PROBLEM SOLVING AND SEARCH

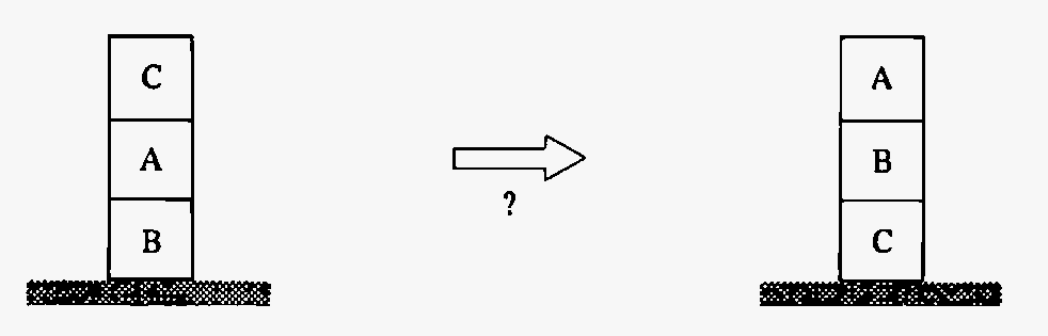
BASIC TECHNIQUES

Ivan Bratko
Faculty of Computer and Information Sc.
Ljubljana University

PROBLEM SOLVING

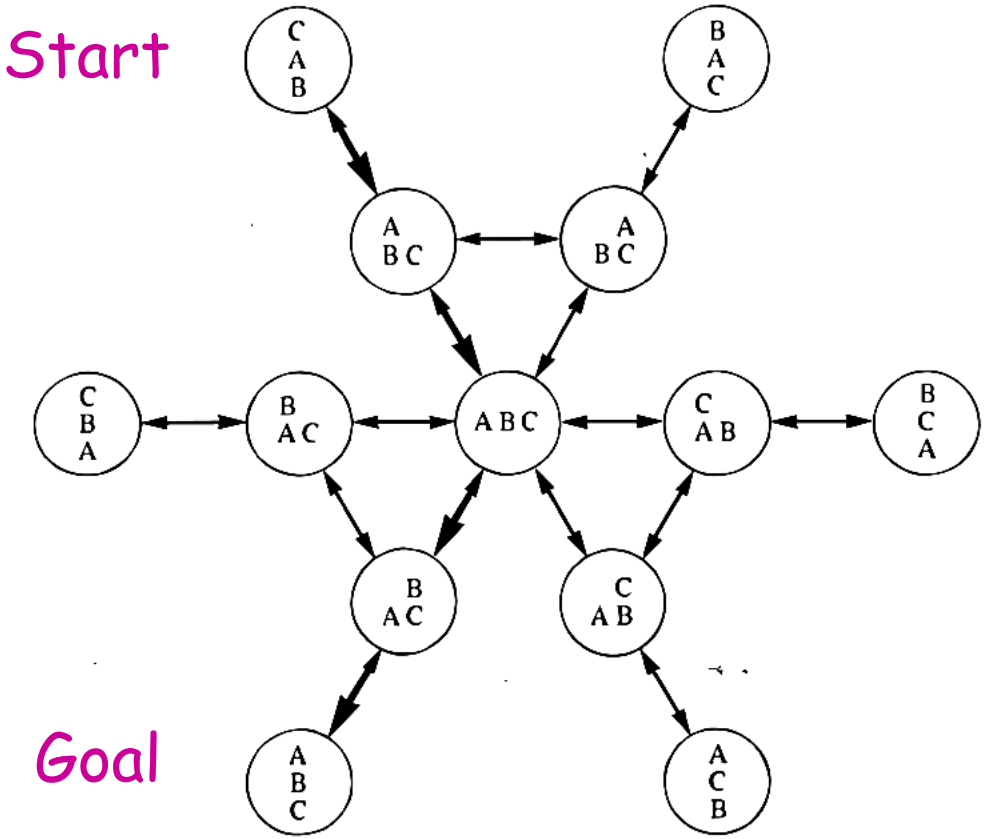
- Problems generally represented as graphs
- Problem solving ~ searching a graph

A problem from blocks world



Find a sequence of robot moves to re-arrange blocks

Blocks world state space



State Space

- State space = Directed graph
- Nodes ~ Problem situations
- Arcs ~ Actions, legal moves

- Problem = (State space, Start, Goal condition)
- Note: several nodes may satisfy goal condition (e.g. in case goal is on(B,C))

- Solving a problem ~ Finding a path in a graph
- Problem solution ~ Path from start to a goal node

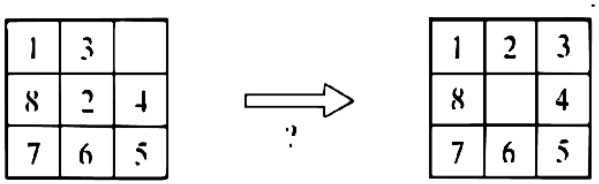
Examples of representing problems in state space

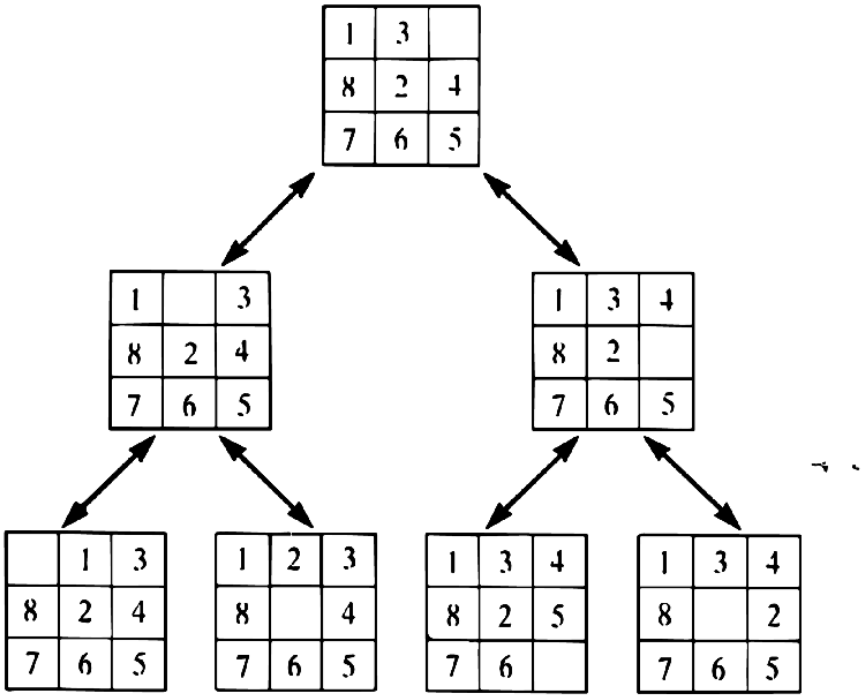
- Blocks world planning
- 8-puzzle, 15-puzzle
- Mobile robots on grid
- Robots in a warehouse
- Timetable construction
- Travelling salesman

How can these problems be represented by graphs?

Propose corresponding state spaces

8-puzzle

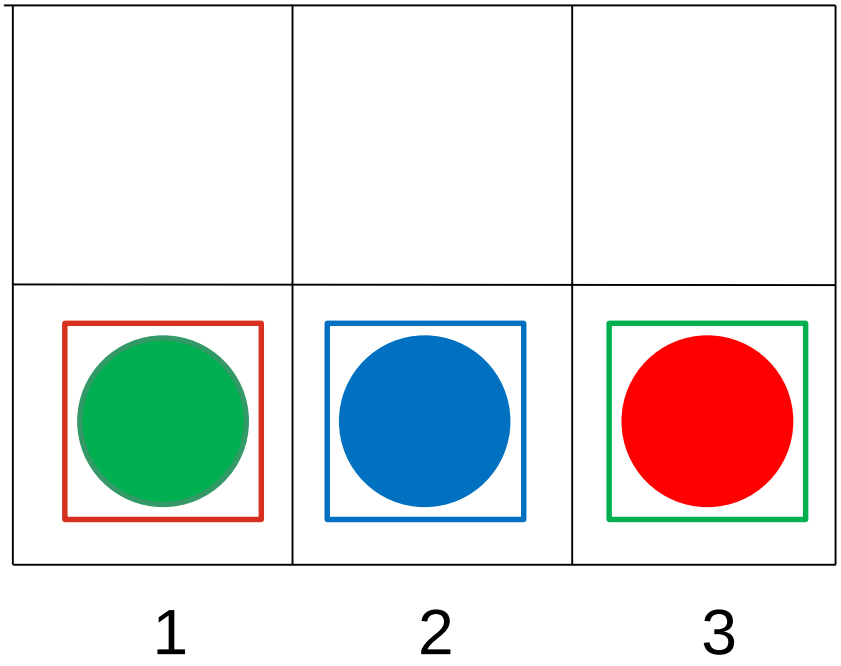

1 3
8 2 4
7 6 5
1 2 3
8 4
7 6 5


1 3
8 2 4
7 6 5
1 3
8 2 4
7 6 5
1 3 4
8 2
7 6 5
1 3
8 2 4
7 6 5
1 2 3
8 4
7 6 5
1 3 4
8 2 5
7 6
1 3 4
8 2
7 6 5

Three robots on grid moving in parallel

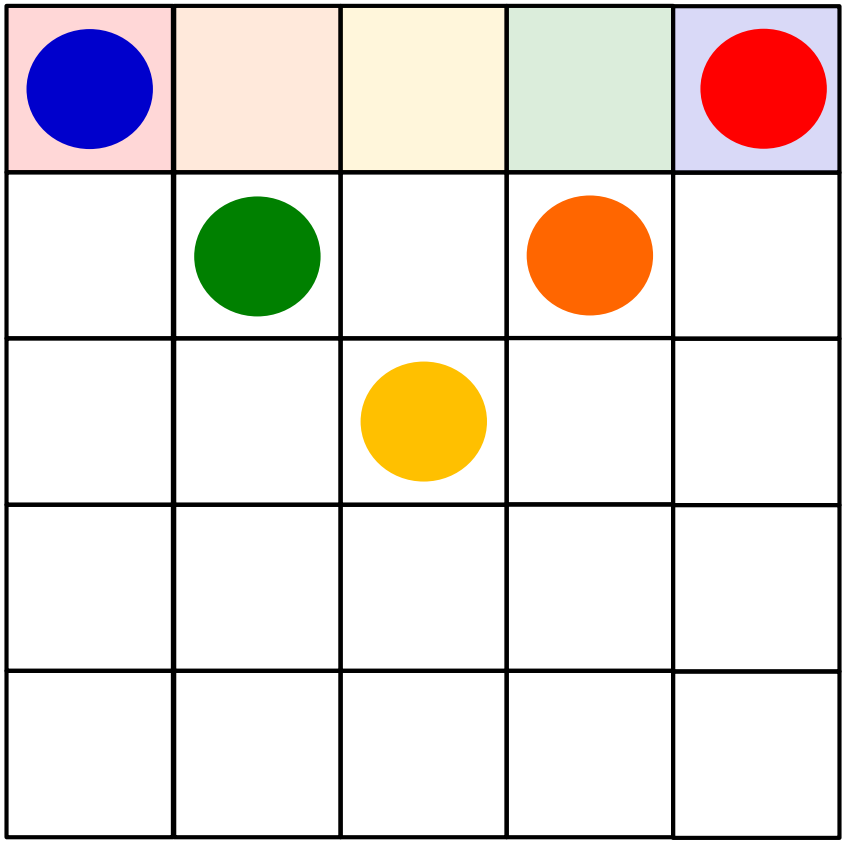
- Robots: red, blue, green; Locations at bottom: 1, 2, 3
- Start state: at(green, 1), at(blue, 2), at (red, 3)
- Goal: at(green, 3), at(blue ,2), at(red, 1)

  Goal locations are indicated by colour squares
- How many time steps are needed for this task?



FIVE ROBOTS ON GRID 5 x 5

Robots creating a rainbow in top row



State spaces for optimisation problems

- Optimisation: **minimise cost** of solution

- In blocks world:
  actions may have different costs
  (blocks have different weights, ...)

- Assign costs to arcs
- Cost of solution = cost of solution path
- That is: sum of arc-costs along the path

More complex examples

- Making a time table
- Production scheduling
- Grammatical parsing
- Interpretation of sensory data
- Modelling from measured data
- Finding scientific theories that account for experimental data
- Machine learning: find model that matches observations

WHAT IS THE CHALLENGE IN SEARCH?

- Small state space: search is trivial
- Large state space: search may be prohibitive
- Size of state space typically **grows exponentially** with length of solution path

SEARCH METHODS

- **Uninformed techniques**:

  systematically search complete graph, unguided

- **Informed methods**:

  Use problem specific information to guide search in promising directions

- What is “promising”?
- Domain specific knowledge required
- ***Heuristics***

Basic search methods - uninformed

- Depth-first search
- Breadth-first search
- Iterative deepening

Informed, heuristic search

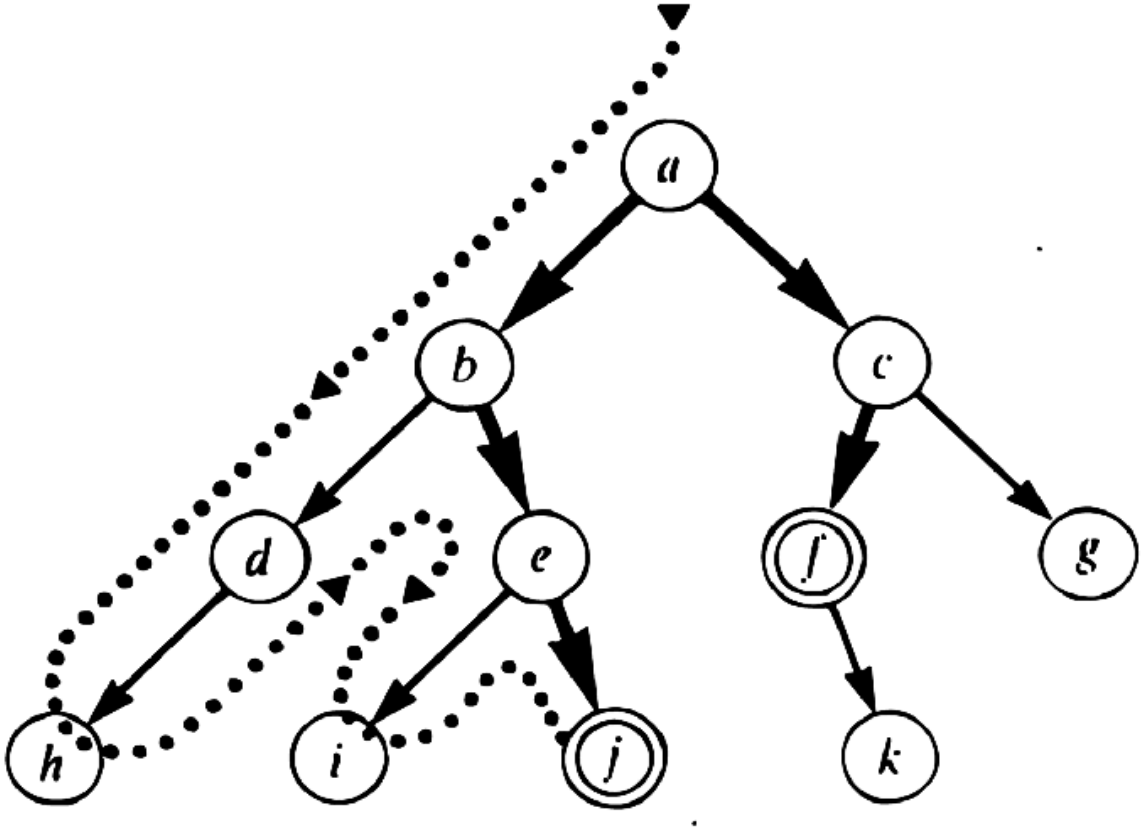
- Best-first search
- Hill climbing, steepest descent
- Algorithm A*
- Beam search
- Algorithm IDA* (Iterative Deepening A*)
- Algorithm RBFS (Recursive Best First Search)
- RTA*
- LRTA*
- ...

Direction of search

- Forward search: from start to goal
- Backward search: from goal to start
- Bidirectional search

Depth-first search

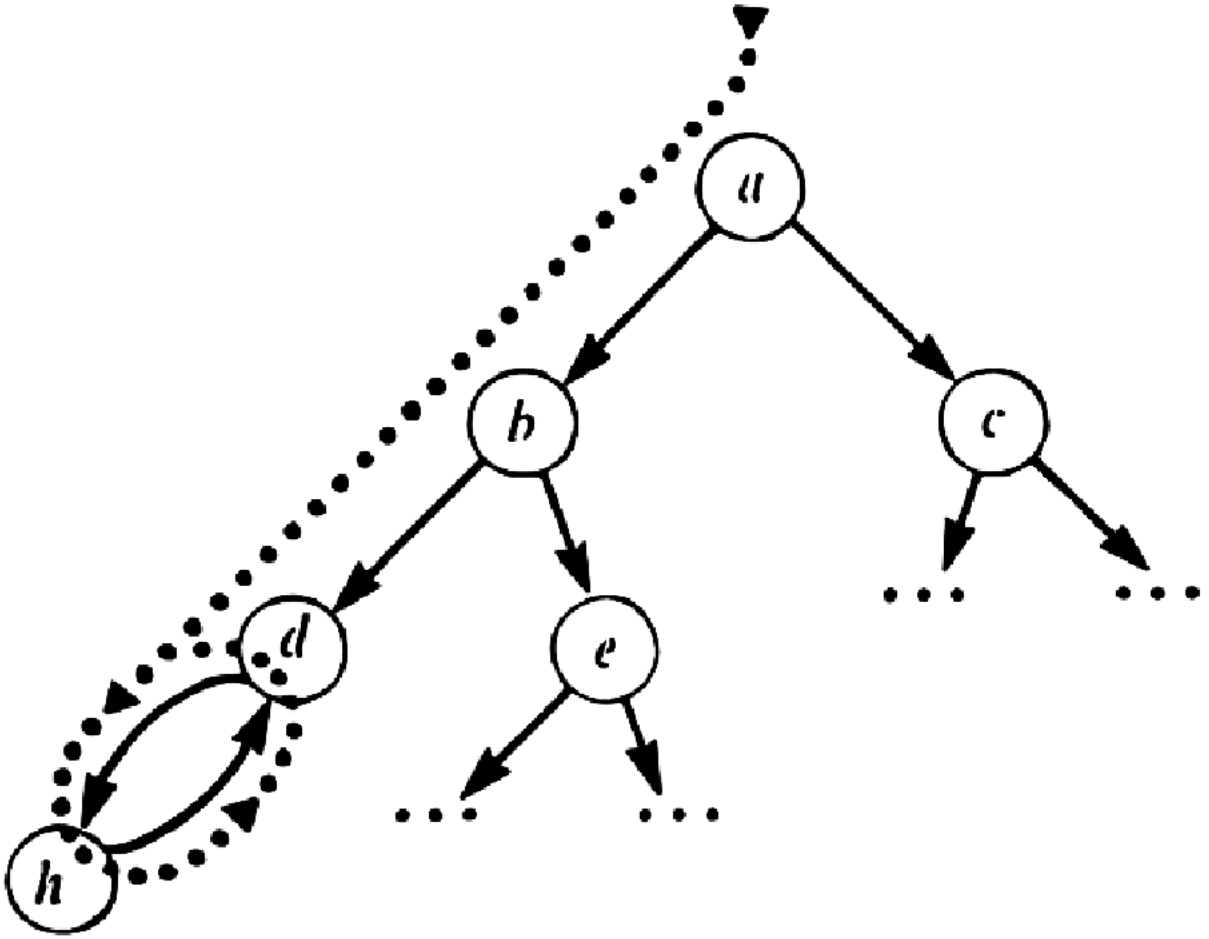
Example: a is start state, f and j are goal states



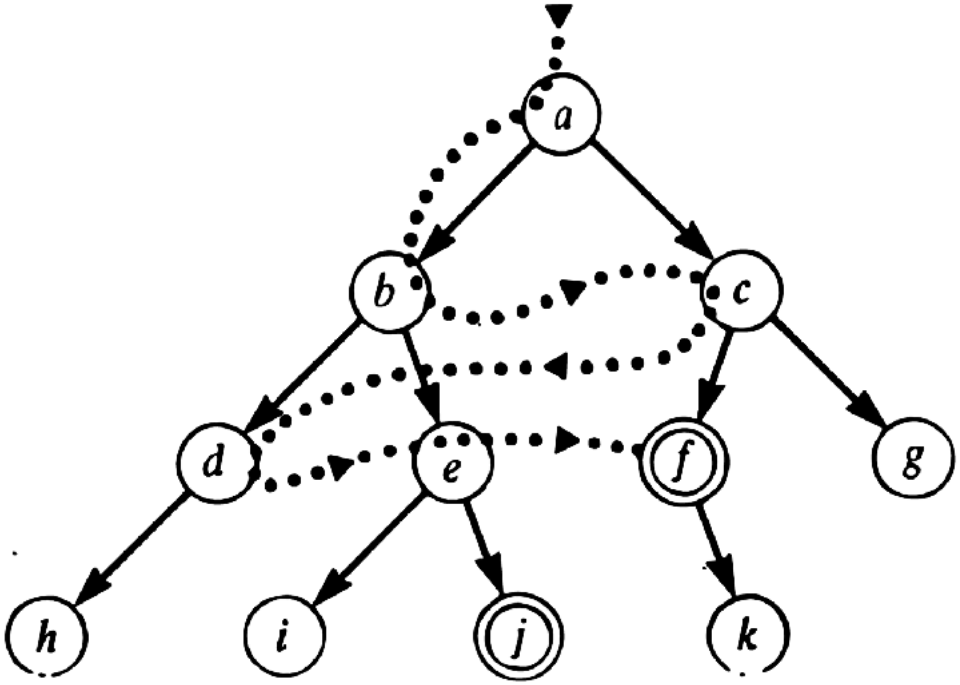
Properties of depth-first search program

- Is not guaranteed to find shortest solution first
- May get lost in infinite loop (should check for cycles)
- **Has low space complexity**: only proportional to depth of search
- Only requires memory to store the current path from start to the current node
- When moving to alternative path, previously searched paths can be forgotten

Depth-first search, problem of looping



Breadth-first search



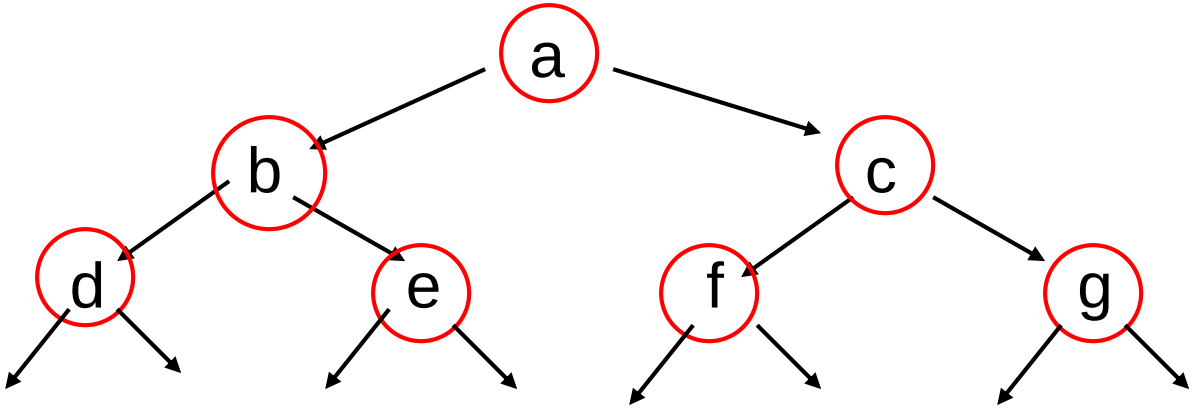
- **Guaranteed to find shortest solution first**
- breadth-first finds solution a-c-f
- depth-first finds a-b-e-j

A an implementation of breadth-first search

- Breadth-first search completes one level before moving on to next level
- **Has to keep in memory all the competing paths** that aspire to be extended to a goal node
- A possible representation of candidate paths: list of lists
- Easiest to store paths in reverse order;

  then, to extend a path, add a node as new head (easier than adding a node at end of list)

Candidate paths as list of lists



[[d,b,a], [e,b,a], [f,c,a], [g,c,a]]

On each iteration: Remove *first* candidate path, extend it and add extensions at *end* of list

ITERATIVE DEEPENING SEARCH

- Depth-limited depth-first search may miss a solution if depth-limit is set too low
- This may be problematic if solution length not known in advance
- Idea: start with small MaxDepth and
  increase MaxDepth until solution found
- For each MaxDepth execute depth-first search

ITERATIVE DEEPENING

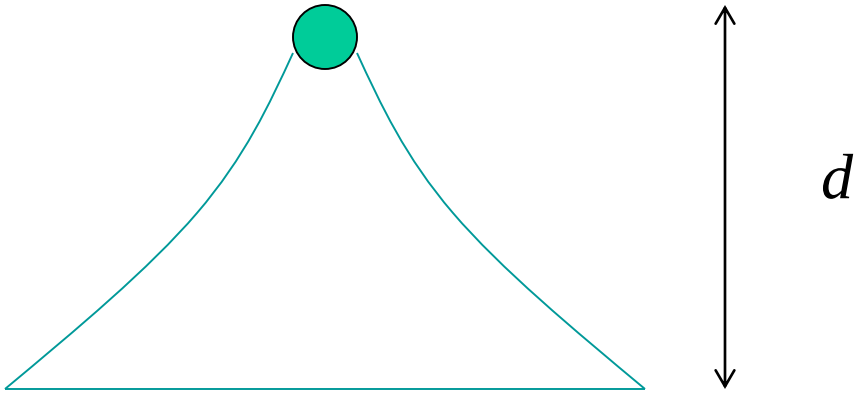
Repeat

depth-limited depth-first search with increasing depth limit

Until solution is found

Complexity of basic search methods

- For simpler analysis consider state-space as a tree
- Uniform branching *b*
- Solution at depth *d*



Number of nodes at level *d* : b^d

Time and space complexity orders

	Time	Space	Shortest solution guaranteed
Breadth-first	b^d	b^d	yes
Depth-first	b^{dmax}	d_{max}	no
Iterative deepening	b^d	d	yes

Time and space complexity

- Breadth-first and iterative deepening guarantee shortest solution
- Breadth-first: high space complexity
- Depth-first: low space complexity, but may search well below solution depth
- Iterative deepening: best performance in terms of orders of complexity

Time complexity of iterative deepening

- Repeatedly re-generates upper levels nodes
- Start node (level 1): *d* times
- Level 2: (*d* -1) times
- Level 3: (*d* -2) times, ...
- Notice: Most work done at last level *d* , typically more than at all previous levels

Overheads of iterative deepening due to re-generation of nodes

- Example: binary tree, *d* =3, #nodes = 15

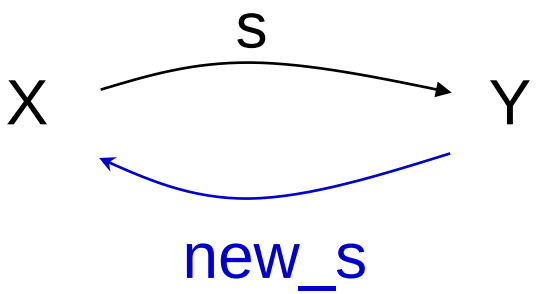
- Breadth-first generates 15 nodes
- Iterative deepening: 26 nodes
- Relative overheads due to re-generation: 26/15

- Generally:

$$\frac{\text{nodes generated by iter. deep}}{\text{nodes generated by breadth-first}} < \frac{b}{b-1}$$

Backward search

- Search from goal to start
- Can be realised by re-defining successor relation as:



For all X, Y: new_s(Y, X) if s(X, Y)

- New goal condition satisfied by start node
- Only applicable if original goal node(s) known
- Under what circumstances is backward search preferred to forward search?
- Depends on branching in forward/backward direction

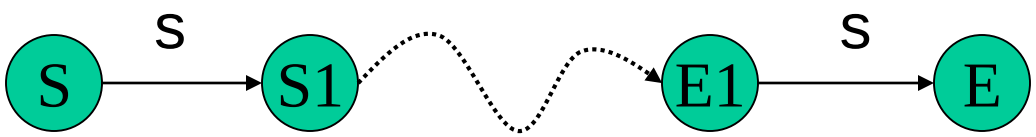
Bidirectional search

- Search progresses from both start and goal
- Standard search techniques can be used on re-defined state space
- Problem situations defined as pairs of form:

  StartNode – GoalNode
- This assumes goal node is known

Re-defining state space for bidirectional search

Original successor relation s:



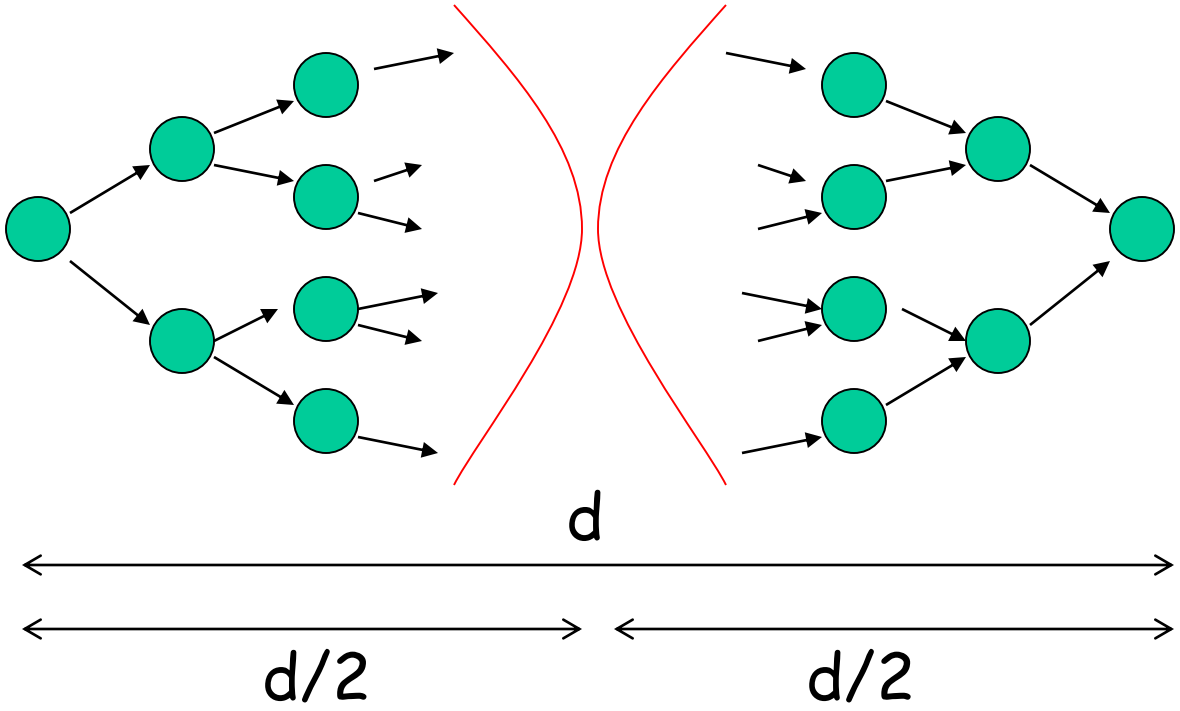
New space: Nodes are pairs of form S - E

new_goal(S - S) % Both ends coincide

new_goal(S - E) if s(S, E) % Ends sufficiently close

Complexity of bidirectional search

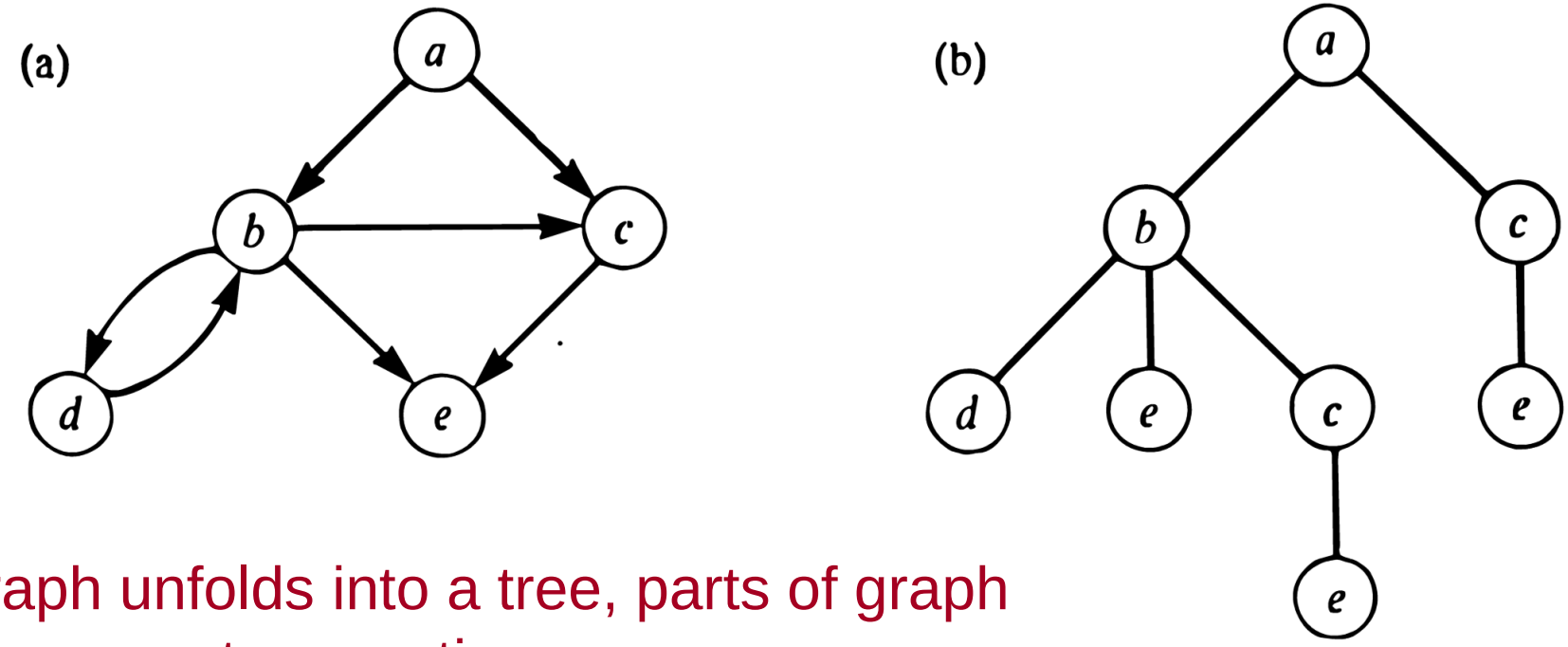
Consider the case: forward and backward branching both b, uniform



Time ~ $b^{d/2} + b^{d/2} << b^d$

Searching graphs

Do our techniques work on graphs, not just trees?



Graph unfolds into a tree, parts of graph may repeat many times

Techniques work, but may become very inefficient

Better: add check for repeated nodes