ADAM PURTEE
UNIVERSITY OF ROCHESTER
CSC 442: ARTIFICIAL INTELLIGENCE

SEARCH ALGORITHMS

LAST TIME:

- Intelligent Agents
- State-space Search Problems

PROBLEM SOLVING AGENTS

- A problem solving agent develops an **action sequence** in order to accomplish a goal. The sequence is obtained by **searching** a possible **state-space** together with a set of possible **actions** and an associated **transition model**.
- Problem solving agents are general purpose and powerful. They can be applied to almost any AI problem, as long as you're flexible about your **level of abstraction**.

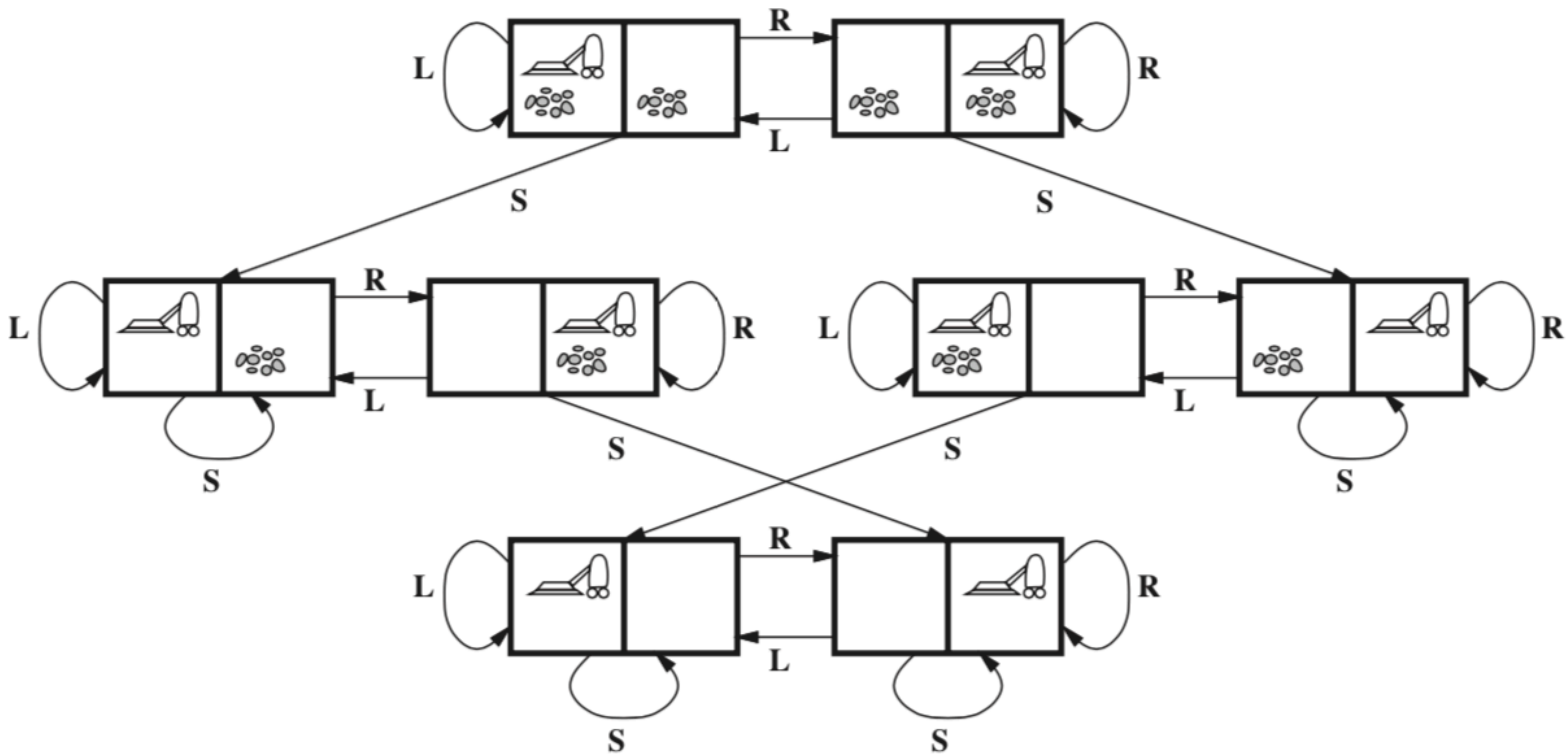
PROBLEM SOLVING AGENTS

- How did we formally define problems?
- How will we search for solutions to problems?

STATE-SPACE SEARCH PROBLEMS

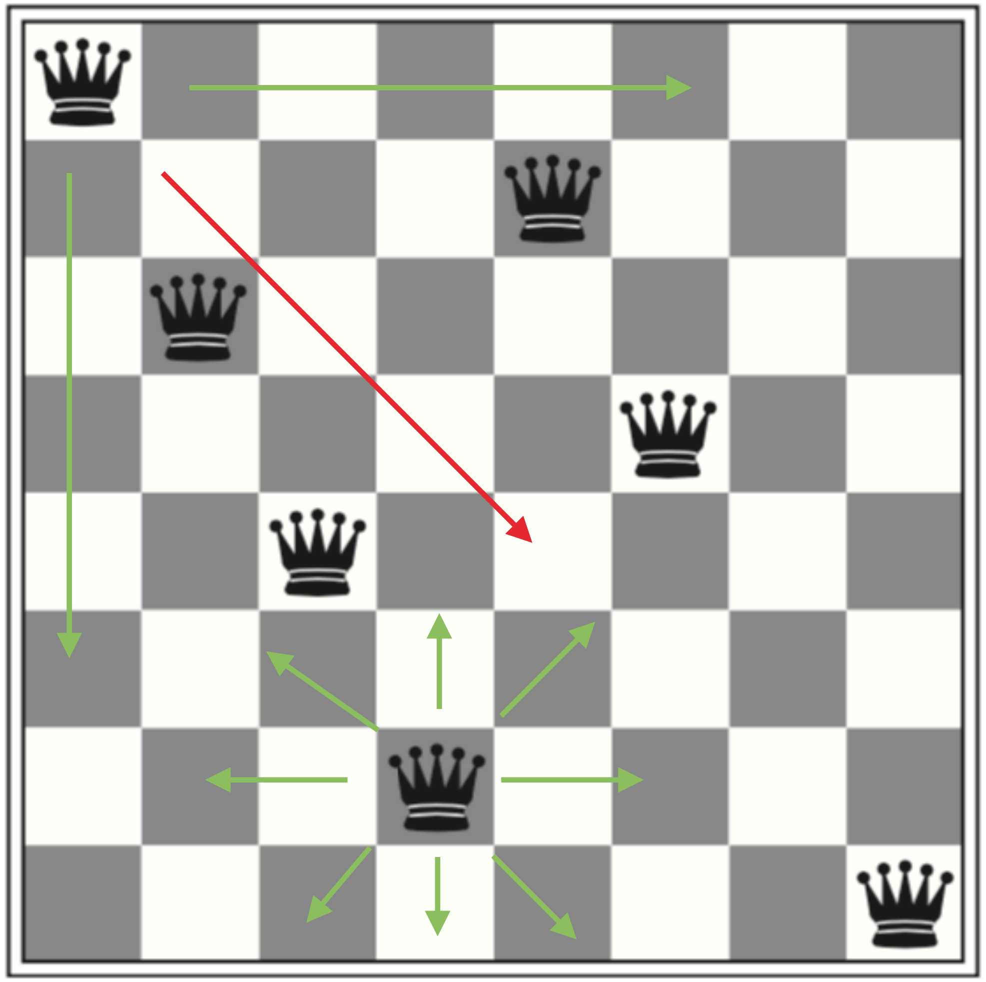
- A **problem** can be defined formally by five components:
  - an **initial state**
  - a set of **possible actions**
  - a **transition model**
  - a **goal test** function
  - a **path cost** function

VACUUM WORLD



THE N-QUEENS PUZZLE

- Can we place queens in every row and column without attacks?

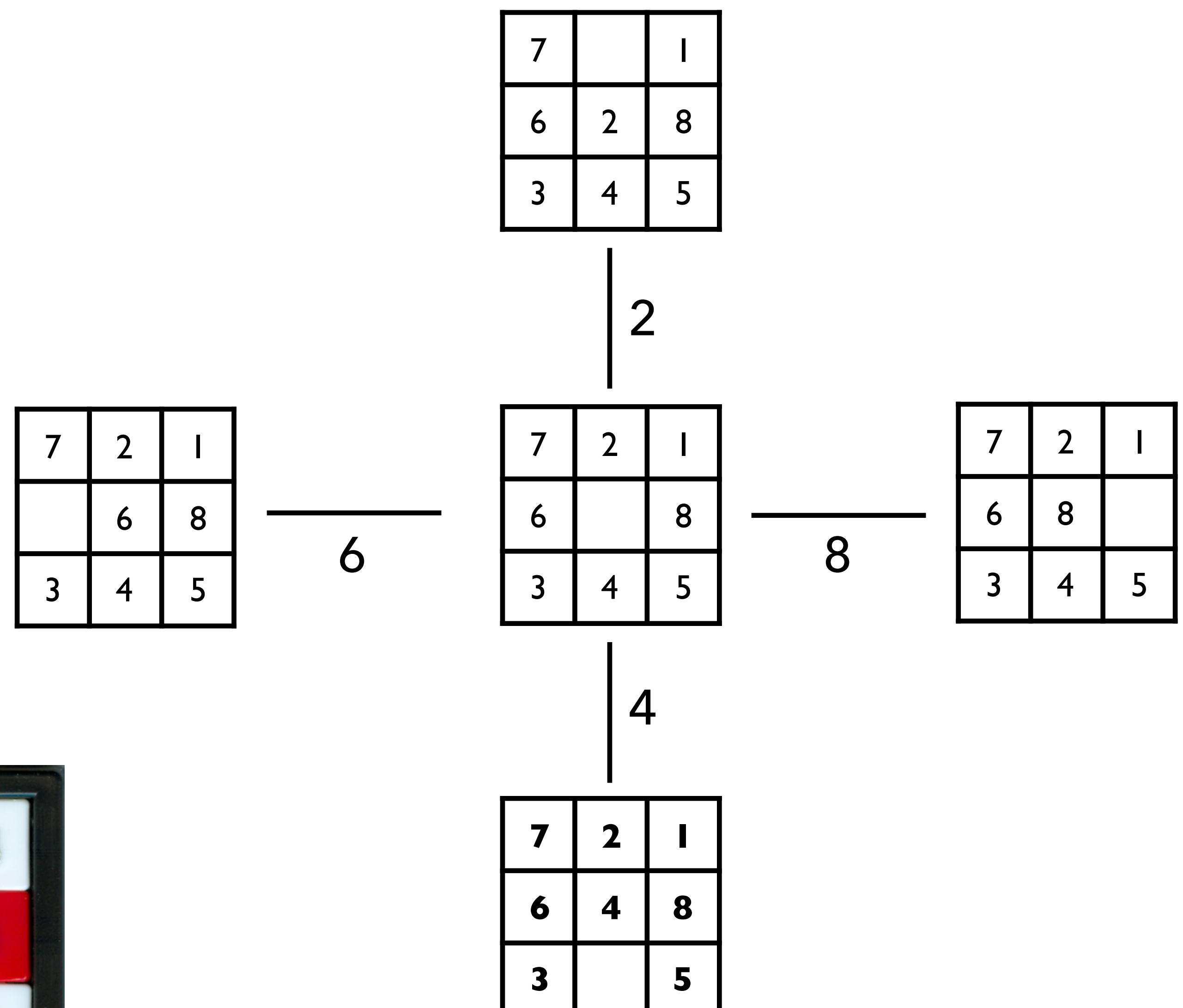


KNUTH'S CONJECTURE

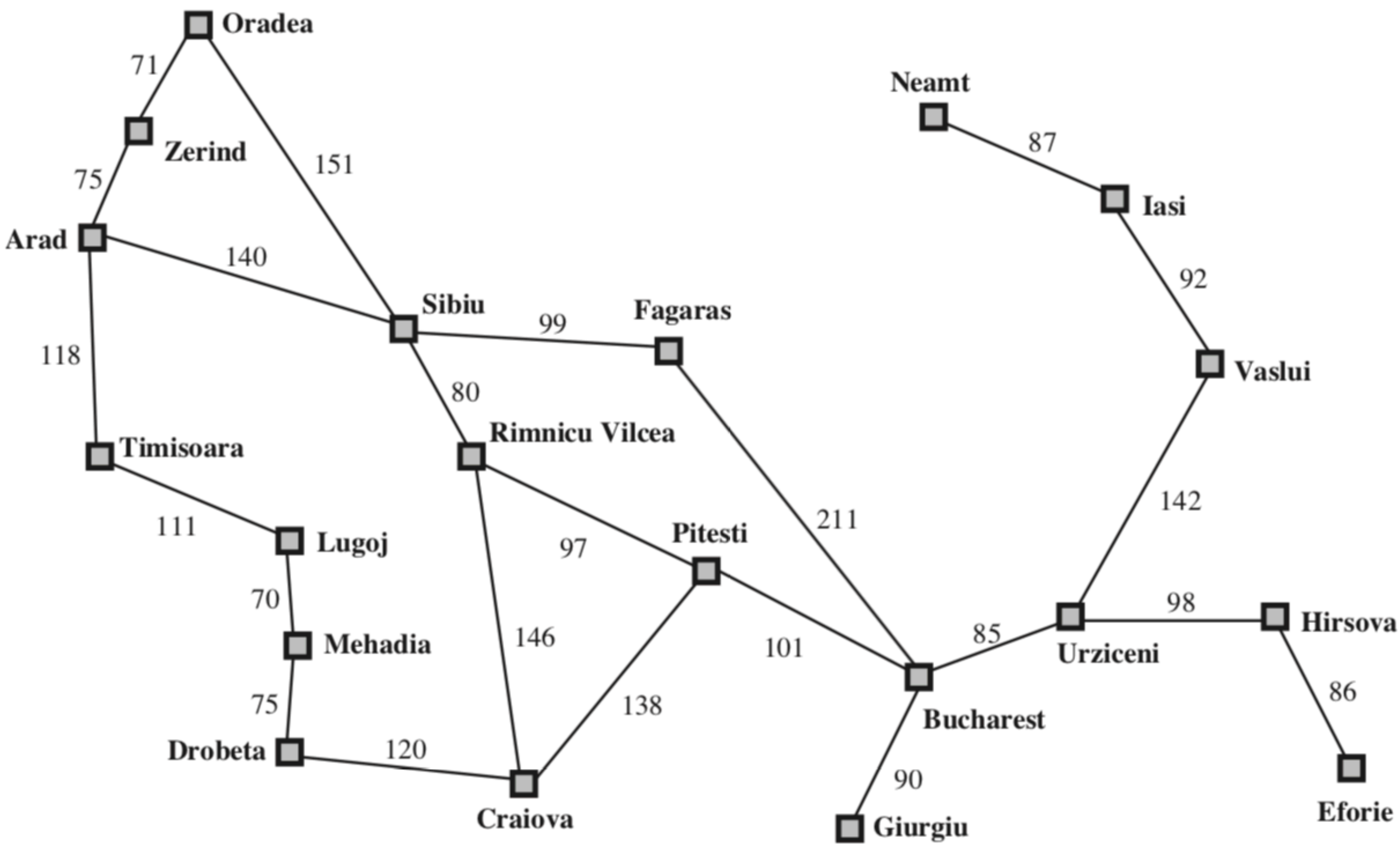
- Starting with the number 4, a sequence of factorial, square root, and floor operations will reach any desired positive integer.

$$\left\lfloor \sqrt{\sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}}} \right\rfloor = 5$$

SLIDING BLOCK PUZZLES



NAVIGATION — ARAD TO BUCHAREST?



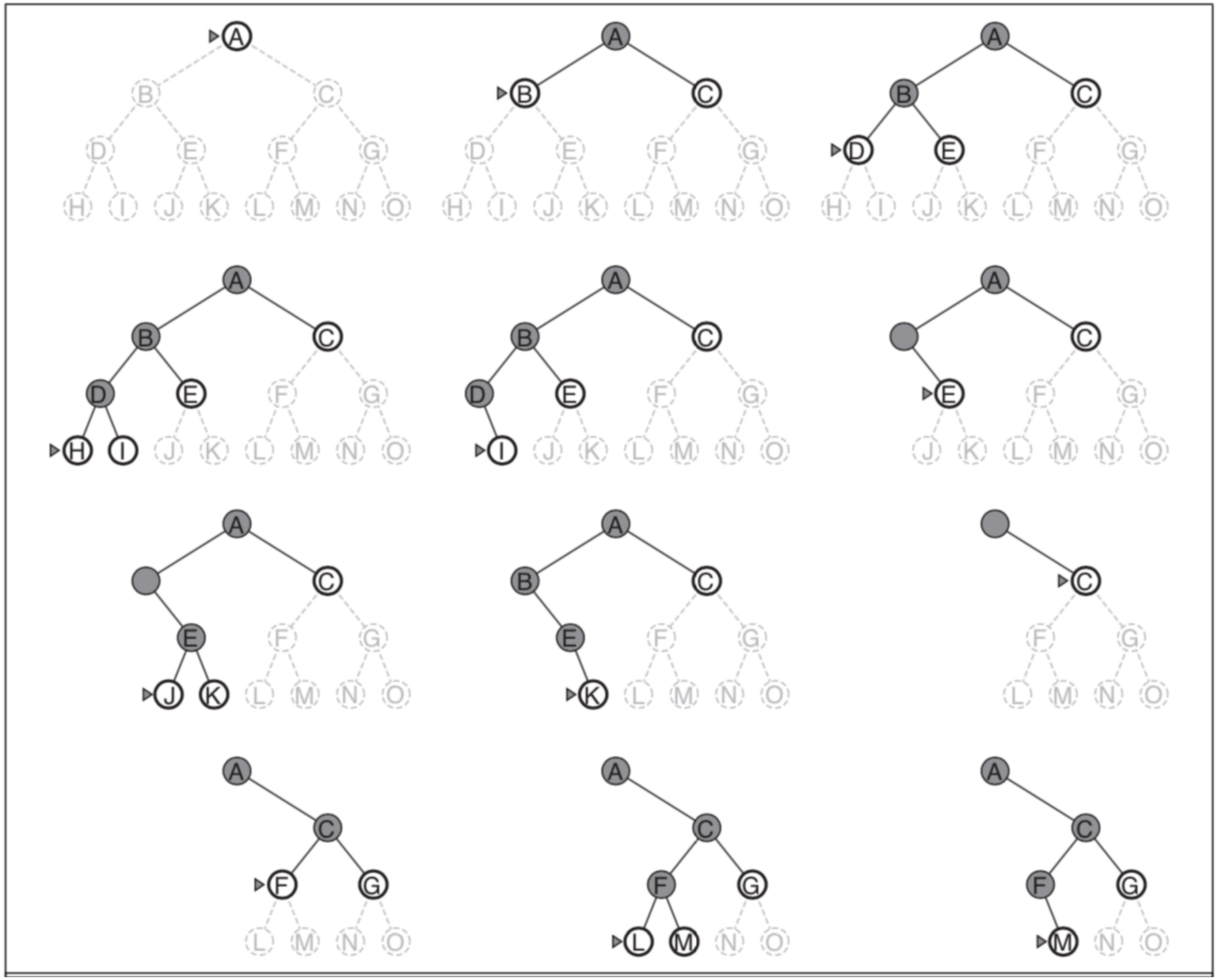
SOLVING STATE-SPACE SEARCH PROBLEMS

- Search problems are solved with **search algorithms**.
- The simplest of these is **depth-first tree search**.

DEPTH-FIRST TREE SEARCH

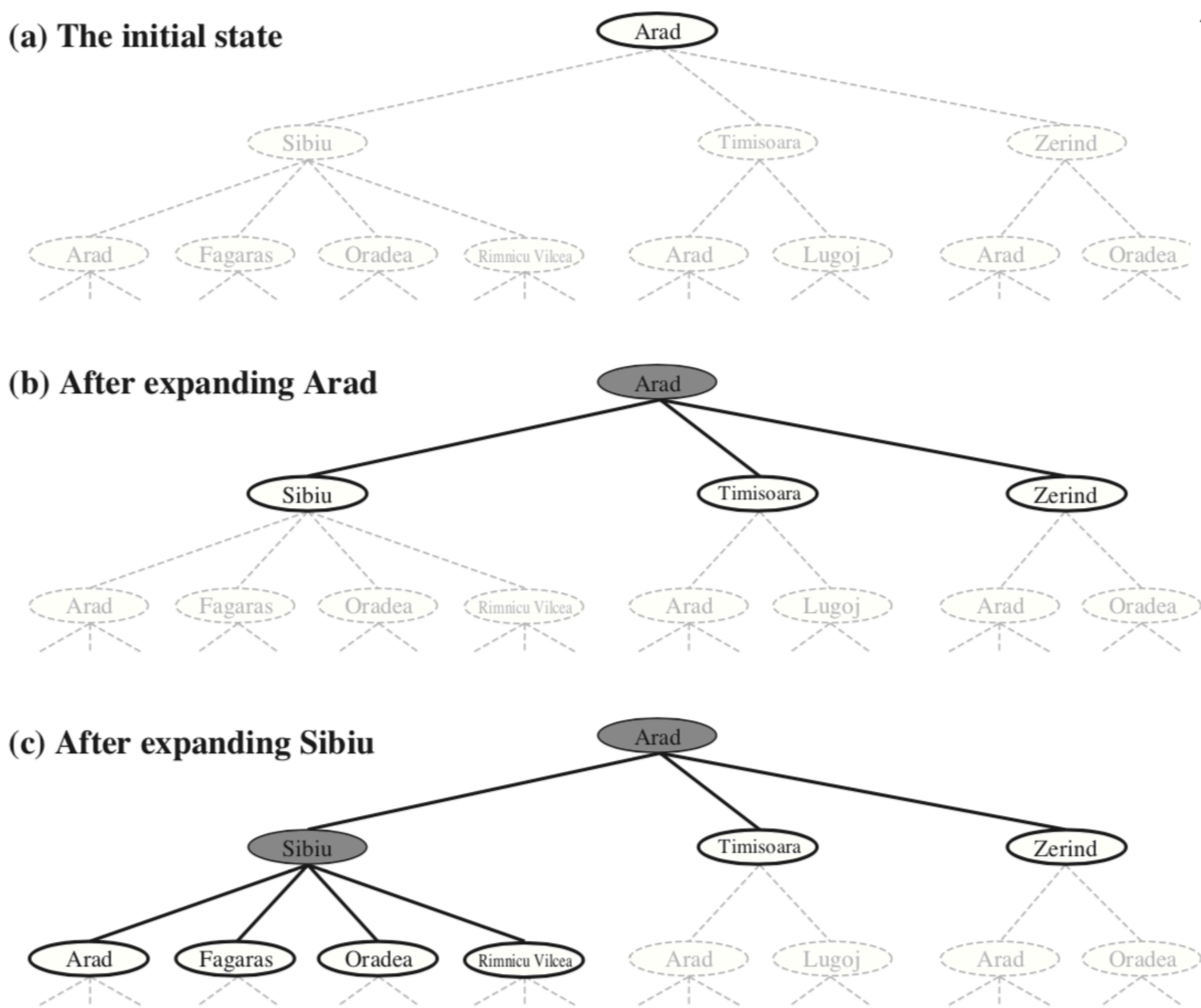
```
function Tree-Search (problem, state) returns a solution
   if Goal(state), return Empty
   for each possible action a:
       let s' = Transition(state, a)
       let seq = Tree-Search (problem, s')
       if seq satisfies Goal, return a + seq
   return failure
```

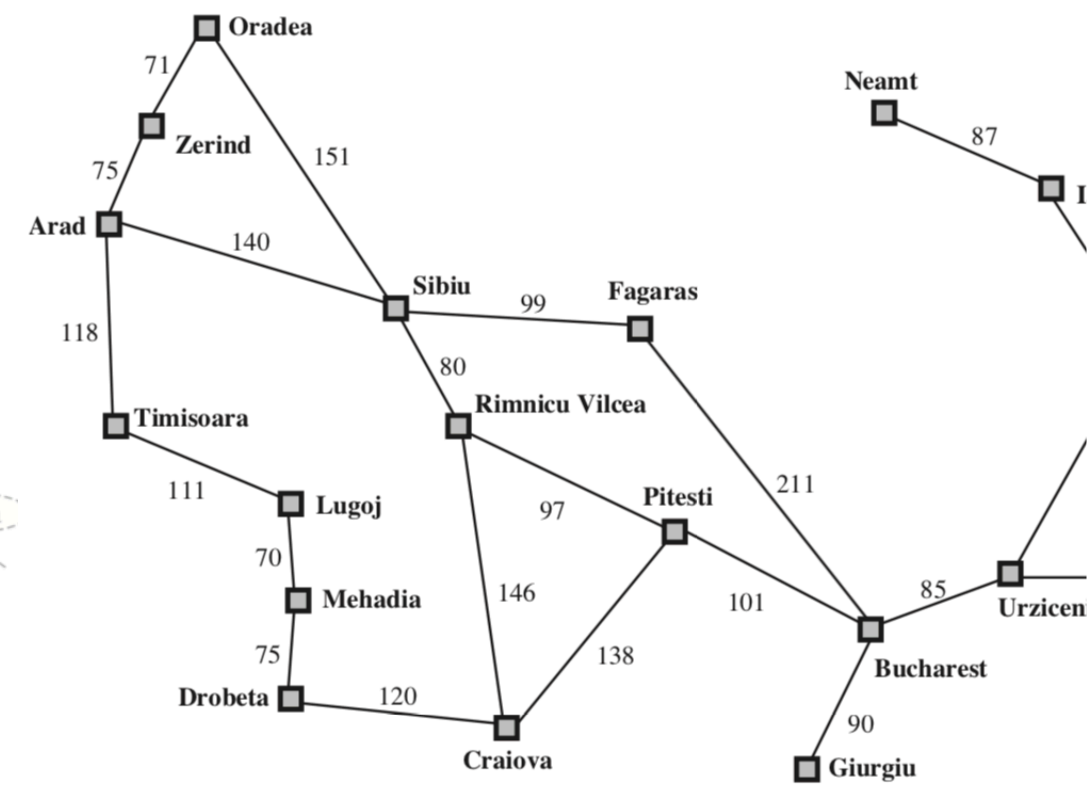
DEPTH FIRST SEARCH



AIMA Figure 3.16, p86

DEPTH FIRST SEARCH





ADAPTING DEPTH-FIRST SEARCH TO GRAPHS

- The biggest challenge with depth-first search is its tendency to get “lost” by wandering off into infinity… even for finite problems, such as navigation.
- A solution is to **remember visited states** by incorporating a hash table (i.e., dictionary).

DEPTH FIRST TREE SEARCH

```
function Tree-Search (problem, state) returns a solution
  if Goal(state), return Empty
  for each possible action a:
     let s' = Transition(state, a)
     let seq = Tree-Search (problem, s')
     if seq satisfies Goal, return a + seq
  return failure
```

DEPTH FIRST TREE SEARCH

```
function Tree-Search (problem, state) returns a solution

  if Goal(state), return Empty
  for each possible action a:
      let s' = Transition(state, a)


      let seq = Tree-Search (problem, s')
      if seq satisfies Goal, return a + seq
  return failure
```

DEPTH FIRST GRAPH SEARCH

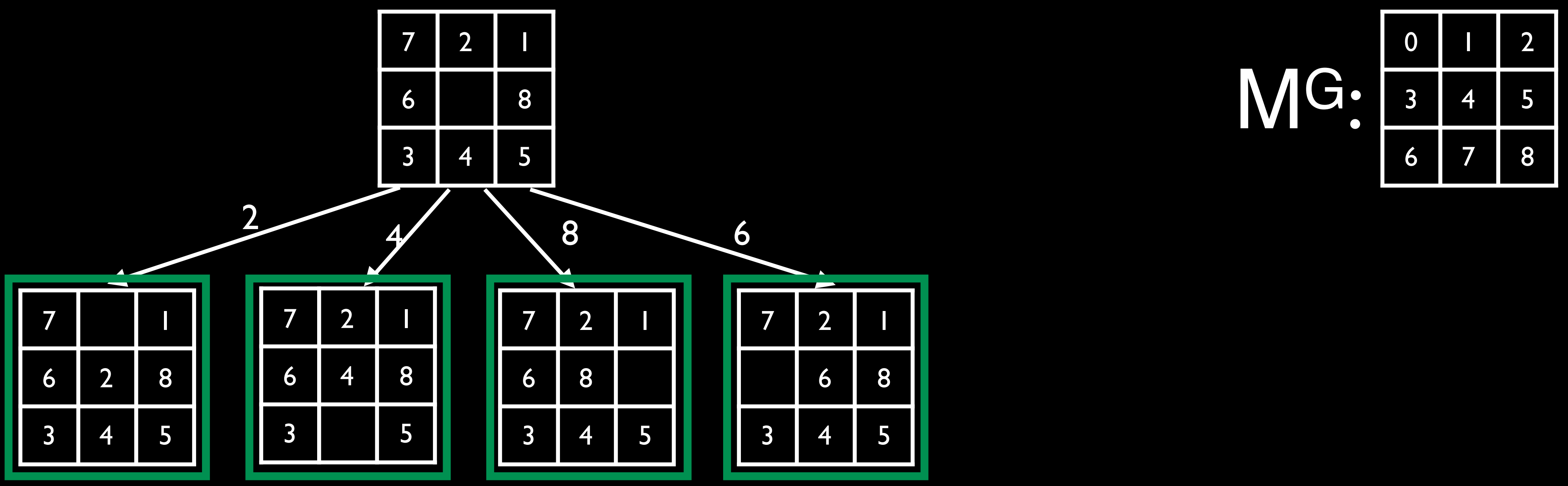
```
function Graph-Search (problem, state) returns a solution
  initialize explored set to empty.
  if Goal(state), return Empty
  for each possible action a:
      let s' = Transition(state, a)
      if s' not in explored:
          add s' to explored
          let seq = Tree-Search (problem, s')
          if seq satisfies Goal, return a + seq
  return failure
```

7	2	1
6		8
3	4	5

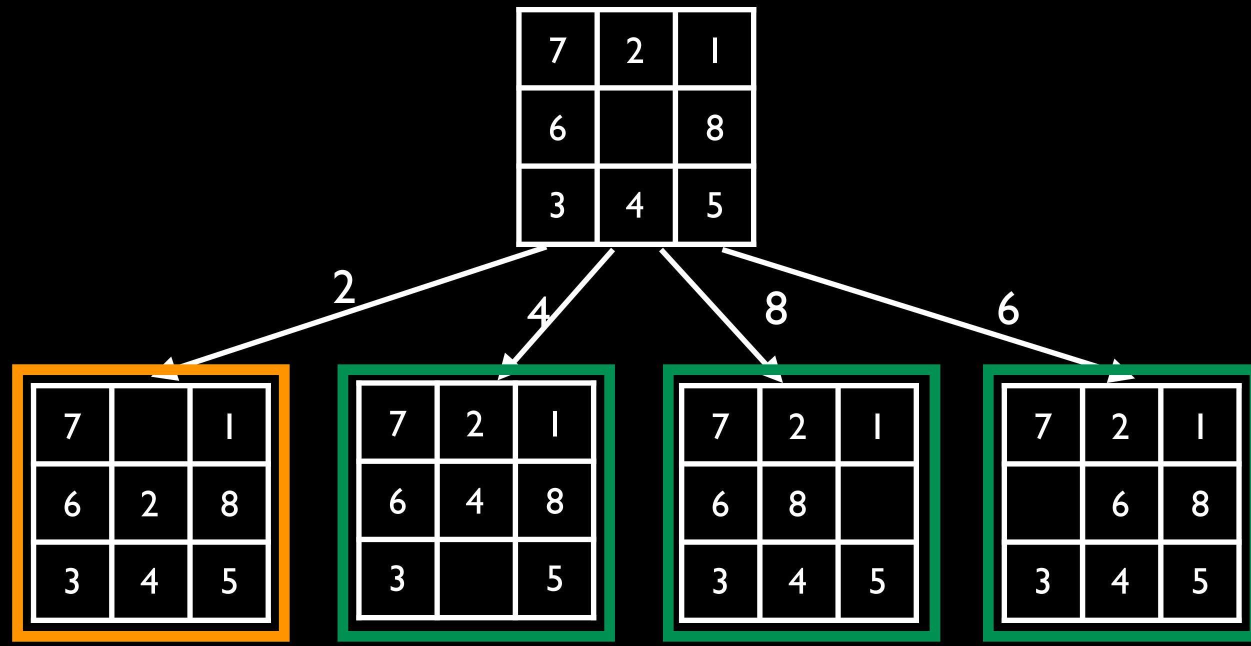
M^G:

1	2	3
4	5	6
7	8	

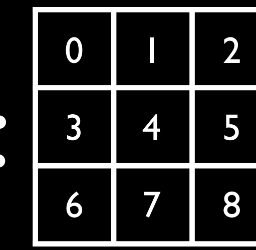
DEPTH FIRST (GRAPH) SEARCH



DEPTH FIRST (GRAPH) SEARCH



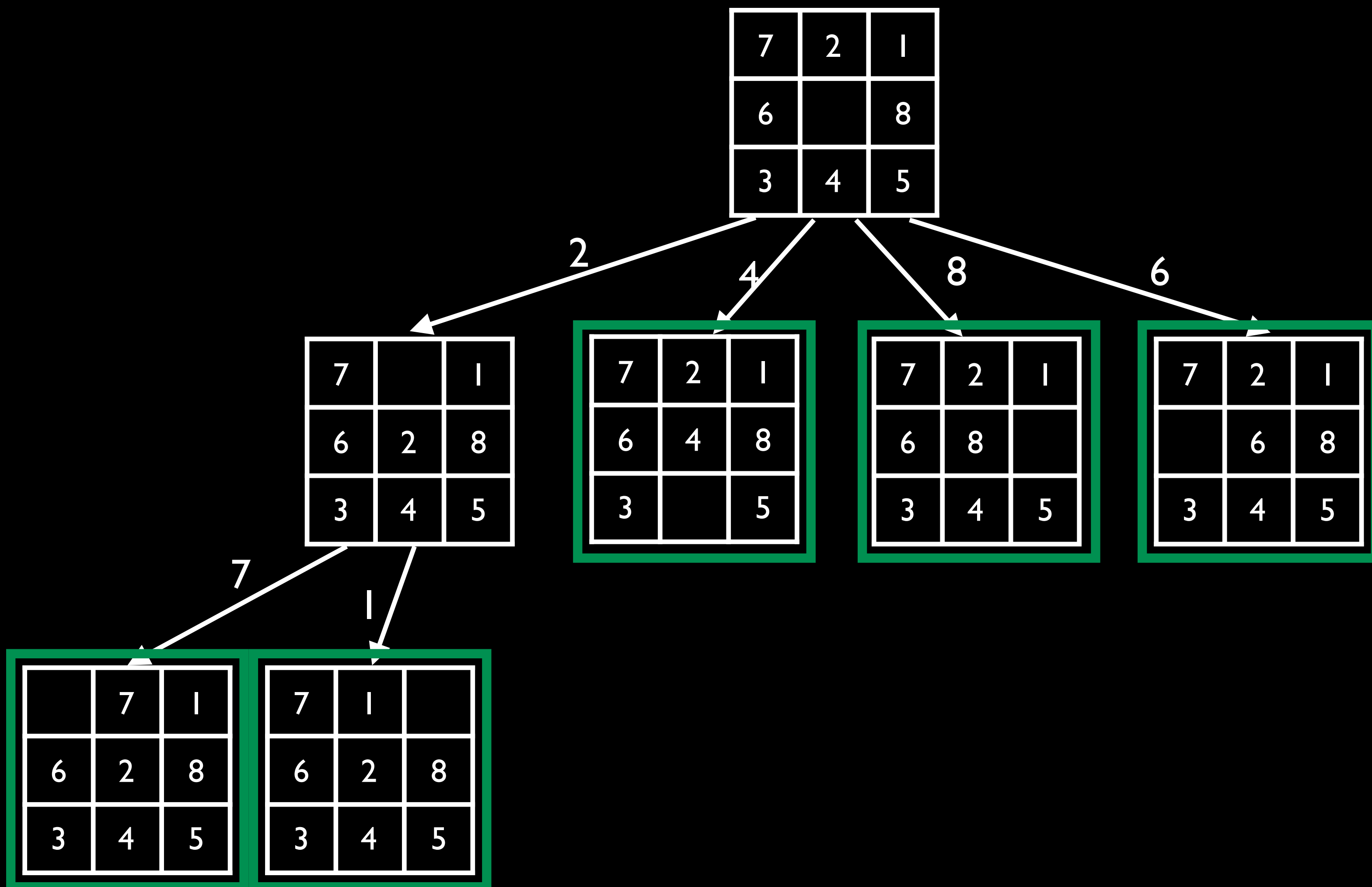
M^G:



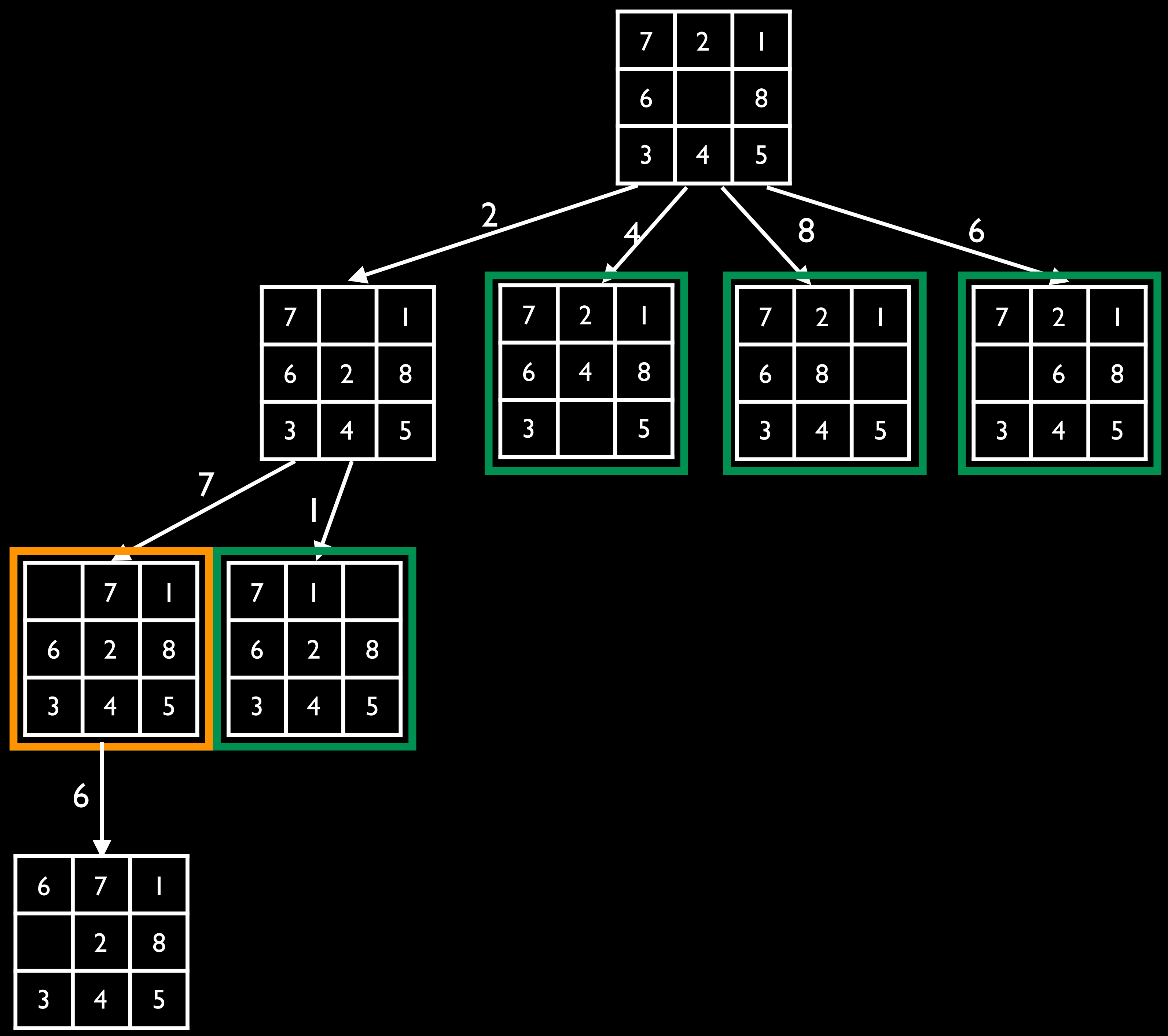
DEPTH FIRST (GRAPH) SEARCH

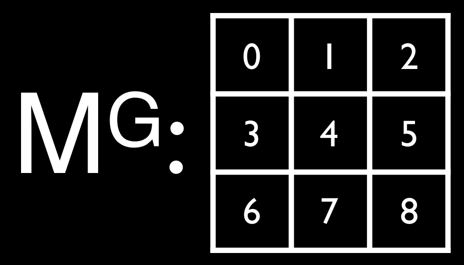
M^G:

0	1	2
3	4	5
6	7	8



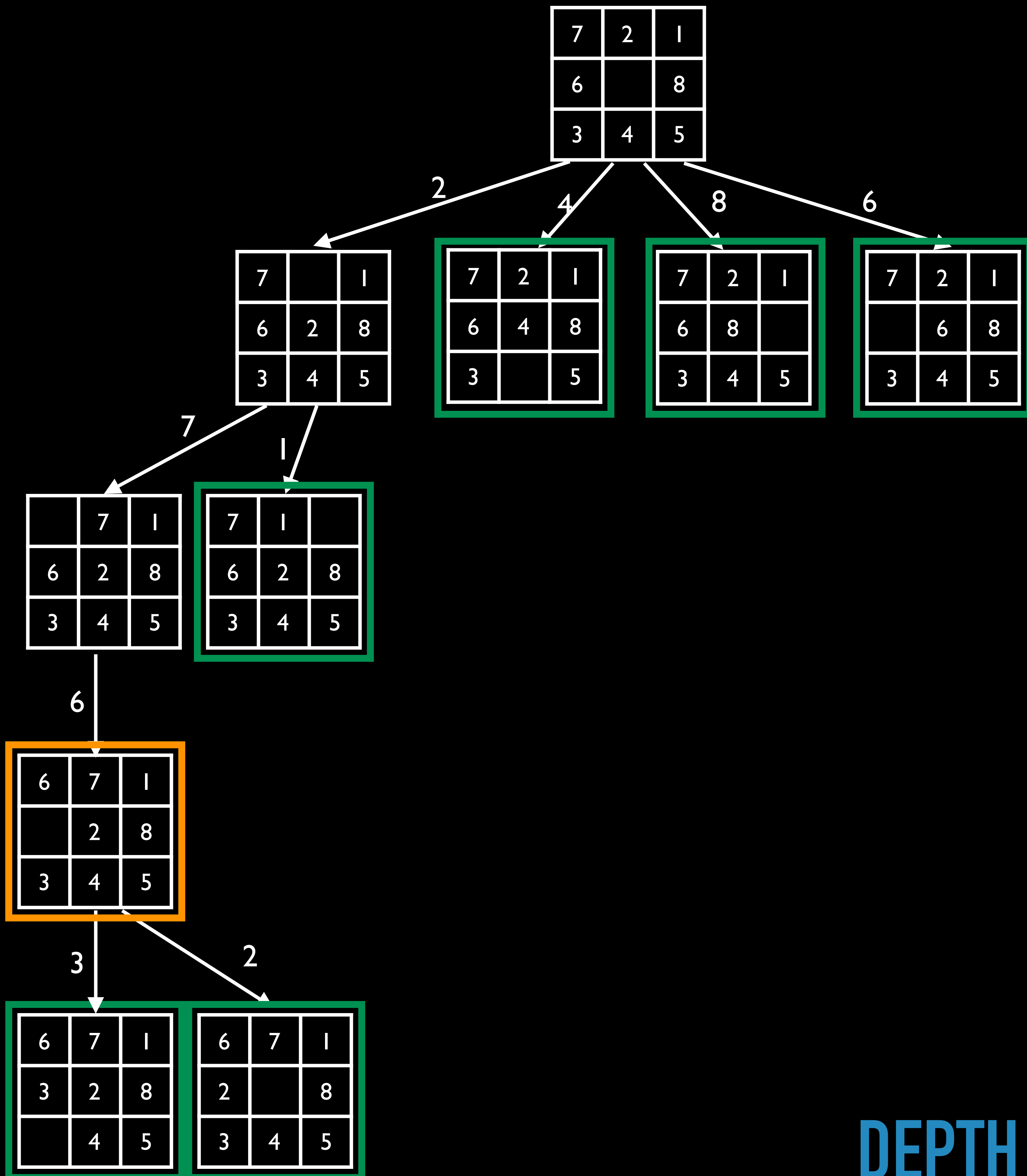
DEPTH FIRST (GRAPH) SEARCH





DEPTH FIRST (GRAPH) SEARCH

DEPTH FIRST (GRAPH) SEARCH



M^G:

0	1	2
3	4	5
6	7	8

SOLVING STATE-SPACE SEARCH PROBLEMS

- Tree-search works great in some applications, but may find sub-optimal solutions in others, or worse, may recurse forever.
- We will discuss the following:
  - Depth First Graph Search
  - Breadth First Search
  - Iterative Deepening
  - Uniform Cost Search
  - Heuristics and A*

Uninformed / Brute Force

Informed / Heuristic

REVIEW YOUR DATA STRUCTURES

- Implementation of problem solving search requires the user to understand the **stack**, **queue**, and **graph** data structures.
- Understanding **linked lists** would be helpful.
- Understanding **dictionaries** (**hash tables**) would also be helpful.
- Relevant AIMA: 3.3-3.7

BACK ON TOPIC

- Before we go any further,
  let's discuss how we can compare search algorithms.

COMPARING SEARCH ALGORITHMS

- Four key properties:
  - Completeness
  - Optimality
  - Time complexity
  - Space complexity

COMPLETENESS AND OPTIMALITY

- A search algorithm is **complete** if it is guaranteed to find a solution when one exists.
- A search algorithm is **optimal** if the solution returned has the lowest path cost (or highest utility) among all possible solutions.

TIME AND SPACE COMPLEXITY

- Computational complexity is a measure of **growth**.
- Specifically, how the resources required by an algorithm grow as a function of problem difficulty.
- E.g., if we double the size of a problem, should our program take twice as long? Less? More?

COMPUTATIONAL COMPLEXITY — TECHNICAL DEFINITION

$$f(x) \in \mathcal{O}(g(x))$$

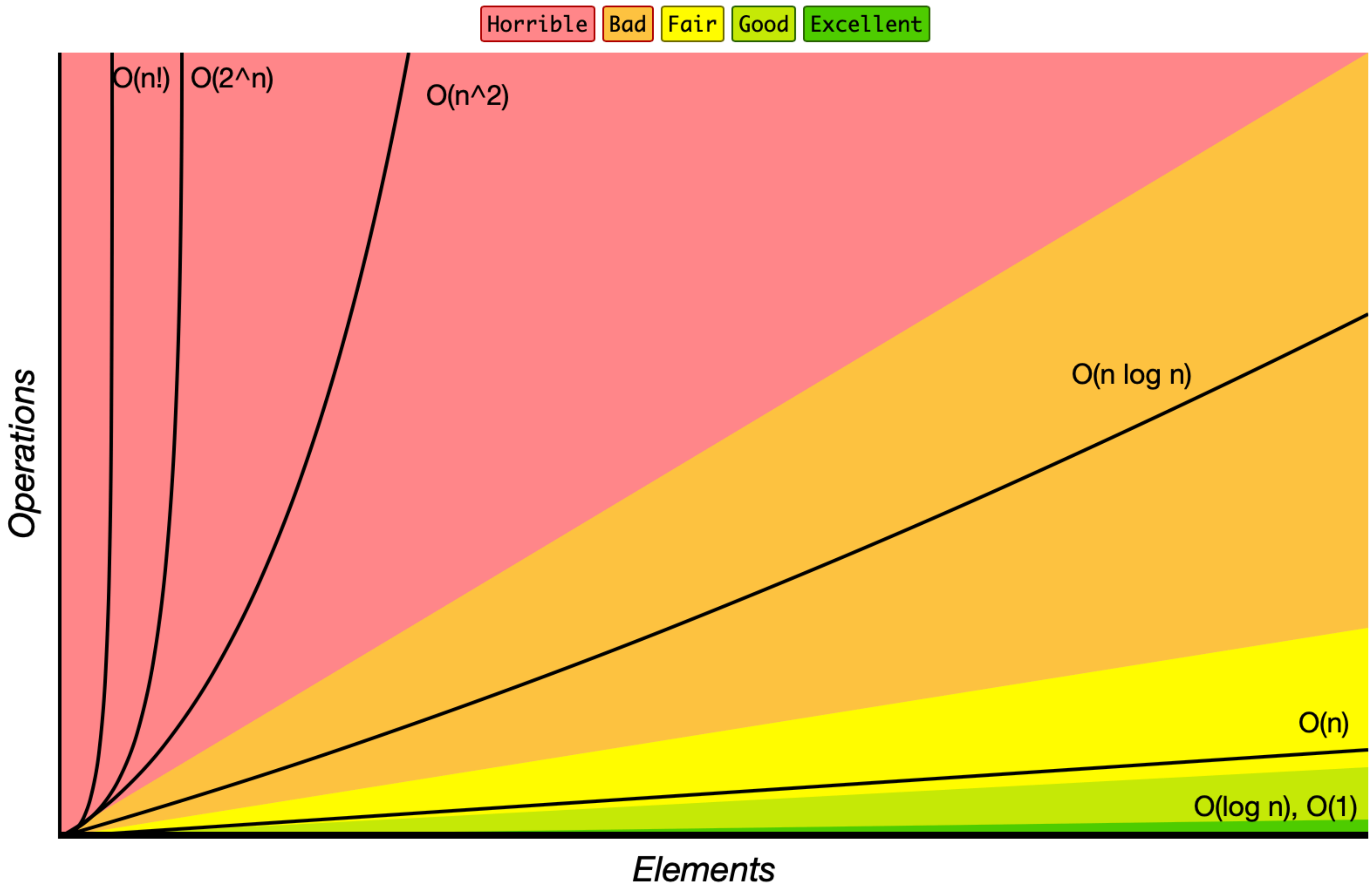
if and only if

$$\exists\ M \in \mathcal{R},\ x_o \in \mathrm{dom}(f)$$

such that

$$|f(x)| \leq M g(x)\ \forall x \geq x_0$$

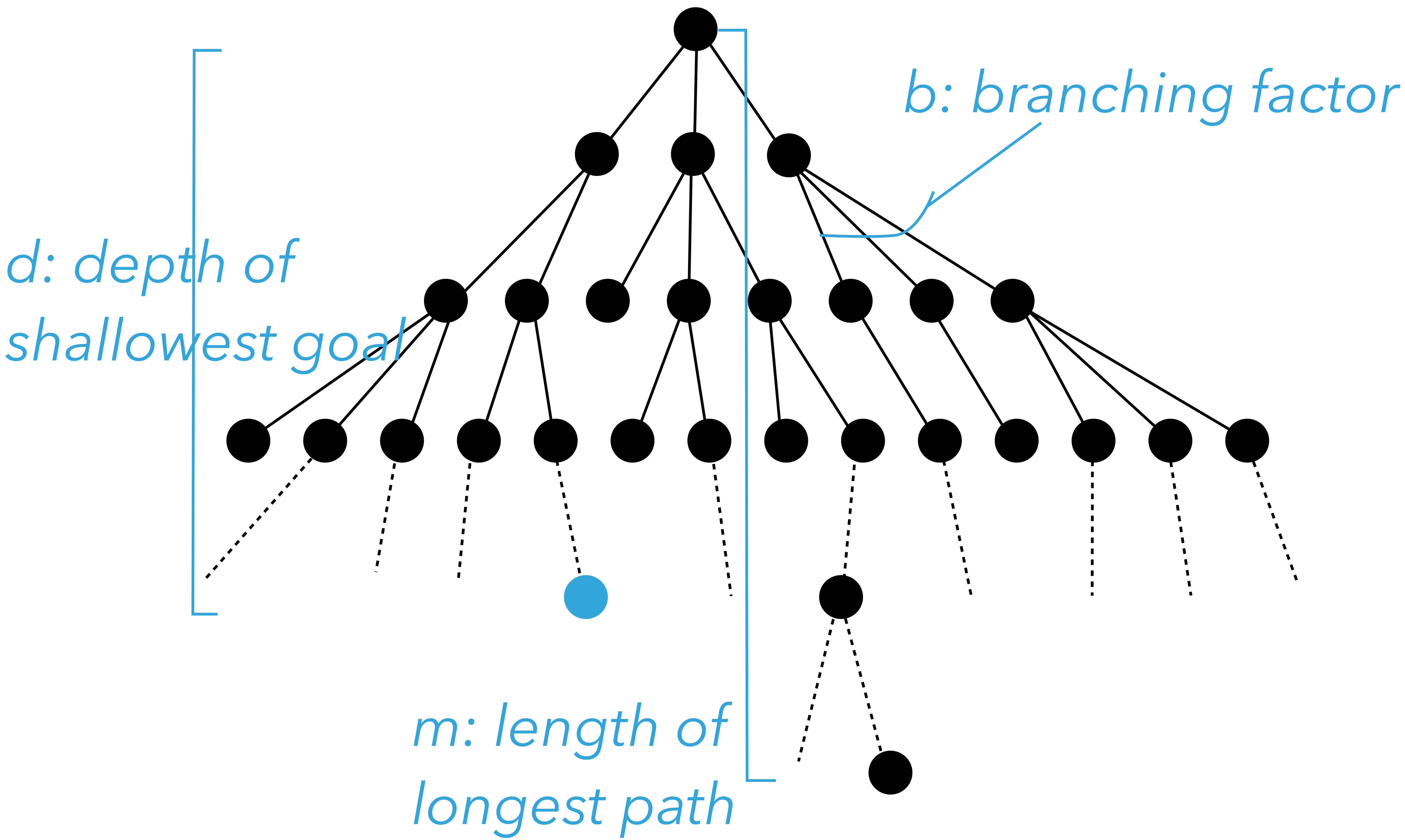
COMPUTATIONAL COMPLEXITY — CLASSIC FIGURE



https://www.bigocheatsheet.com

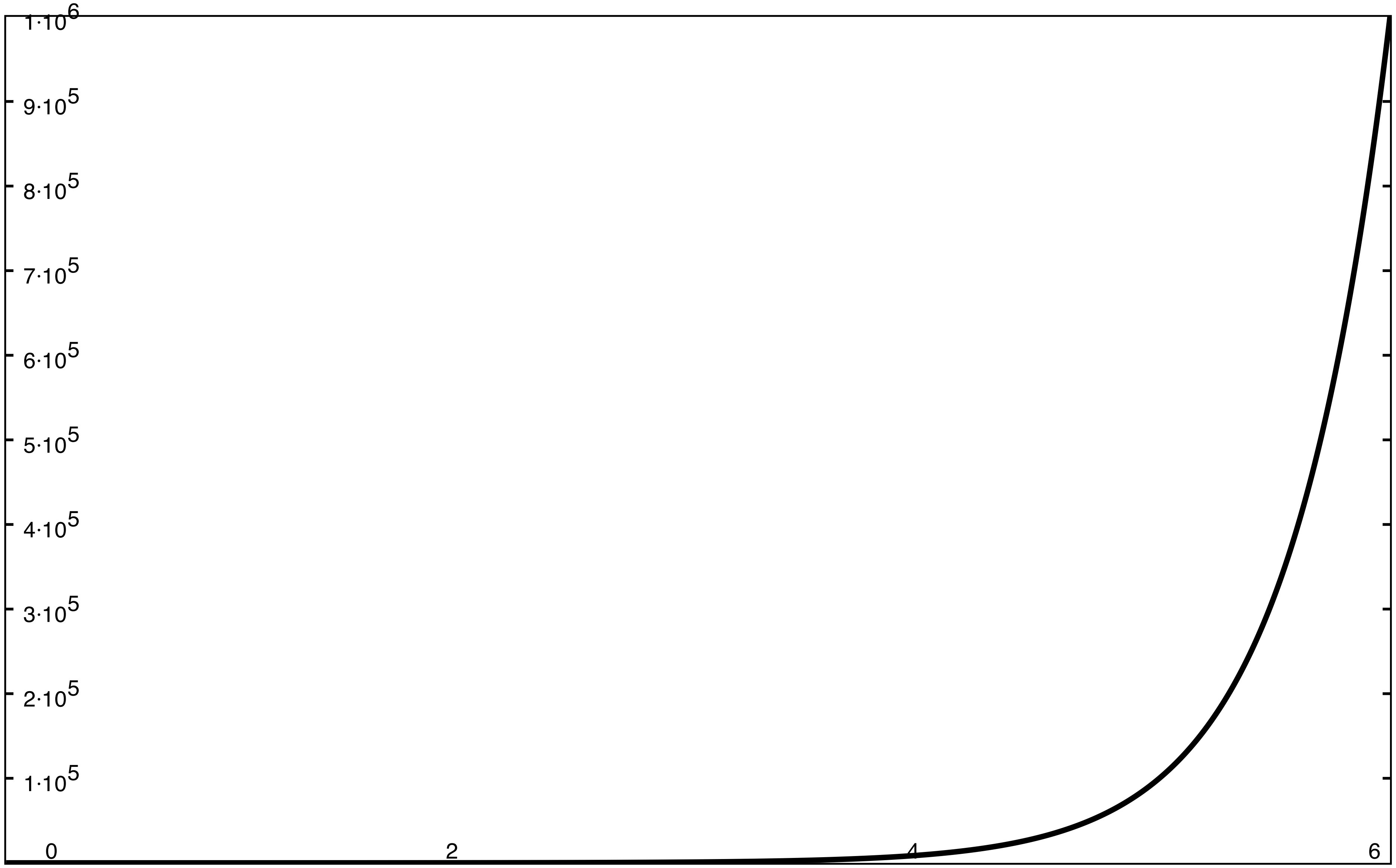
COMPARING SEARCH ALGORITHMS

- How do we characterize the “size” of a state-space problem?
  - **b** – the branching factor (avg/max)
  - **m** – the depth of the deepest state (can be infinite!)
  - **d** – the depth of the shallowest solution


b: branching factor
d: depth of shallowest goal
m: length of longest path

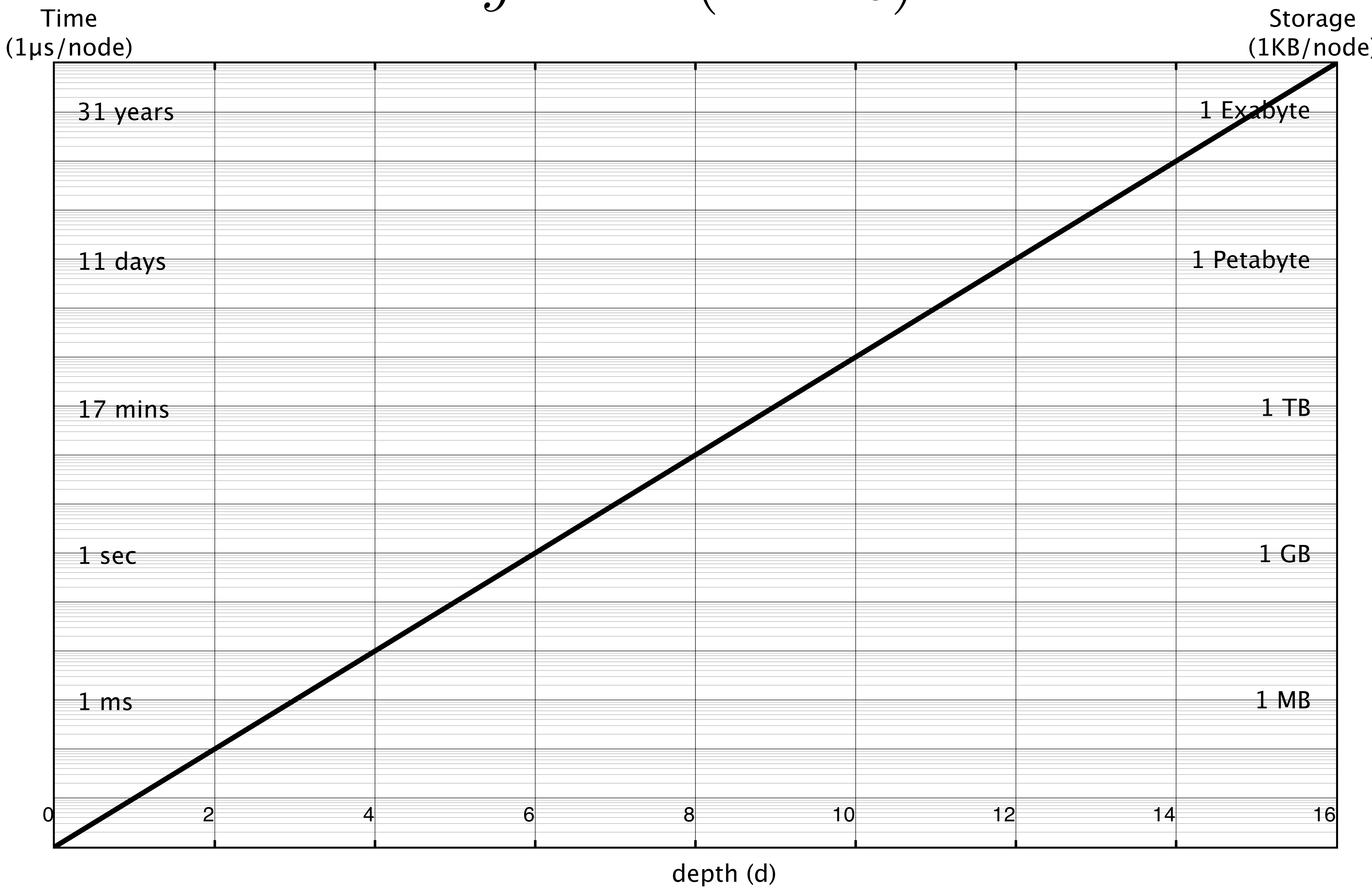
EXPONENTIAL COMPLEXITY

$$y = b^d \quad (b = 10)$$



EXPONENTIAL COMPLEXITY

$$y = b^d \quad (b = 10)$$



AIMA

"Exponential complexity search problems cannot be solved by uninformed methods for any but the smallest instances."

SEARCH ALGORITHMS

- Now we know that we want **complete**, **optimal**, algorithms with low **time complexity** and **space complexity.**

- How does Depth-First Tree search stack up?

SEARCH ALGORITHMS — DFTS

- Complete… ?
- Optimal… ?
- Time complexity … ?
- Space complexity … ?

SEARCH ALGORITHMS — DFTS

- Complete… No ✗
- Optimal… No ✗
- Time complexity … $\mathcal{O}(b^m)$ ✗
- Space complexity … $\mathcal{O}(bm)$ ✓

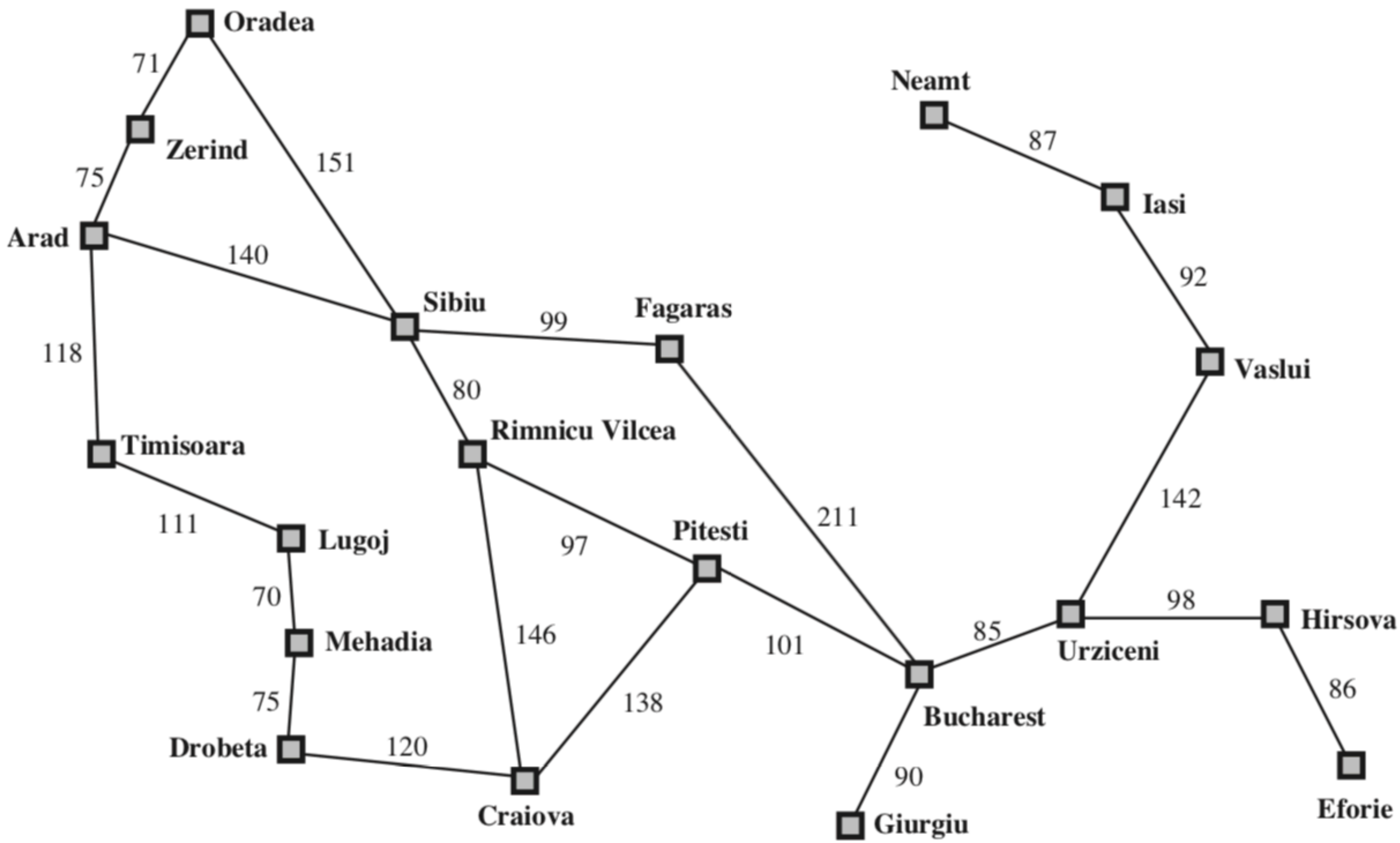
SEARCH ALGORITHMS — DEPTH FIRST GRAPH SEARCH

- Complete… ?
- Optimal… ?
- Time complexity … ?
- Space complexity … ?

SEARCH ALGORITHMS — DEPTH FIRST GRAPH SEARCH

- Complete… No ✗
- Optimal… No ✗
- Time complexity … $\mathcal{O}(b^m)$ ✗
- Space complexity … $\mathcal{O}(b^m)$ ✗

ROUTE FINDING



BREADTH-FIRST SEARCH

- Expand all the nodes in a level before expanding any of their children
- Expand the shallowest unexpanded node
- Use a FIFO queue for the frontier

BREADTH FIRST SEARCH

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)   /* chooses the shallowest node in frontier */
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
           frontier ← INSERT(child, frontier)
```

AIMA Figure 3.11, p82

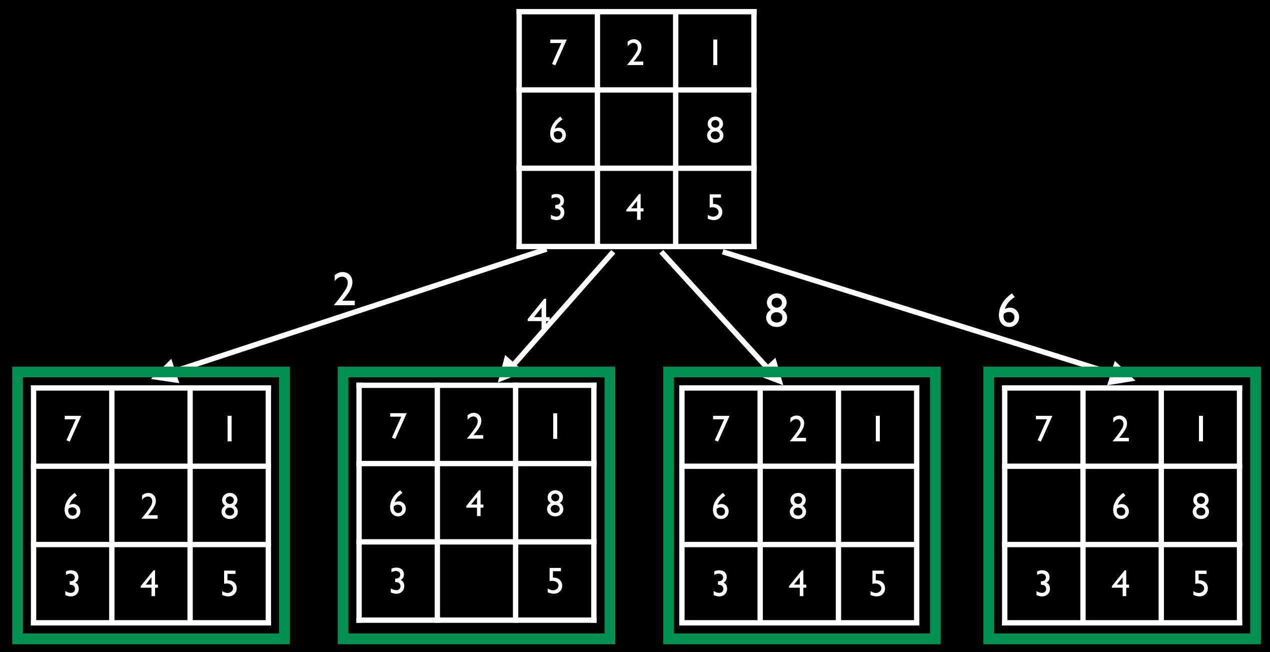
BREADTH FIRST SEARCH

7	2	1
6		8
3	4	5

M^G:

0	1	2
3	4	5
6	7	8

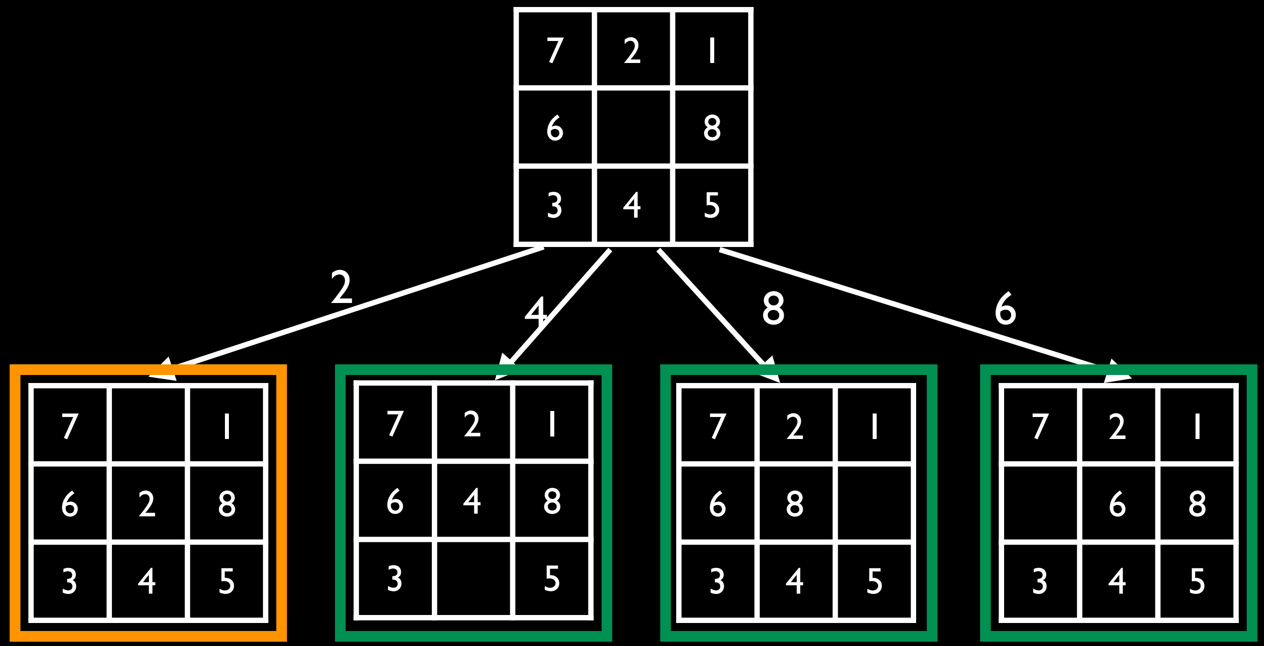
BREADTH FIRST SEARCH



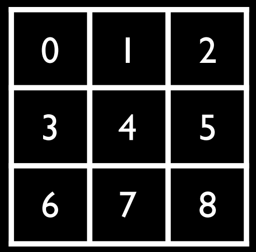
M^G:

0	1	2
3	4	5
6	7	8

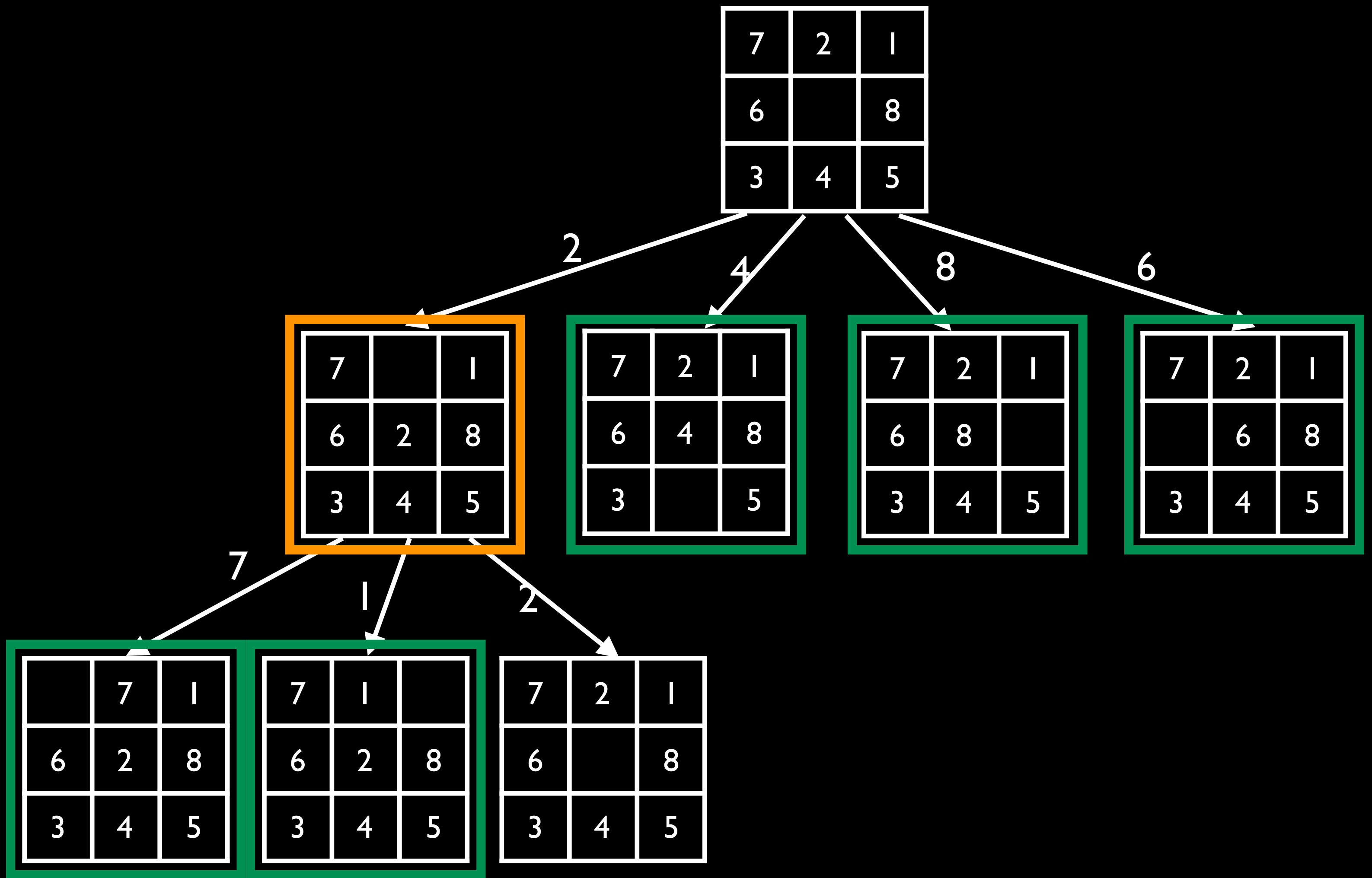
BREADTH FIRST SEARCH

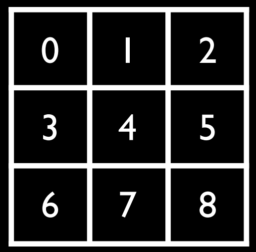


M^G:

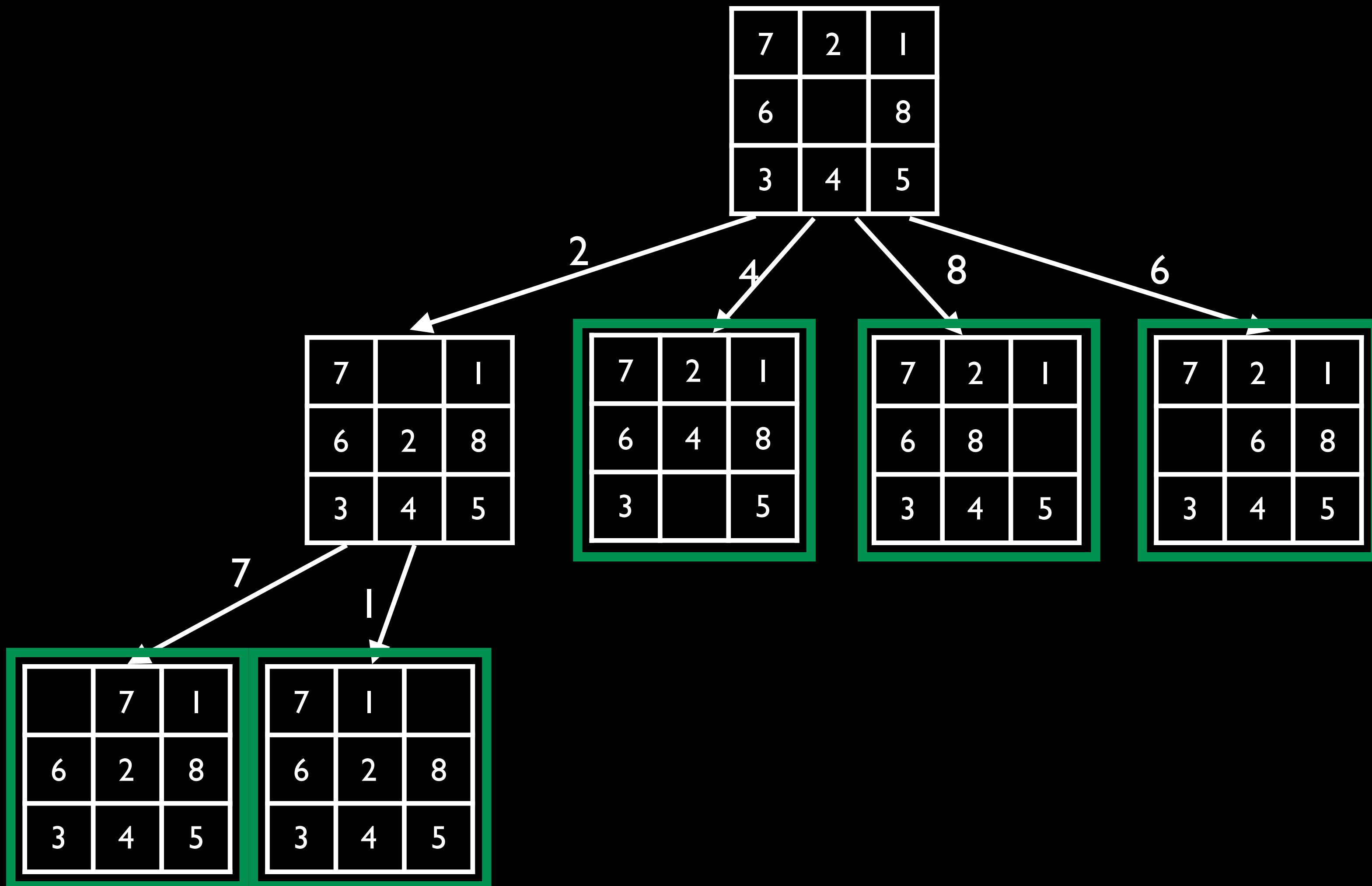


BREADTH FIRST SEARCH



MG: 

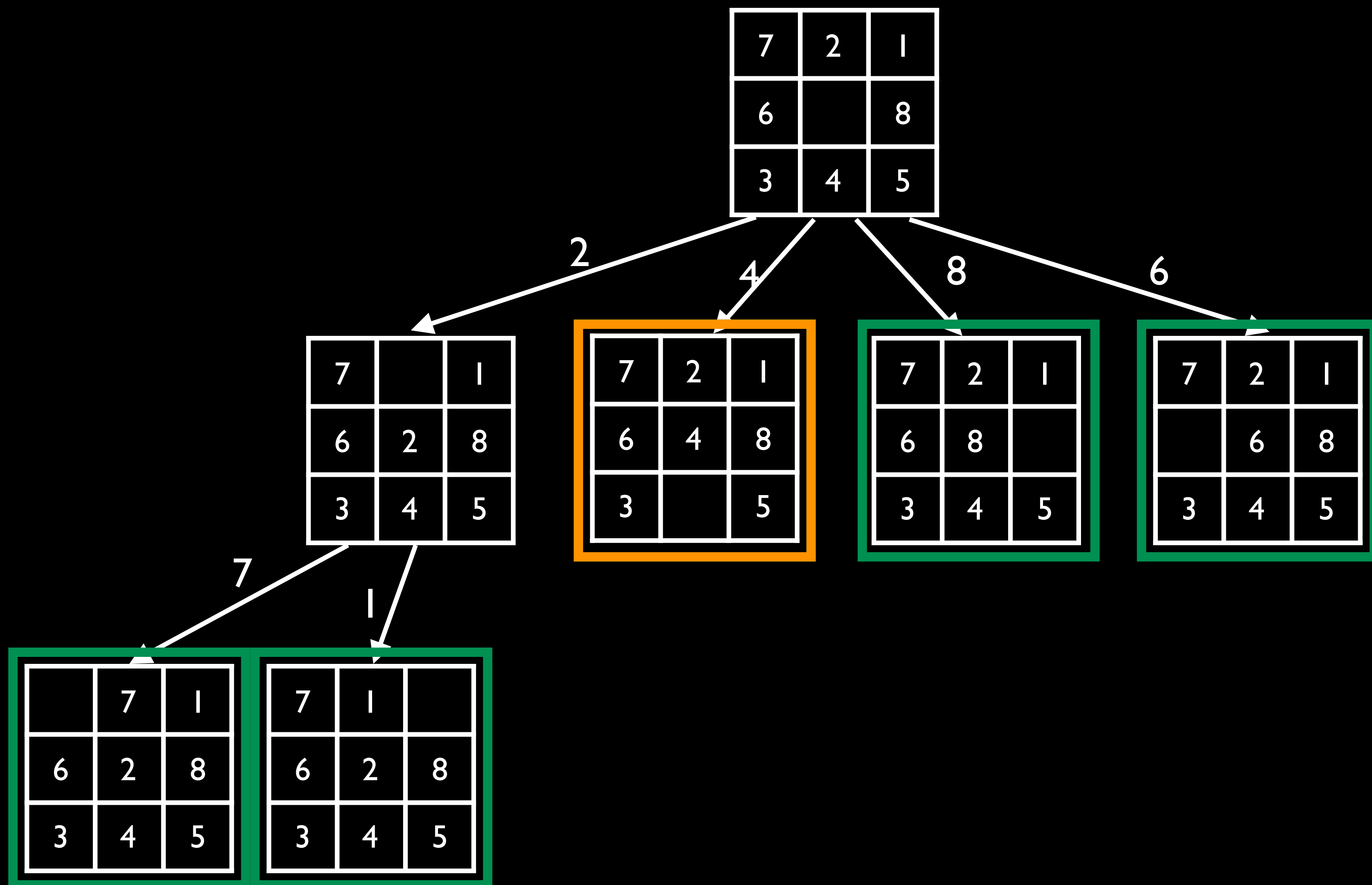
BREADTH FIRST SEARCH



M^G:

0	1	2
3	4	5
6	7	8

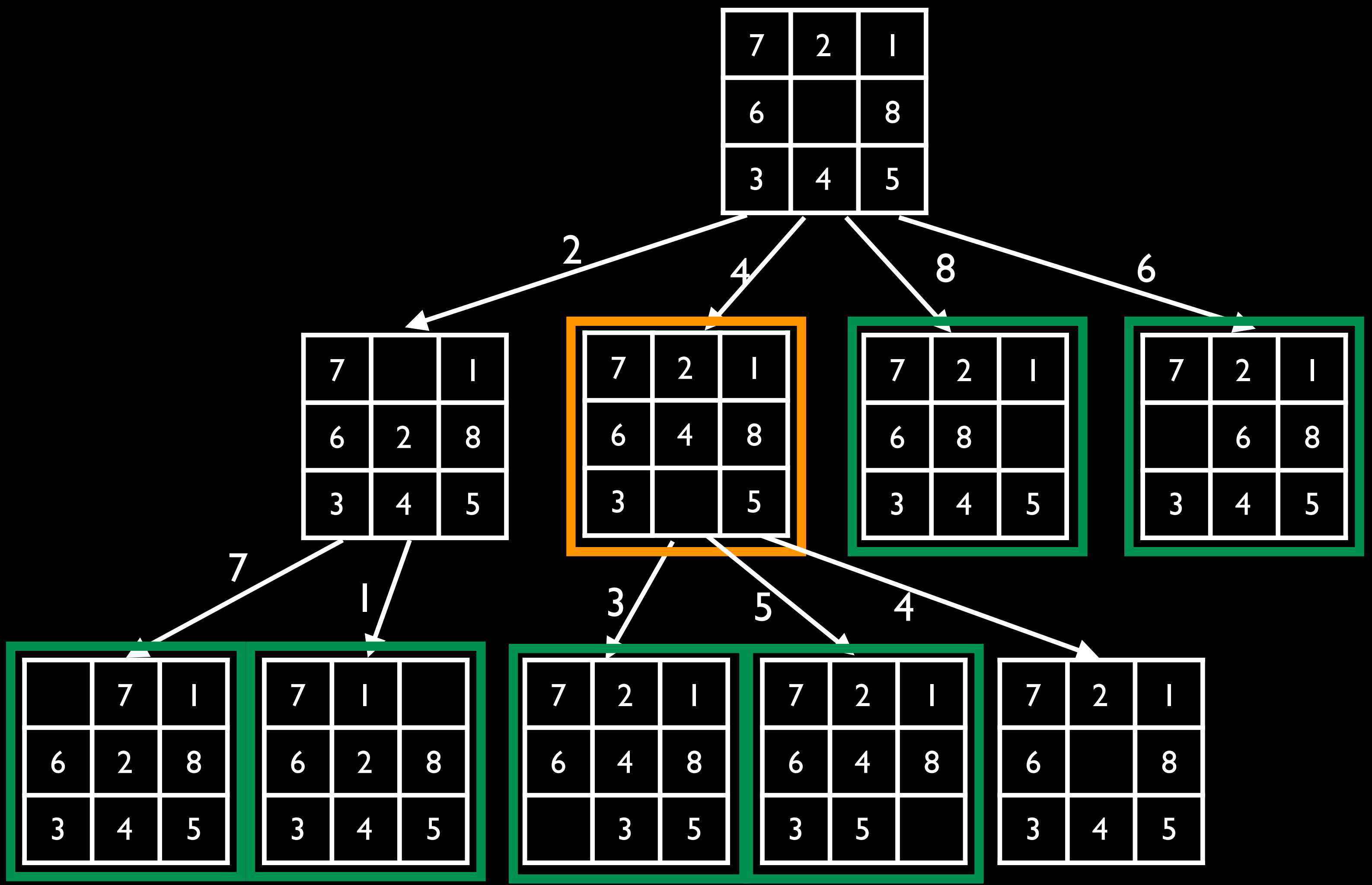
BREADTH FIRST SEARCH

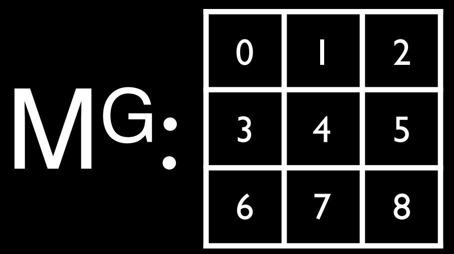


M^G:

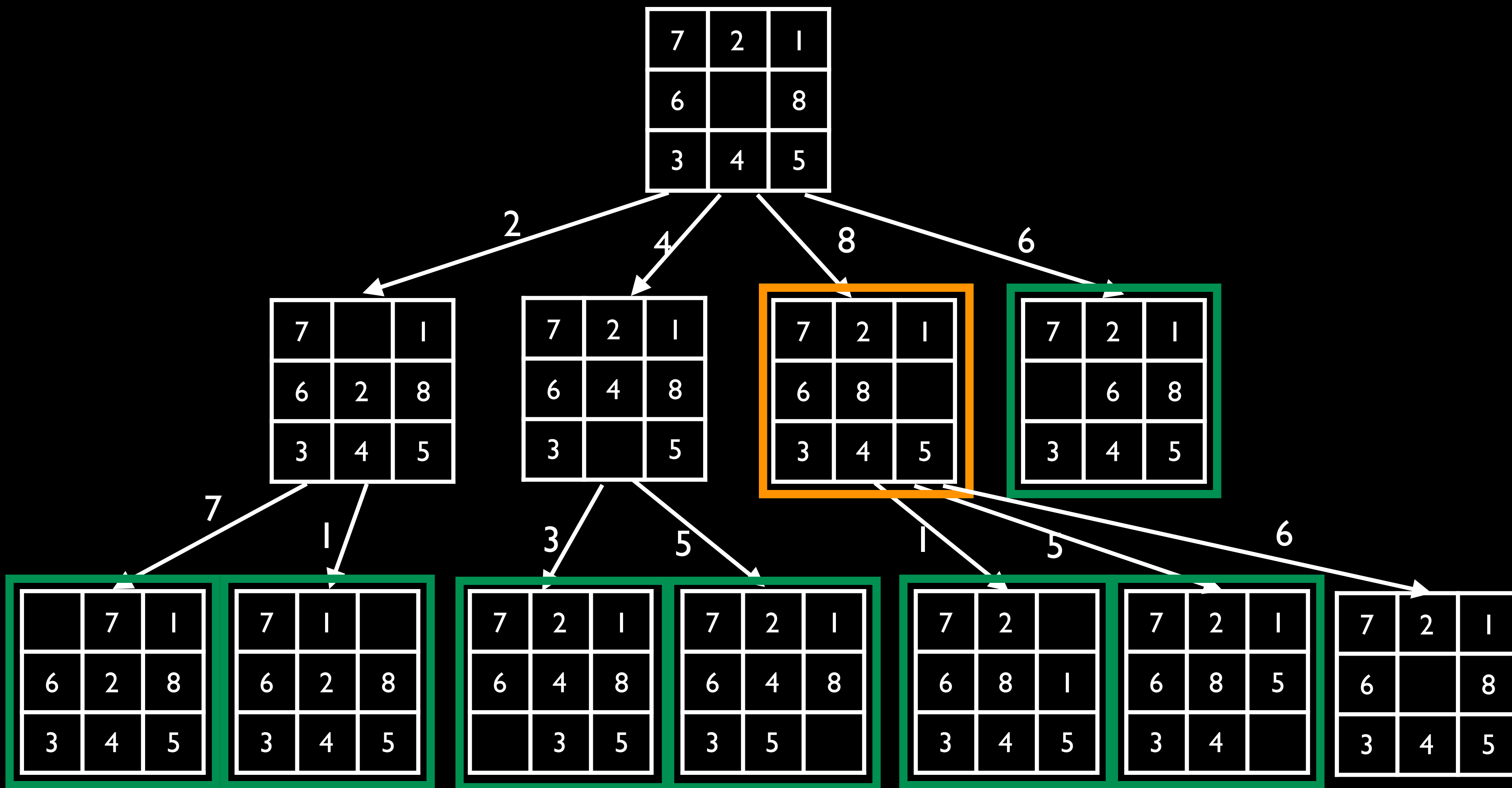
0	1	2
3	4	5
6	7	8

BREADTH FIRST SEARCH





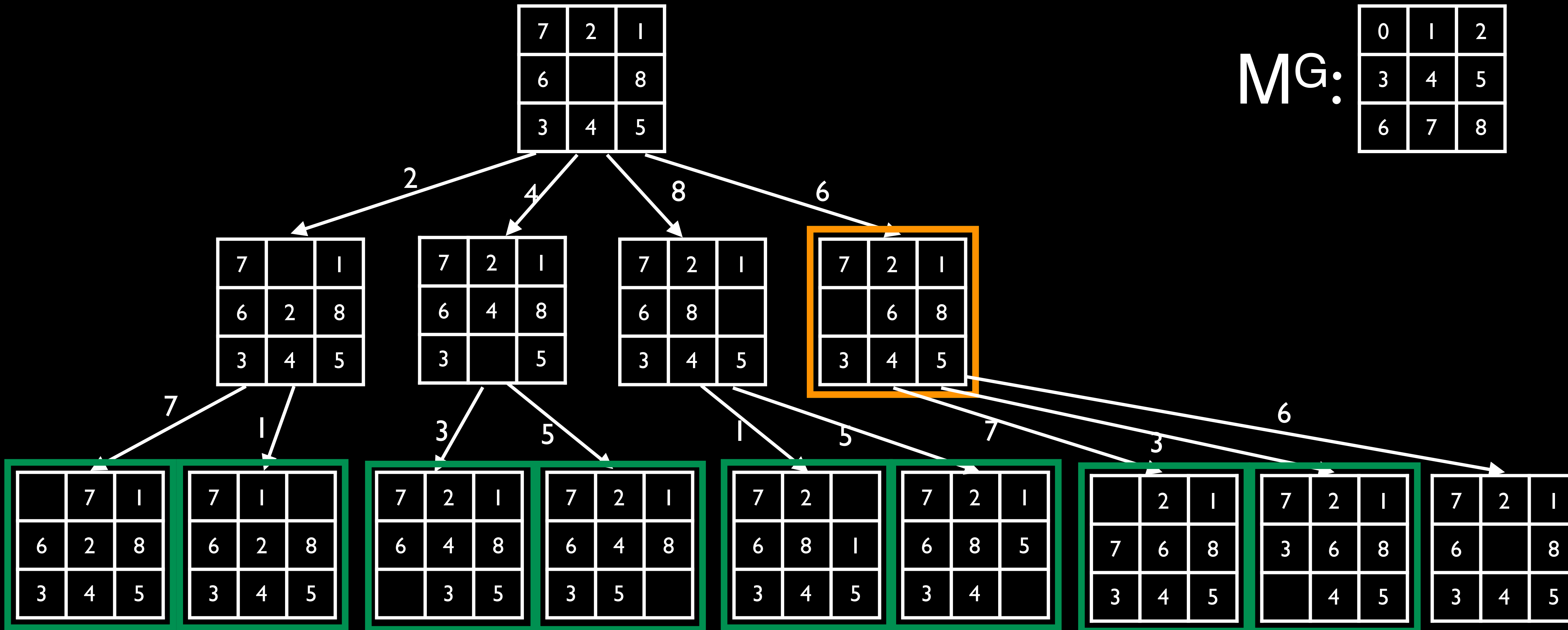
BREADTH FIRST SEARCH



M^G:

0	1	2
3	4	5
6	7	8

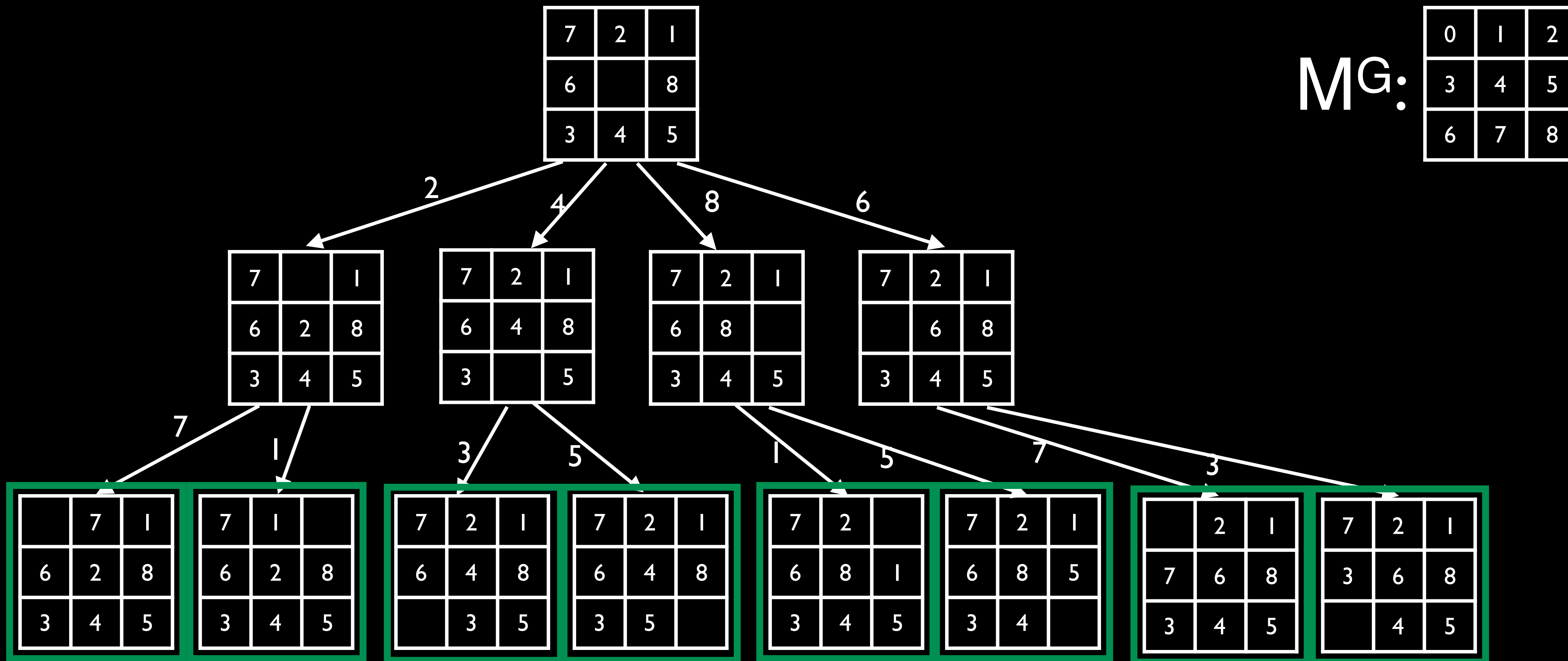
BREADTH FIRST SEARCH



M^G:

0	1	2
3	4	5
6	7	8

BREADTH FIRST SEARCH



BREADTH FIRST SEARCH

SEARCH ALGORITHMS — BFS

- Complete… ?
- Optimal… ?
- Time complexity … ?
- Space complexity … ?

SEARCH ALGORITHMS — BFS

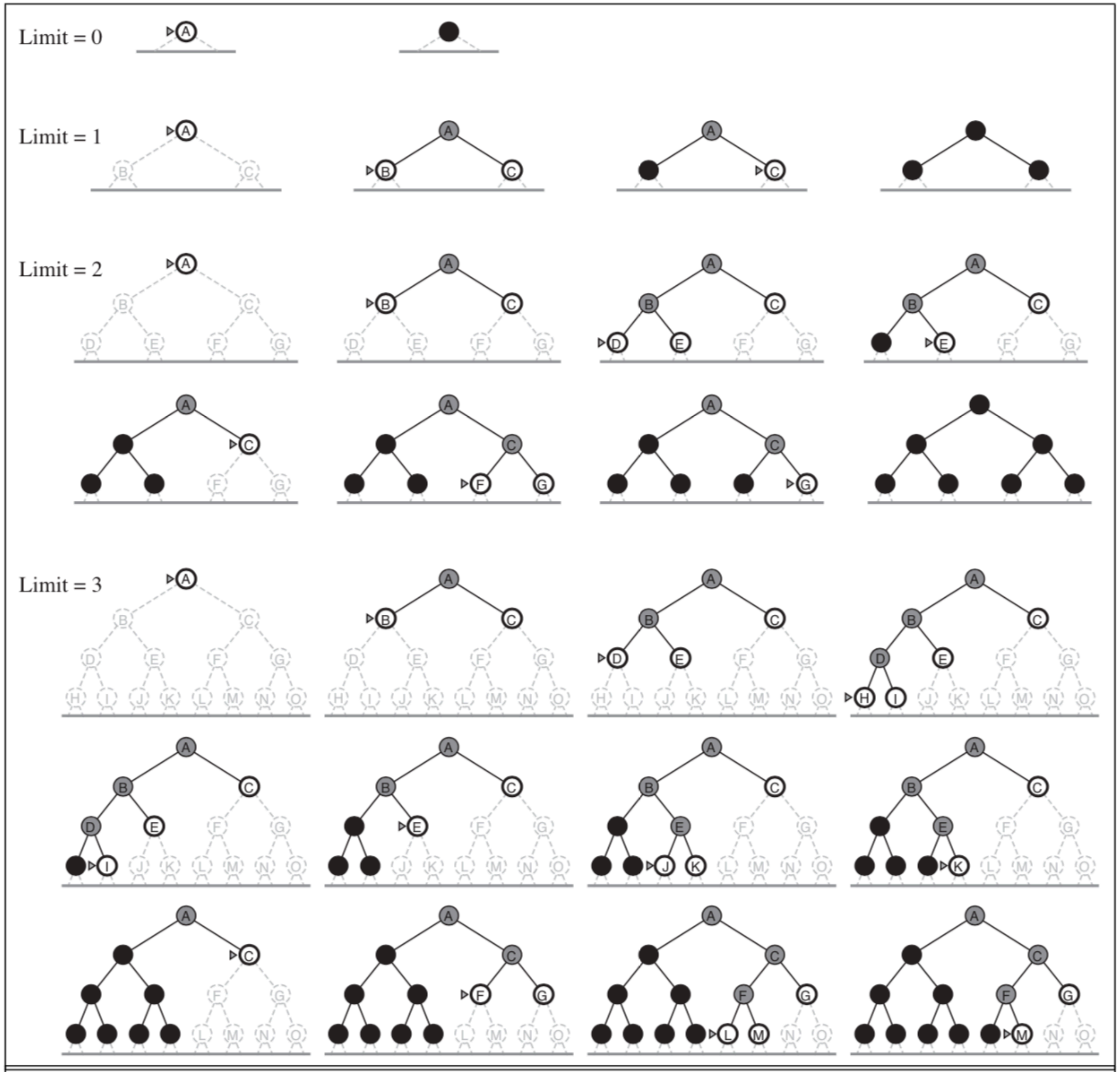
- Complete… Yes! ✔
- Optimal… Yes! ✔
- Time complexity … $\mathcal{O}(b^d)$ ✘
- Space complexity … $\mathcal{O}(b^d)$ ✘

BREADTH FIRST SEARCH

- Pros – solution obtained has fewest number of steps (so it's **optimal** for uniform action costs), and **complete** – even for infinite graphs (assuming finite branching factor and infinite memory).
- Cons – **intractable memory requirements** for nontrivial graphs, if steps have nonuniform cost then solution may be suboptimal.

ITERATIVE DEEPENING

- Systematically explore (via depth-first tree-search) all states at depth one, then restart and explores all states to depth two, then restart, …



AIMA Figure 3.19, p89

ITERATIVE DEEPENING SEARCH

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
     cutoff_occurred? ← false
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        result ← RECURSIVE-DLS(child, problem, limit − 1)
        if result = cutoff then cutoff_occurred? ← true
        else if result ≠ failure then return result
     if cutoff_occurred? then return cutoff else return failure
```

AIMA Figure 3.17, p88

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
   for depth = 0 to ∞ do
      result ← DEPTH-LIMITED-SEARCH(problem, depth)
      if result ≠ cutoff then return result
```

AIMA Figure 3.18, p89

ITERATIVE DEEPENING SEARCH

- Performs depth-first search (without recording visited states) at each step, so individual steps have low memory requirements.
- Because of the exponential nature of branching, the total amount of work performed (i.e., computation time) is bounded by the branching factor of what would result from breadth-first search.
- Space complexity – $\mathcal{O}(bd)$
- Time complexity – ?

IDS — TIME COMPLEXITY

$$\mathcal{O}(b^d)$$

- **Total cost:**

$$(d+1) + db + (d-1)b^2 + \cdots + 3b^{d-2} + 2b^{d-1} + b^d$$

ITERATIVE DEEPENING

- Performs depth-first search (without recording visited states) at each step, so individual steps have low memory requirements.
- Because of the exponential nature of branching, the total amount of work performed (i.e., computation time) is bounded by the branching factor of what would result from breadth-first search.
- Space complexity – $\mathcal{O}(bd)$ ✓
- Time complexity – $\mathcal{O}(b^d)$ ✗

ITERATIVE DEEPENING

- Pros – **complete**, **optimal** (for uniform action costs), **memory efficient**.
- Cons – still exponential time.

UNINFORMED STRATEGIES

	BFS	DFGS	DFTS	IDS
Complete?	✓	✓	✗	✓
Optimal?	✓*	✗	✗	✓*
Time	$O(b^d)$	$O(b^m)$	$O(b^m)$	$O(b^d)$
Space	$O(b^d)$	$O(b^m)$	$O(bm)$	$O(bd)$

* If step costs are identical (see book)

AIMA

"Iterative deepening is the preferred uninformed search method when the search space is large and the depth of the solution is not known."

NEXT TIME

- A* and Heuristic Search
- Read textbook: 3.5-3.7

We stoped here on Monday.

We covered DFS, BFS, IDS,
and analysis of graph search algorithms.

We have not yet discussed UCS or A*.