Better Search

Improved Uninformed Search

CIS 32

Functionally

PROJECT 1: **Lunar Lander Game**

- Demo + Concept
- Open-Ended: No *One Solution*
- Menu of Point Options
- Get Started NOW!!!
- Demo After Spring Break

Today:

Wrap up Basic Search

Improvements on Uninformed Search

Re-Cap on Problem Solving Through Search

Agents that solve problems through Search (as opposed to Behavior-Based Agents)

Goal is **given** in the agent specification

Actions/Operations are **abstracted**

State-Space is formalized (i.e. modeled for the agent to search)

State-Space Modeled as a **Tree Structure**

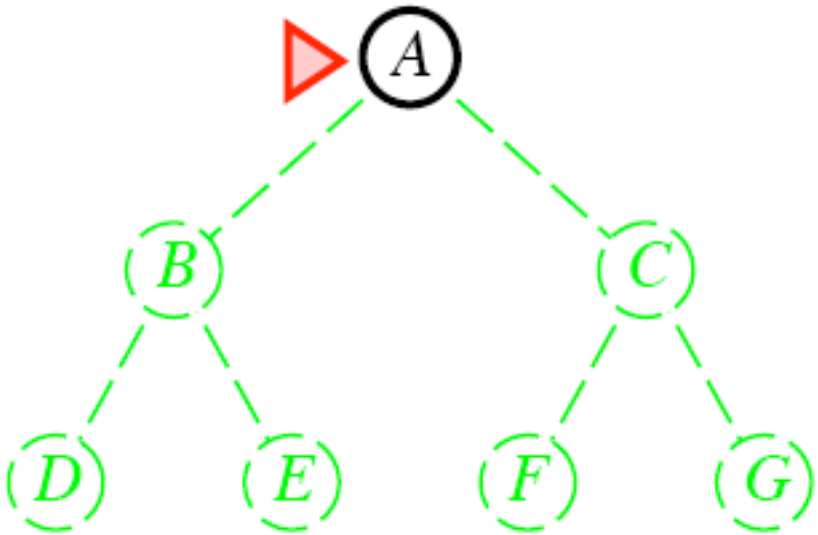
Uninformed Search:

- Breath First Search
- Uniform Search
- Depth First Search

Breadth First Search

- Start by expanding initial state — gives tree of depth 1.
- Then expand all nodes that resulted from previous step

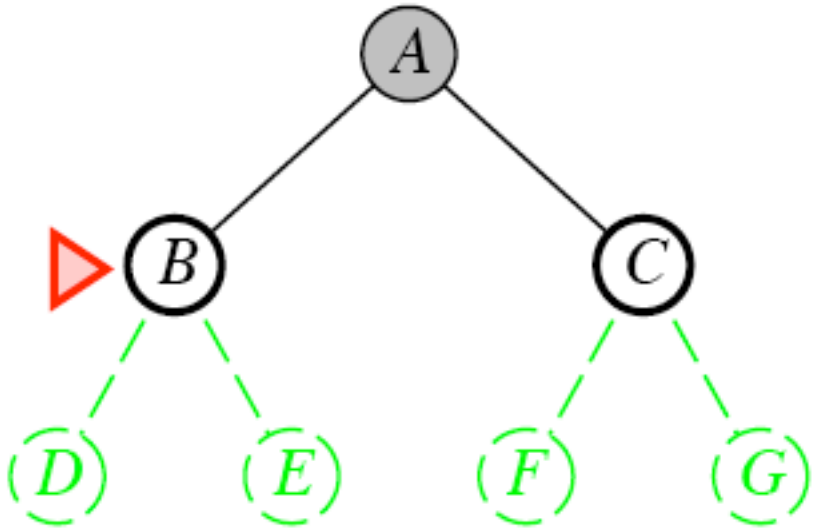
  — gives tree of depth 2.
- Then expand all nodes that resulted from previous step, and so on.
- Expand nodes at depth n before level n + 1.



Breadth First Search

- Start by expanding initial state — gives tree of depth 1.
- Then expand all nodes that resulted from previous step

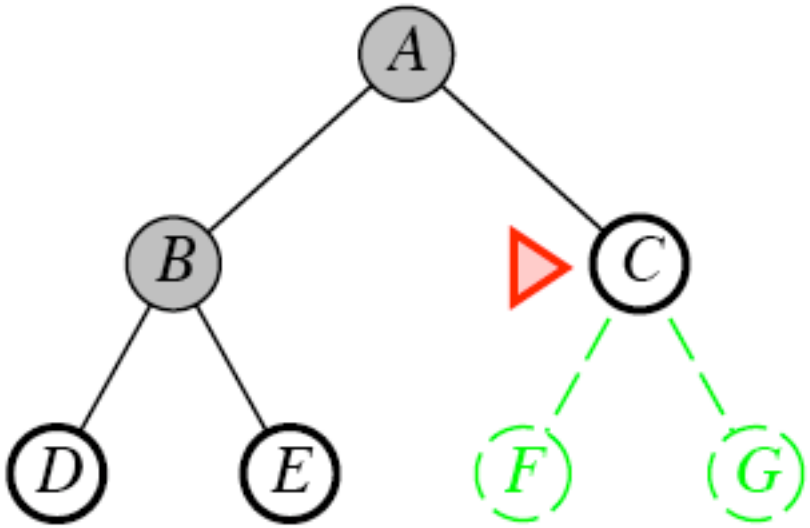
  — gives tree of depth 2.
- Then expand all nodes that resulted from previous step, and so on.
- Expand nodes at depth n before level n + 1.



Breadth First Search

- Start by expanding initial state — gives tree of depth 1.
- Then expand all nodes that resulted from previous step

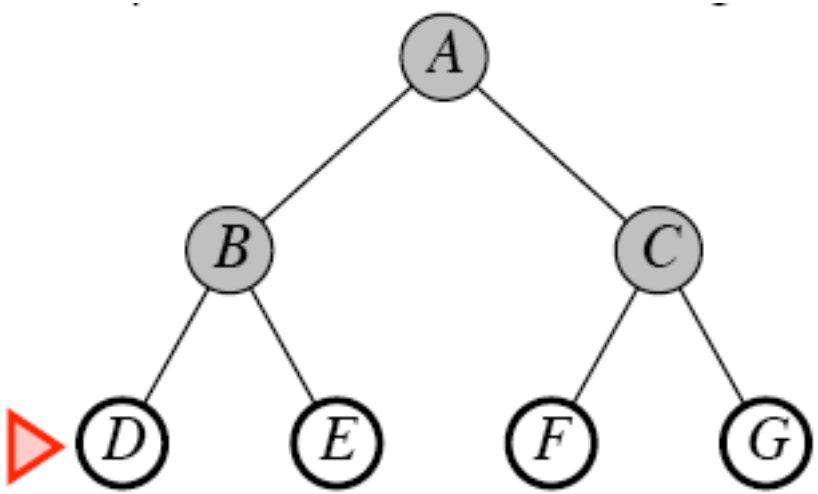
  — gives tree of depth 2.
- Then expand all nodes that resulted from previous step, and so on.
- Expand nodes at depth n before level n + 1.



Breadth First Search

- 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

- **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

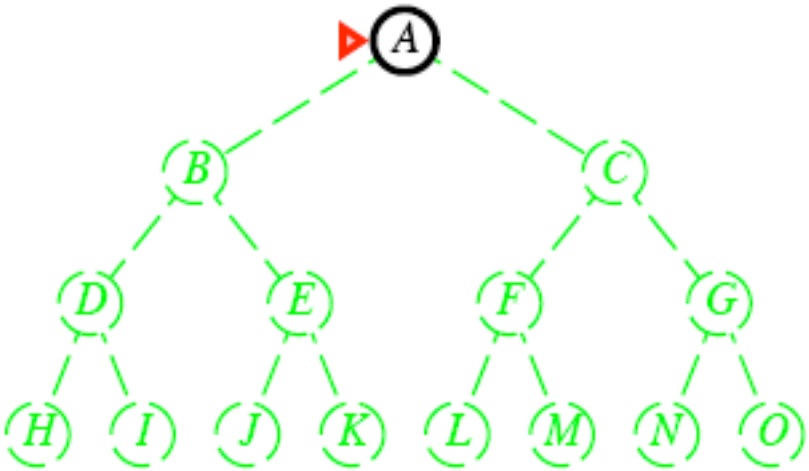
$$1 + b + b^2 + \cdots + b^d$$

nodes before reaching solution — *exponential*.

- How else can we search the State-Space?

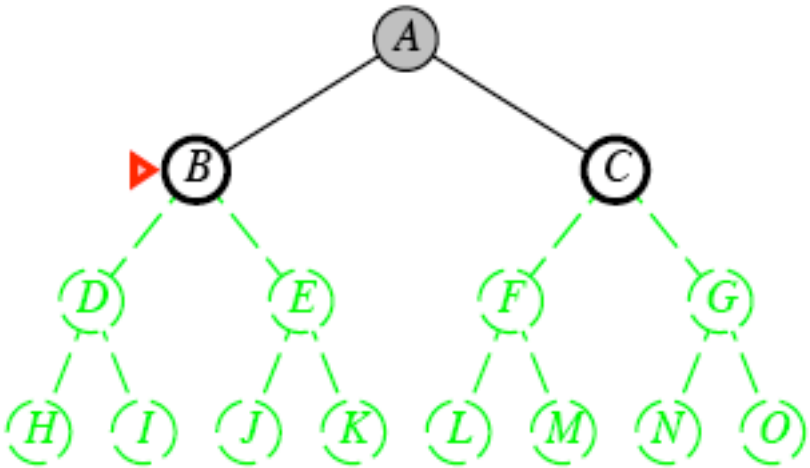
Depth First Search

- Start by expanding initial state.
- Pick one of nodes resulting from **first** step, and expand it.
- Pick one of nodes resulting from **second** step, and expand it, and so on.
- Always expand deepest node.
- Follow one "branch" of search tree.



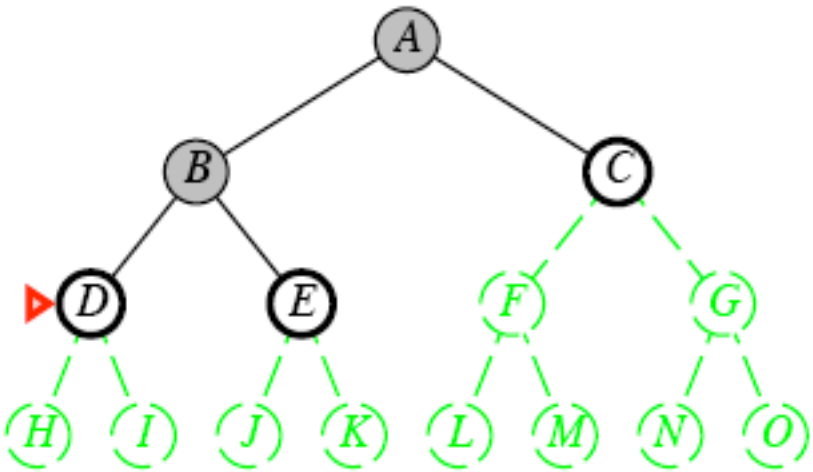
Depth First Search

- Start by expanding initial state.
- Pick one of nodes resulting from **first** step, and expand it.
- Pick one of nodes resulting from **second** step, and expand it, and so on.
- Always expand deepest node.
- Follow one "branch" of search tree.



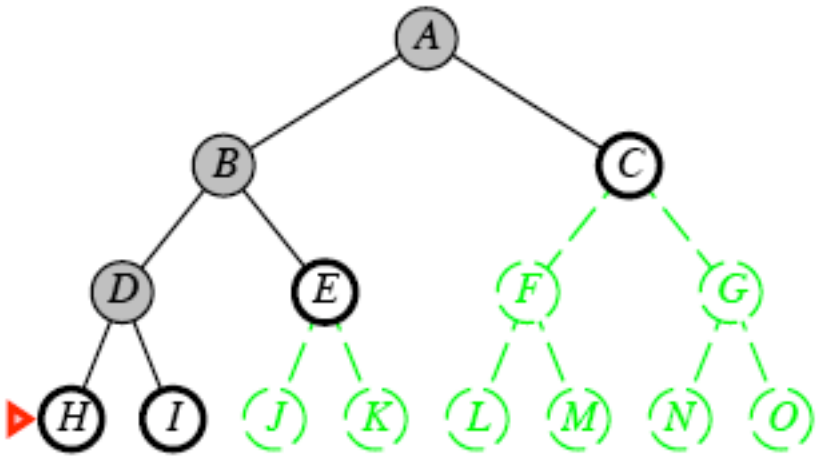
Depth First Search

- Start by expanding initial state.
- Pick one of nodes resulting from **first** step, and expand it.
- Pick one of nodes resulting from **second** step, and expand it, and so on.
- Always expand deepest node.
- Follow one "branch" of search tree.



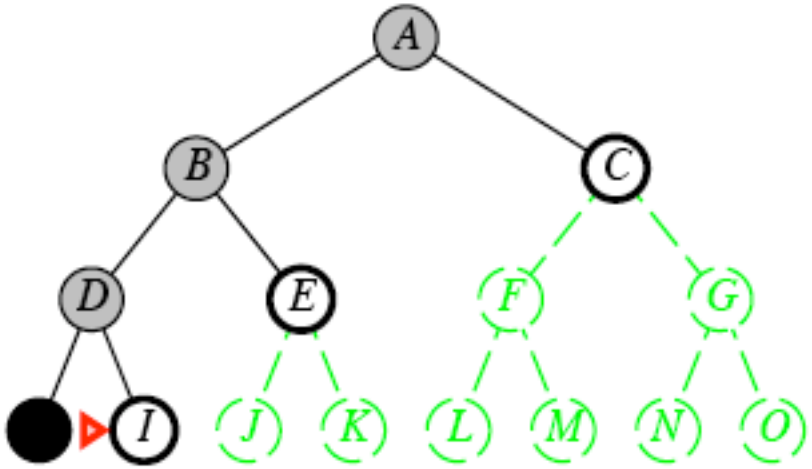
Depth First Search

- Start by expanding initial state.
- Pick one of nodes resulting from **first** step, and expand it.
- Pick one of nodes resulting from **second** step, and expand it, and so on.
- Always expand deepest node.
- Follow one "branch" of search tree.



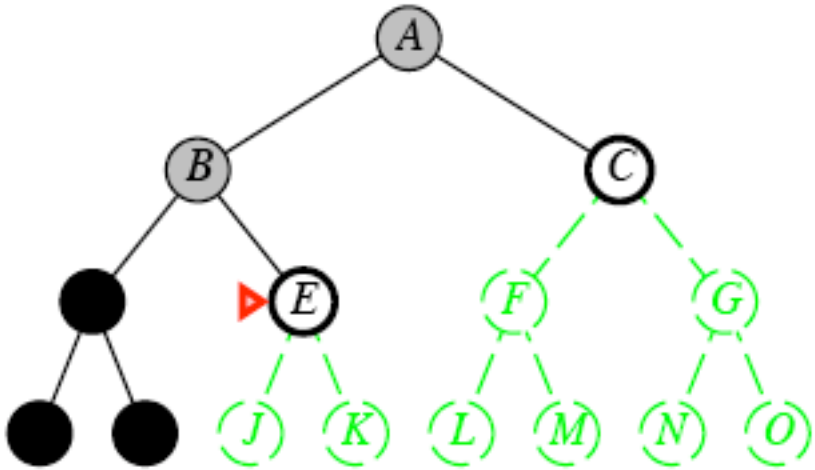
Depth First Search

- Start by expanding initial state.
- Pick one of nodes resulting from **first** step, and expand it.
- Pick one of nodes resulting from **second** step, and expand it, and so on.
- Always expand deepest node.
- Follow one "branch" of search tree.



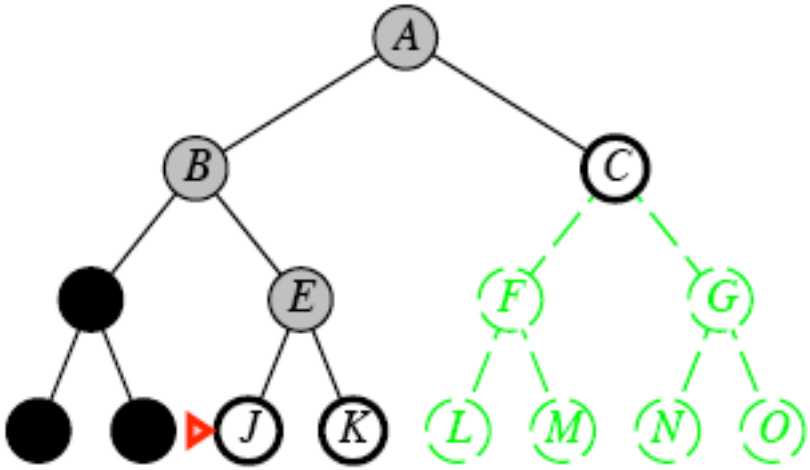
Depth First Search

- Start by expanding initial state.
- Pick one of nodes resulting from **first** step, and expand it.
- Pick one of nodes resulting from **second** step, and expand it, and so on.
- Always expand deepest node.
- Follow one "branch" of search tree.



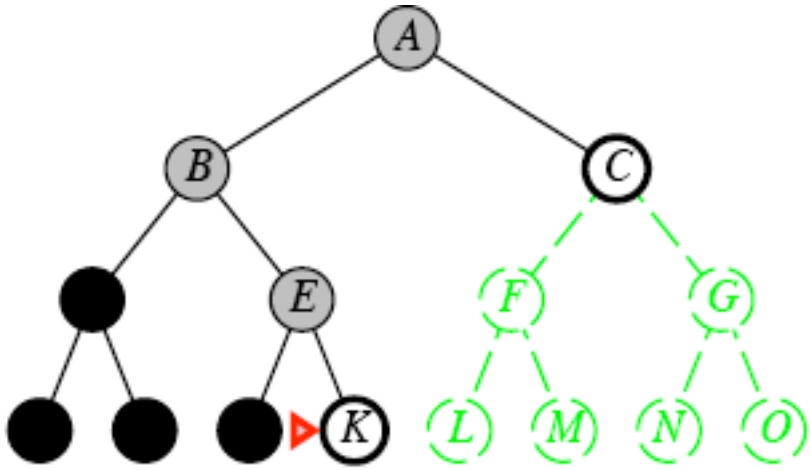
Depth First Search

- Start by expanding initial state.
- Pick one of nodes resulting from **first** step, and expand it.
- Pick one of nodes resulting from **second** step, and expand it, and so on.
- Always expand deepest node.
- Follow one "branch" of search tree.



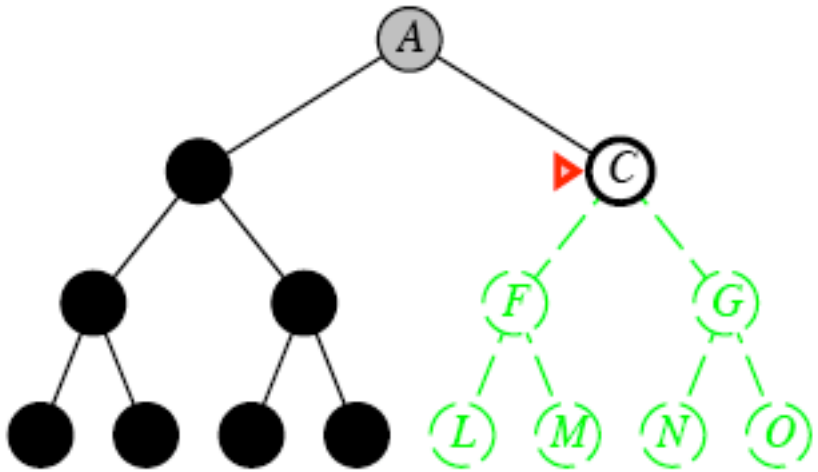
Depth First Search

- Start by expanding initial state.
- Pick one of nodes resulting from **first** step, and expand it.
- Pick one of nodes resulting from **second** step, and expand it, and so on.
- Always expand deepest node.
- Follow one "branch" of search tree.



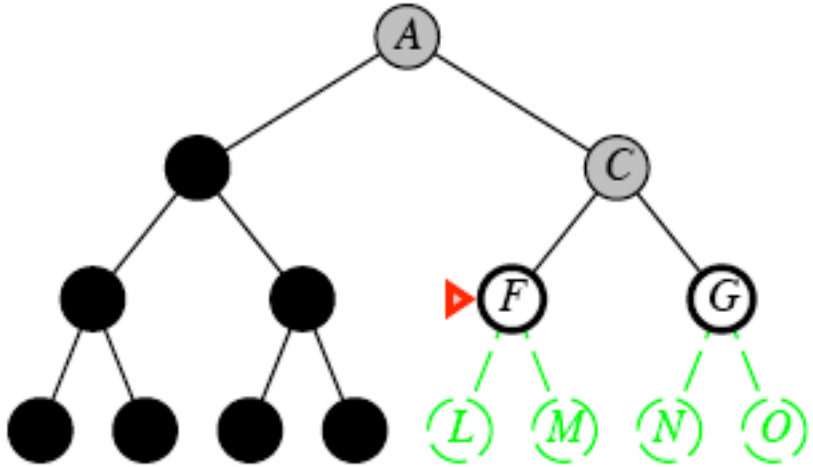
Depth First Search

- Start by expanding initial state.
- Pick one of nodes resulting from **first** step, and expand it.
- Pick one of nodes resulting from **second** step, and expand it, and so on.
- Always expand deepest node.
- Follow one "branch" of search tree.



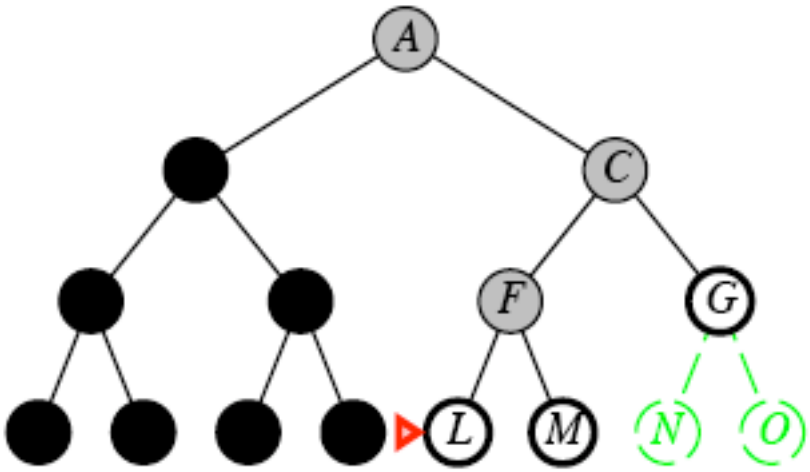
Depth First Search

- Start by expanding initial state.
- Pick one of nodes resulting from **first** step, and expand it.
- Pick one of nodes resulting from **second** step, and expand it, and so on.
- Always expand deepest node.
- Follow one "branch" of search tree.



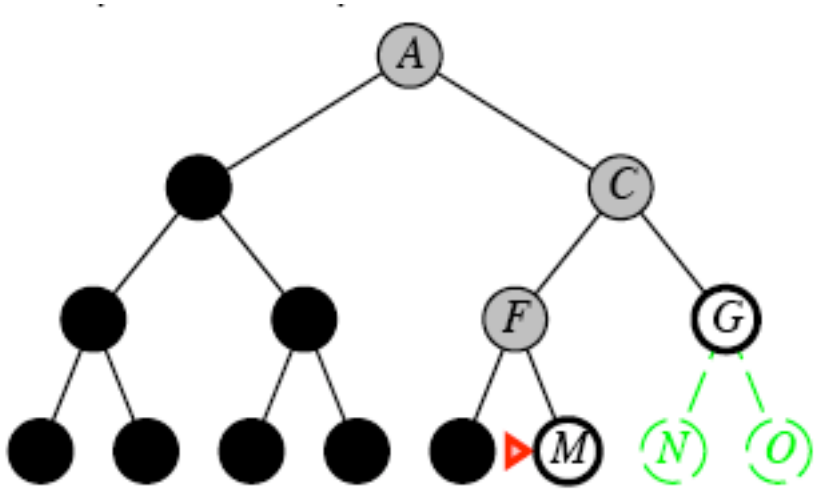
Depth First Search

- Start by expanding initial state.
- Pick one of nodes resulting from **first** step, and expand it.
- Pick one of nodes resulting from **second** step, and expand it, and so on.
- Always expand deepest node.
- Follow one "branch" of search tree.



Depth First Search

- Start by expanding initial state.
- Pick one of nodes resulting from **first** step, and expand it.
- Pick one of nodes resulting from **second** step, and expand it, and so on.
- Always expand deepest node.
- Follow one "branch" of search tree.



Depth First Search

```
/* 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;
}
```

(Agenda = OPEN List = the "Fringe" = Frontier)

Depth-First Search

Disadvantages:

• Depth first search is *not guaranteed* to find a solution if one exists.

• Solution found is *not guaranteed* to be the best.

Advantages:

• 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. (Linear space requirement)

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.

- An Optimal Solution is called **admissible.**

Improvements on Depth-First and Breadth-First

- Breadth-first search is *complete* but **expensive**.
- Depth-first search is **cheap** but *incomplete*
- Can't we do better than this?
- Basic search (depth 1st, breadth 1st) can be improved:
- Improvements:
  - depth limited search;
  - iterative deepening.
- But we will see that even with such improvements, search is hopelessly unrealistic for real problems.

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 expanded (with no solution on it), then it won't terminate.
- Introduce a *depth limit* on branches to be expanded.
- Don't expand a branch below this depth.

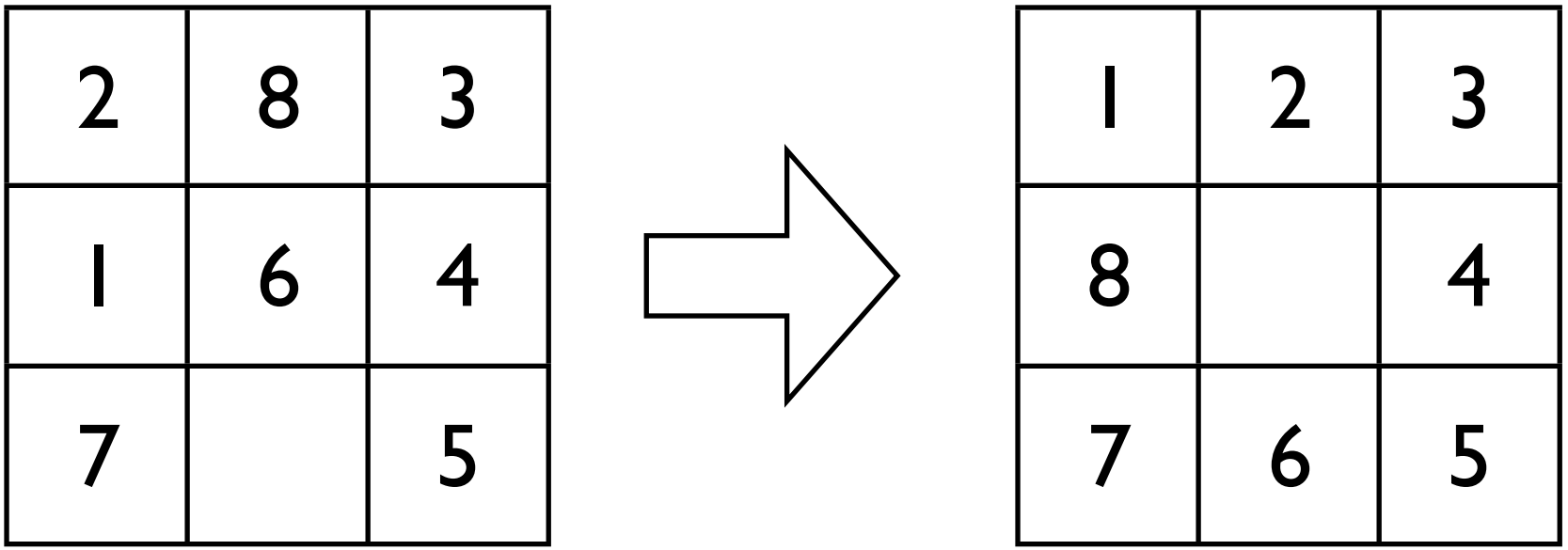
Depth Limited Algorithm

```
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; // Expanding the node
   if goal state in new nodes then {
      return solution;
   }
   if depth(node) < depth limit then {
      add new nodes to front of agenda;
   }
}
```

(Agenda = OPEN List = the "Fringe" = Frontier)

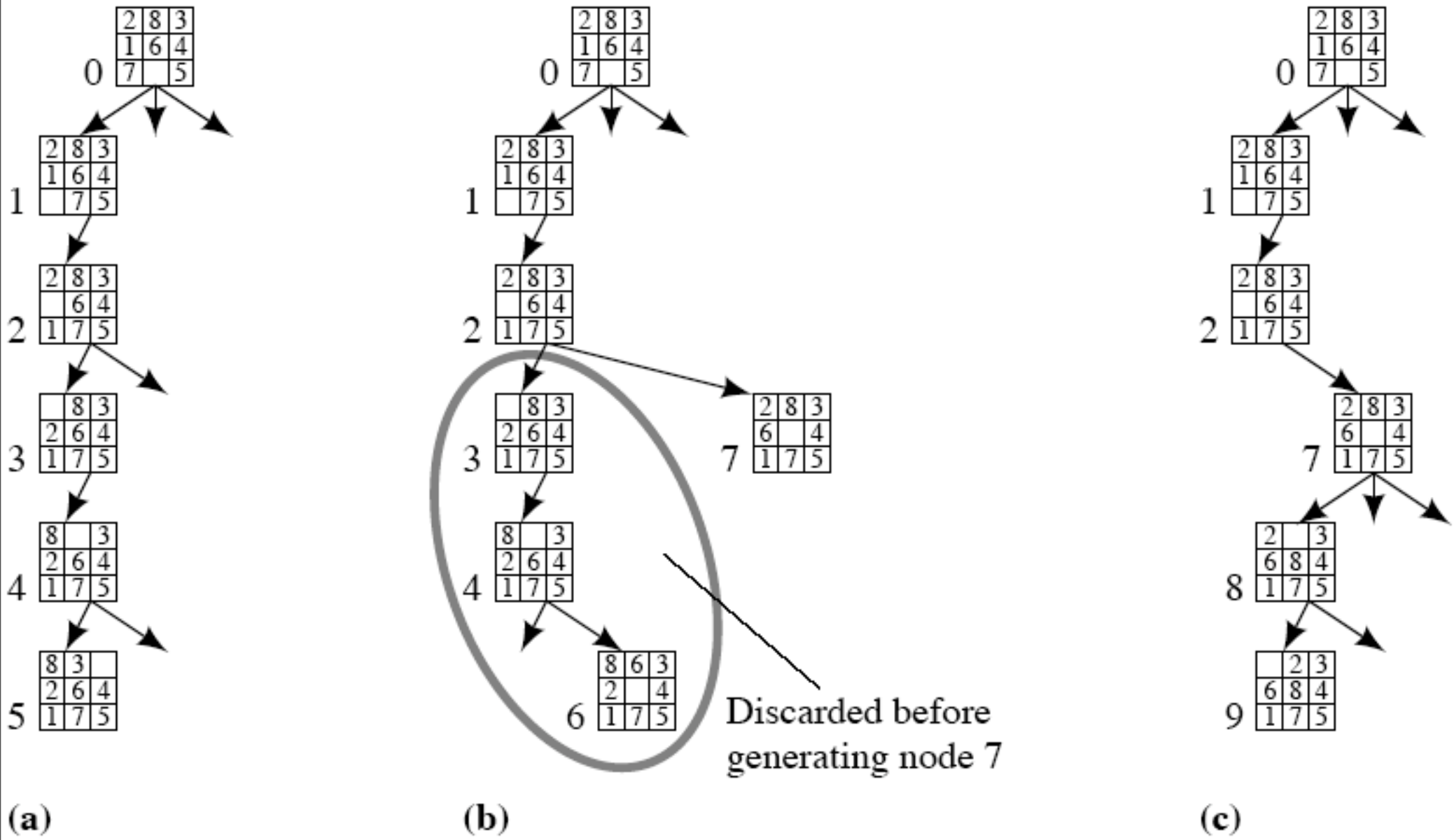
8-Puzzle Example

- For the 8-puzzle set up:



- Search tree (with State Symbolizing the Nodes)

8-Puzzle Search (Depth Limited at Level 5)



Chronological Backtracking

- Hit the depth bound of level 5, we don't add any more nodes to the agenda.
- Then we pick the next node off the agenda. (Node 7 in this case)
- This has the effect of moving the search back to the last node above depth limit that that is "partially 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

- Depth-limited search is not complete:
  - If we choose a max-depth such that the shortest-path solution is longer.
  - We choose a max-depth deeper than the shortest goal node, and then suffer the consequences of depth-first search and hit a *less optimal goal* first.
- Complete Solution: *Iterative deepening*
- Basic idea repeat depth-limited-search for all depths until solution:

– depth-limited-search (depth = 1) ; if solution found, return it;

– otherwise depth-limited-search (depth = 2); if solution found, return it;

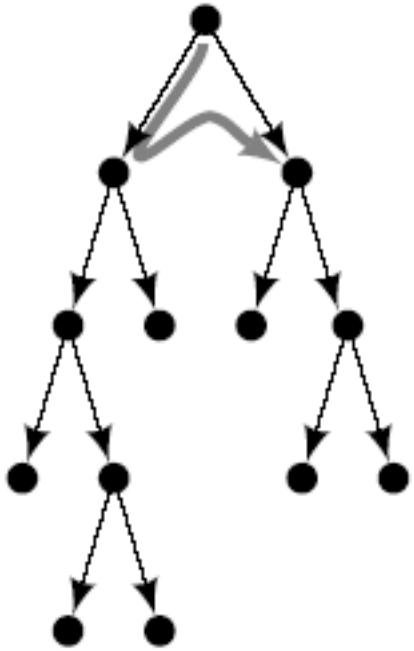
– otherwise depth-limited-search (depth = n); if solution found, return it;

– otherwise, . . .

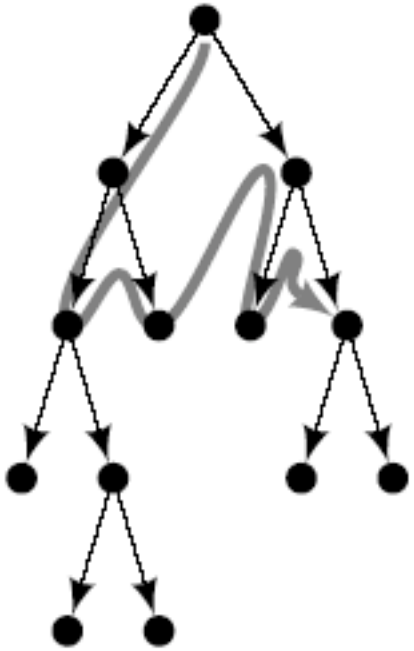
Iterative Deepening Search (Algorithm)

```
depth limit = 1;
repeat {
   // depth_limited_search is a sub-routine
   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 */
```

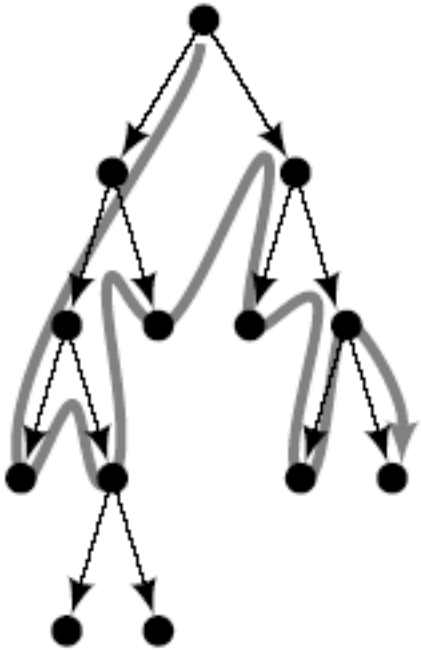
Depth-Limited Search



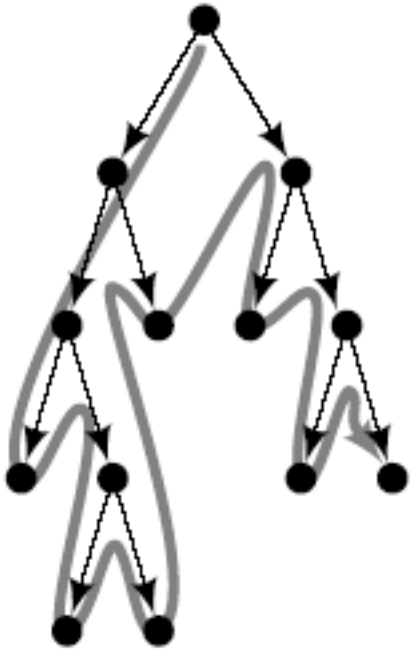
Depth bound = 1



Depth bound = 2



Depth bound = 3



Depth bound = 4

Efficiency of Iterative Deepening

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*.

Number of Nodes Generated for breadth-first search to level *d*:

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

Iterative Deepening

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

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

(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.

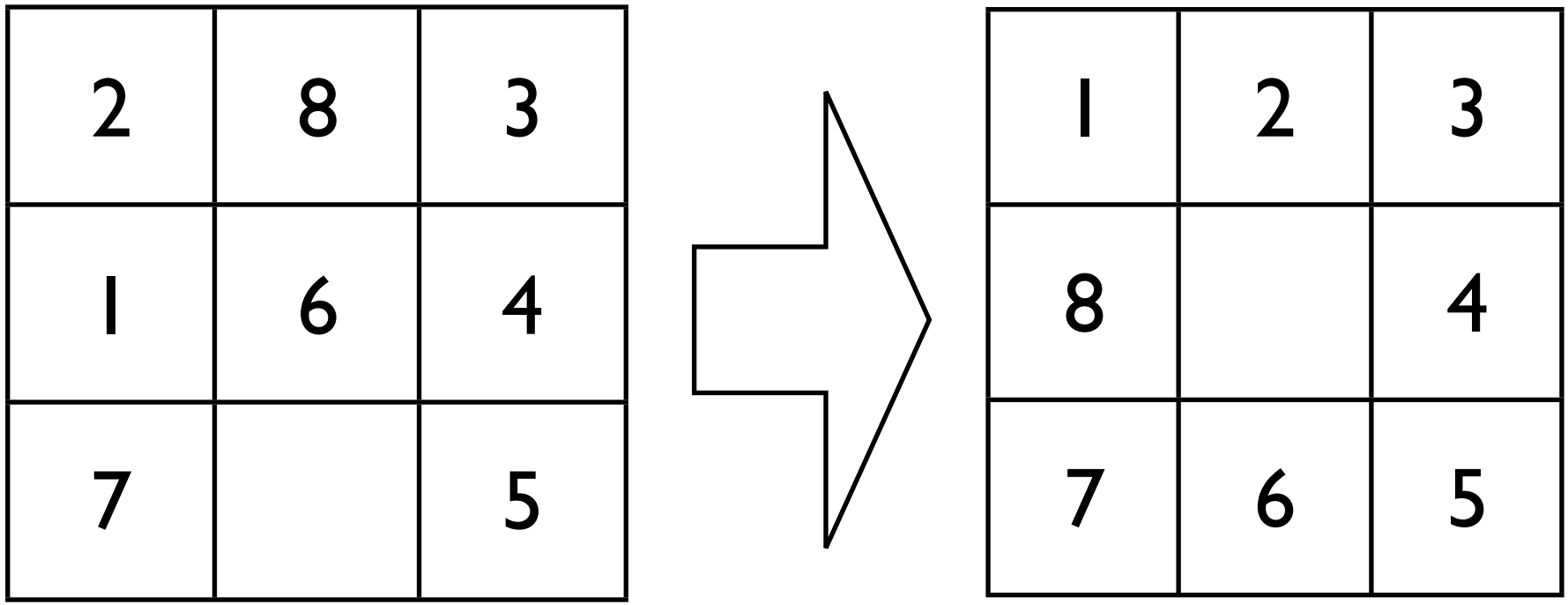
Breadth first search would require examining 111, 111 nodes, with memory requirement of 100, 000 nodes.

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

Takes *11%* longer in this case.

8-puzzle Example (with Iterative Deepening)

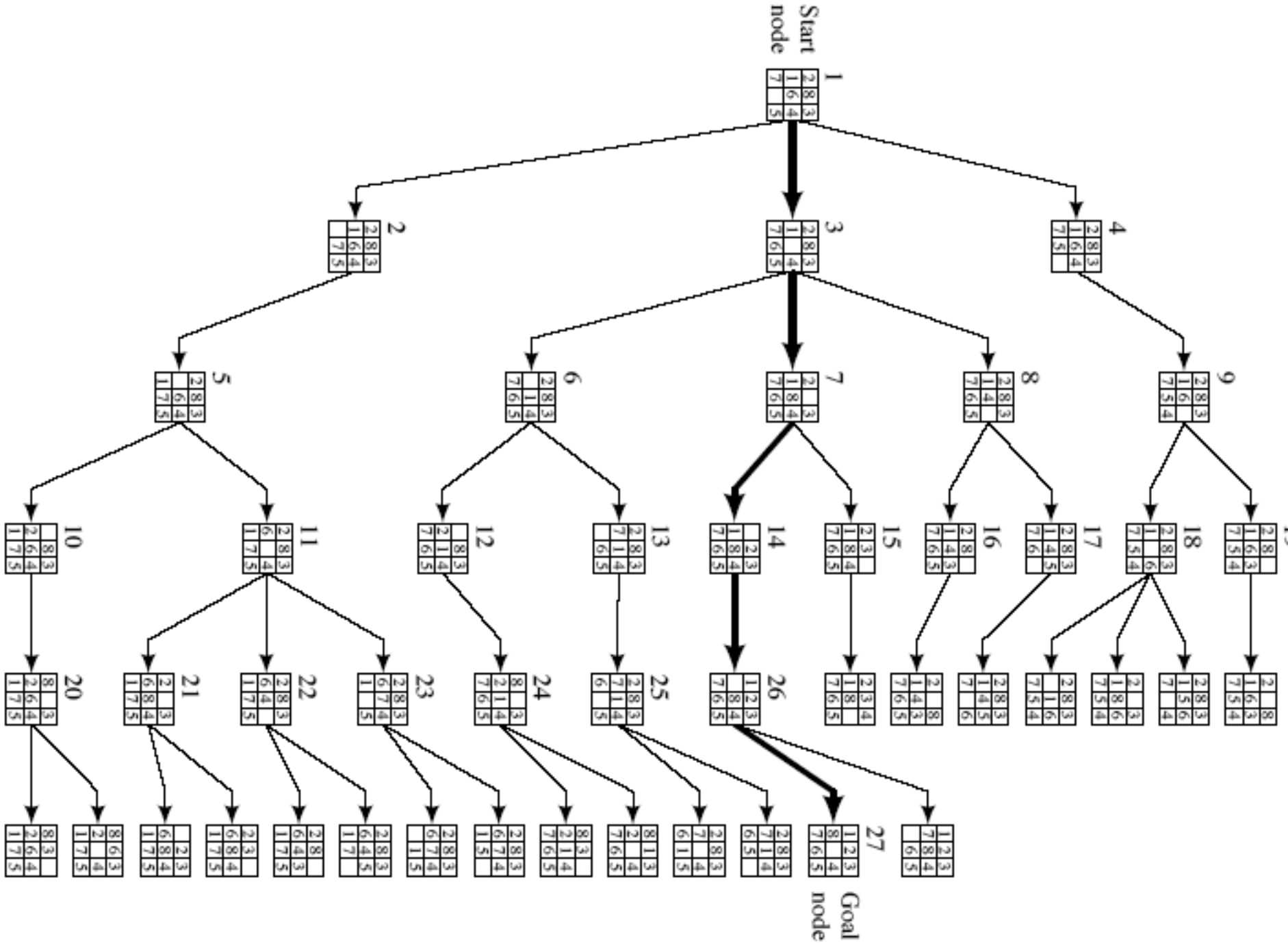
For the 8-puzzle setup as:



What would iterative deepening search look like?

Well, it would explore the search space, (draw it on the board!)

Expanded Search Space



- States would be expanded in the order:

  1. 1

  2. 1, 2, 3, 4

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

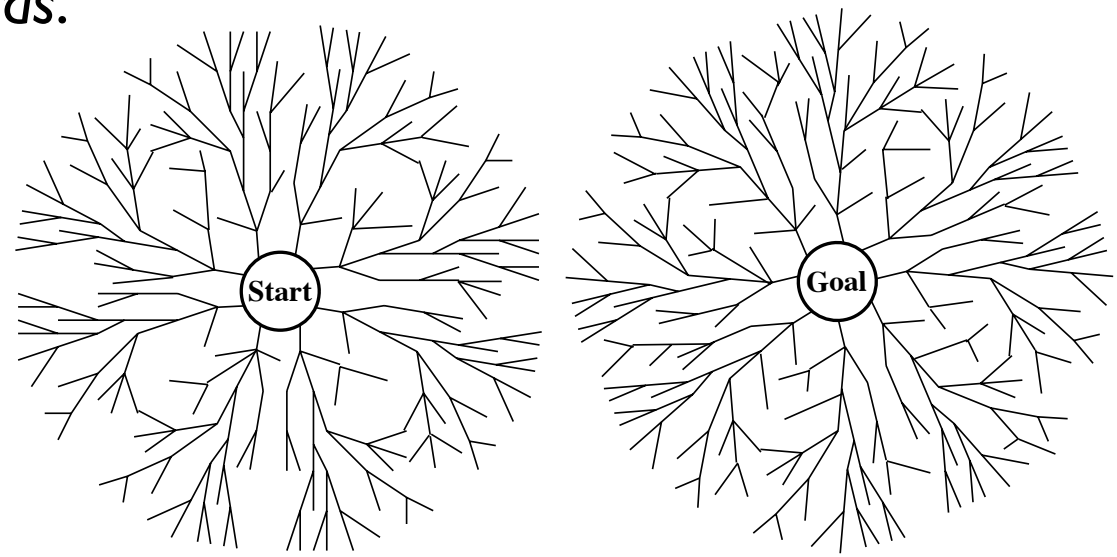
  4. 1, 2, 5, 10, 11, 3, 6, 13, 13, 7, 14, 15, 8, 16, 17, 4, 9, 18, 19.

  5. . . .

- Note that these are the states *visited*, not the nodes on the agenda (remember depth-first search has at most *bd* 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*, which works well in navigation problems (i.e. driving from Boston to New York).

More difficult to use in problems with implicit goals (i.e. Checkmate(X))

• Rather than doing one search of b^d, we do two $b^{d/2}$ searches.

• Much *more efficient*.

Bi-Directional Search

- Example:
  - Suppose $b = 10$, $d = 6$.
  - Breadth first search will examine ____ nodes.
  - Bi-directional search will examine ____ nodes.
- Can combine different search strategies in different directions.
- For large d, is still *impractical*!

Bi-Directional Search

- Example:
  - Suppose b = 10, d = 6.
  - Breadth first search will examine b^d = 1,000,000 nodes.
  - Bi-directional search will examine $b^{d/2} + b^{d/2}$ =1,000 + 1,000 nodes.
- Can combine different search strategies in different directions.
- For large d, is still *impractical*!

Summary

• This lecture has looked at some more efficient techniques than *breadth first* and *depth first search*.

  – depth-limited search;

  – iterative-deepening search; and

  – bidirectional search.

• These all improve on *depth-first* and *breadth-first search*.

• However, all **fail** for big enough problems (too large state space).

• Next lecture, we will look at approaches that cut down the size of the state-space that is searched.