PROBLEM SOLVING AGENTS

Overview

Aims of the this lecture:

- introduce *problem solving*;
- introduce *goal formulation*;
- show how problems can be stated as *state space search*;
- show the importance and role of *abstraction*;
- introduce *undirected search*:
  - breadth 1st search;
  - depth 1st search.
- define main performance measures for search.

Problem Solving Agents

- Lecture 1 introduced *rational agents*.
- Now consider agents as *problem solvers*:

  Systems which set themselves *goals* and find *sequences of actions* that achieve these goals.
- What is a problem?

  A *goal* and a *means* for achieving the goal.
- The goal specifies the state of affairs we want to bring about.
- The means specifies the operations we can perform in an attempt to bring about the means.
- The difficulty is deciding which operations and what *order* to carry out the operations.

- Operation of problem solving agent:

```
/* s is sequence of actions */
repeat {
    percept = observeWorld();
    state = updateState(state, p);
    if s is empty then {
        goal = formulateGoal(state);
        prob = formulateProblem(state, goal);
        s = search(prob);
    }
    action = first(s);
    s = remainder(s);
}
until false; /* i.e., forever */
```

- Key difficulties:
  - `formulateGoal(...)`
  - `formulateProblem(...)`
  - `search(...)`
- It isn't easy to see how to tackle any of these.
- Here we will concentrate mainly on search.

Goal Formulation

- Where do an agent's goals come from?
  - Agent is a *program* with a *specification*.
  - Specification is to maximise performance measure.
  - Should *adopt goal* if achievement of that goal will maximise this measure.
- Goals provide a *focus* and *filter* for decision-making:
  - *focus*: need to consider how to achieve them;
  - *filter*: need not consider actions that are incompatible with goals.
- For this course, we will assume that an agent is given its goals.

Problem Formulation

- Once goal is determined, formulate the problem to be solved.
- First determine set of possible states S of the problem.
- Then problem has:
  - *initial state* — the starting point, s_0;
  - *operations* — the actions that can be performed, $\{o_1, \ldots, o_n\}$.
  - *goal* — what you are aiming at — subset of S.

- The initial state together with operations determines *state space* of problem.
- Operations cause *changes* in state.
- Solution is a sequence of actions such that when applied to initial state s_0, we have goal state.
- What does this look like?

Examples of Toy Problems

- *Example 1*: The 8 puzzle.

  Do the following transformation, moving tile from occupied space to filled space.

2	8	3
1	6	4
7		5

⇒

1	2	3
8		4
7	6	5

- Initial state as shown above.
- Goal state as shown above.
- Operations:
  - o_1: move any tile to left of empty square to right;
  - o_2: ?
  - o_3: ?
  - o_4: ?

- What state space does this define?

- Example 2: The *n* queens problem from chess.
- Place *n* queens on chess board so that no queen can be taken by another.
- Initial state: empty chess board.
- Goal state: *n* queens on chess board, one occupying each space, so that none can take others.
- Operations: place queen in empty square.

Solution Cost

- For most problems, some solutions are better than others:
  - in 8 puzzle, number of moves to get to solution;
  - number of moves to checkmate;
  - length of distance to travel.
- Mechanism for determining *cost* of solution is *path cost function*.
- This is the length of the path through the state-space from the initial state to the goal state.

- As an example, consider the following state in the 8-puzzle:

2	8	3
1	6	4
7		5

- How many moves are there to the solution?

- There are five moves:

1.
2.
3.
4.
5.

- What are they?
- What does the path through the solution space look like?

Problem Solving as Search

- In the state space view of the world, finding a solution is finding a path through the state space.
- When we solve a problem like the 8-puzzle we have some idea of what constitutes the next best move.
- It is hard to program this kind of approach.
- Instead we start by programming the kind of repetitive task that computers are good at.
- A *brute force* approach to problem solving involves *exhaustively searching* through the space of *all possible* action sequences to find one that achieves goal.

- Systematically generate a *search tree*
- The tree is built by taking the initial state and identifying some states that can be obtained by applying a single operator.
- These new states become the *children* of the initial state in the tree.
- These new states are then examined to see if they are the goal state.
- If not, the process is repeated on the new states.
- We can formalise this description by giving an algorithm for it.

- General algorithm for search:

```
agenda = initial state;
while agenda not empty do{
   pick node from agenda;
   new nodes = apply operations to state;
   if goal state in new nodes
   then {
          return solution;
      }
   add new nodes to agenda;
}
```

- Note the difference between *state space* and *search tree*.
- State space is every possible state and the relationships between them.
  - It is static.
- Search tree the set of states the agent has looked at (is looking at) and some of the relationships between them.
  - It is dynamic.

- Question: How to pick states for expansion?
- Two obvious solutions:
  - depth first search;
  - breadth first search.

Breadth First Search

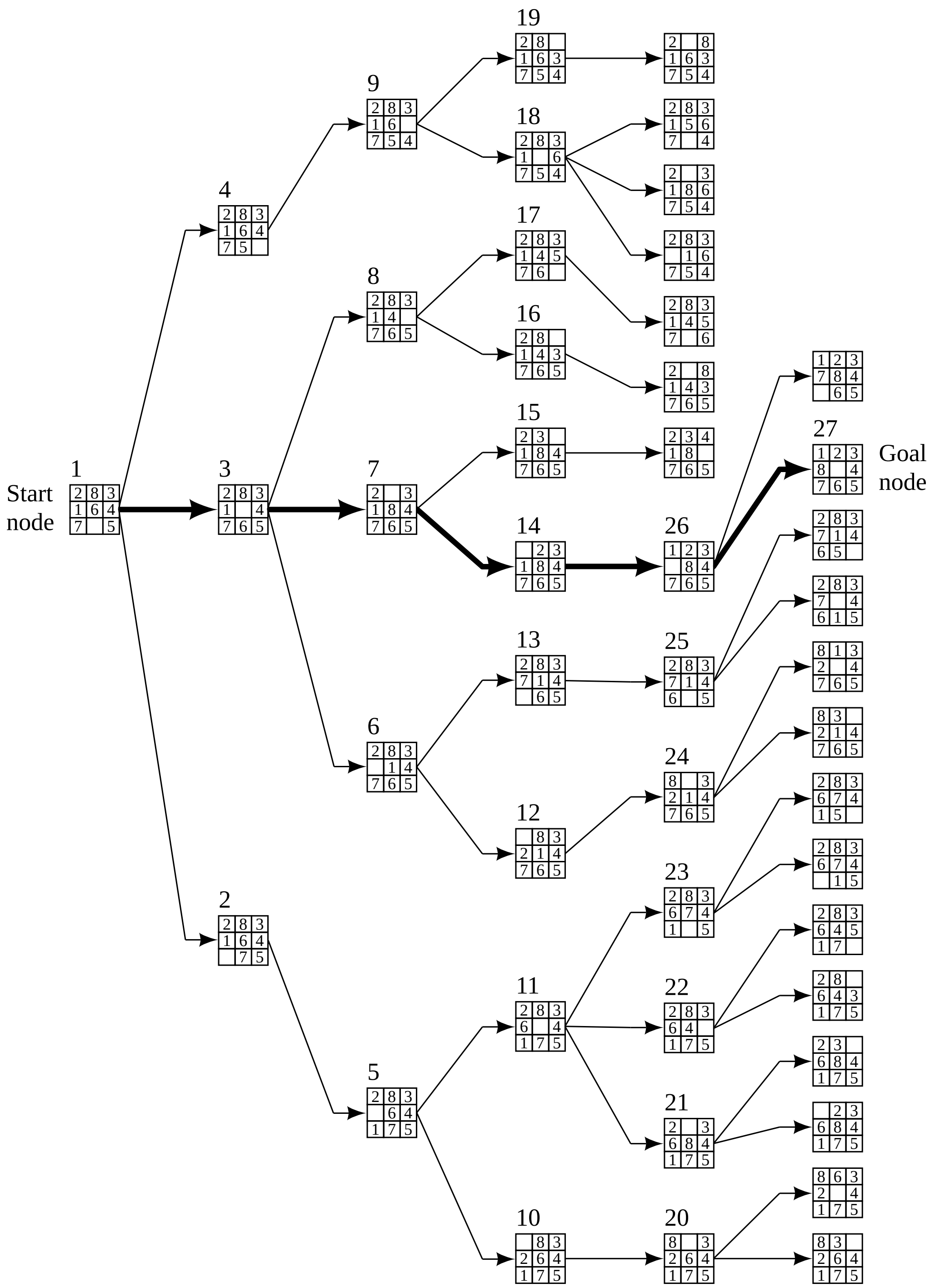
- Start by *expanding* initial state — gives tree of depth 1.
- Then expand *all* nodes that resulted from previous step — gives tree of depth 2.
- Then expand *all* nodes that resulted from previous step, and so on.
- Expand nodes at depth n before level $n + 1$.

```
/* Breadth first search */

agenda = initial state;

while agenda not empty do
{
    pick node from front of agenda;
    new nodes = apply operations to state;
    if goal state in new nodes then
    {
         return solution;
    }

    APPEND new nodes to END of agenda;
 }
```

- For the 8-puzzle as so:

2	8	3
1	6	4
7		5

- We have the following state space:


Start node
Goal node
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27

- Given this numbering of the states, the agenda would look like

1. 1
2. 2, 3, 4
3. 3, 4, 5
4. 4, 5, 6, 7, 8
5. 5, 6, 7, 8, 9
6. 6, 7, 8, 9, 10, 11.
7. . . .

- Advantage: *guaranteed* to reach a solution if one exists.
- If all solutions occur at depth n, then this is good approach.
- Disadvantage: time taken to reach solution!
- Let b be *branching factor* — average number of operations that may be performed from any level.
- If solution occurs at depth d, then we will look at

$$1 + b + b^2 + \cdots + b^d$$

nodes before reaching solution — *exponential*.

- Time for breadth first search:

Depth	Nodes	Time
0	1	1 msec
1	11	.01 sec
2	111	.1 sec
4	11,111	11 secs
6	10^6	18 mins
8	10^8	31 hours
10	10^{10}	128 days
12	10^{12}	35 years
14	10^{14}	2500 years
20	10^{20}	3^{15} years

- *Combinatorial explosion!*

Importance of ABSTRACTION

- When formulating a problem, it is crucial to pick the right level of *abstraction*.
- Example: Given the task of driving from where I used live in Manhatatan to Boston.
- Some possible actions…
  - depress clutch;
  - turn steering wheel right 10 degrees;

… inappropriate level of *abstraction*.

Too much *irrelevant detail*.

- Better level of abstraction:
  - Take the FDR drive north
  - Take the Cross County turnoff
  - Merge onto the Hutchinson River Parkway

  … and so on.
- Getting abstraction level right lets you focus on the specifics of problem and is one way to combat the combinatorial explosion.
- (Tell that to Mapquest/Google Maps).

Depth First Search

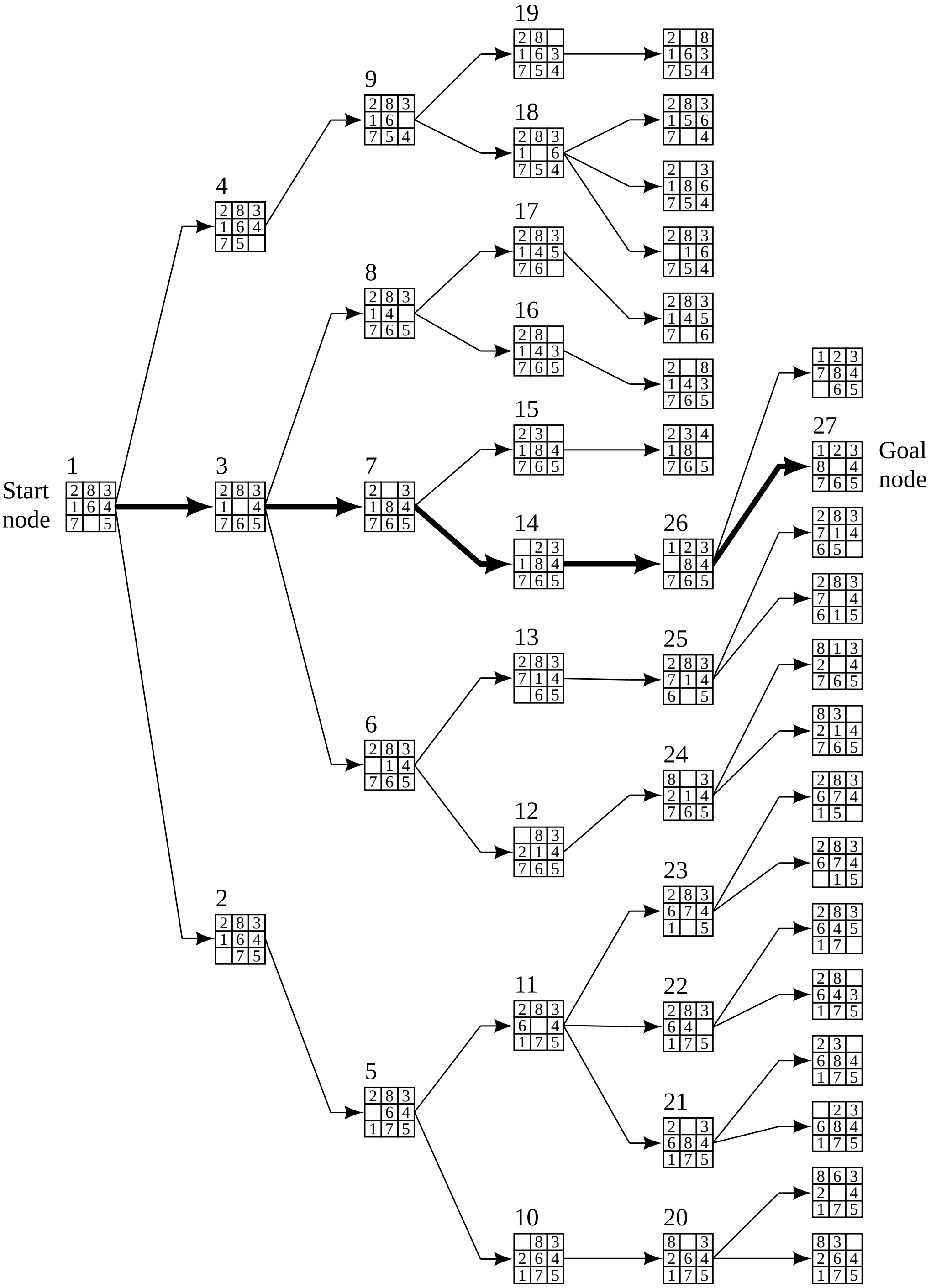
- Start by expanding initial state.
- Pick one of nodes resulting from 1st step, and expand it.
- Pick one of nodes resulting from 1nd step, and expand it, and so on.
- Always expand *deepest* node.
- Follow one “branch” of search tree.

```
/* Depth first search */

agenda = initial state;

while agenda not empty do
{
    pick node from front of agenda;
    new nodes = apply operations to state;
    if goal state in new nodes then
    {
         return solution;
    }

put new nodes on FRONT of agenda;
}
```

- For the 8-puzzle as so:

2	8	3
1	6	4
7		5

- We have the following state space:


Start
node
Goal
node
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27

- Given this numbering of the states, the agenda would look like

  1. 1
  2. 2, 3, 4
  3. 5, 3, 4
  4. 10, 11, 3, 4
  5. 20, 11, 3, 4
  6. …

- Depth first search is *not* guaranteed to find a solution if one exists.
- However, if it *does* find one, amount of time taken is much less than breadth first search.
- *Memory requirement* is much less than breadth first search.
- Solution found is *not* guaranteed to be the best.

Performance Measures for Search

- *Completeness*:
  Is the search technique *guaranteed* to find a solution if one exists?
- *Time complexity*:
  How many computations are required to find solution?
- *Space complexity*:
  How much memory space is required?
- *Optimality*:
  How good is a solution going to be w.r.t. the path cost function.

Algorithmic Improvements

- Are then any *algorithmic* improvements we can make to basic search algorithms that will improve overall performance?
- Try to get *optimality* and *completeness* of breadth 1st search with *space efficiency* of depth 1st.
- Not too much to be done about time complexity :-(

Depth Limited Search

- Depth first search has some desirable properties — space complexity.
- But if wrong branch is expanded (with no solution on it), then it won't terminate.
- Idea: introduce a *depth limit* on branches to be expanded.
- Don't expand a branch below this depth.

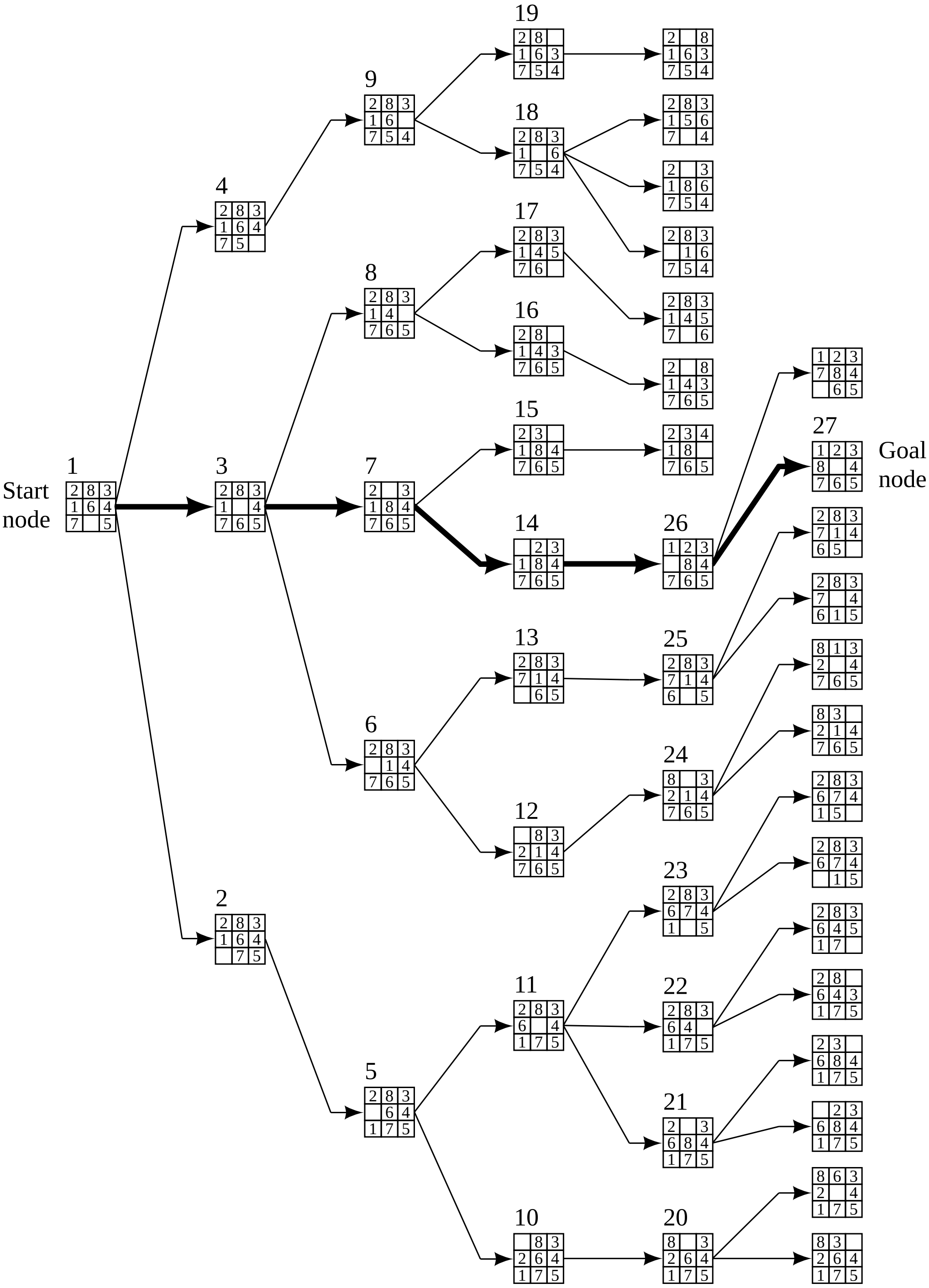
- General algorithm for depth limited search:

```
depth limit = max depth to search to;
agenda = initial state;
while agenda not empty do
  take node from front of agenda;
  new nodes = apply operations to node;
  if goal state in new nodes then {
    return solution;
  }
  if depth(node) < depth limit then {
    add new nodes to front of agenda;
  }
}
```

- For the 8-puzzle as so:

2	8	3
1	6	4
7		5

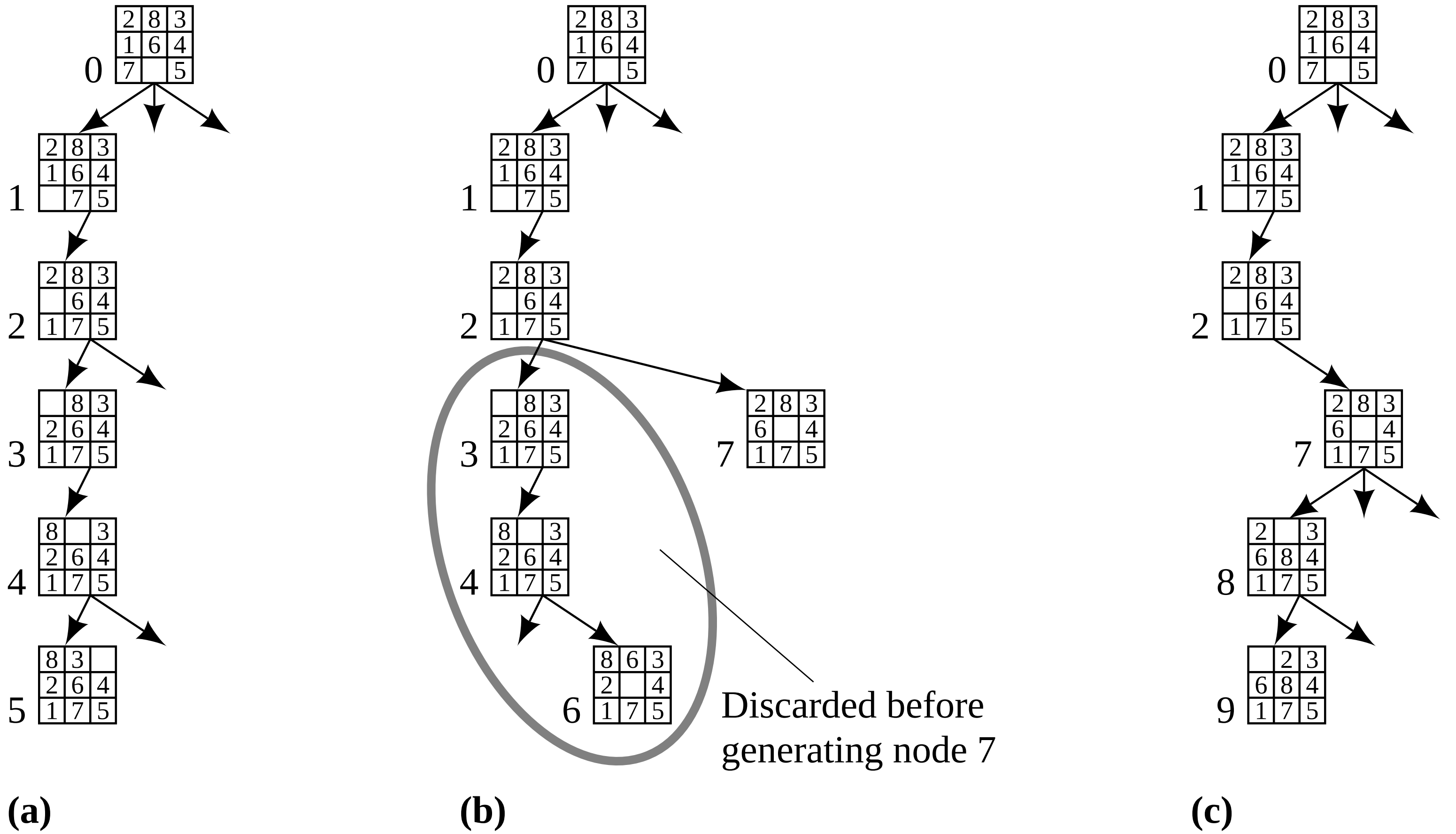
- We have the following state space:


Start node
Goal node
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27

- Given this numbering of the states, a depth limited search with depth limit of three would have an agenda that looks like

  1. 1
  2. 2, 3, 4
  3. 5, 3, 4
  4. 10, 11, 3, 4
  5. 11, 3, 4
  6. 3, 4
  7. 6, 7, 8, 4
  8. 12, 13, 7, 8, 4
  9. 13, 7, 8, 4
  10. . . .

- Let's look at the search tree in more detail:



- So, when we hit the depth bound, we don’t add any more nodes to the agenda.
- Then we pick the next node off the agenda.
- This has the effect of moving the search back to the last node above depth limit that that is “partly expanded”.
- This is known as *chronological backtracking*.
- The effect of the depth limit is to force the search of the whole state space down to the limit.
- We get the completeness of breadth-first (down to the limit), with the space cost of depth first.

Iterative Deepening

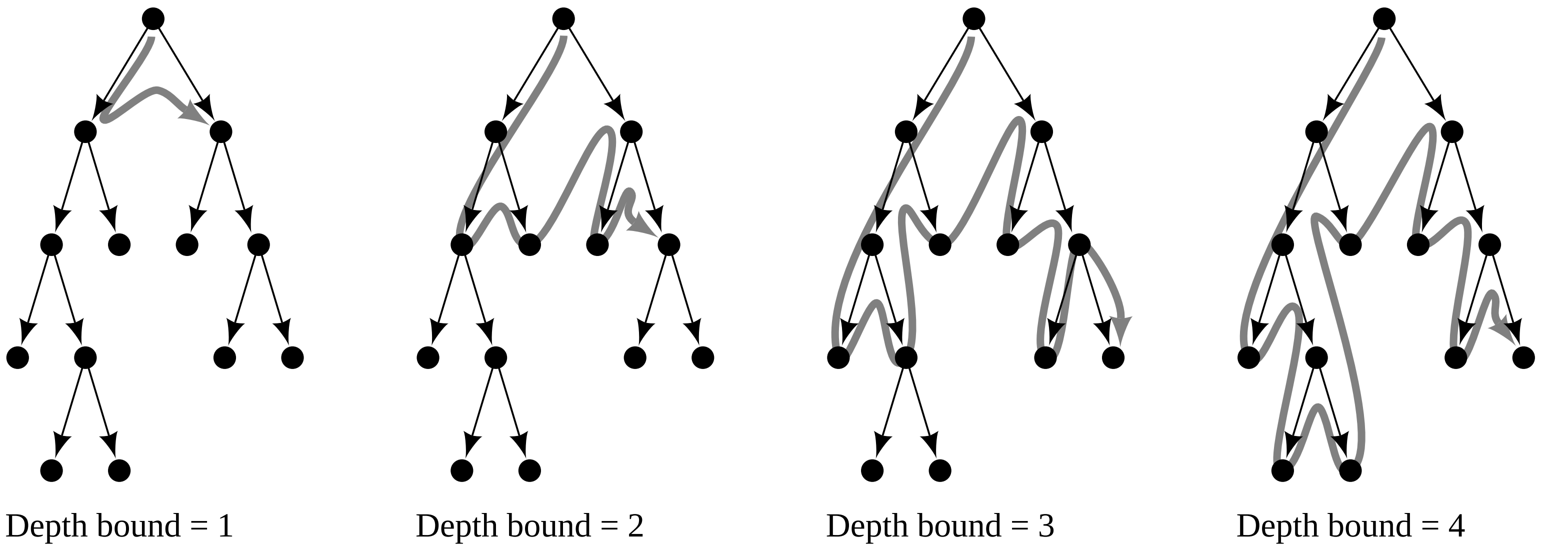
- Unfortunately, if we choose a max depth for d.l.s. such that shortest solution is longer, d.l.s. is not complete.
- Iterative deepening an ingenious *complete* version of it.
- Basic idea is:
  - do d.l.s. for depth 1; if solution found, return it;
  - otherwise do d.l.s. for depth n; if solution found, return it;
  - otherwise, . . .
- So we *repeat* d.l.s. for all depths until solution found.

- General algorithm for iterative deepening search:

```
depth limit = 1;
repeat {
  result = depth_limited_search(
    max depth = depth limit;
    agenda = initial node;
  );
  if result contains goal then {
    return result;
  }
  depth limit = depth limit + 1;
} until false; /* i.e., forever */
```

- Calls d.l.s. as subroutine.

- The search covers the whole state space down to the depth limit.



- The order it searches the nodes changes for each depth limit.

- Note that in iterative deepening, we *re-generate nodes on the fly.* Each time we do call on depth limited search for depth d, we need to regenerate the tree to depth $d-1$.
- Isn't this inefficient?
- Tradeoff *time* for *memory*.
- In general we might take a *little* more time, but we save a *lot* of memory.
- Now for breadth-first search to level d:

$$\begin{aligned} N_{bf} &= 1 + b + b^2 + \ldots b^d \\ &= \frac{b^{d+1} - 1}{b - 1} \end{aligned}$$

- In contrast a complete depth-limited search to level j:

$$N^j_{df} = \frac{b^{j+1} - 1}{b - 1}$$

- (This is just a breadth-first search to depth j.)
- In the worst case, then we have to do this to depth d, so expanding:

$$
\begin{aligned}
N_{id} &= \sum_{j=0}^{d} \frac{b^{j+1} - 1}{b - 1} \\
&\vdots \\
&= \frac{b^{d+2} - 2b - bd + d + 1}{(b - 1)^2}
\end{aligned}
$$

- For large d:

$$\frac{N_{id}}{N_{bf}} = \frac{b}{b-1}$$

- So for high branching and relatively deep goals we do a small amount more work.
- Example: Suppose $b = 10$ and $d = 5$.

  Breadth first search would require examining $111,111$ nodes, with memory requirement of $100,000$ nodes.

  Iterative deepening for same problem: $123,456$ nodes to be searched, with memory requirement only 50 nodes.

  Takes 11% longer in this case.

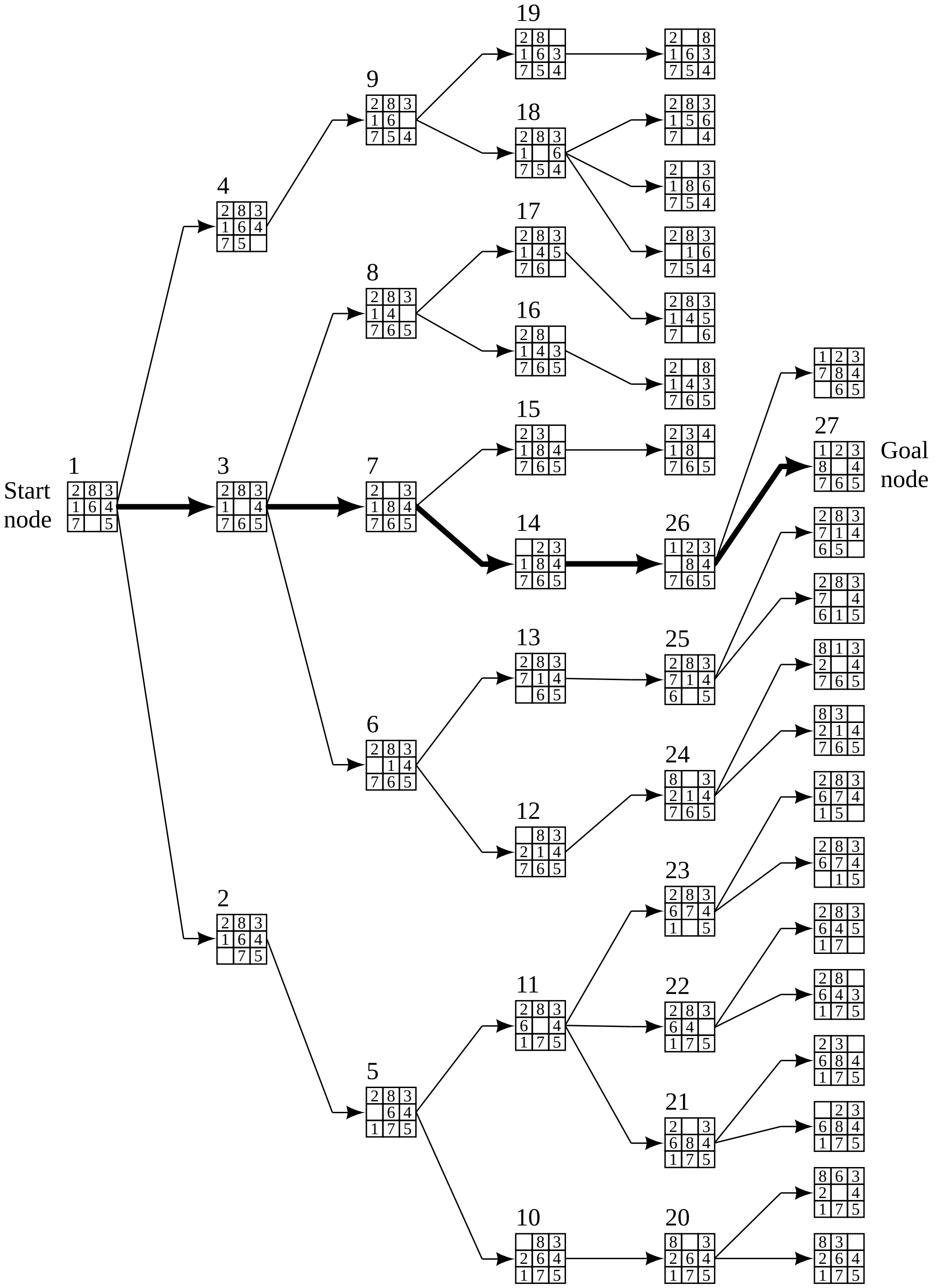
- For the 8-puzzle setup as:

2	8	3
1	6	4
7		5

⇒

1	2	3
8		4
7	6	5

- What would iterative deepening search look like?
- Well, it would explore the state space:


Start node
Goal node

- In the following way.
- States would be expanded in the order:

1. 1
2. 1, 2, 3, 4
3. 1, 2, 5, 3, 6, 7, 8, 4, 9.
4. 1, 2, 5, 10, 11, 3, 6, 12, 13, 7, 14, 15, 8, 16, 17, 4, 9, 18, 19.
5. ...

- Note that these are the states *visited*, not the nodes on the agenda.

Bi-directional Search

- Suppose we search from *the goal state backwards* as well as from *initial state forwards*.
- Involves determining *predecessor* nodes to goal, and then looking at predecessor nodes to this, ...
- Rather than doing one search of b^d, we do *two* $b^{d/2}$ searches.
- *Much* more efficient.

- Example:

  Suppose $b = 10$, $d = 6$.

  Breadth first search will examine nodes.

  Bidirectional search will examine nodes.

- Can combine different search strategies in different directions.
- For large *d*, is still impractical!

Summary

- This lecture introduced the basics of problem solving.
- In particular it discussed *state space* models and looked at the basic techniques for solving them.
  - Search for the goal.
  - Path through state space is the solution.
- We also looked at two techniques for search:
  - Breadth first.
  - Depth first.