Lecture 7

Search Wrap Up,

Intro to Constraint Satisfaction Problems

Lecture Overview

- A few more points about the material from Lecture 6 (more than a recap)
- Other advanced search algorithms
- Intro to CSP (time permitting)

A* properties

We showed that A* is optimal and complete, under certain conditions

A* properties

We showed that A* is optimal and complete, under certain conditions

Which of the following conditions is not needed?

A. Arc costs are bounded above 0

B. Branching factor is finite

C. h(n) is an underestimate of the cost of the shortest path from n to a goal

A* properties

D. The costs around a cycle must sum to zero

We showed that A* is optimal and complete, under certain conditions

Which of the following conditions is not needed?

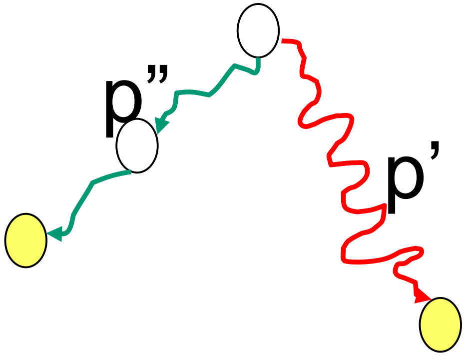
A. Arc costs are bounded above 0

B. Branching factor is finite

C. h(n) is an underestimate of the cost of the shortest path from n to a goal

D. The costs around a cycle must sum to zero

Remember proof for optimality



- Let p^* be the optimal solution path, with cost c^*.

- Let p' be a suboptimal solution path. That is $c(p') > c^*$. p* • Let p'' be a sub-path of p* on the frontier.

we know that $\boxed{f(p^*) < f(p')}$ because at a goal node $\boxed{f(goal) = c(goal)}$

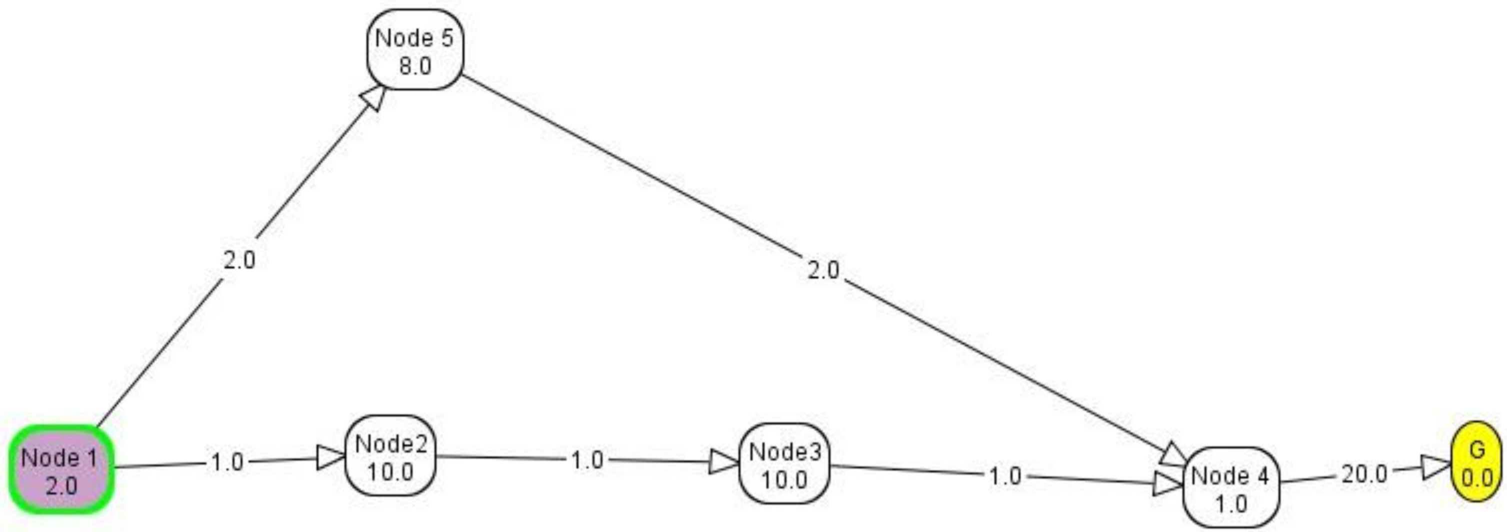
A* properties

and $\boxed{f(p'') \leq f(p^*)}$ because *h* is admissible (see proof in previous class)

thus $\boxed{f(p'') < f(p')}$

Any sup-path of the optimal solution path will be expanded before *p'*

Run A* on this example (file "Astar" in course syllabus") to see how A* starts off going down the suboptimal path (through N5) but then recovers and never expands it, because there are always subpaths of the optimal path through N2 on the frontier with lower f value.

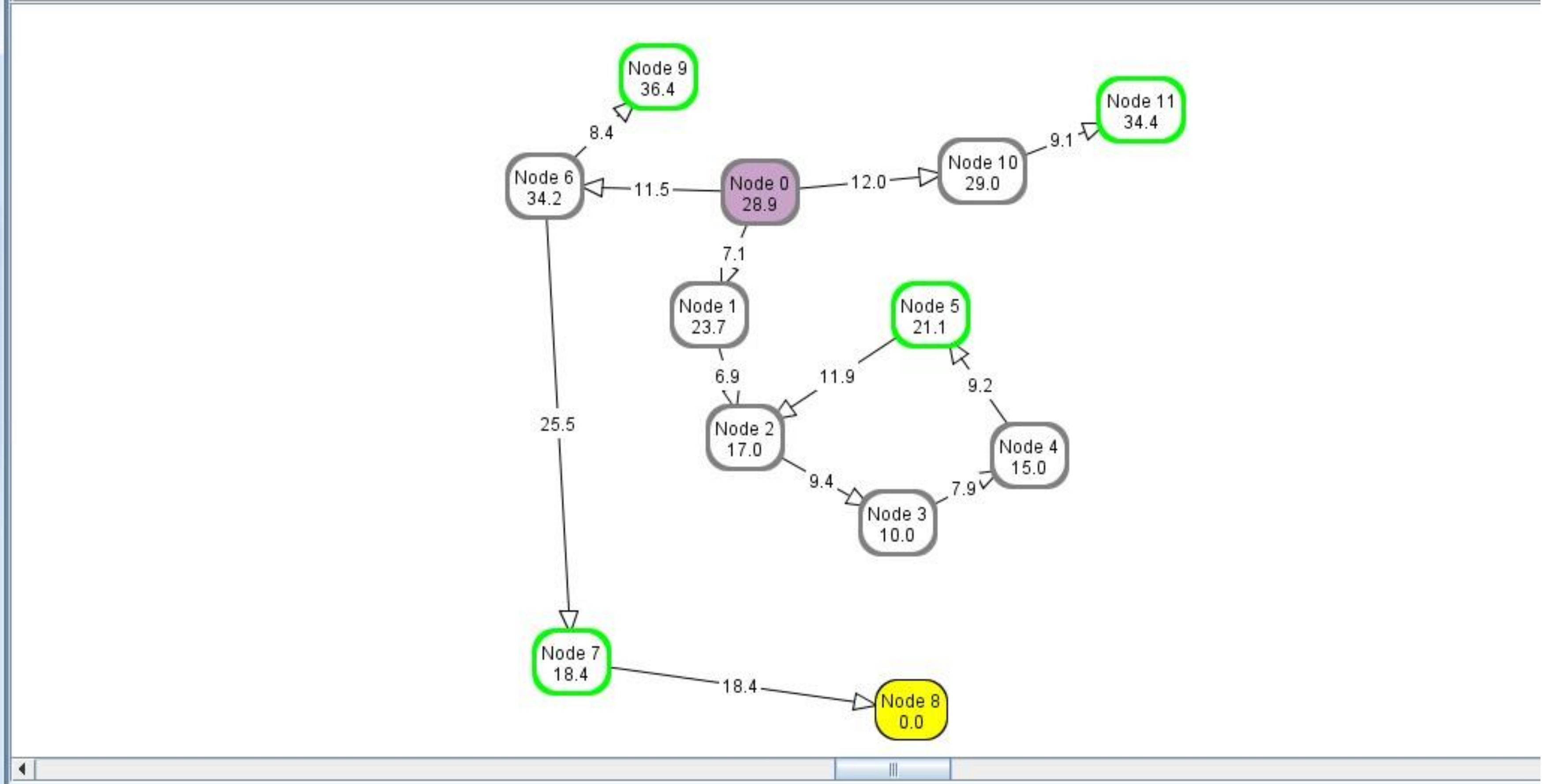


Why is A* complete

It does not get caught in cycles

- Let f* be the cost of the (an) optimal solution path p* (unknown but finite if there exists a solution)
- Each sub-path p of p* will be expanded before p*
    - See previous proof
- With positive (and > ε) arc costs, the cost of any other path p on the frontier would eventually exceed f*
- This happens at depth no greater than (f^* / c_{min}), where c_{min} is the minimal arc cost in the search graph

See how it works on the “misleading heuristic” problem in AI space:



Algorithm Selected: A*

PREVIOUS PATH2:

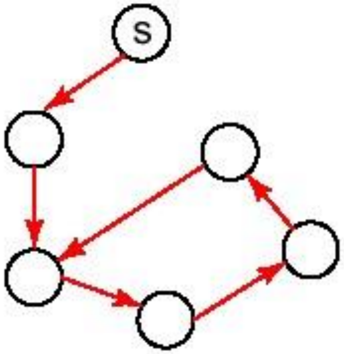
Node 0 --> Node 1 --> Node 2 --> Node 3 --> Node 4

NEW FRONTIER:

Node: Node 7	Path Cost: 37.0	h(Node 7): 18.4	f-value: 55.4	Path: Node 0 --> Node 6 --> Node 7
Node: Node 11	Path Cost: 21.1	h(Node 11): 34.4	f-value: 55.5	Path: Node 0 --> Node 10 --> Node 11
Node: Node 9	Path Cost: 19.9	h(Node 9): 36.4	f-value: 56.3	Path: Node 0 --> Node 6 --> Node 9
Node: Node 5	Path Cost: 40.5	h(Node 5): 21.1	f-value: 61.6	Path: Node 0 --> Node 1 --> Node 2 --> Node 3 --> Node 4 --> Node 5

Why is A* complete

A* does not get caught into the cycle because f(n) of sub paths in the cycle eventually (at depth <= 55.4/6.9) exceed the cost of the optimal solution 55.4 (N0->N6->N7->N8) 9

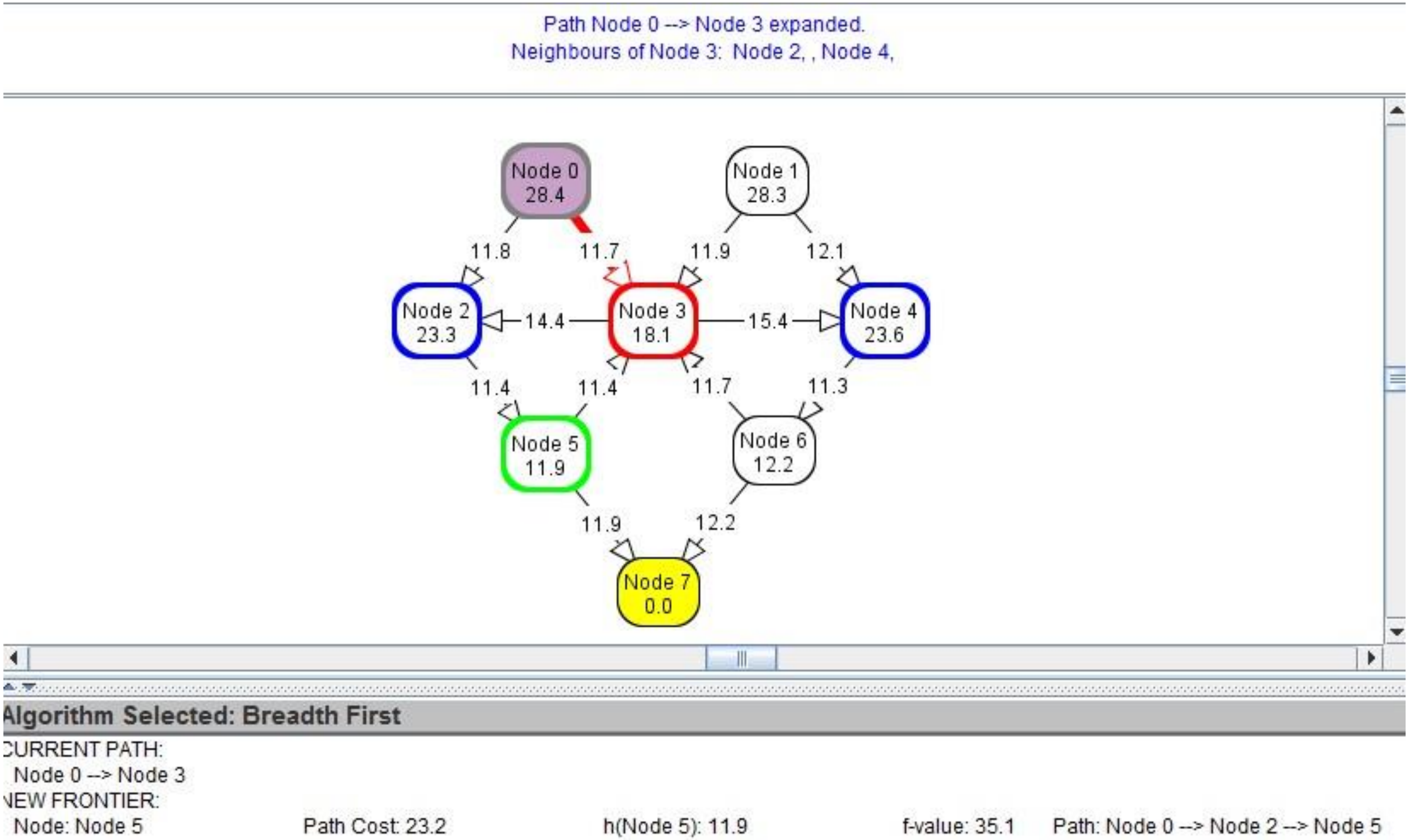


Cycle Checking

- If we want to get rid of cycles, but we also want to be able to find multiple solutions
- Do cycle checking
- In BFS-type search algorithms
- Cycle checking requires time linear in the length of the expanded path

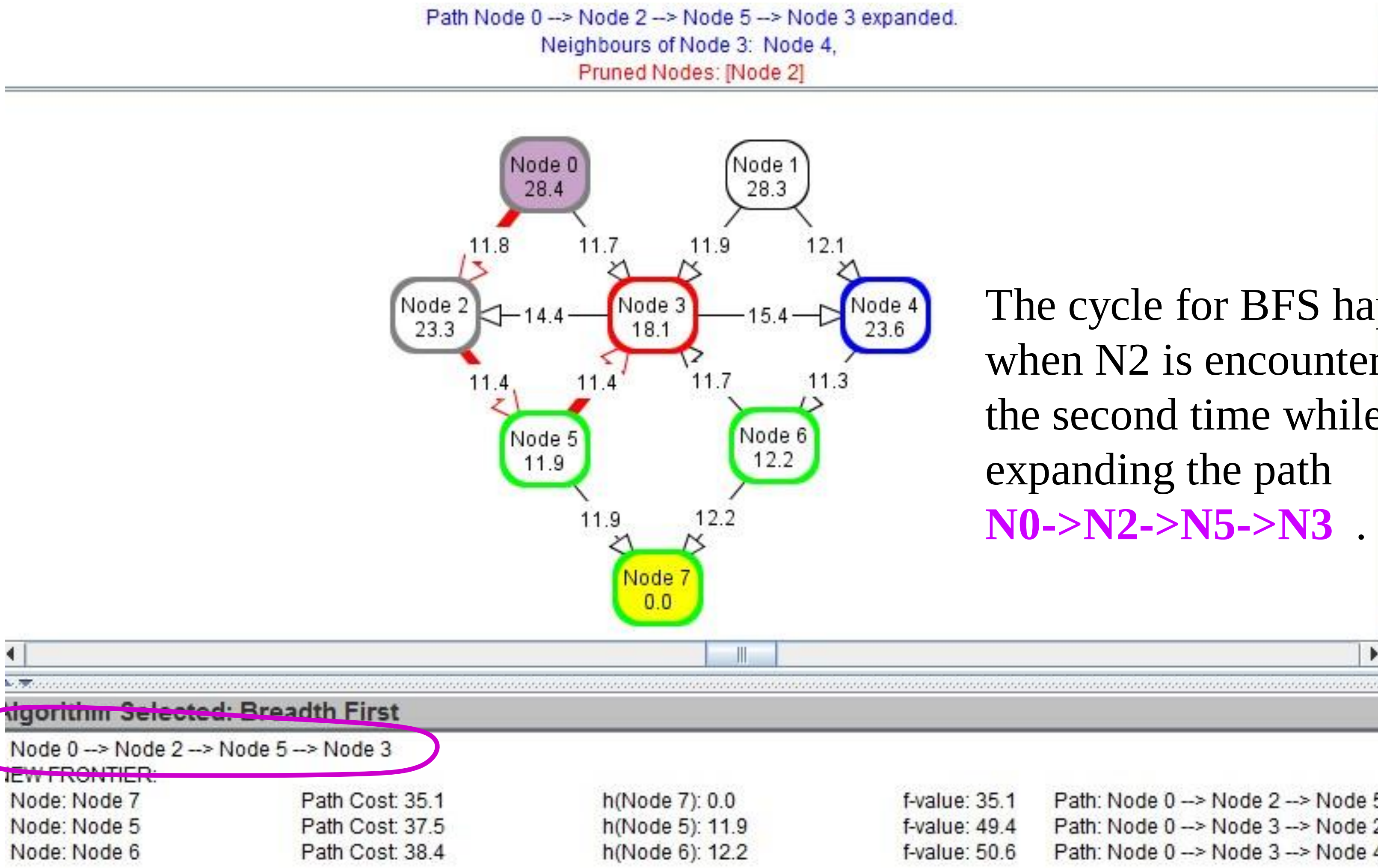
- Need to make sure that the node i being re-visited was first visited as part of the current path, not by a different path on the frontier
- In DFS-type search algorithms
- Since there is only one path on the frontier, if a node is being re-visited it is part of a cycle.
- We can do cheap cycle checks: as low as constant time (i.e. independent of path length)

Breadth First Search



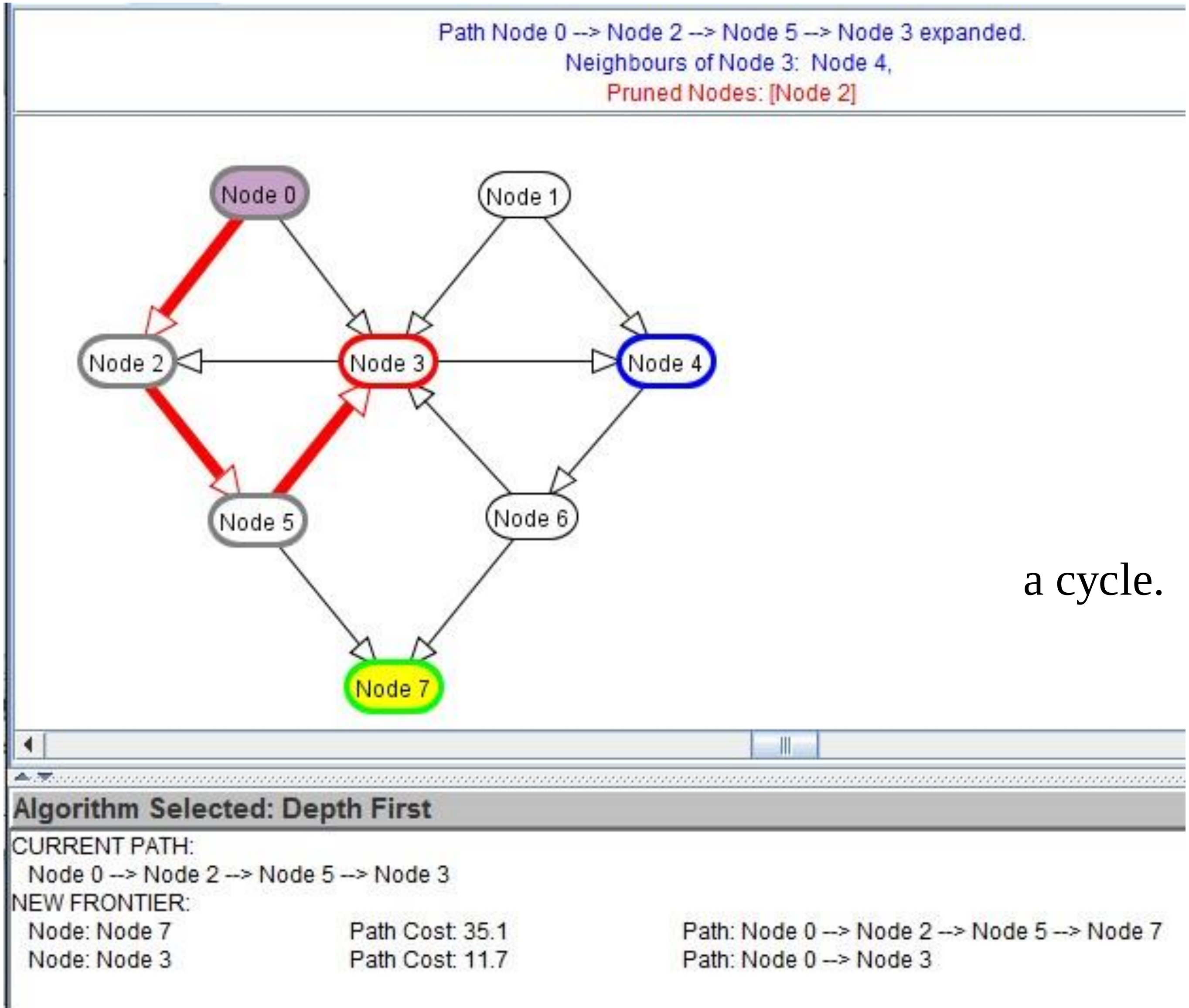
Since BFS keeps multiple subpaths going, when a node is encountered for the

Breadth First Search

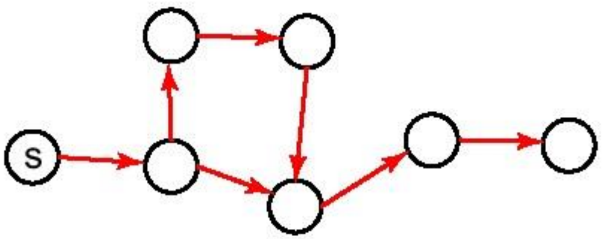


The cycle for BFS happens when N2 is encountered for the second time while expanding the path **N0->N2->N5->N3** .

Depth First Search



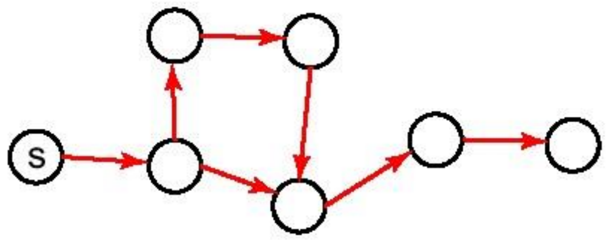
Since DFS looks at one path at a time, when a node is encountered for the second time (e.g. Node 2 while expanding N0, N2, N5, N3) it is guaranteed to be part of a cycle.



Multiple Path Pruning

If we only want one path to the solution

- Can prune path to a node n that has already been reached via a previous path
  - Subsumes cycle check
- Must make sure that we are not pruning a shorter path to the node



Multiple Path Pruning

If we only want one path to the solution

- Can prune path to a node n that has already been reached via a previous path
  - Subsumes cycle check
- Must make sure that we are not pruning a shorter path to the node
  - Is this always necessary?

    Or are there algorithms that are guaranteed to always find the shortest path to any node in the search space?

Algorithm X always find the optimal path to any node *n* in the search space first

" Whenever search algorithm X expands the first path p ending in node n, this is the lowest-cost path from the start node to n (if all costs ≥ 0)"

This is true for

A. Lowest Cost Search First

B. A*

C. Both of the above

D. None of the above

Algorithm X always find the optimal path to any node *n* in the search space first

"Whenever search algorithm X expands the first path p ending in node n, this is the lowest-cost path from the start node to n (if all costs ≥ 0)"

This is true for

A. Lowest Cost Search First

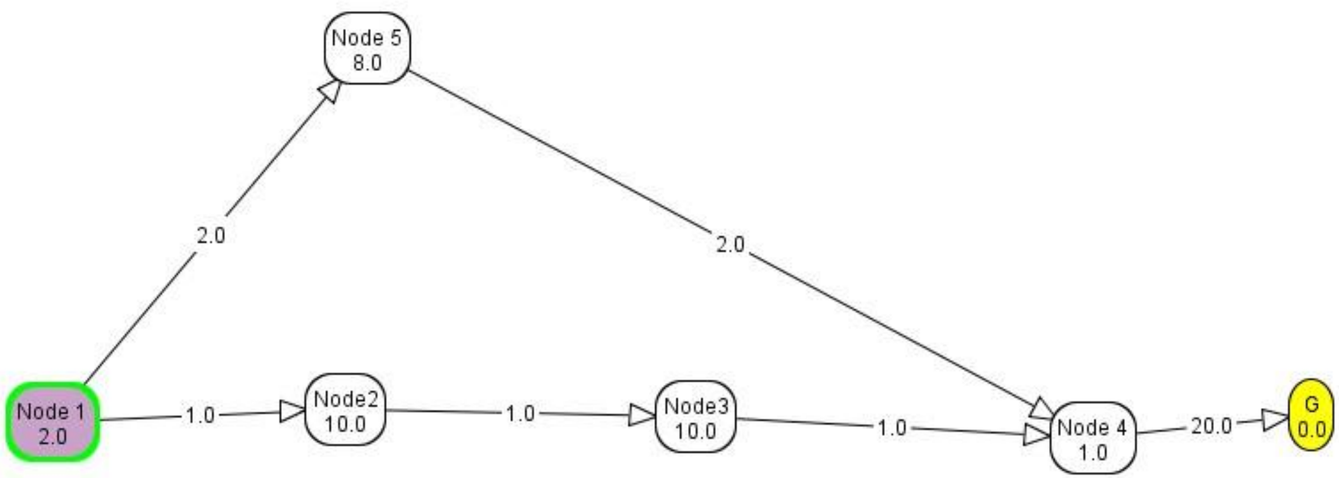
B. A*

C. Both of the above

D. None of the above

- Only LCSF, which always expand the path with the lowest cost by construction

Below is the counter-example for A*: it expands the upper path to n first, so if we prune the second path at the bottom, we miss the optimal solution



Special conditions on the heuristic can recover the guarantee of LCFS for A*: the monotone restriction (See P&M text, Section 3.7.2)

Branch-and-Bound Search

One way to combine DFS with heuristic guidance h(n) and f(n)

- Follows exactly the same search path as depth-first search
- But to ensure optimality, it does not stop at the first solution found
- It continues, after recording upper bound on solution cost
- upper bound: UB = cost of the best solution found so far

- When a path pis selected for expansion:
- Compute lower bound LB(p) = f(p)

  - If LB(p) □UB, remove pfrom frontier without expanding it (pruning)

  - Else expand p, adding all of its neighbors to the frontier

Branch-and-Bound Analysis

- Is Branch-and-Bound optimal? A.   YES, with no

  further conditions

  B.   NO

C. Only if h(n) is admissible

D. Only if there are no cycles

Branch-and-Bound Analysis

- Is Branch-and-Bound optimal? A. YES, with no

further conditions

B. NO

C. Only if h(n) is admissible. Otherwise, when checking LB(p) □UB, if the answer is yes but h(p) is an overestimate of the actual cost of p, we remove a possibly optimal solution

D. Only if there are no cycles

Branch-and-Bound Analysis

- Complete? (...even when there are cycles)

A. YES

C. It depends on initial UB

D. It depends on h

B. NO

Branch-and-Bound Analysis

- Complete ? (..even when there are cycles)

IT DEPENDS on whether we can initialize UB to a finite value, i.e. we have a reliable overestimate of the solution cost. If we don`t, we need to use ∞, and BB can be caught in a cycle

Branch-and-Bound Search

One way to combine DFS with heuristic guidance

- Follows exactly the same search path as depth-first search

- But to ensure optimality, it does not stop at the first solution found

- It continues, after recording upper bound on solution cost
- upper bound: UB = cost of the best solution found so far • Initialized

  to □ or any overestimate of optimal solution cost

- When a path pis selected for expansion:
- Compute lower bound LB(p) = f(p)

  - If LB(p) □UB, remove pfrom frontier without expanding it (pruning)

  - Else expand p, adding all of its neighbors to the frontier

Search Methods so Far

uninformed | Uninformed but using arc cost | Informed (goal directed)

	Complete	Optimal	Time	Space
DFS	N	N	$O(b^m)$	O(mb)
BFS	Y	Y	$O(b^m)$	$O(b^m)$
IDS	Y	Y	$O(b^m)$	O(mb)
LCFS (when arc costs available)	Y Costs > 0	Y Costs >=0	$O(b^m)$	$O(b^m)$
Best First (when havailable)	N	N	$O(b^m)$	$O(b^m)$
A* (when arc costs > 0 and h admissible)	Y	Y	$O(b^m)$ Optimally Efficient	$O(b^m)$

Search Methods so Far

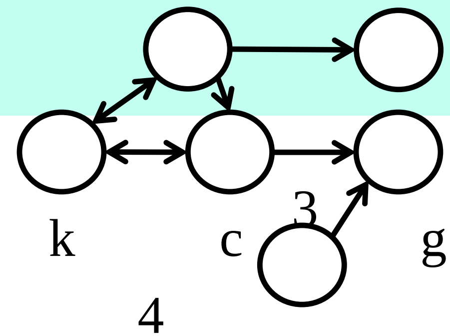
uninformed | Uninformed but using arc cost | Informed (goal directed)

	Complete	Optimal	Time	Space
DFS	N	N	$O(b^m)$	O(mb)
BFS	Y	Y	$O(b^m)$	$O(b^m)$
IDS	Y	Y	$O(b^m)$	O(mb)
LCFS (when arc costs available)	Y Costs > 0	Y Costs >=0	$O(b^m)$	$O(b^m)$
Branch-and-Bound			$O(b^m)$	O(bm)

Best First (when h available)	N	N	$O(b^m)$	$O(b^m)$
A* (when arc costs > 0 and h admissible)	Y	Y	$O(b^m)$ Optimally Efficient	$O(b^m)$
Branch-and-Bound	N (Y with finite initial bound)	Y If h admissible	$O(b^m)$	$O(bm)$

Dynamic Programming

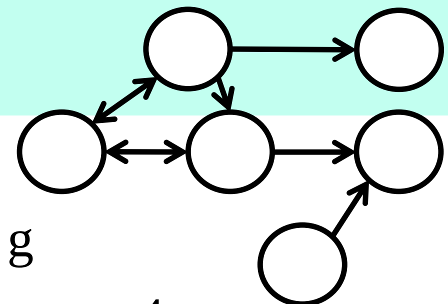
- Idea: for statically stored graphs, build a table of dist(n):
- The actual distance of the shortest path from any node n to a goal g
- This is the perfect h 2 b 1 2 h



- How could we implement that? z 1
- For each node n in the search space,
  - ✓run one of the search algorithms we have seen so far in the backwards graph (arcs reversed),
  - ✓Using the goal as start state
  - ✓And n as the goal

Dynamic Programming

- Idea: for statically stored graphs, build a table of dist(n):
- The actual distance of the shortest path from any node n to a goal g
- This is the perfect h 2 b 1 2 h



- How could we implement that? k c 3 g

4

- For each node n in the search space, z 1
  - ✓run one of the search algorithms we have seen so far in the backwards graph (arcs reversed),
  - ✓Using the goal as start state
  - ✓And n as the goal

Which algorithm should we use?

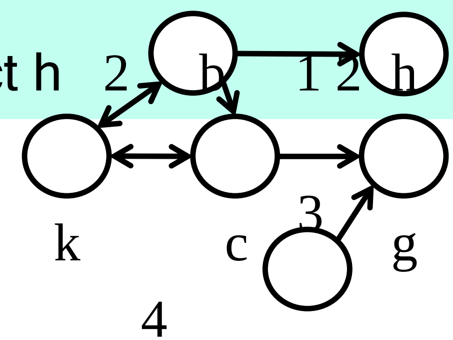
C. LCSF with multiple path pruning

Dynamic Programming

It is the only one guaranteed to find the shortest path from s to any node, and does not need h

We want multiple path pruning because we are only interested in the first, shortest path from each node to s

- Idea: for statically stored graphs, build a table of dist(n):
  - The actual distance of the shortest path from node n to a goal g
    - This is the perfect h



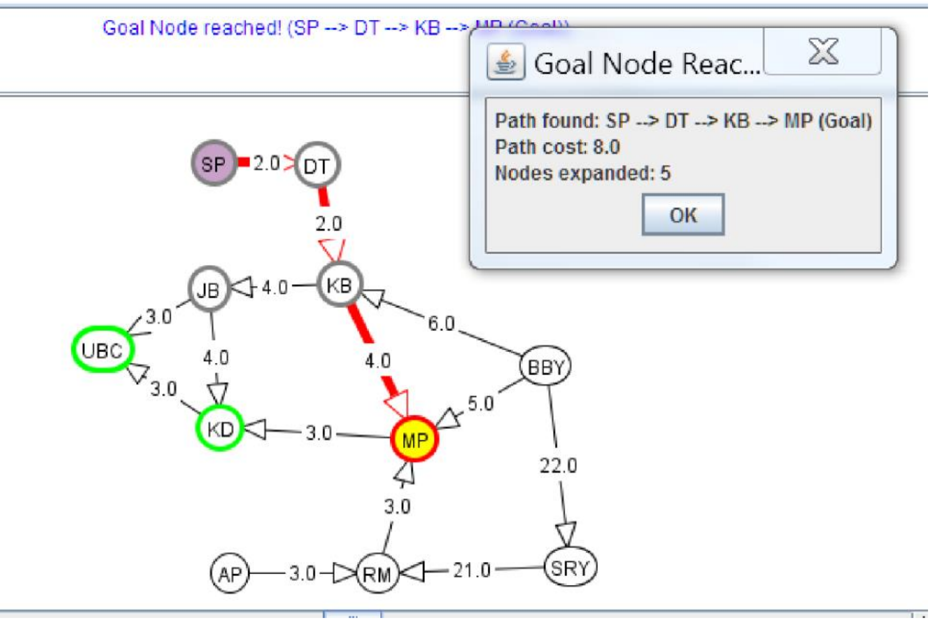
- How could we implement that? z 1
- For each node n in the search space,

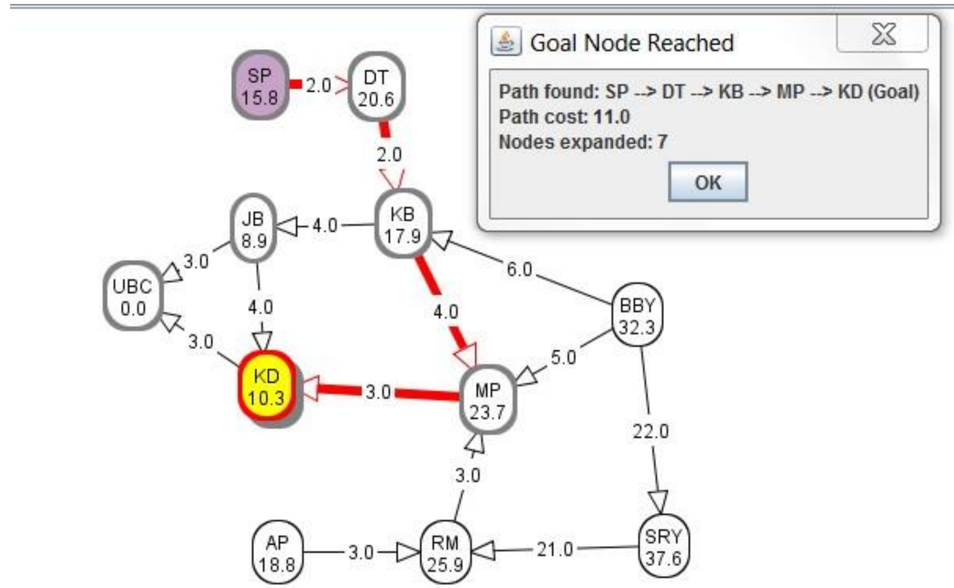
Dynamic Programming

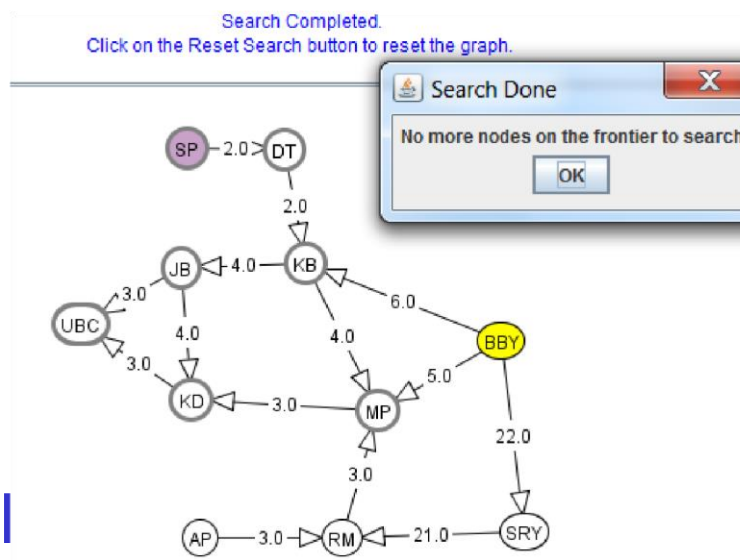
✓run LCSF with MP pruning in the backwards graph (arcs reversed),

✓Using the goal as start state

- You can manually simulate how this works by generating the backward graph in AISpace: do invert graph, in createmode



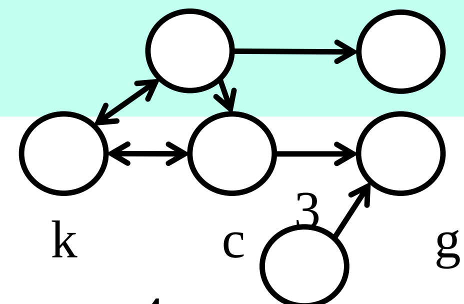




LCSF on I... nple steps)

Dynamic Programming

- Idea: for statically stored graphs, build a table of dist(n):
- The actual distance of the shortest path from node n to a goal g
- This is the perfect h 2 b 1 2 h



- How could we implement that? z 1
- For each node n in the search space,
  - ✓run LCFS with MP pruning in the backwards graph (arcs reversed),
  - ✓Using the goal as start state

- When it’s time to act (forward): for each node nalways pick neighbor m that minimizes distance to goal
- Problems?
- Needs space to explicitly store the full search graph
- The dist function needs to be rec

Lecture Overview

- Recap of Lecture 9
- Other advanced search algorithms
- Intro to CSP (time permitting)

Iterative Deepening A* (IDA*)

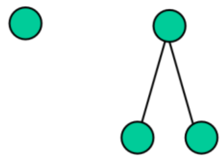
Branch & Bound (B&B) can still get stuck in infinite (or extremely long) paths

- Search depth-first, but to a fixed depth, as we did for Iterative Deepening

Iterative Deepening DFS (IDS) in a Nutshell

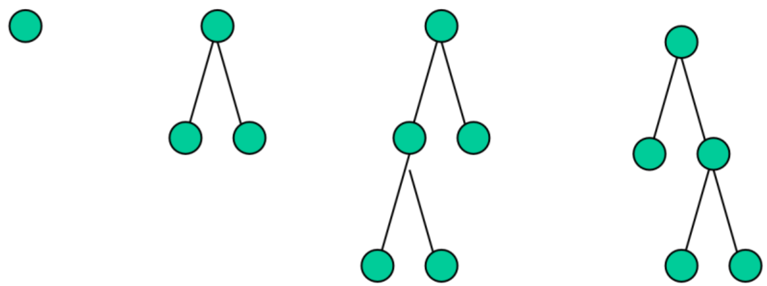
- Use DFS to look for solutions at depth 1, then 2, then 3, etc
  - Depth-bounded depth-first search

depth = 1



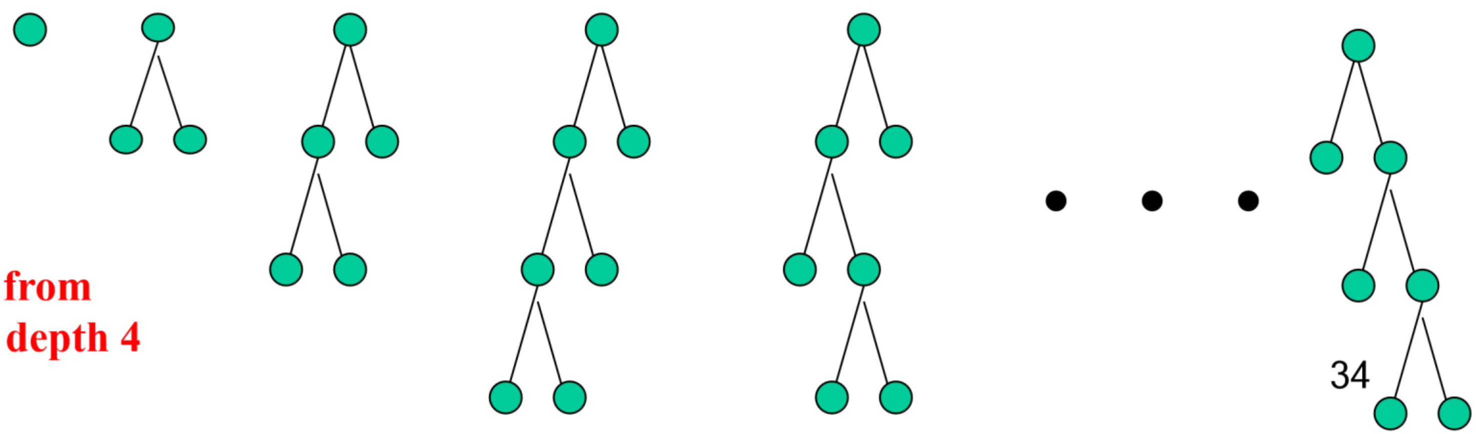
If no goal re-start from scratch and get to depth 2

depth = 2



If no goal re-start from scratch and get to depth 3

depth = 3

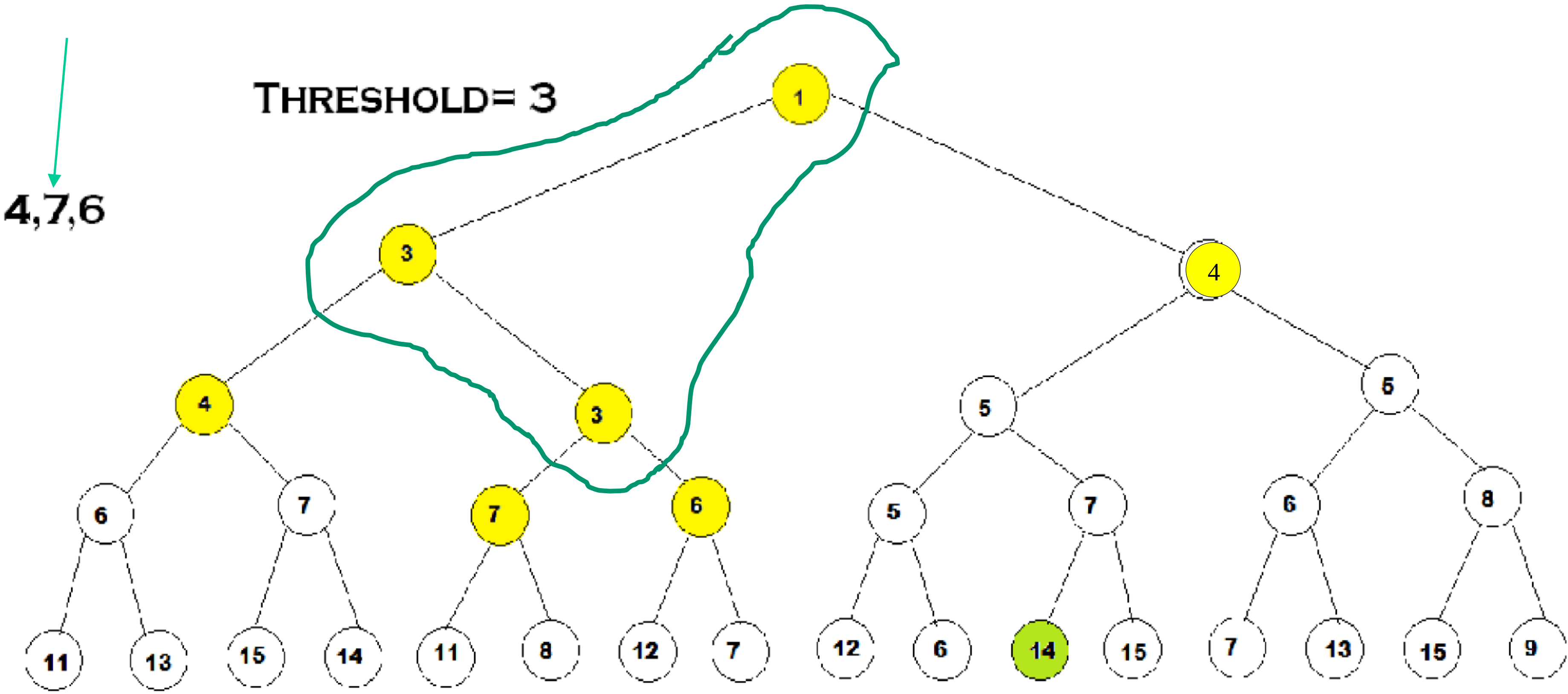


If no goal re-start from scratch and get to depth 4

– For depth D, ignore any paths with longer length

Iterative Deepening A* (IDA*)

- Like Iterative Deepening DFS
  - But the "depth" bound is measured in terms of f
  - IDA* is a bit of a misnomer
    - The only thing it has in common with A* is that it uses the f value $f(p) = cost(p) + h(p)$
    - It does NOT expand the path with lowest f value. It is doing DFS!
    - But f-value-bounded DFS doesn't sound as good ...
- Start with f-value = f(s) (s is start node)
- If you don' t find a solution at a given f-value
  - Increase the bound: to the minimum of the f-values that exceeded the previous bound
- Will explore all nodes n with f value $\leq$ f min (optimal one)
- Under the same conditions for the optimality of A*



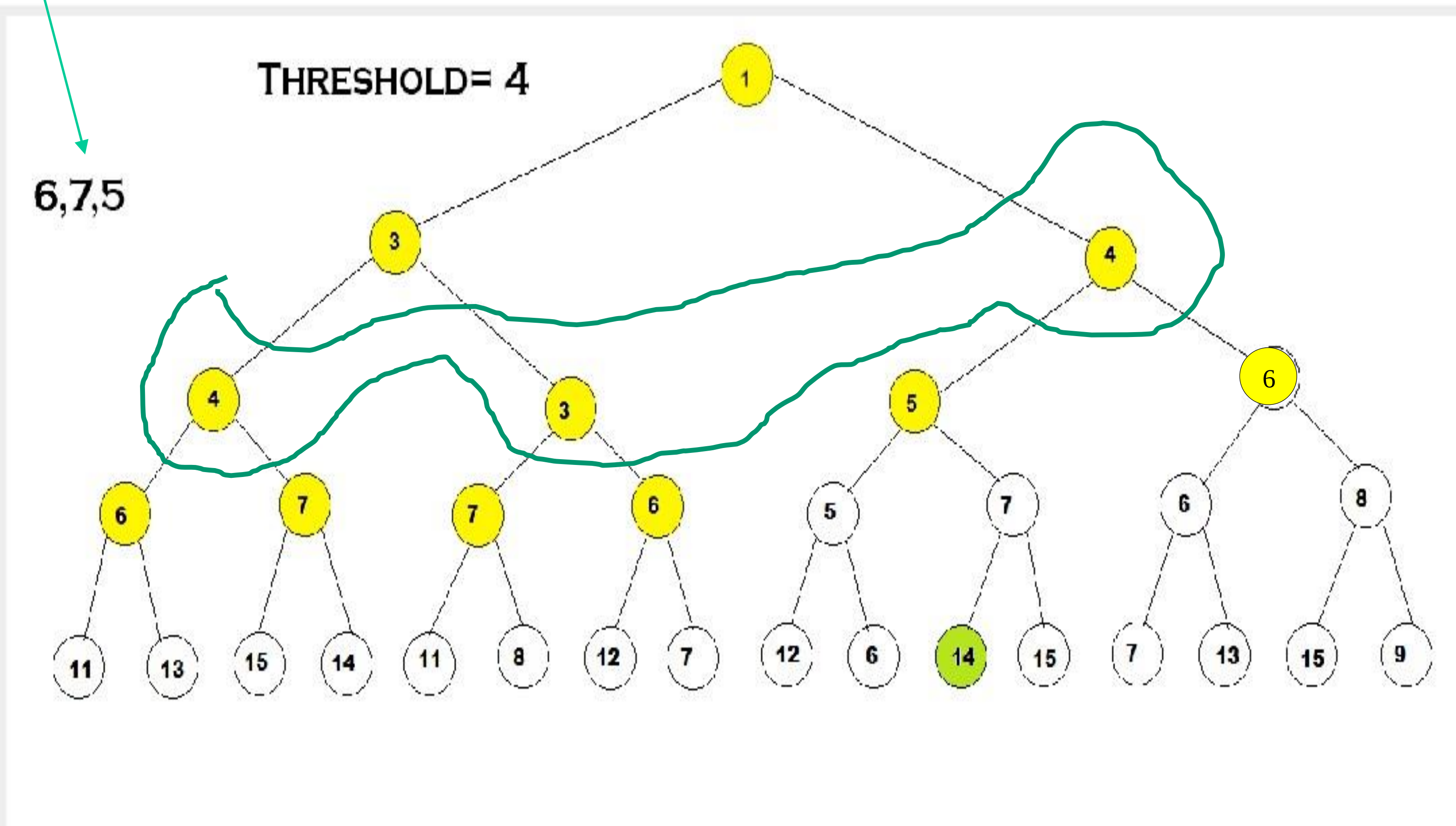
Numbers inside nodes are their f scores

The algorithm would have started with a bound of 1 (f of the start state).

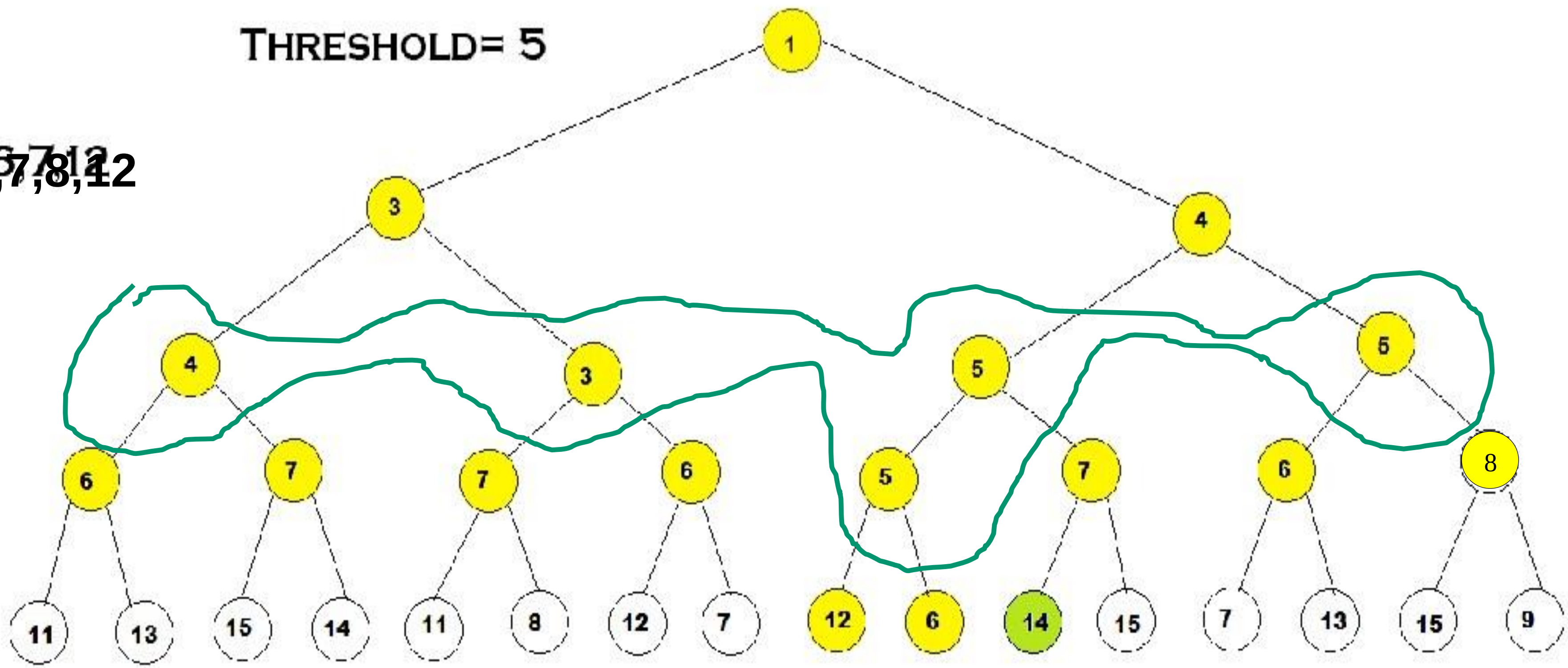
The current bound of 3 is the minimum of the f values found to exceed bound = 1 (i.e. 3 and 4) in that iteration

f values found to exceed the bound of 4 in this iteration

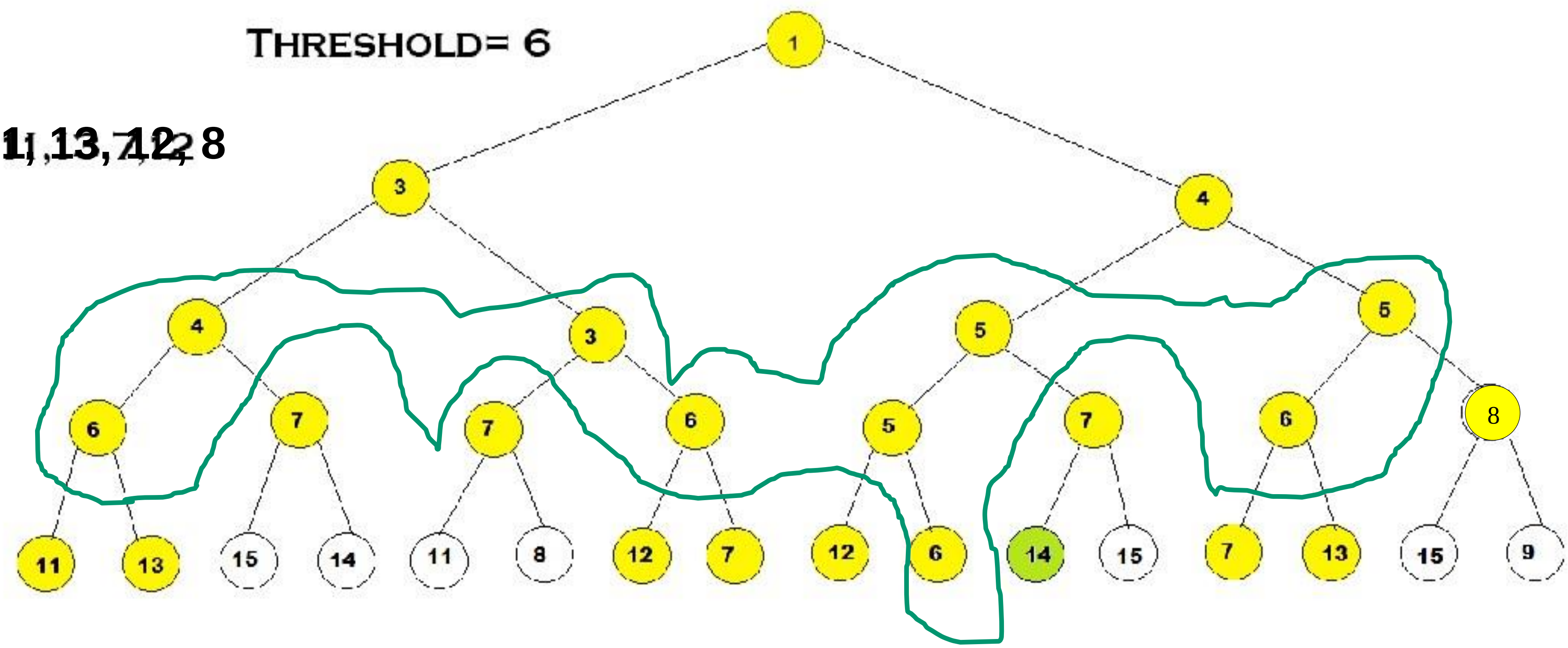
6,7,5


THRESHOLD= 4
1
3
4
4
3
5
6
6
7
7
6
5
7
6
8
11
13
15
14
11
8
12
7
12
6
14
15
7
13
15
9

6,7,8,12



11, 13, 7, 12, 8



And so on…

Analysis of Iterative Deepening A* (IDA*)

- Complete and optimal under the same conditions as A*
- Time complexity: $O(b^m)$
- Same as DFS, even though we visit paths multiple times (see slides on uninformed IDS)
- Space complexity: $O(bm)$
- Same as DFS and IDS
- Compared to Branch and Bound:
- Advantages
- does not need a finite overestimate of the solution cost to be complete
- Does not need to keep searching after finding a solution

- Disadvantage: multiple re-expansions of nodes

Search Methods so Far

uninformed | Uninformed but using arc cost | Informed (goal directed)

	Complete	Optimal	Time	Space
DFS	N	N	$O(b^m)$	O(mb)
BFS	Y	Y	$O(b^m)$	$O(b^m)$
IDS	Y	Y	$O(b^m)$	O(mb)
LCFS (when arc costs available)	Y Costs > 0	Y Costs >=0	$O(b^m)$	$O(b^m)$
Best First (when havailable)	N	N	$O(b^m)$	$O(b^m)$

A* (when arc costs > ε and h admissible)	Y	Y	$O(b^m)$ Optimally Efficient	$O(b^m)$
Branch-and-Bound	N (Y with finite initial bound)	Y If h admissible	$O(b^m)$	O(bm)
IDA*				

Search Methods so Far

uninformed   Uninformed but using arc cost   Informed (goal directed)

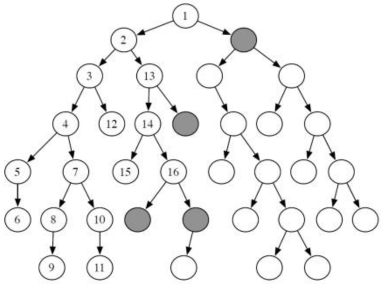
	Complete	Optimal	Time	Space
DFS	N	N	$O(b^m)$	O(mb)

Heuristic DFS

BFS	Y	Y	O(b^m)	O(b^m)
IDS	Y	Y	O(b^m)	O(mb)
LCFS (when arc costs available)	Y Costs > 0	Y Costs >=0	O(b^m)	O(b^m)
Best First (when havailable)	N	N	O(b^m)	O(b^m)
A* (when arc costs > з and h admissible)	Y	Y	O(b^m) Optimally Efficient	O(b^m)
Branch-and-Bound	N (Y with finite initial bound)	Y If h admissible	O(b^m)	O(bm)

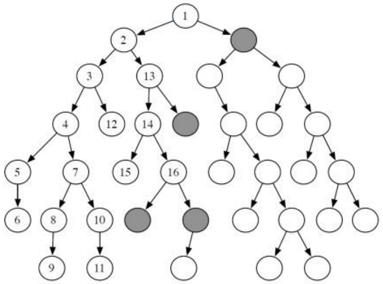
IDA* (when arc costs > ε and h admissible)	Y	Y	O(b^m)	O(bm) 42

- Other than IDA*, how else can we use heuristic information in DFS?



Heuristic DFS

- Other than IDA*, how else we use heuristic information in DFS?
  - When we expand a node, we put all its neighbours on the frontier
  - In which order? Matters because DFS uses a LIFO stack
    - ✓Can use heuristic guidance: h or f
    - ✓Perfect heuristic f: would solve problem without any backtracking



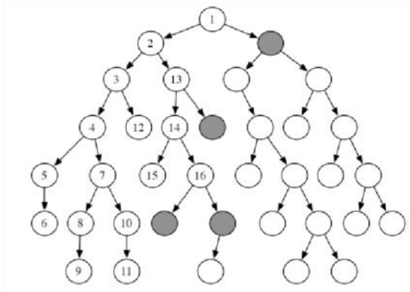
  - Heuristic DFS is very frequently used in practice
  - Simply choose promising branches first

Heuristic DFS

- Other than IDA*, how else we use heuristic information in
  - Based on any kind of information available

  DFS?

  - When we expand a node, we put all its neighbours on the frontier
  - In which order? Matters because DFS uses a LIFO stack
    - ✓Can use heuristic guidance: h or f
    - ✓Perfect heuristic f: would solve problem without any backtracking



  - Heuristic DFS is very frequently used in practice
  - Simply choose promising branches first

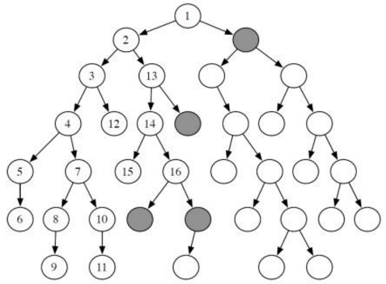
Heuristic DFS

- Other than IDA*, how else we use heuristic information in
  - Based on any kind of information available Does it have to be admissible?

A. Yes B. No C. It depends

DFS?

  - When we expand a node, we put all its neighbours on the frontier
  - In which order? Matters because DFS uses a LIFO stack
    - ✓Can use heuristic guidance: h or f
    - ✓Perfect heuristic f: would solve problem without any backtracking



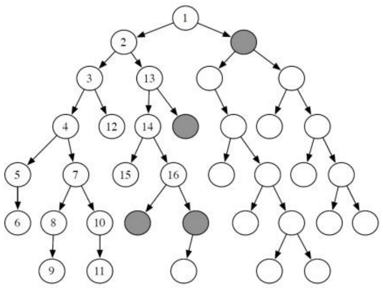
Heuristic DFS

- Other than IDA*, how else we use heuristic information in
  - Heuristic DFS is very frequently used in practice
  - Simply choose promising branches first

B. No • Based on any kind of information available
We are still doing DFS, i.e. following each path all the way to end before trying
any Does heuristic have to be admissible? other.

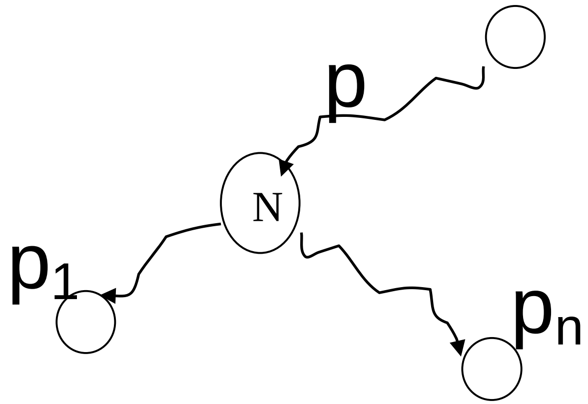
Heuristic DFS and More

- Can we combine this with IDA* ? Yes
- DFS with an f-value bound (using admissible heuristic h)
- putting neighbors onto frontier in a smart order (using some heuristic h')
- Can, of course, also choose h' = h



Memory-bounded A*

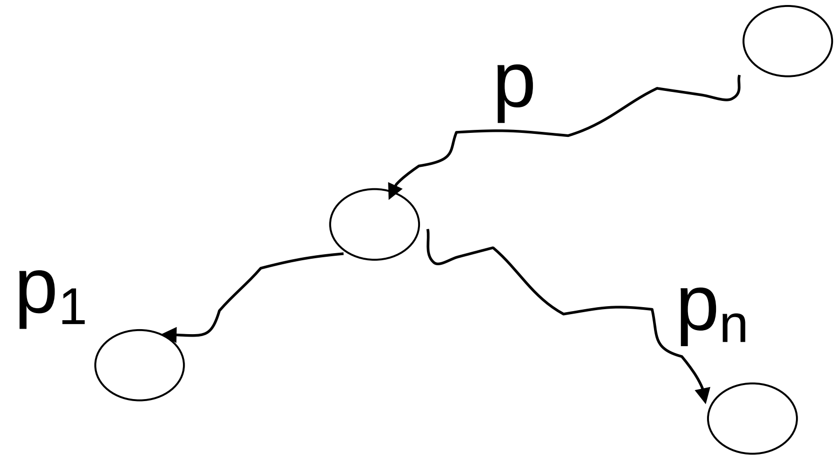
- Iterative deepening A* and B & B use little memory
- What if we have some more memory (but not enough for regular A*)?
- Do A* and keep as much of the frontier in memory as possible • When running out of memory
  - ✓ delete worst paths (highest f) from frontier (e.g. p_1, ..p_n below)
  - ✓ Backup the value of the deleted paths to a common ancestor (e.g. N below) - Way to remember the potential value of the "forgotten" paths



The corresponding subtrees get regenerated only when all other paths have been shown to be worse than the “forgotten” path

MBA*: Compute New h(p)

If we want to prune subpaths p_1, p_2, .., p_n below and “back up” their value to common ancestor p



New h(p) max minMin[[(cost(pcost(p_i)$_i$ –) -cost(p)) + h(p cost(p)

$_i$h(p)], $_i$)], Old h(p)Old h(p)

i

$(cost(p_i) - cost(p)) + h(p_i)$ gives the estimated cost of the pruned subpath from p to p_i

Min $[(cost(p_i) - cost(p)) + h(p_i)]$ gives the pruned subpath with the most promising estimated cost

Taking the max with Old h(p) gives the tighter h value for p

Memory-bounded A*

Details of the algorithm are beyond the scope of this course but

- It is complete, if there is any reachable solution, i.e. a solution at a depth manageable by the available memory
- it is optimal if the optimal solution is reachable ○ Otherwise it returns the best reachable solution given the available memory

- Often used in practice: considered one of the best algorithms for finding optimal solutions under memory limitations
- It can be bogged down by having to switch back and forth among a set of candidate solution paths, of which only a few fit in memory

Recap (Must Know How to Fill This

	Selection	Complete	Optimal	Time	Space
DFS					
BFS					
IDS					

LCFS					
Best First					
A*					
B&B					
IDA*					
MBA*					

Recap (Must Know How to Fill This

	Selection	Complete	Optimal	Time	Space
DFS	LIFO	N	N	$O(b^m)$	$\mathbf{O(mb)}$
BFS	FIFO	Y	Y	$O(b^m)$	$O(b^m)$
IDS	LIFO	Y	Y	$O(b^m)$	$\mathbf{O(mb)}$
LCFS	min cost	Y **	Y **	$O(b^m)$	$O(b^m)$
Best First	min h	N	N	$O(b^m)$	$O(b^m)$
A*	min f	Y**	Y**	$O(b^m)$	$O(b^m)$
B&B	LIFO + pruning	Y**	Y**	$O(b^m)$	$\mathbf{O(mb)}$

IDA*	LIFO	Y**	Y**	$O(b^m)$	$\boldsymbol{O(mb)}$
MBA*	min f	Y**	Y**	$O(b^m)$	$O(b^m)$

** Needs conditions: you need to know what they are

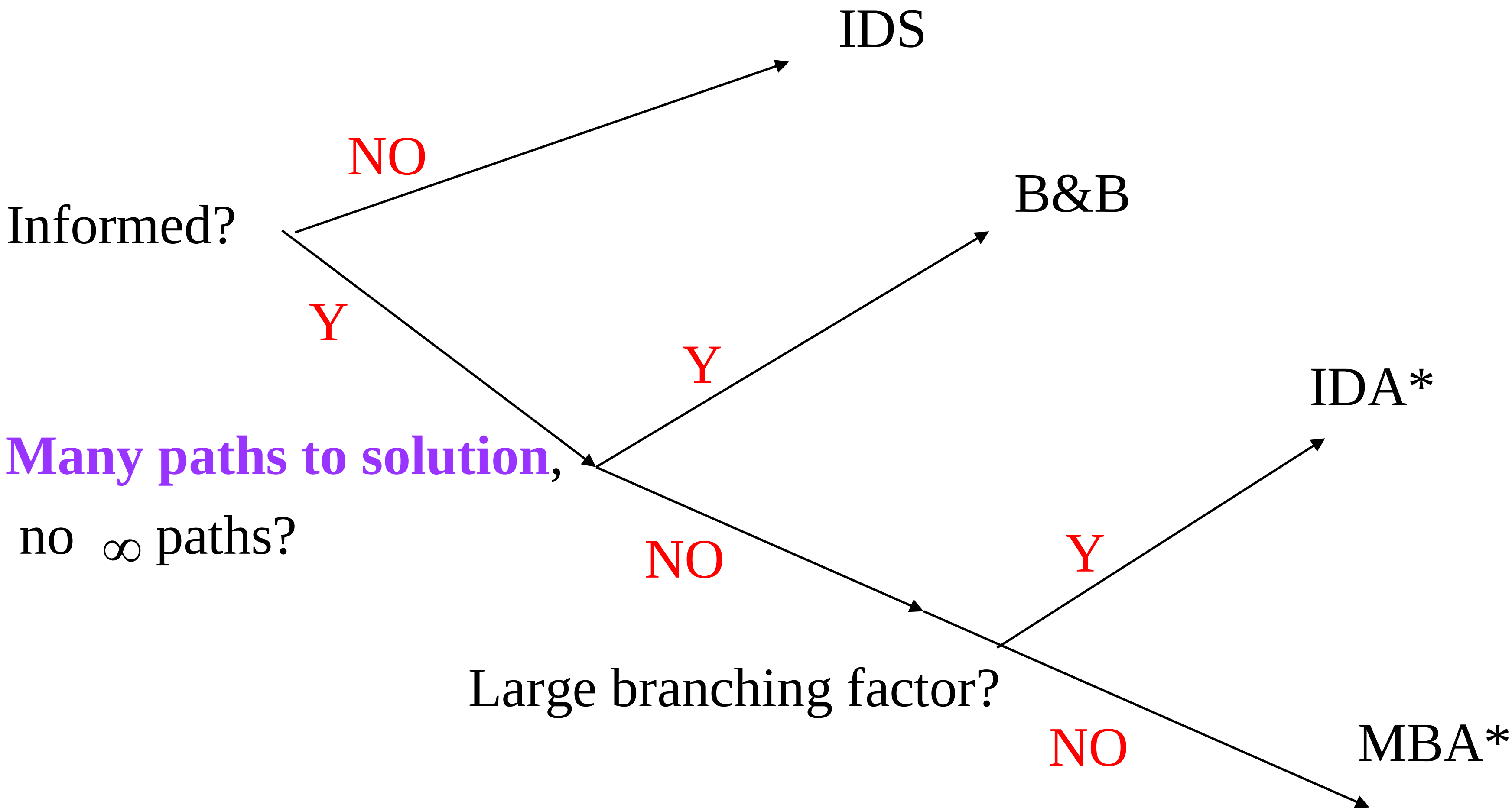
Algorithms Often Used in Practice

	Selection	Complete	Optimal	Time	Space
DFS	LIFO	N	N	$O(b^m)$	$\boldsymbol{O(mb)}$
BFS	FIFO	Y	Y	$O(b^m)$	$O(b^m)$
IDS	LIFO	Y	Y	$O(b^m)$	$\boldsymbol{O(mb)}$
LCFS	min cost	Y **	Y **	$O(b^m)$	$O(b^m)$

Best First	min h	N	N	$O(b^m)$	$O(b^m)$
A*	min f	Y**	Y**	$O(b^m)$	$O(b^m)$
B&B	LIFO + pruning	Y**	Y**	$O(b^m)$	$\boldsymbol{O(mb)}$
IDA*	LIFO	Y	Y	$O(b^m)$	$\boldsymbol{O(mb)}$
MBA*	min f	Y**	Y**	$O(b^m)$	$O(b^m)$

** Needs conditions: you need to know what they are[53]

Search in Practice



These are indeed general guidelines, specific problems might yield different choices

Remember Deep Blue?

Deep Blue’s Results in the second tournament:

- second tournament: won 3 games, lost 2, tied 1
- 30 CPUs + 480 chess processors

- Searched 126.000.000 nodes per sec
- Generated 30 billion positions per move reaching depth 14 routinely

- Iterative Deepening with evaluation function (similar to a heuristic) based on 8000 features (e.g., sum of worth of pieces: pawn 1, rook 5, queen 10)

Sample applications

- An Efficient A* Search Algorithm For Statistical Machine Translation. 2001 (DMMT '01 Proceedings of the workshop on Data-driven methods in machine translation - Volume 14)
- The Generalized A* Architecture. Journal of Artificial Intelligence Research (2007)
- Machine Vision … Here we consider a new compositional model for finding salient curves.
- Factored A*search for models over sequences and trees. IJCAI 2003
- It starts by saying… The primary challenge when using A* search is to find heuristic functions that simultaneously are admissible, close to actual completion costs, and efficient to calculate…
- applied to NLP and Bioinformatics

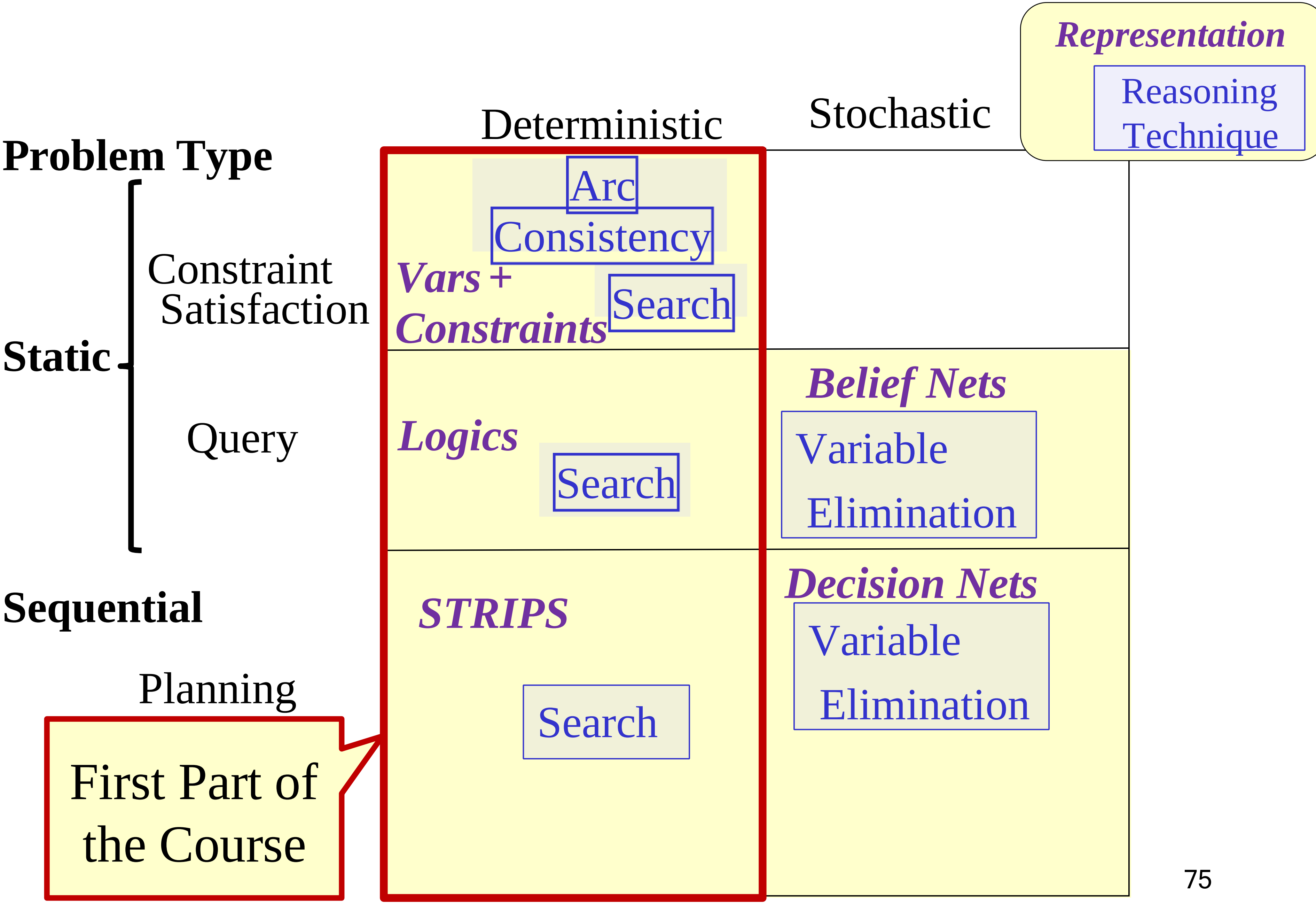
- Recursive Best-First Search with Bounded Overhead (AAAI 2015)
- “We show empirically that this improves performance in several domains, both for optimal and suboptimal search, and also yields a better linear space anytime heuristic search. RBFSCR is the first linear space best-first search robust enough to solve a variety of domains with varying operator costs.”

Learning Goals for search

- Identify real world examples that make use of deterministic, goal-driven search agents
- Assess the size of the search space of a given search problem.
- Implement the generic solution to a search problem.
- Apply basic properties of search algorithms:

-completeness, optimality, time and space complexity of search algorithms.

- Select the most appropriate search algorithms for specific problems.
- Define/read/write/trace/debug different the search algorithms covered
- Implement cycle checking and multiple path pruning for different algorithms
- Identify when they are appropriate
- Construct heuristic functions for specific search problems
- Formally prove A* optimality. 57
- Understand general ideas behind Dynamic Programming and MBA*


Representation
Reasoning Technique
Deterministic
Stochastic
Problem Type
Static
Constraint Satisfaction
Query
Sequential
Planning
Arc Consistency
Vars + Constraints
Search
Logics
Search
STRIPS
Search
Belief Nets
Variable Elimination
Decision Nets
Variable Elimination
First Part of the Course

Course Overview

Environment

Standard vs Specialized Search

- We studied general state space search in isolation
- Standard search problem: search in a state space
- State is a "black box" - any arbitrary data structure that supports three problem-specific routines:
- goal test: goal(state)
- finding successor nodes: neighbors(state)

Markov Processes

Value Iteration

- if applicable, heuristic evaluation function: h(state)

- We will see more specialized versions of search for various problems

Course Overview

Representation

Reasoning Technique

Environment

Problem Type		Deterministic	Stochastic
Static	Constraint Satisfaction	*Vars + Constraints* Arc Consistency Search	
	Query	*Logics* Search	*Belief Nets* Variable Elimination
Sequential	Planning	*STRIPS* Search	*Decision Nets* Variable Elimination

We will look at Search in Specific R&R Systems

- Constraint Satisfaction Problems (CPS):
- State
- Successor function
- Goal test
- Solution
- Heuristic function • Query:
- State
- Successor function
- Goal test
- Solution
- Heuristic function
- Planning
- State
- Successor function
- Goal test

Course Overview

Environment

- Solution
- Heuristic function

Markov Processes

Value Iteration

Representation

Reasoning Technique

Problem Type		Deterministic	Stochastic
Static	Constraint Satisfaction	***Vars + Constraints*** Arc Consistency, Search	
	Query	***Logics*** Search	***Belief Nets*** Variable Elimination
Sequential	Planning	***STRIPS*** Search	***Decision Nets*** Variable Elimination

We'll start from CPS

Course Overview

Environment

Lecture Overview

- A few more points about the material from Lecture 6 (more than a recap)
- Other advanced search algorithms
- Intro to CSP (time permitting)

Markov Processes

Value Iteration

- Constraint Satisfaction Problems (CPS):
  - State
  - Successor function
  - Goal test
  - Solution
  - Heuristic function

We will look at Search for CSP

- Query:
- State
- Successor function
- Goal test
- Solution
- Heuristic function
- Planning

- State
- Successor function
- Goal test
- Solution
- Heuristic function

Lecture Overview

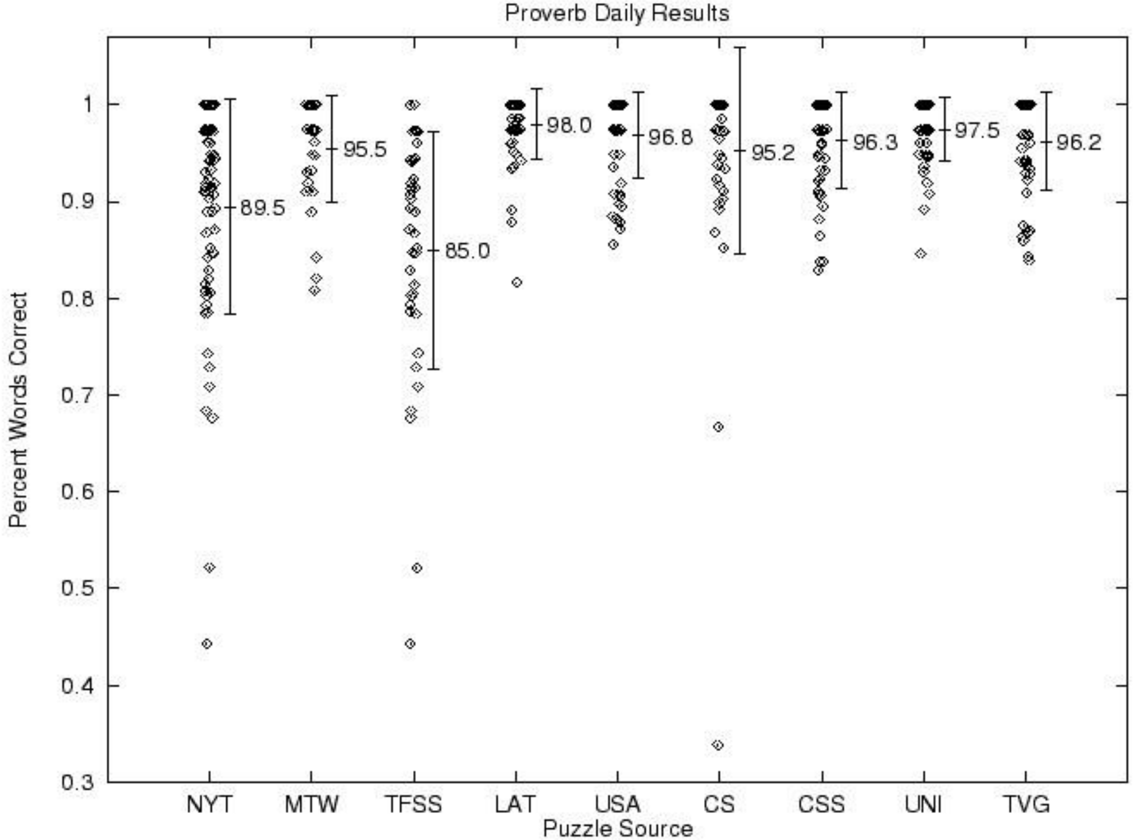
- Recap of previous lecture
- Other advanced search algorithms
- Intro to CSP (time permitting)

Daily Puzzles

370 puzzles from 7 sources.

Summary statistics:

- 95.3% words correct (miss three or four words per puzzle)
- 98.1% letters correct
- 46.2% puzzles completely correct



P	O	L	O	N	E		P	A	L	O	M	I	N	O
A	S	I	M	O	V		I	S	O	L	A	T	E	D
S	L	E	E	V	E		T	H	W	A	R	T	E	D
T	I	G	G	E	R		C	O	R	N	Y			
A	N	E	A	L	E		A	R	I	D		J	A	M
				E	S	P	I	E	S		L	O	G	O
S	E	A	O	T	T	E	R		E	E	E	N	O	N
A	B	B	O	T		A	N	A		U	S	A	G	E
B	O	O	Z	E	S		S	N	A	P	S	H	O	T
E	N	V	Y		P	L	I	N	T	H				
R	Y	E		H	I	E	S		T	E	A	S	E	T
			K	A	R	E	L		I	M	P	A	L	E
M	A	R	I	N	A	R	A		M	I	A	S	M	A
A	B	E	R	D	E	E	N		E	S	C	H	E	R
J	U	N	K	Y	A	R	D		S	M	E	A	R	S

Source: *Michael Littman*

CSPs: Crossword Puzzles - Proverb

Constraint Satisfaction Problems (CSP)

- In a CSP
  - state is defined by a set of variables V_i with values from domain D_i
  - goal test is a set of constraints specifying
    1. allowable combinations of values for subsets of variables (hard constraints)
    2. preferences over values of variables (soft constraints)

Dimensions of Representational Complexity (from lecture 2)

- Reasoning tasks (Constraint Satisfaction / Logic&Probabilistic Inference / Planning)
- Deterministic versus stochastic domains

Some other important dimensions of complexity:

•• Explicit state or **Explicit state** features or or **features** relations or **relations**

- Flat or hierarchical representation
- Knowledge given versus knowledge learned from experience
- Goals versus complex preferences

- Single-agent vs. multi-agent

Explicit State vs. Features (Lecture 2)

How do we model the environment?

- You can enumerate the possible states of the world
- A state can be described in terms of features
- Assignment to (one or more) features

- Often the more natural description • 30 binary features can represent

$$\boxed{2^{30}=1{,}073{,}741{,}824 \text{ states}}$$

Variables/Features and Possible Worlds

- Variable: a synonym for feature
- We denote variables using capital letters
- Each variable V has a domain dom(V) of possible values
- Variables can be of several main kinds:
- Boolean: |dom(V)| = 2
- Finite: |dom(V)| is finite

- Infinite but discrete: the domain is countably infinite
- Continuous: e.g., real numbers between 0 and 1
- Possible world:
- Complete assignment of values to each variable
- This is equivalent to a state as we have defined it so far
  - ✓ Soon, however, we will give a broader definition of state, so it is best to start distinguishing the two concepts .

Example (lecture 2)

Mars Explorer Example

Weather	{S, C}
Temperature	[-40, 40]

Longitude [0, 359]

Latitude $\boxed{[0, 179]}$

One possible world (state) $\boxed{\{S, -30, 320, 210\}}$

Number of possible (mutually exclusive) worlds (states)

$$\boxed{2 \times 81 \times 360 \times 180}$$

Product of cardinality of each domain

… always exponential in the number of variables

Constraint Satisfaction Problems (CSP)

- Allow for usage of useful general-purpose algorithms with more power than standard search algorithms

- They exploit the multi-dimensional nature of the problem and the structure provided by the goal

set of constraints, *not* black box.