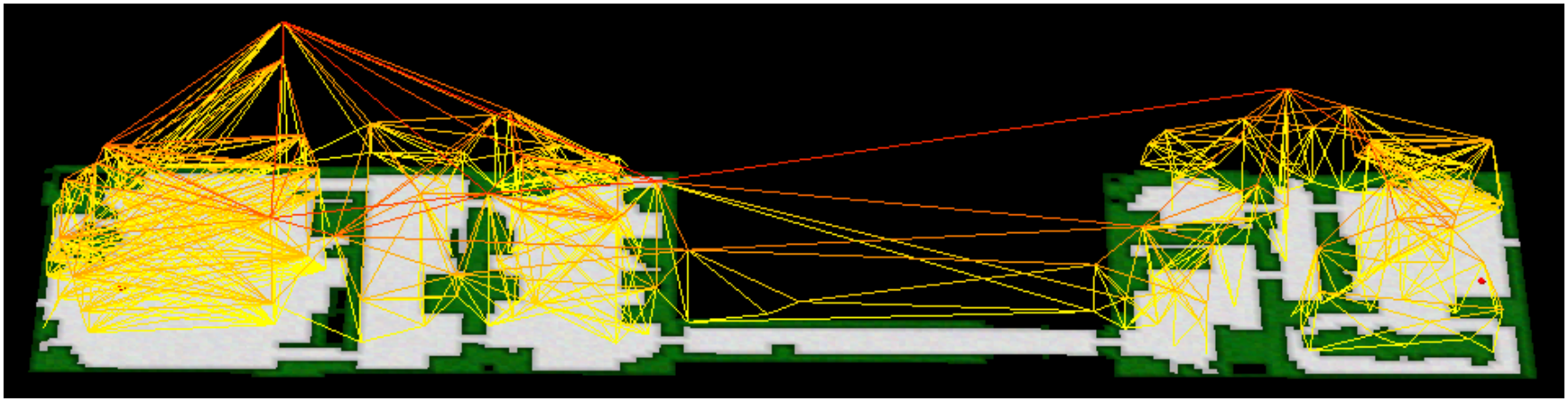


Contraction Hierarchies on Subgoal Gs

- Contraction hierarchies: 4,543 nodes and 32,552 edges

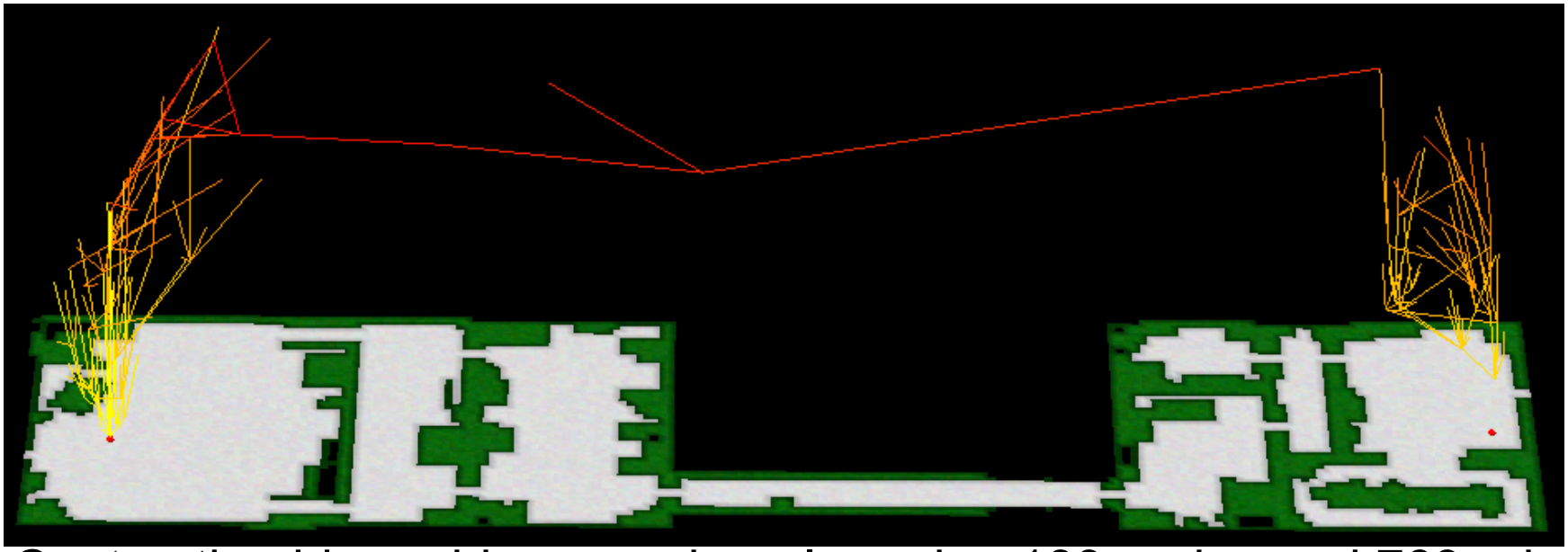


- Contraction hierarchies on subgoal graphs: 183 nodes and 763 edges

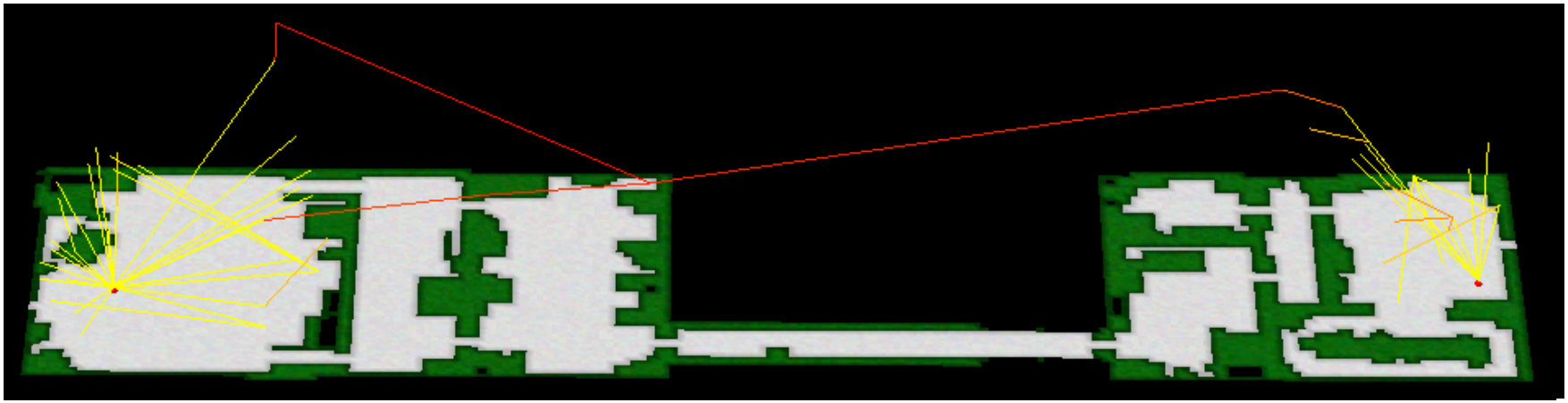


Contraction Hierarchies on Subgoal Gs

- Contraction hierarchies: 4,543 nodes and 32,552 edges



- Contraction hierarchies on subgoal graphs: 183 nodes and 763 edges



Contraction Hierarchies on Subgoal Gs

	% discarded over CH		Factor of improvement over CH				
	Nodes	Edges	Search	Refine	Connect + Search	Connect + Search + Refine	Memory
game	97.98	97.75	5.18	2.22	3.67	3.39	10.85
maze	94.44	98.10	10.39	1.68	6.19	2.69	12.83
room	98.18	98.83	4.61	1.82	3.59	3.01	17.78
random	60.40	53.37	1.07	0.79	1.03	0.96	1.84

- Uras and Koenig; Understanding Subgoal Graphs by Augmenting Contraction Hierarchies; Proceedings of the International Joint Conference on Artificial Intelligence; 2018.

Contraction Hierarchies

Contraction Hierarchies and Constraints

Daniel D. Harabor
Monash University
http://harabor.net/daniel

Joint work with Peter Stuckey

AAMAS Tutorial
2019-05-14

The BCH Query Algorithm

BCH = Bi-directional Dijkstra Search + Contraction Hierarchies.
Introduced in [Geisberger *et al.*, 2008; Geisberger *et al.*, 2012].

Offline

Divide the contracted graph $G = (V, E)$ into two:

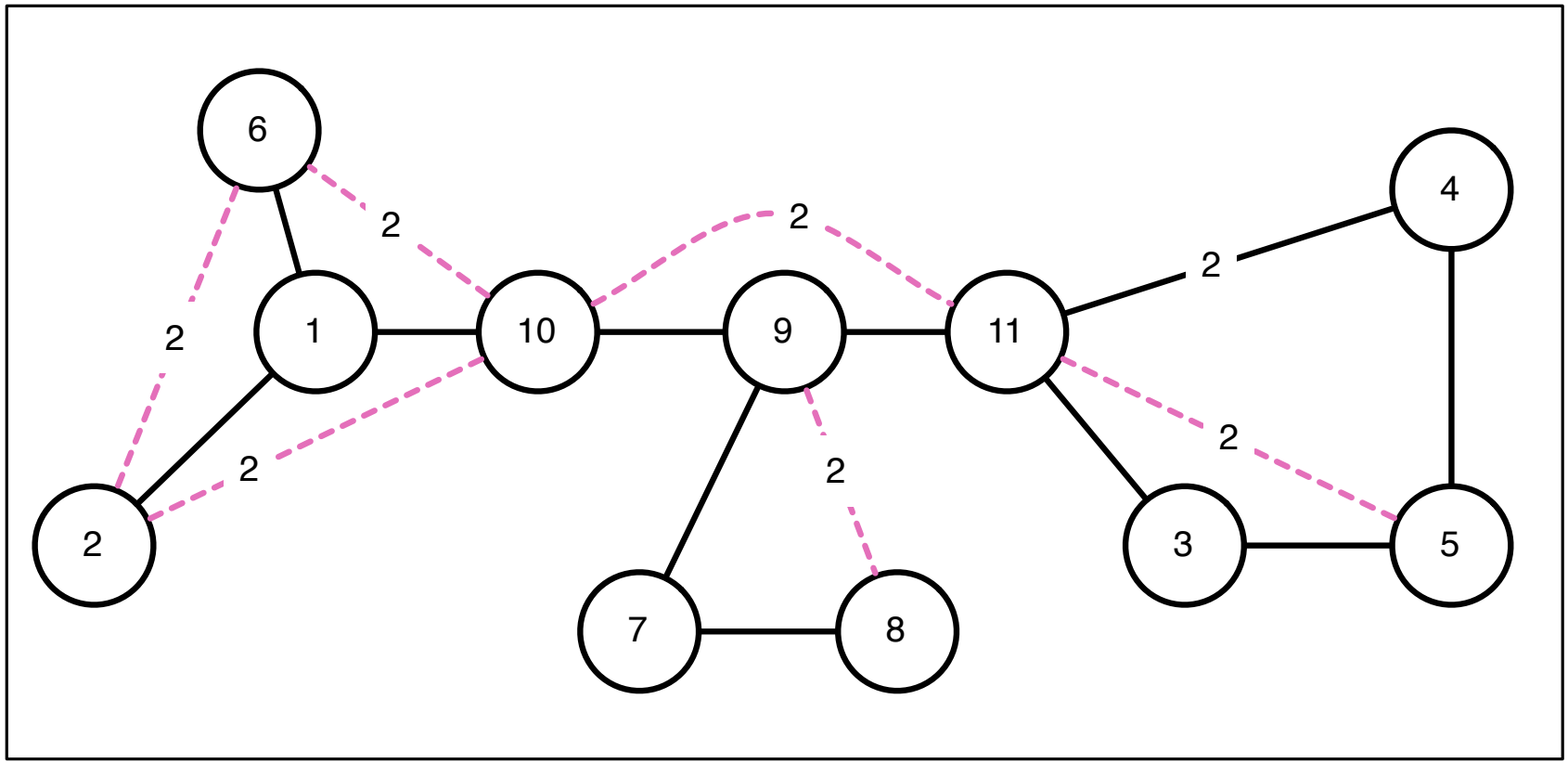
- $G_\uparrow = (V, E_\uparrow = \{(u, v) \in E \mid u \prec v\})$
- $G_\downarrow = (V, E_\downarrow = \{(u, v) \in E \mid u \succ v\})$
- $\prec$ and $\succ$ compare the contraction order of pairs of nodes.

Online

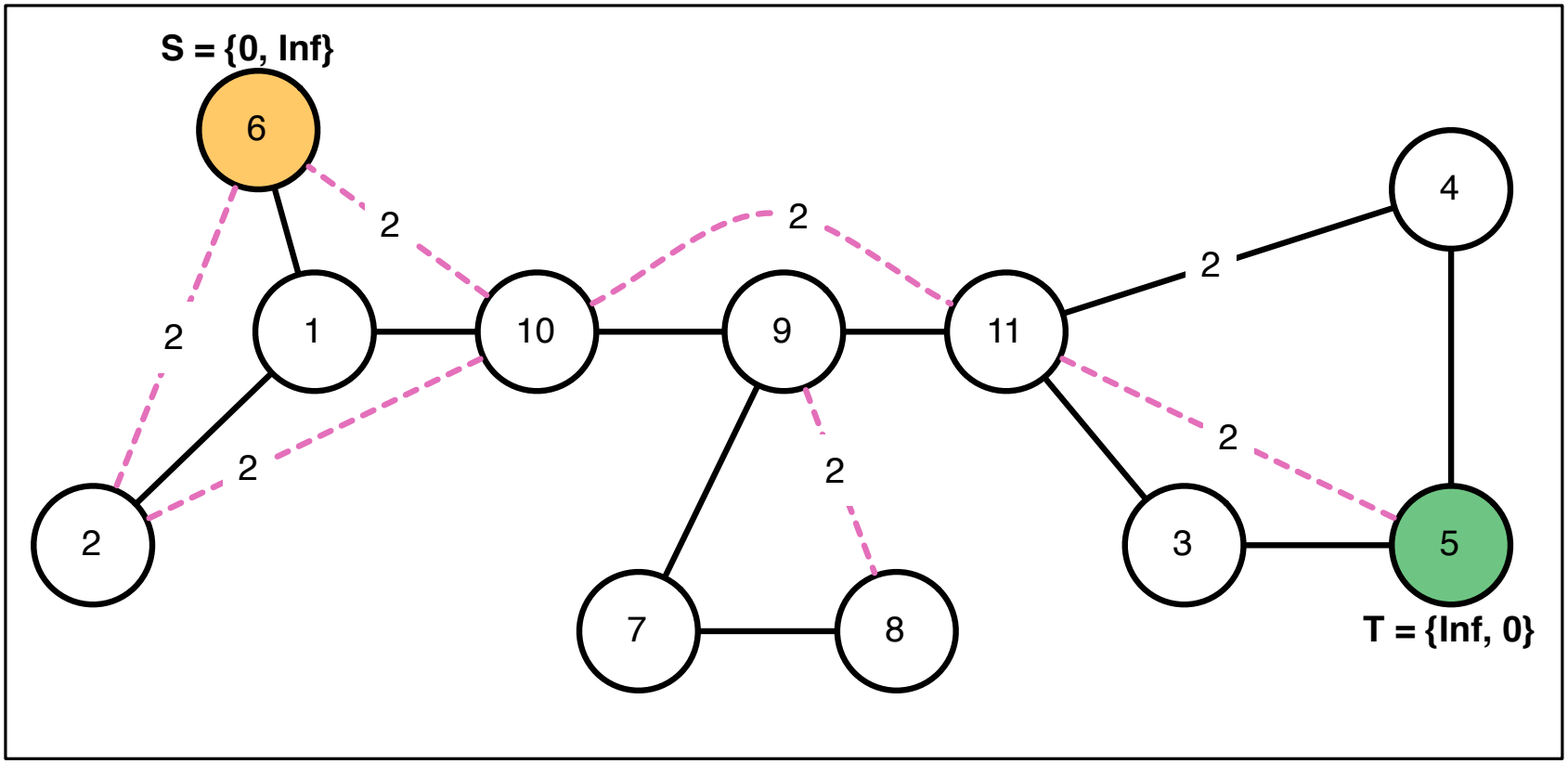
Perform a bi-directional Dijkstra search:

- Forward search in $G_\uparrow$ (relax only outgoing "up" edges).
- Backward search in $G_\downarrow$; (relax only incoming "up" edges).
- Expansions can be interleaved or sequential.

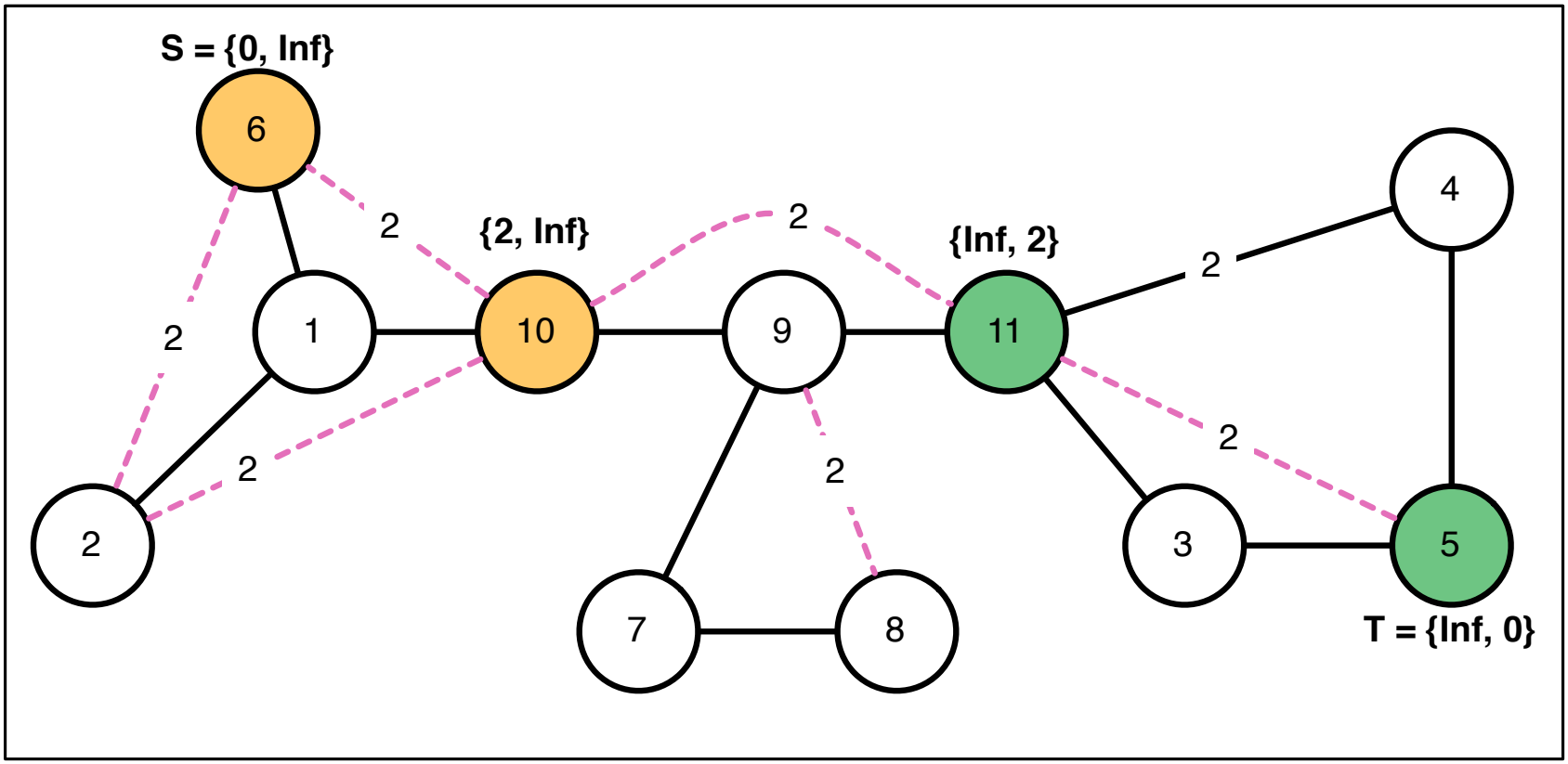
BCH Example



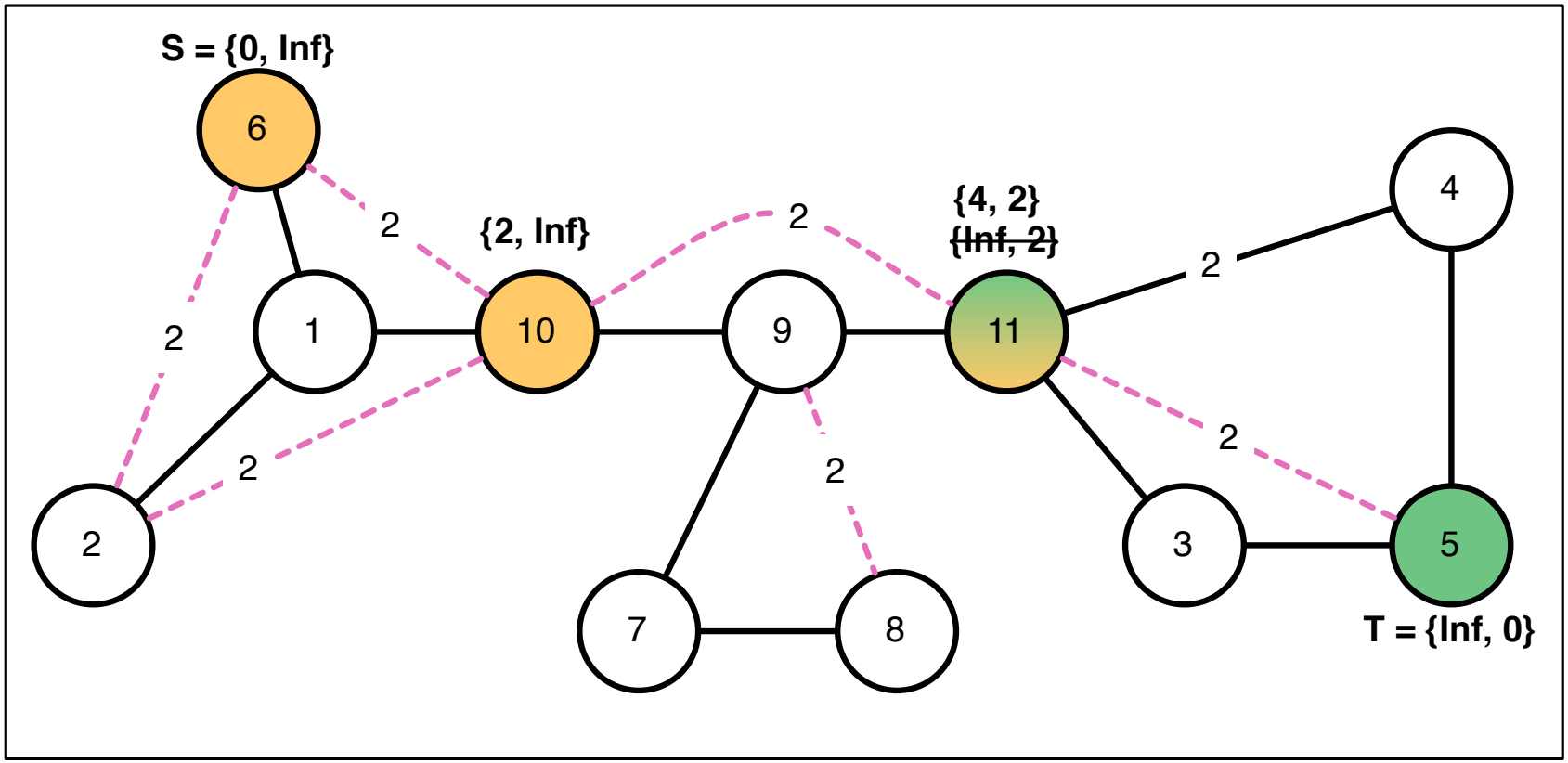
BCH Example



BCH Example



BCH Example



Deconstructing BCH

The following results appear in [Geisberger *et al.*, 2008]:

ch-path

For every *optimal path* in G there exists a cost equivalent path with prefix $\langle s, ..., k \rangle$ found in $G_{\uparrow}$ and suffix $\langle k, ...t \rangle$ found in $G_{\downarrow}$.

apex-node

Every *ch-path* has a node which is lexically largest among all nodes in the path.

Deconstructing BCH

CH-SPSP [Harabor and Stuckey, 2018]

Find a path $\langle s = v_1, \ldots, v_k, \ldots v_n = t \rangle$ where

$$\min \sum_{i=1}^{(n-1)} c_{v_i, v_{i+1}}$$

Subject to:

1. $v_i \prec v_{i+1}$ for all $1 \leq i < k$
2. $v_i \succ v_{i+1}$ for all $k \leq i < n$

Graph Traversal Policies

Successor Types

1. up-up successors
2. up-down successors
3. down-down successors
4. down-up successors

Graph Traversal Policies

Successor Types

1. up-up successors
2. up-down successors
3. down-down successors
4. ~~down-up successors~~

Graph Traversal Policies

Successor Types

1. up-up successors
2. up-down successors
3. down-down successors
4. ~~down-up successors~~

Traversal Strategies

- Always Up: expand successors of type 1 and 3. This strategy is employed by the bi-directional algorithm **BCH**.
- Up-then-Down: expand successors of type 1, 2 and 3. We combine this strategy with A* search to derive the uni-directional query algorithm **FCH**.

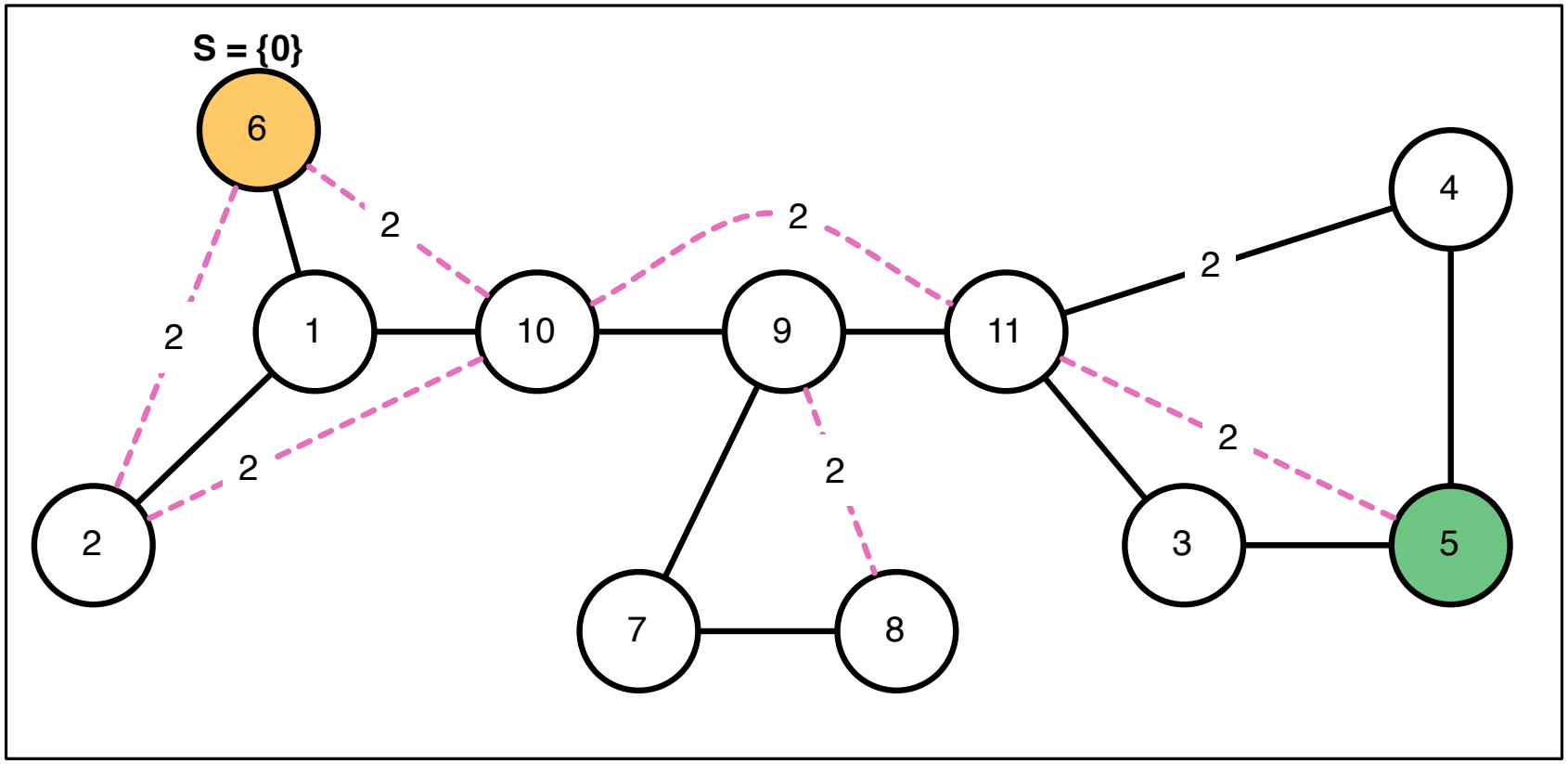
The FCH Query Algorithm

FCH is an variation on A* search which computes only optimal *ch-paths*. Query performance is similar to plain A* (seconds).

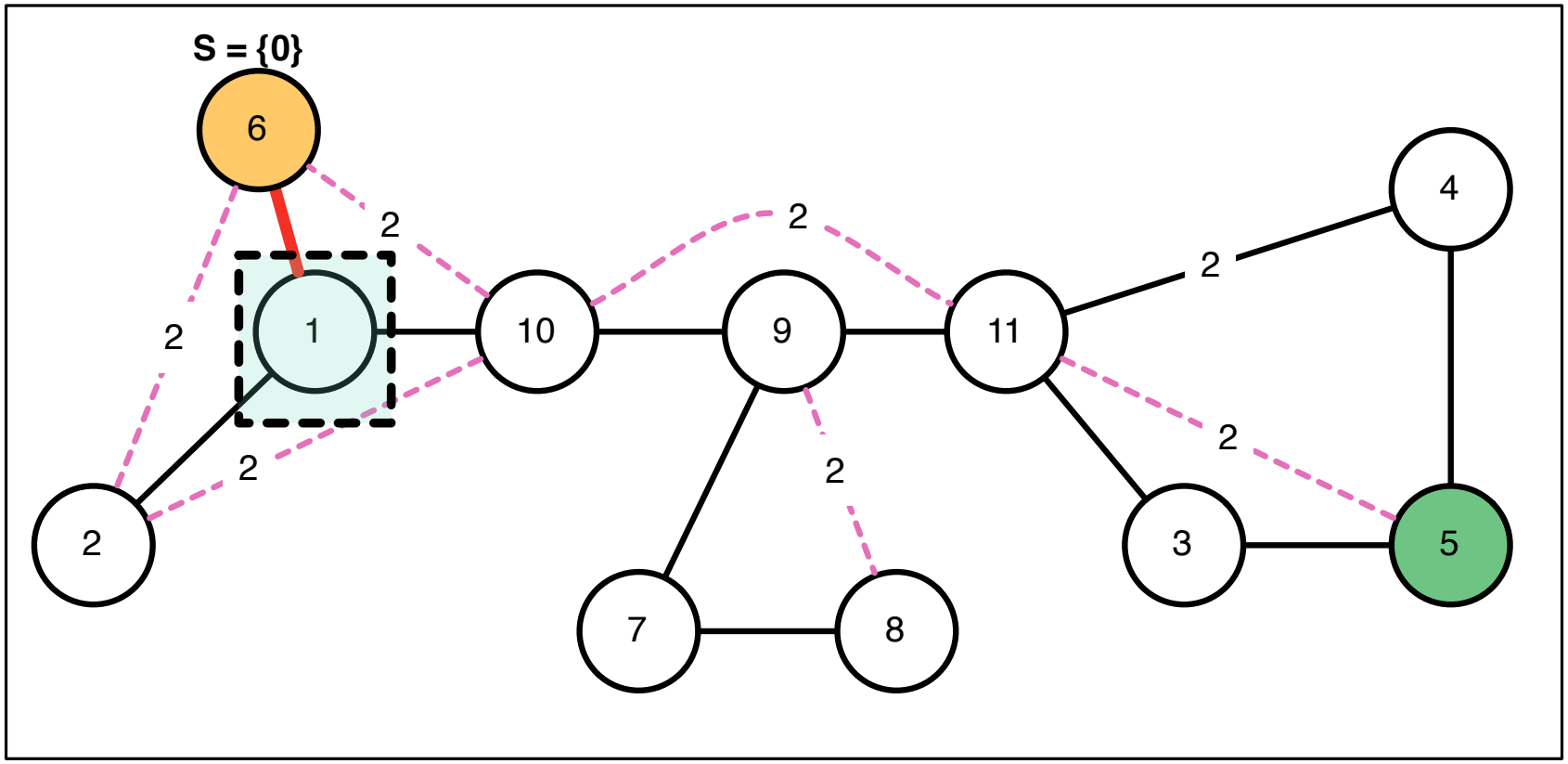
FCH can be easily and effectively combined with many standard pruning methods including Bounding Boxes. This algorithm:

- Reasons about the target in relation to the current node (e.g. could *this* edge appear on an optimal path?)
- Requires preprocessing (i.e. extra time and extra memory).
- Is a type of Geometric Container [Wagner *et al.*, 2005]

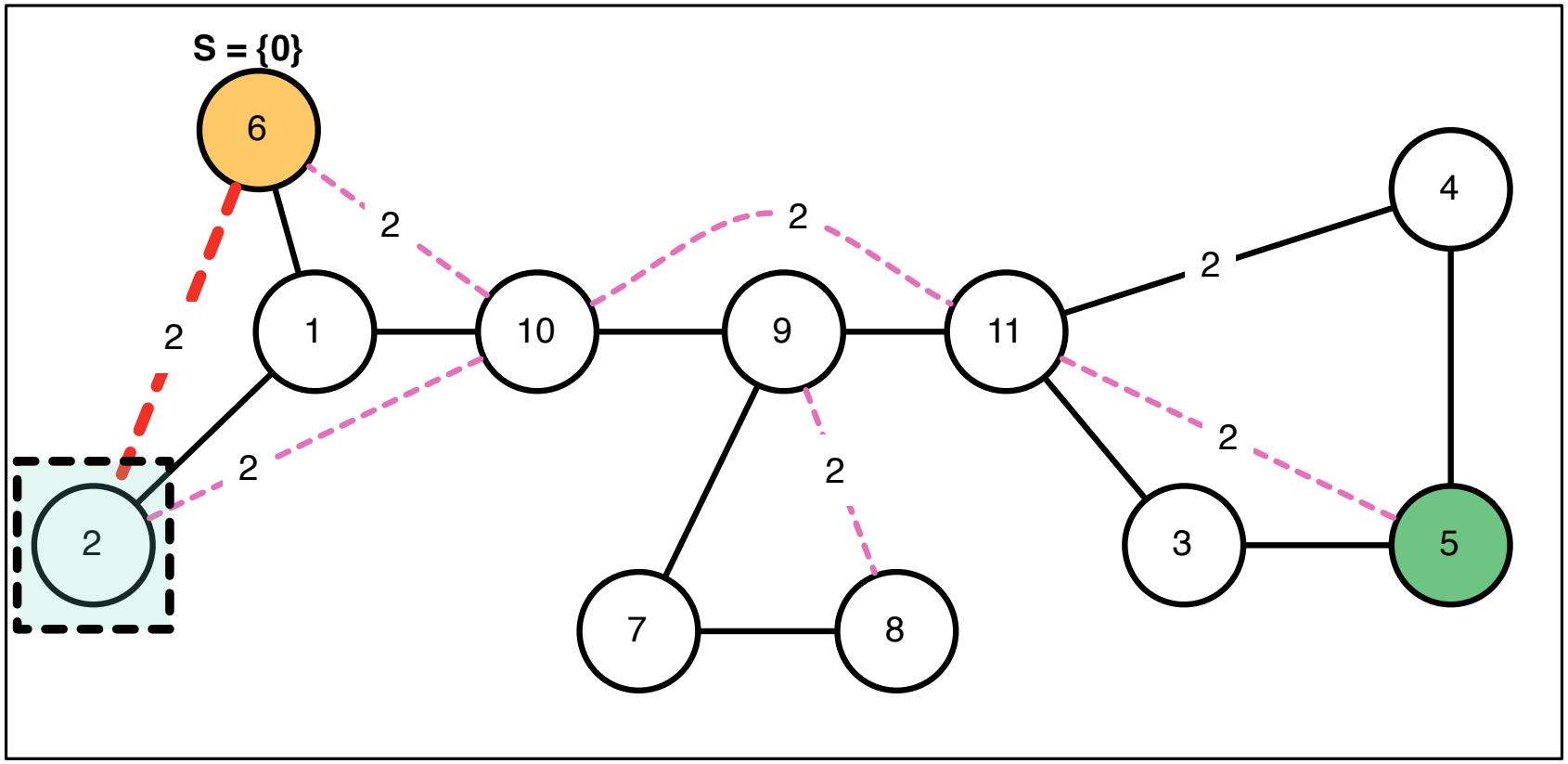
FCH+BB Example



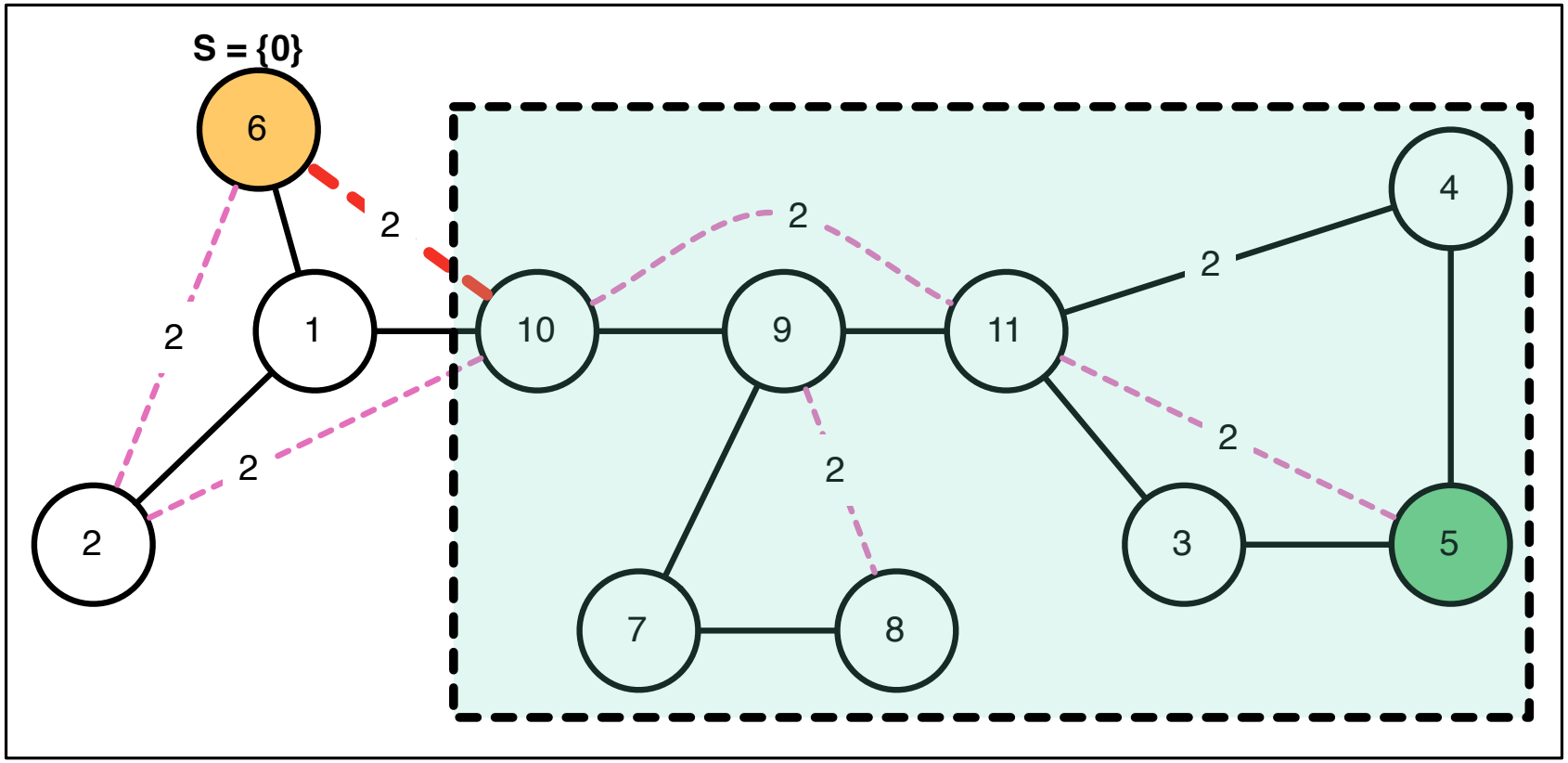
FCH+BB Example



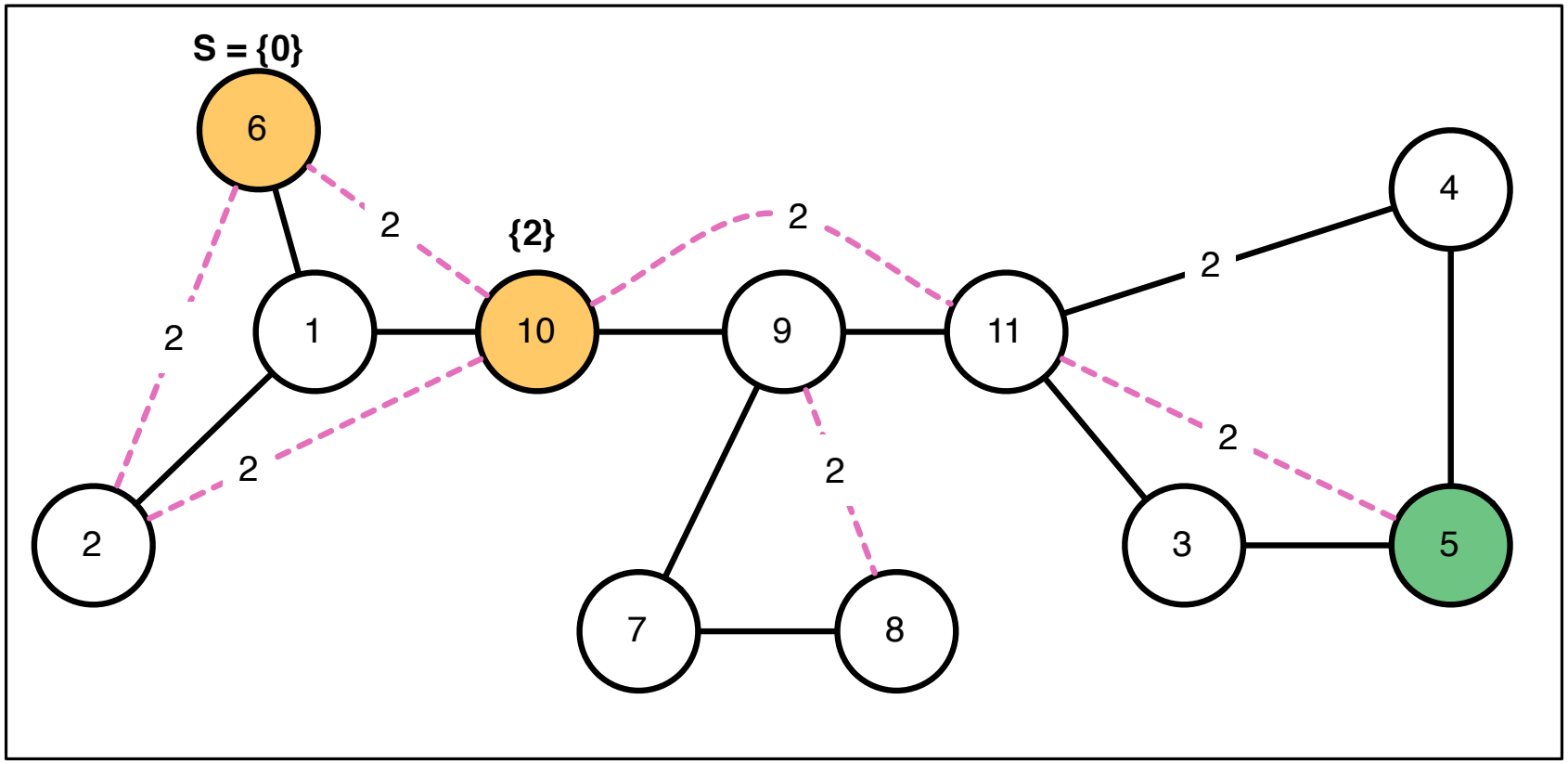
FCH+BB Example



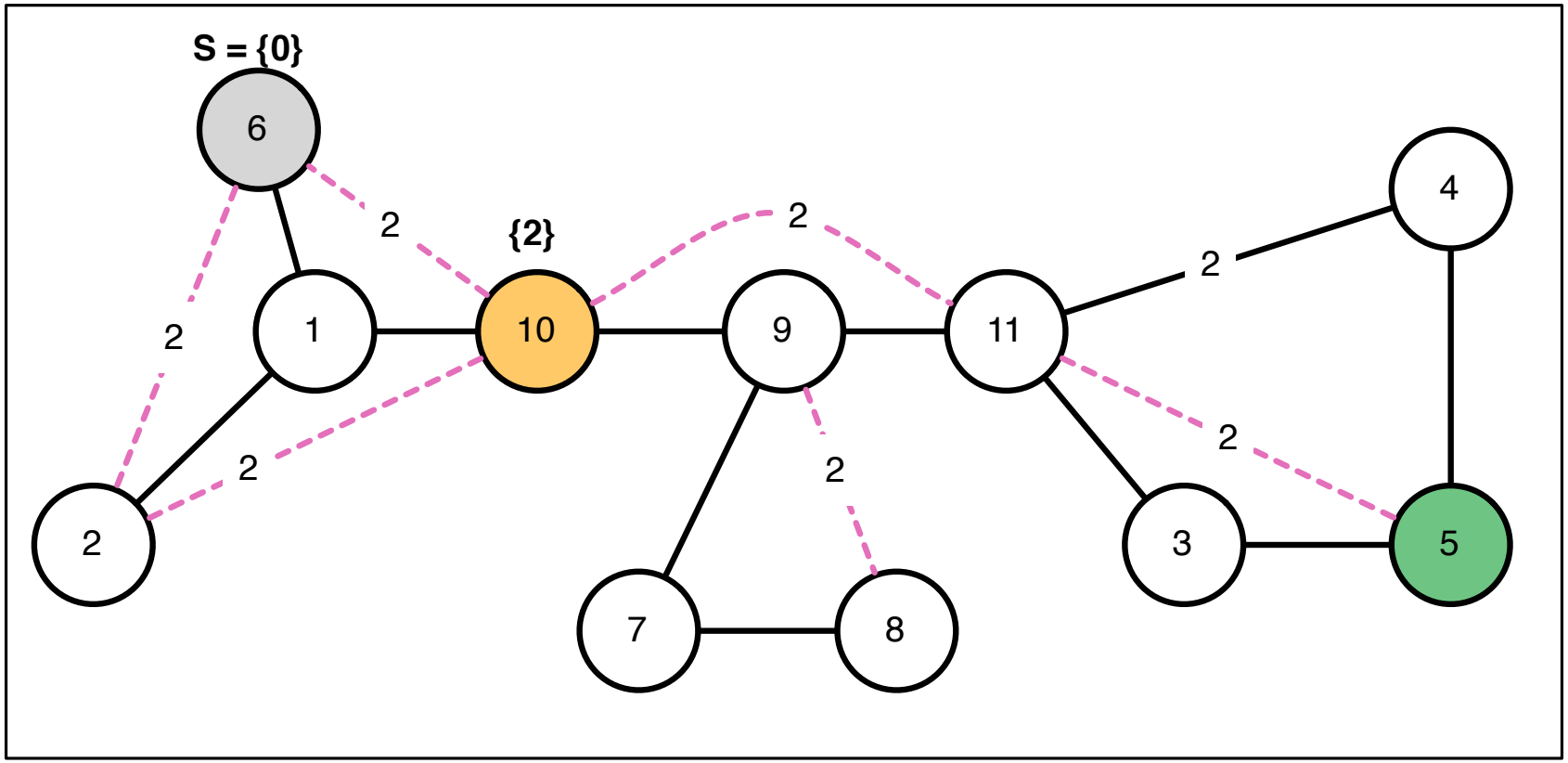
FCH+BB Example



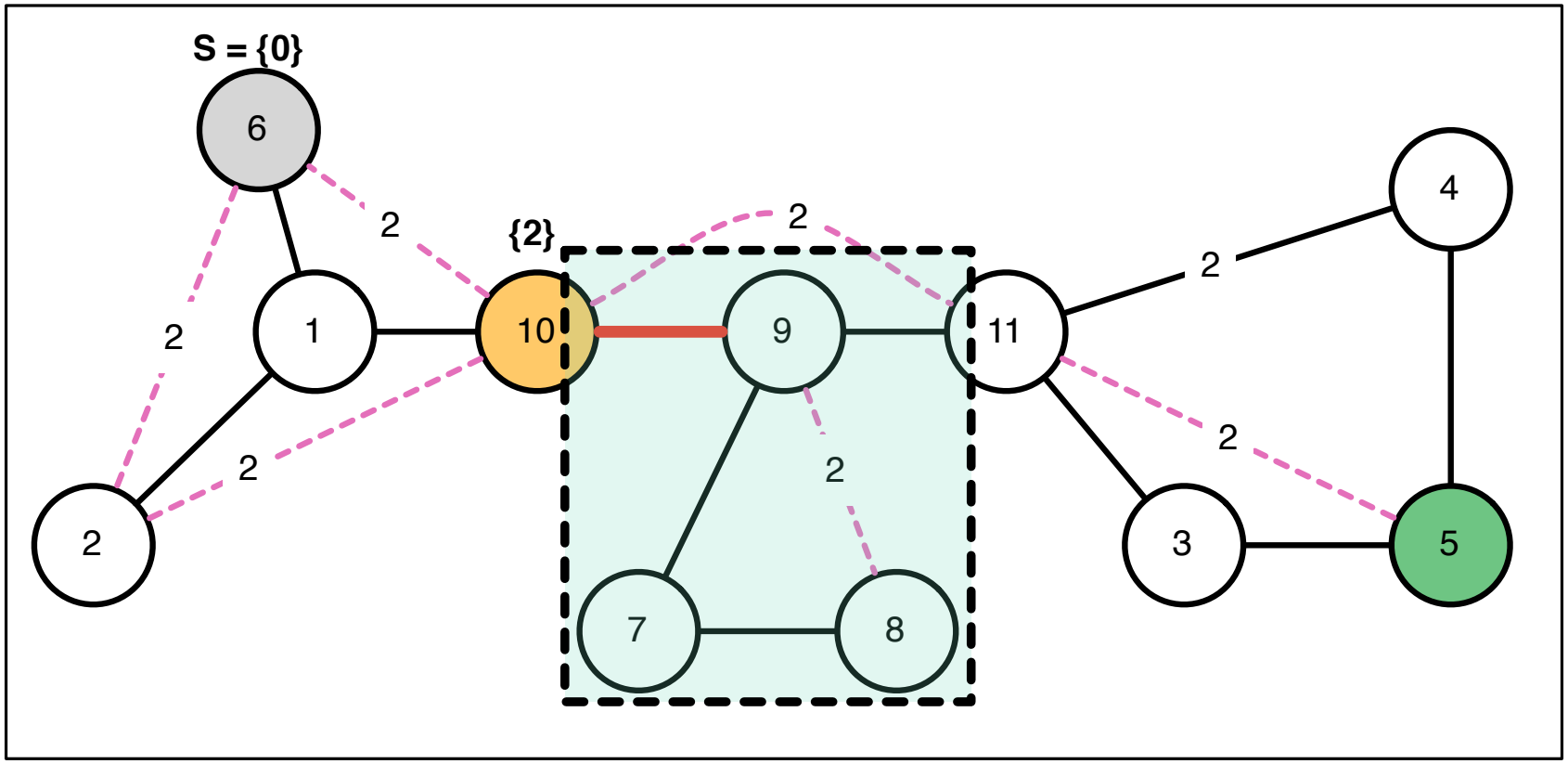
FCH+BB Example



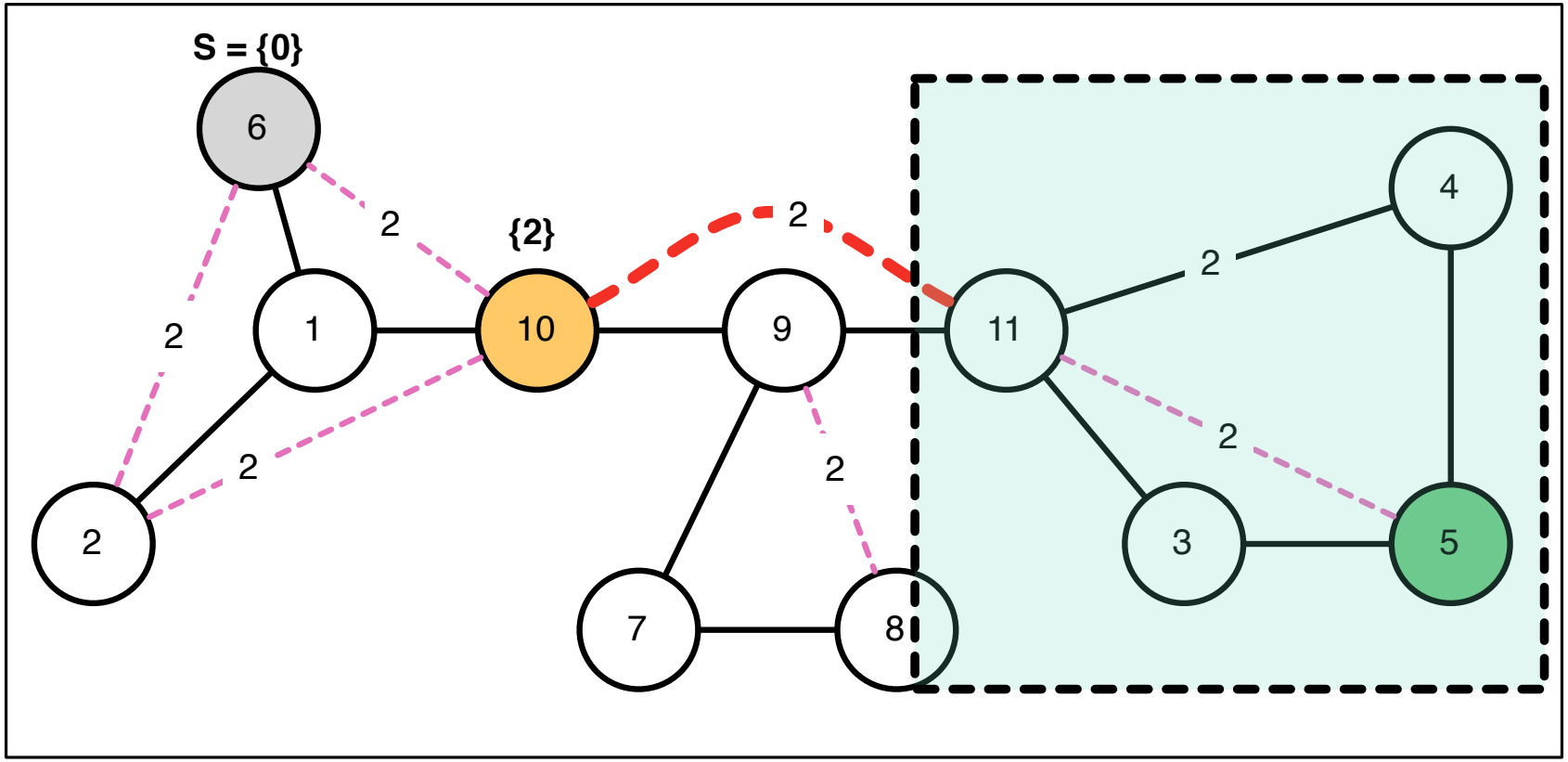
FCH+BB Example



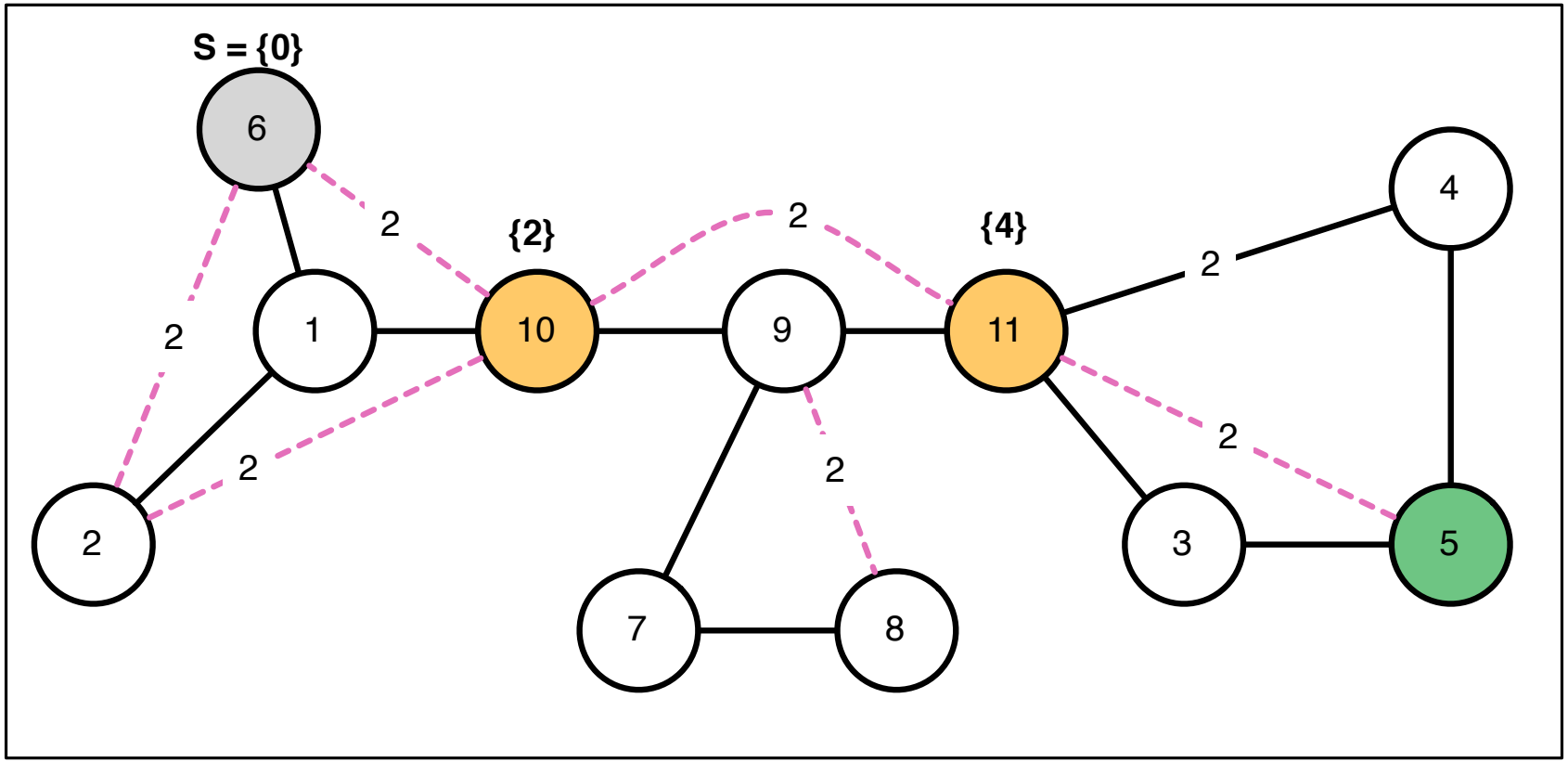
FCH+BB Example



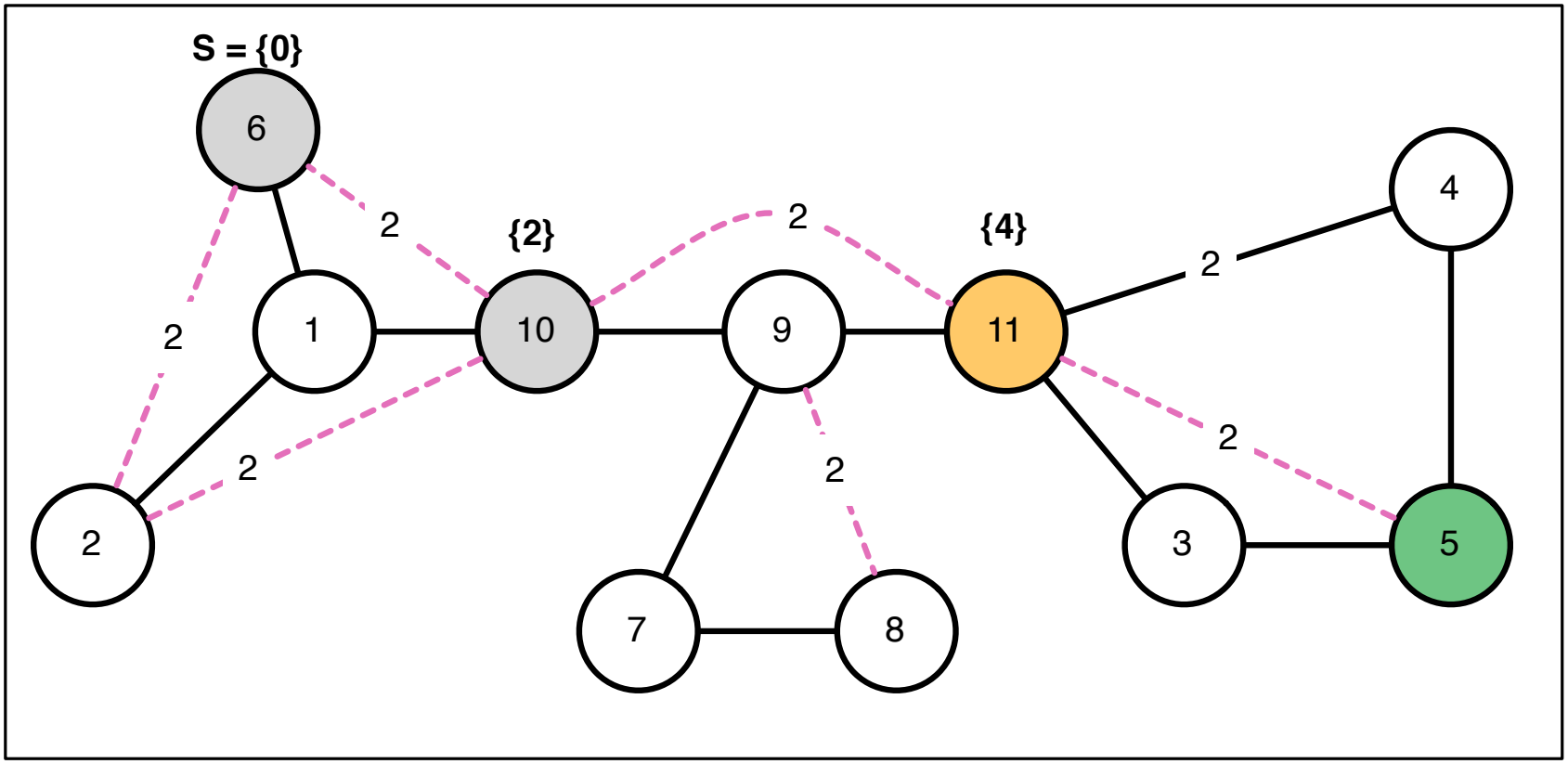
FCH+BB Example



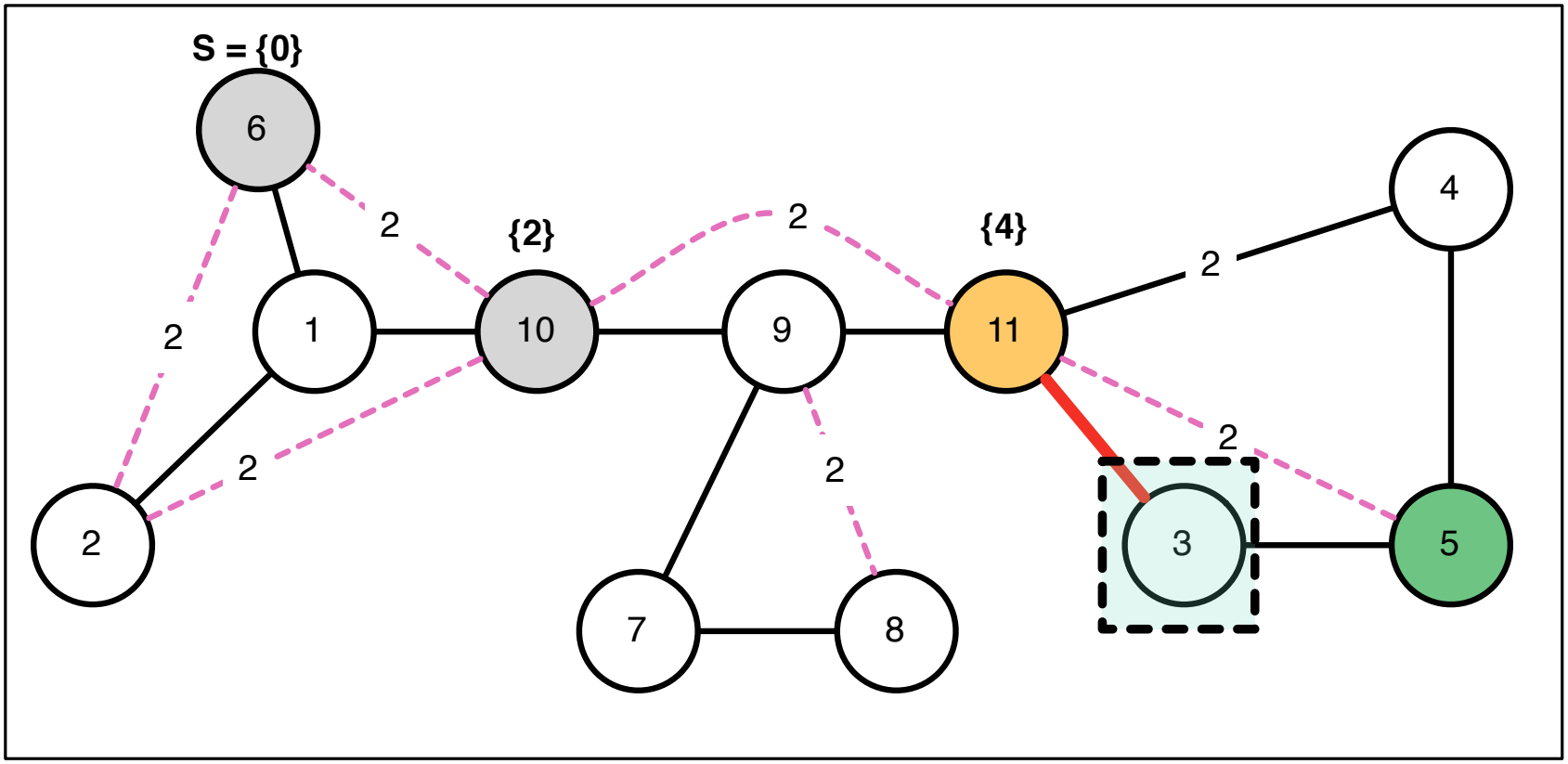
FCH+BB Example



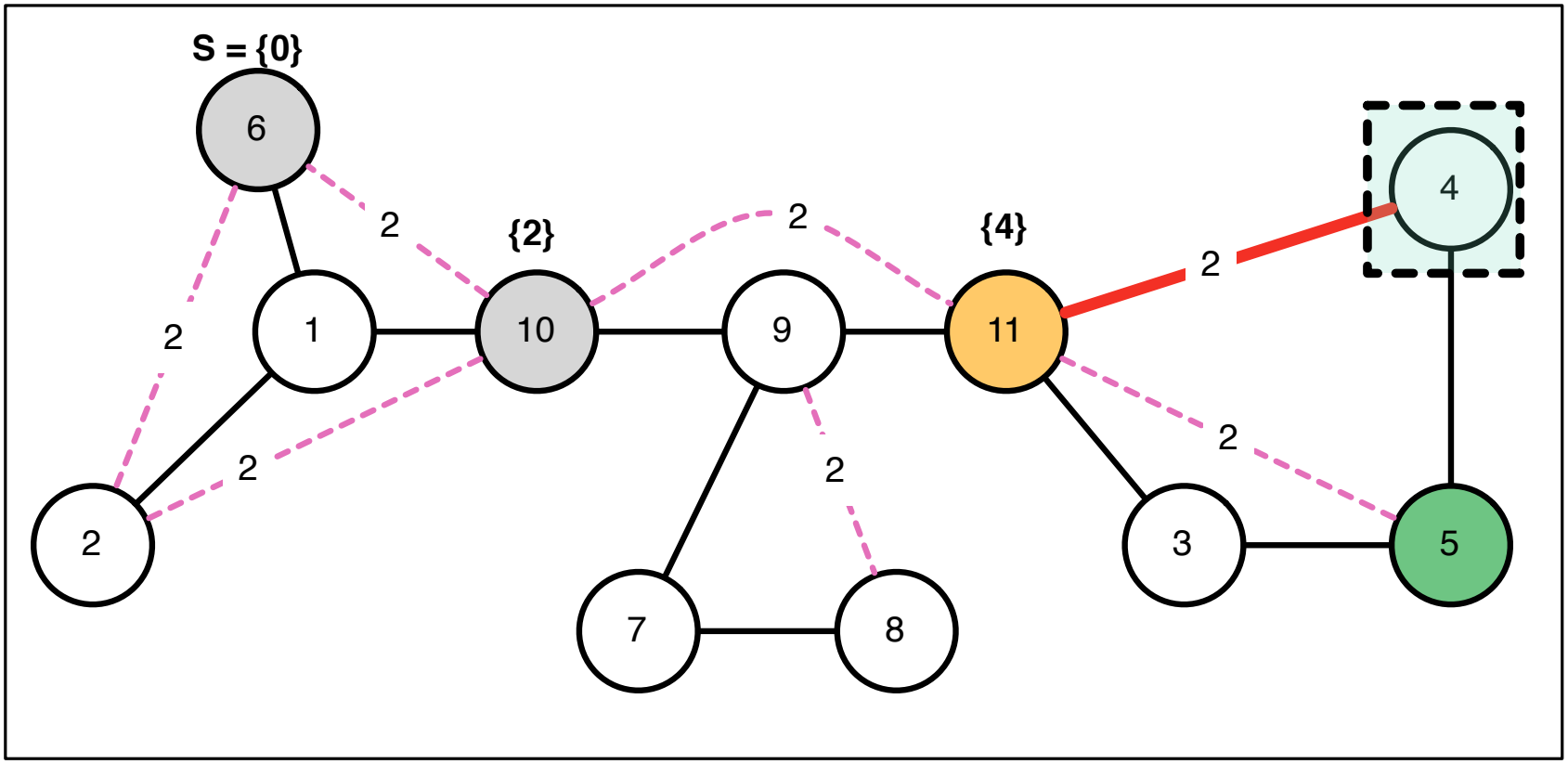
FCH+BB Example



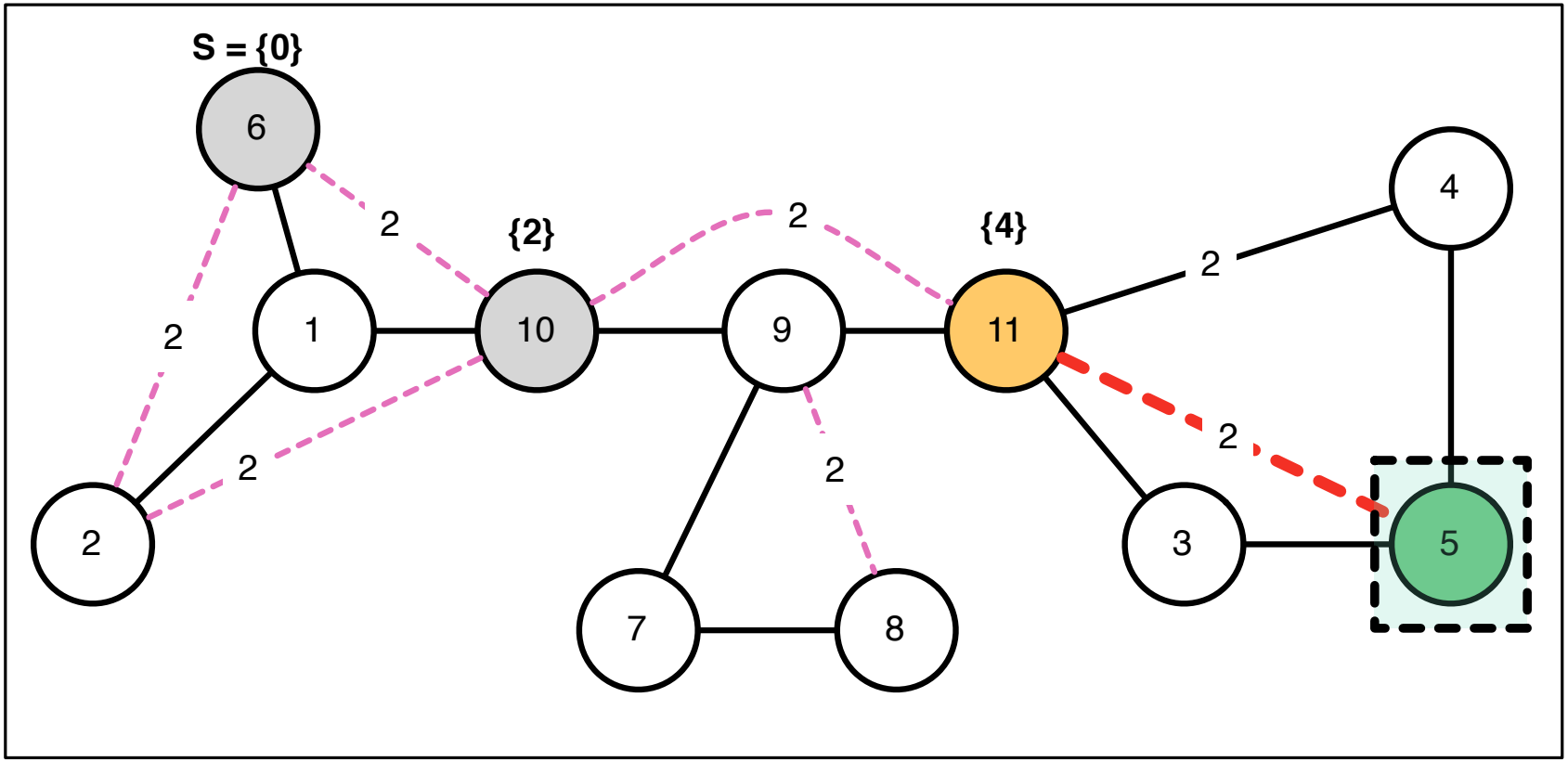
FCH+BB Example



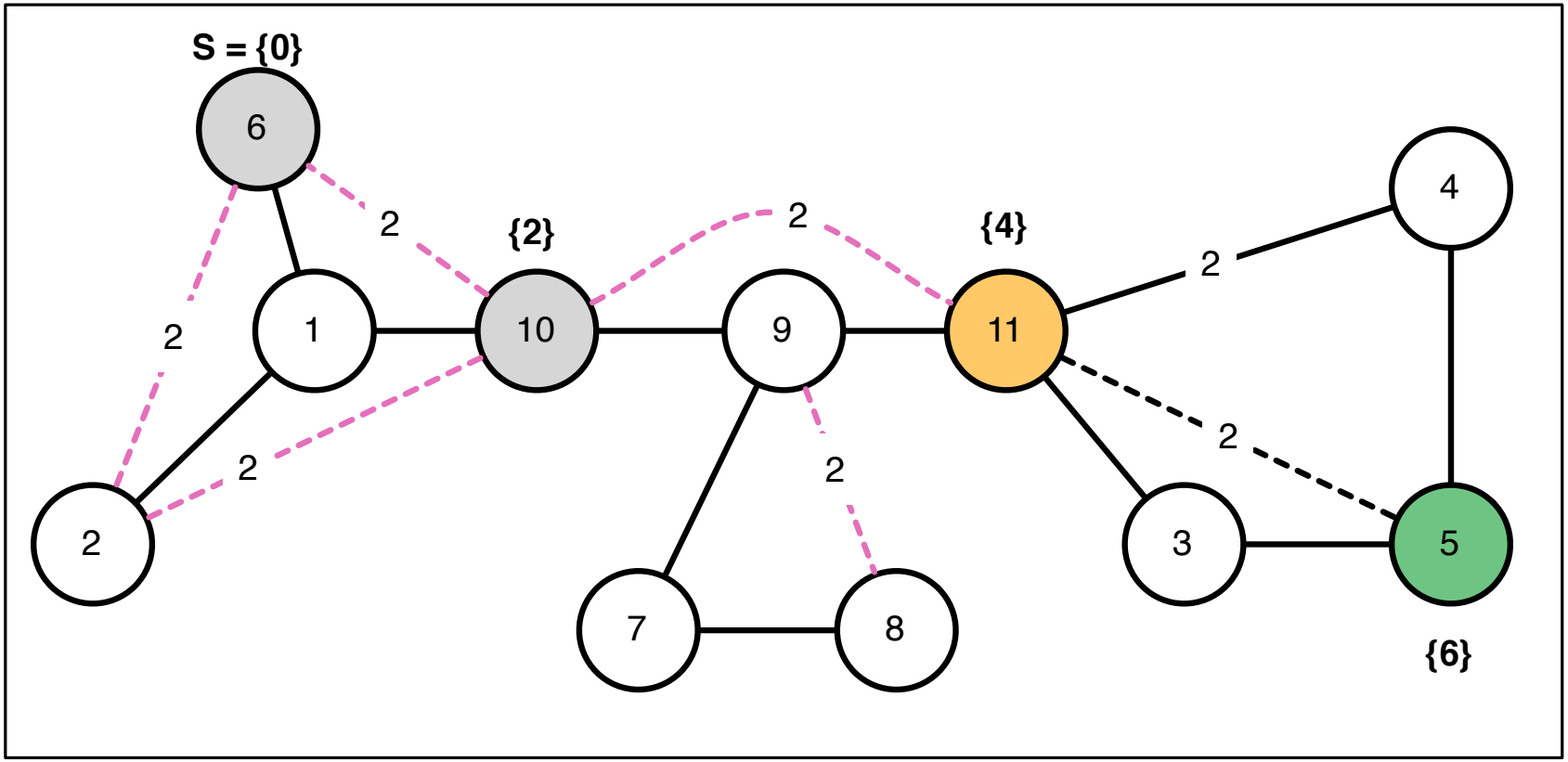
FCH+BB Example



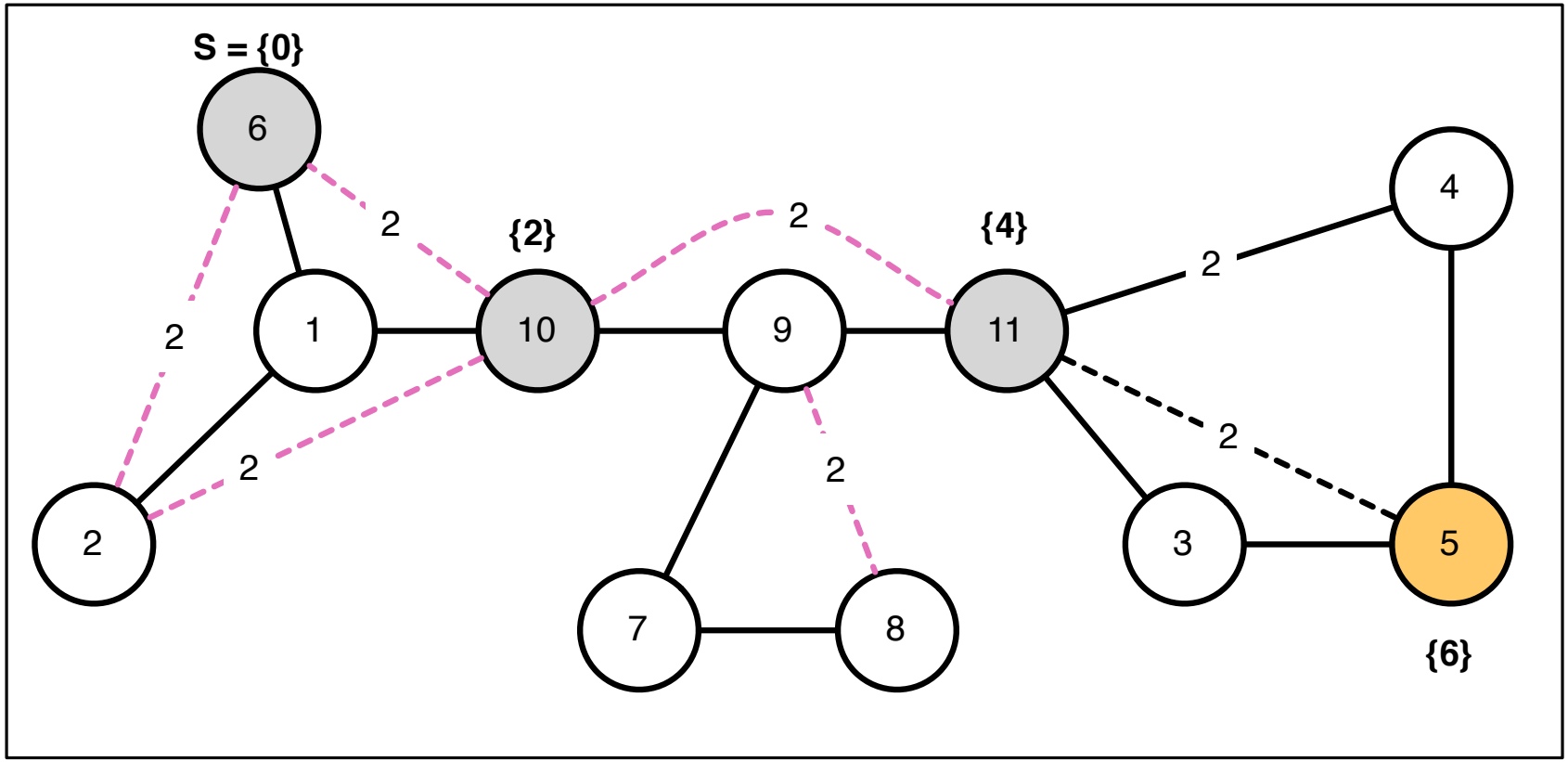
FCH+BB Example



FCH+BB Example



FCH+BB Example



Some Experimental Results

Benchmarks

Graph	$\|V\|$	$\|E\|$		
		#Input	#Shortcuts	Total
NY-d	264346	733846	920078	1653924
BAY-d	321270	800172	808952	1609124
COL-d	435666	1057066	1062850	2119916
FLA-d	1070376	2712798	2697836	5410634

Setup

- 1000 instances for each map, 5 runs per instance.
- MacBook Pro 13,2 machine (16GB RAM, OSX 10.12.6).
- From-scratch C++ implementations of all algorithms.
- `https://bitbucket.org/dharabor/pathfinding`

Comparisons

Uni-directional variants:

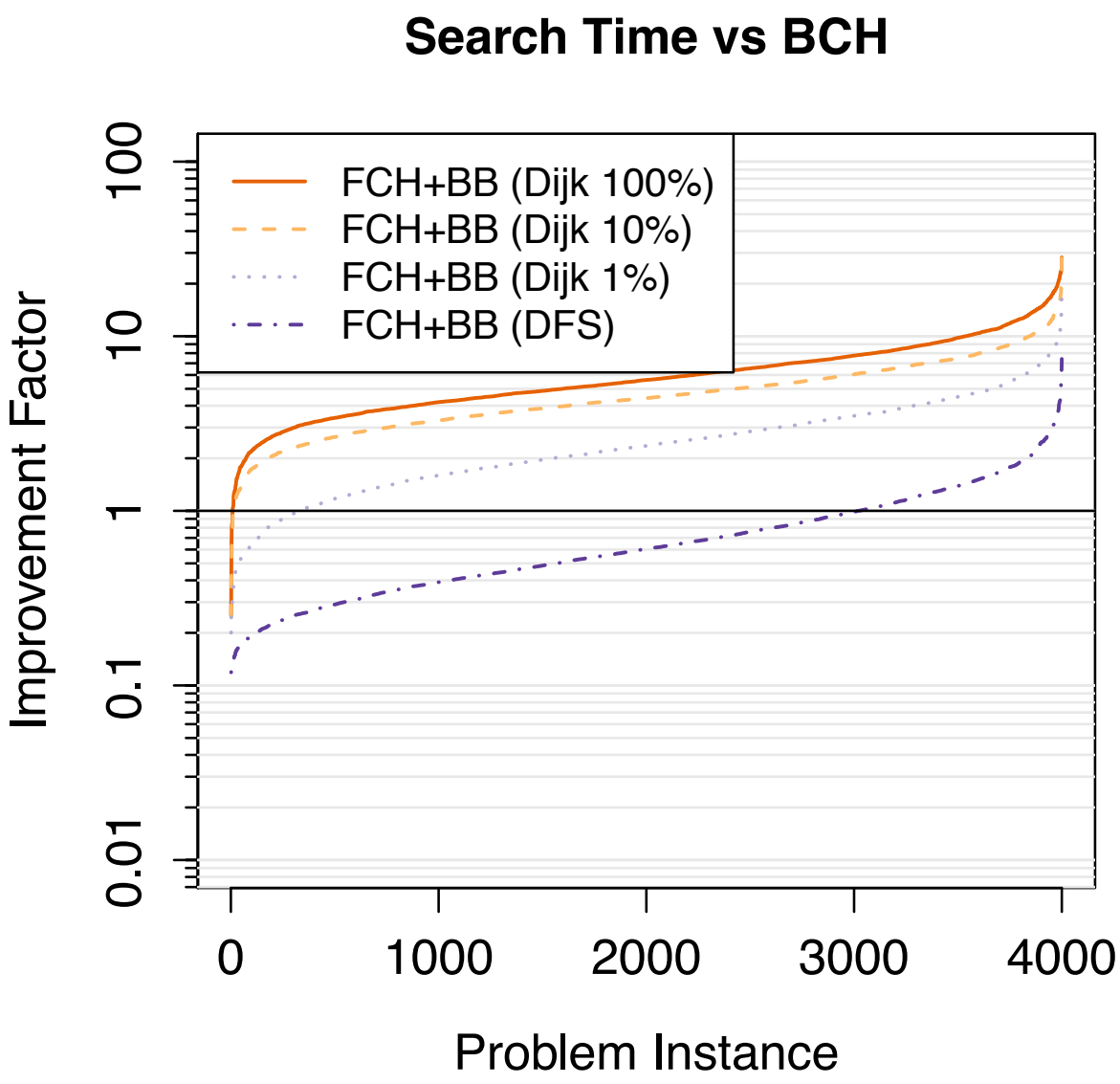
- FCH+BB (DFS) — Very light pre-proc. ($\approx$1 second)
- FCH+BB (Dijk 1%) – Light pre-proc. (minutes)
- FCH+BB (Dijk 10%) – Moderate pre-proc. ($<$ 1 hour)
- FCH+BB (Dijk 100%) – Extensive pre-proc. (many hours)

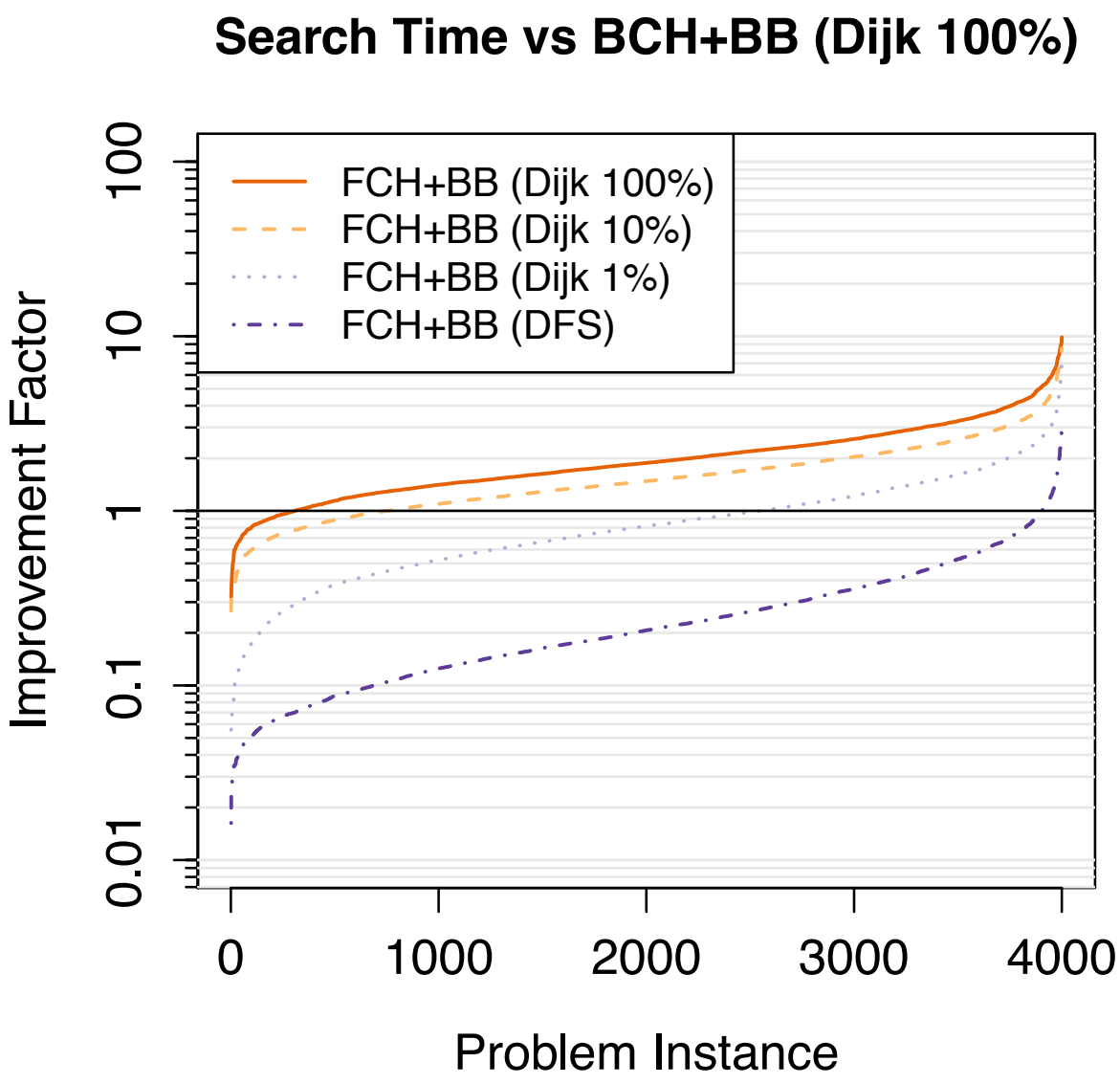
Bi-directional variants:

- BCH
- BCH+BB (Dijk 100%) – Extensive pre-proc. (many hours)

Comparisons vs BCH

Experiment: Time speedup per search (higher is better).



Comparisons vs BCH+BB

Experiment: Time speedup per search (higher is better).