CS440/ECE448: Intro to Artificial Intelligence

Lecture 3: Systematic search

Prof. Julia Hockenmaier
juliahmr@illinois.edu

http://cs.illinois.edu/fa11/cs440

Review

Different kinds of agents:
reflex-based, model-based, goal-based, utility-based, learning-based

How do we evaluate agents?
External performance measure

What is the task environment like:
observable?, known?, deterministic? sequential?, static?

When is an agent rational?

Answer 1: When an agent chooses actions that bring it closer to the goal.

Answer 2: When an agent chooses actions that it expects to bring it closer to the goal

Answer 2 is correct.

Problem solving as search

Problem solving as search

Problem solving

- Finding *any* solution (*goal-driven*)
- Finding *the cheapest* solution (*utility-driven*)

Uninformed (blind) search (goal-driven):

- Algorithms: breadth-first; depth-first

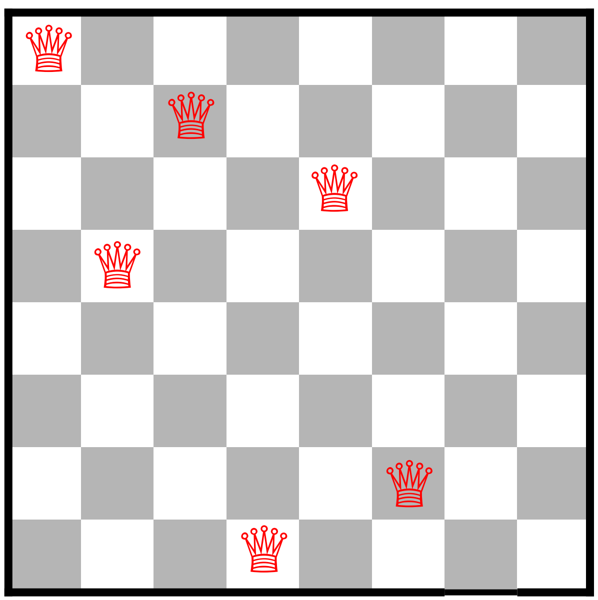
Informed (heuristic) search (utility-driven):

- Search costs; admissible heuristics
- Algorithms: greedy best first; A* search

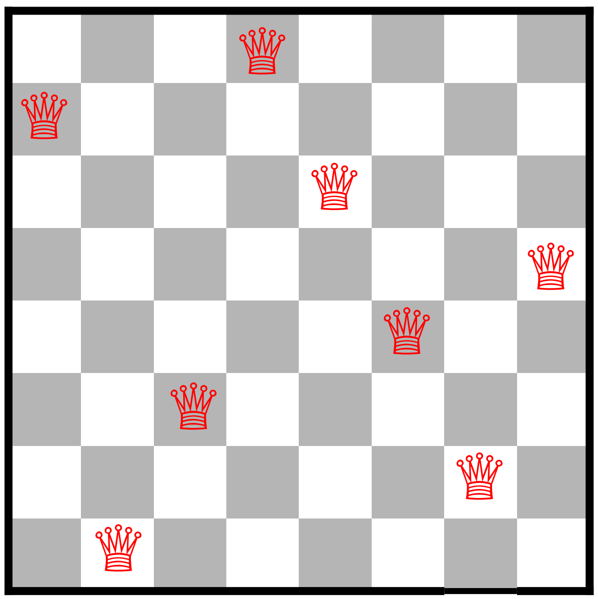
Problem solving

The 8 queens problem

Can you place 8 queens on a chess board so that they don't attack each other?

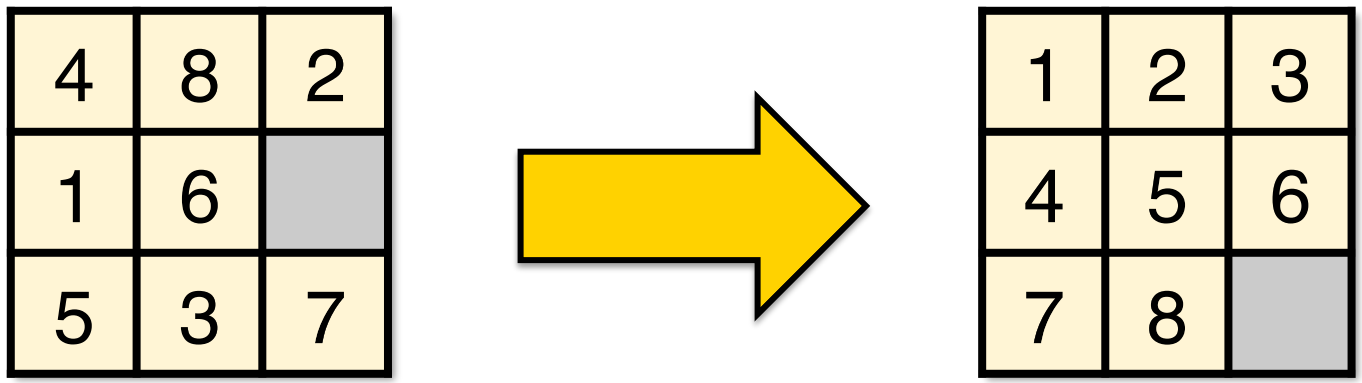


This doesn't work!



Phew!

The 8-puzzle



Initial State

Goal

Four possible actions:

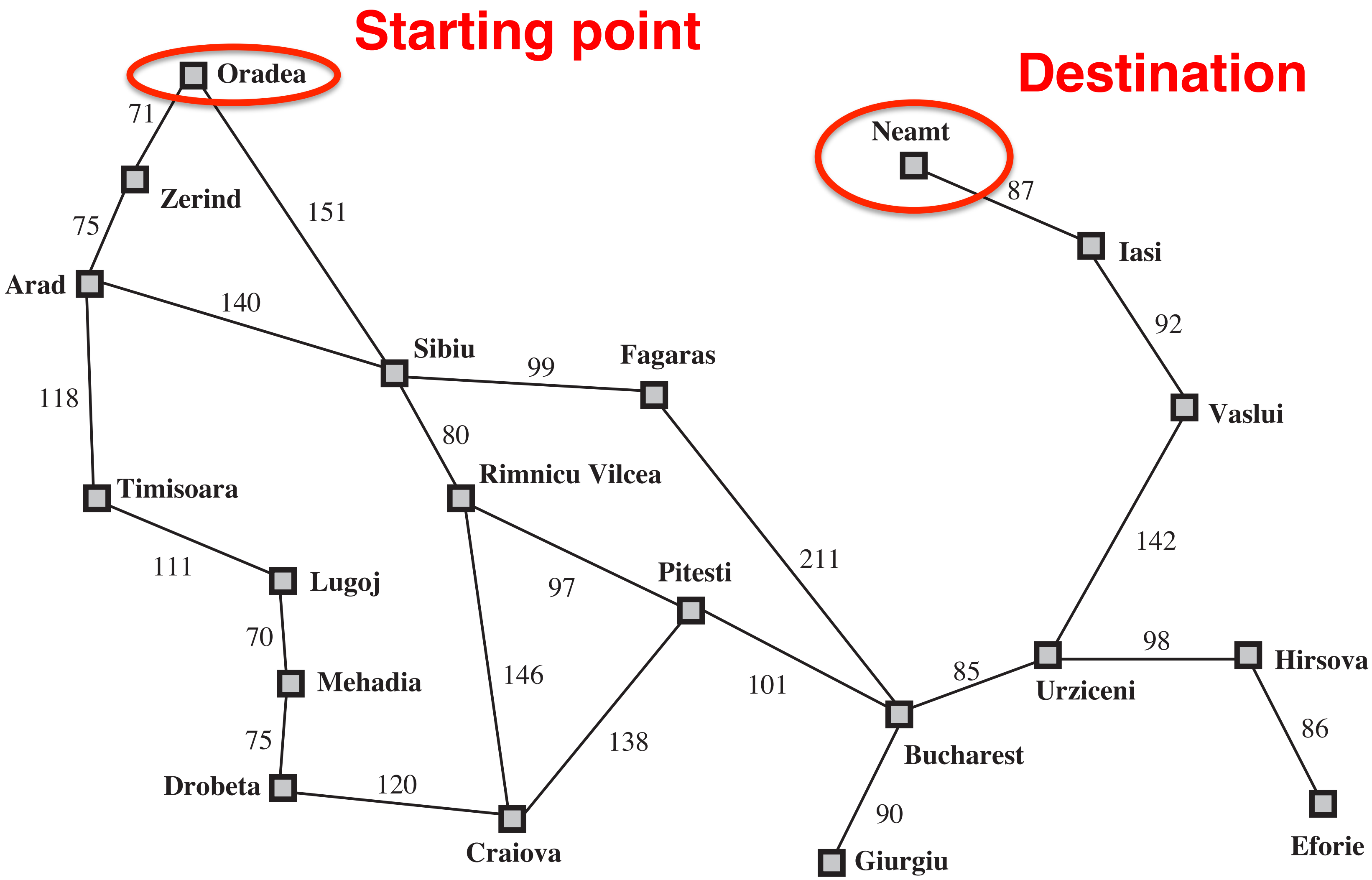
MoveTileUp
MoveTileDown
MoveTileLeft
MoveTileRight

Cryptarithmetic

```
     send            forty
                       ten
+    more          +   ten
---------          ---------
=   money          =  sixty
```

Find a letter/digit substitution that forms a natural & correct arithmetic expression

The route-finding problem



This lecture: assumptions

Today's methods work when the environment is:

1. **observable**
   (Agent perceives all it needs to know)
2. **known**
   (Agent knows the effects of each action)
3. **deterministic**
   (Each action always has the same outcome)

In such environments, the solution to any problem is a **fixed sequence of actions**.

Solving a problem

1. Formulate a **goal**
   goal = a (set of) state(s) to be in

2. Define the corresponding **problem**
   problem = what actions and states to consider

3. Find the **solution** to the problem
   solution = a sequence of actions to reach goal

4. Execute the solution

Implementing problem solving

We need:

- a data structure to represent **states**
- a designated **initial state**
- a function that maps states to states to represent **actions** (operators)
- a boolean predicate (goal test) on states that checks whether a state is a **goal state**

Representing states

8-queens: a set of chessboard positions

```
{}, {a4}, {a4, b6}, …
```

8-puzzle: a list of nine numbers

```
<1,2,8,5,4,0,6,7,3>
```

Cryptarithmetic: a tuple of three sets

(unassigned letters, unassigned digits, assignments)

```
({e, m, n, o, r, s, y,}
 {0, 1, 2, 3, 4, 5, 6, 7, 9}
 {d:8})
```

The initial state

8-queens: the empty board `{}`

Cryptarithmetic:
(all letters, all digits, no assignments)
```
({d e, m, n, o, r, s, y,}
 {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
 {})
```

8-puzzle: a random board
```
<1,2,8,5,4,0,6,7,3>
```

Actions

8-queens:

```
placeQueen(e4, s):
   if e4 ∉ s:   (precondition)
   return s ∪ {e4} (effect)
```

Cryptarithmetic:

```
replaceLetterWithDigit(l,d, s):
   if l ∈ s.unassignedLetters
       and d ∈ s.unassignedDigits:
   return (s.unassignedLetters\{l},
           s.unassignedDigits\{d},
           s.assignments ∪ {l:d})
```

The cost of actions

Actions may have different costs:
The cost of driving from A to B depends on the distance (and traffic conditions)

We may need a **cost function** which calculates the (exact) cost of each action.

We may want to find the **lowest-cost solution** (otherwise, we're happy with the first one we find)

Goal test

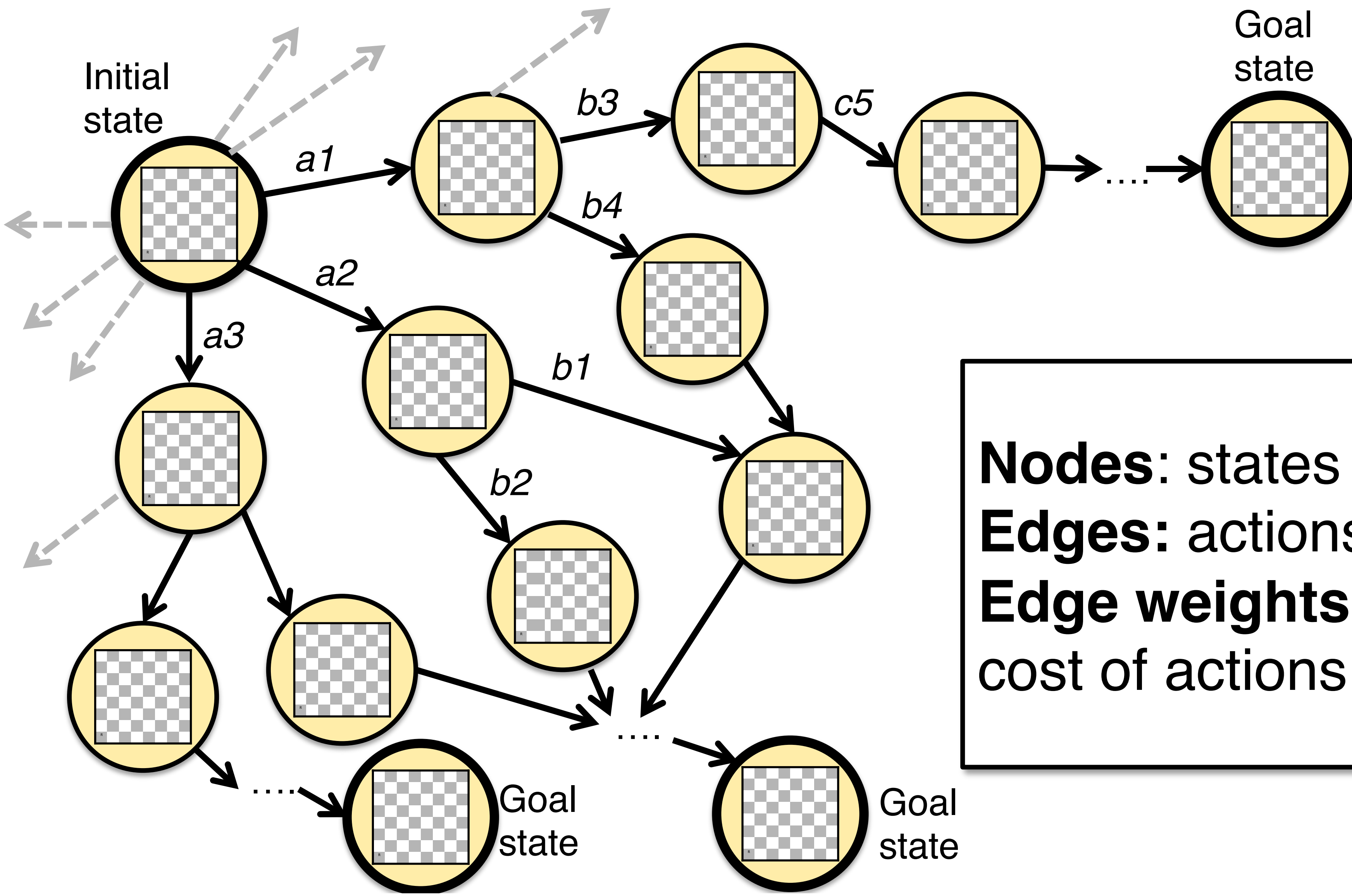
8-queens:

Are eight queens placed, and no queen attacks another queen?

Cryptarithmetic:

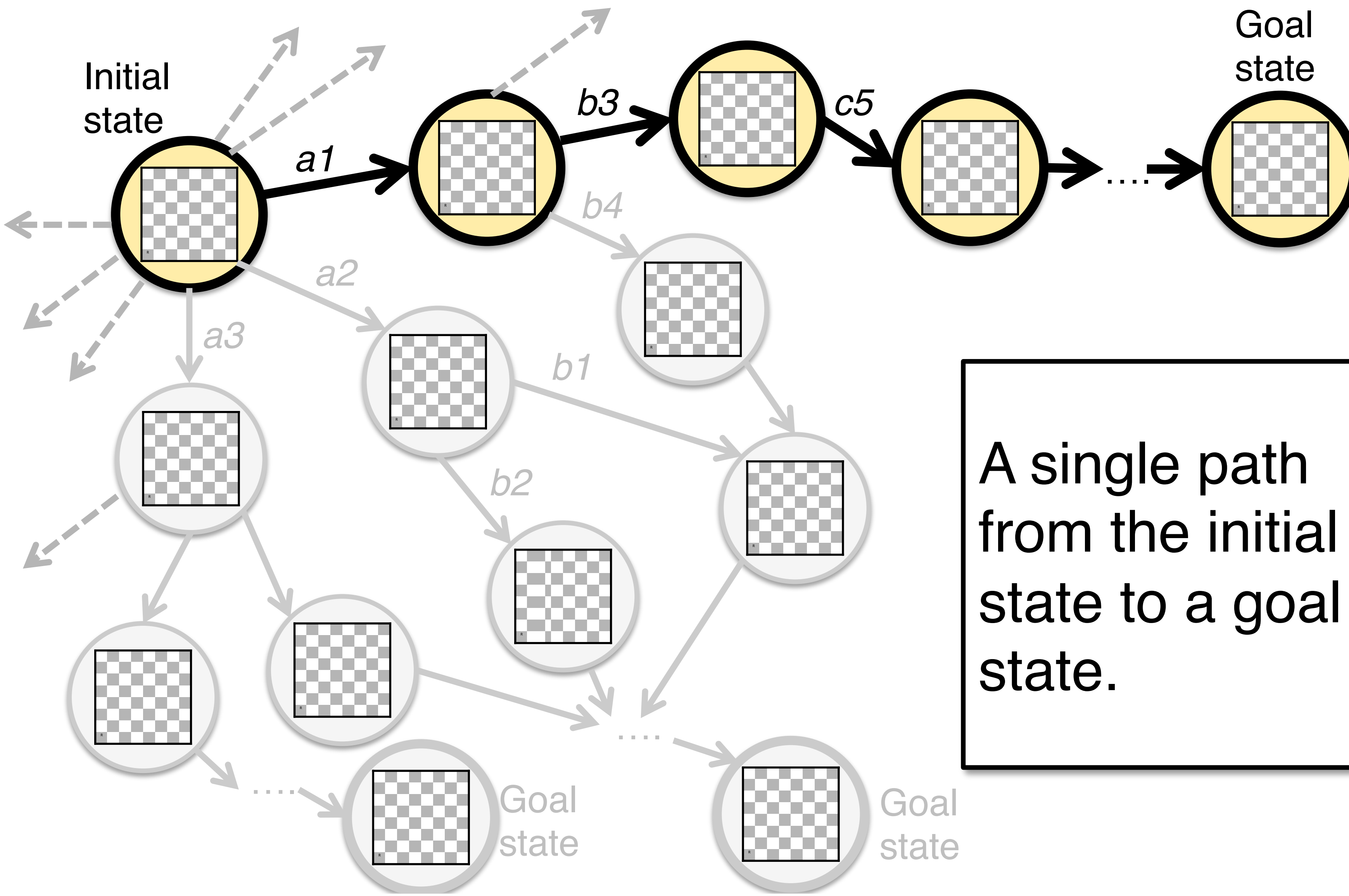
Have all letters been replaced by different digits, are there no leading zeros, and are all calculations correct?

State-space graph



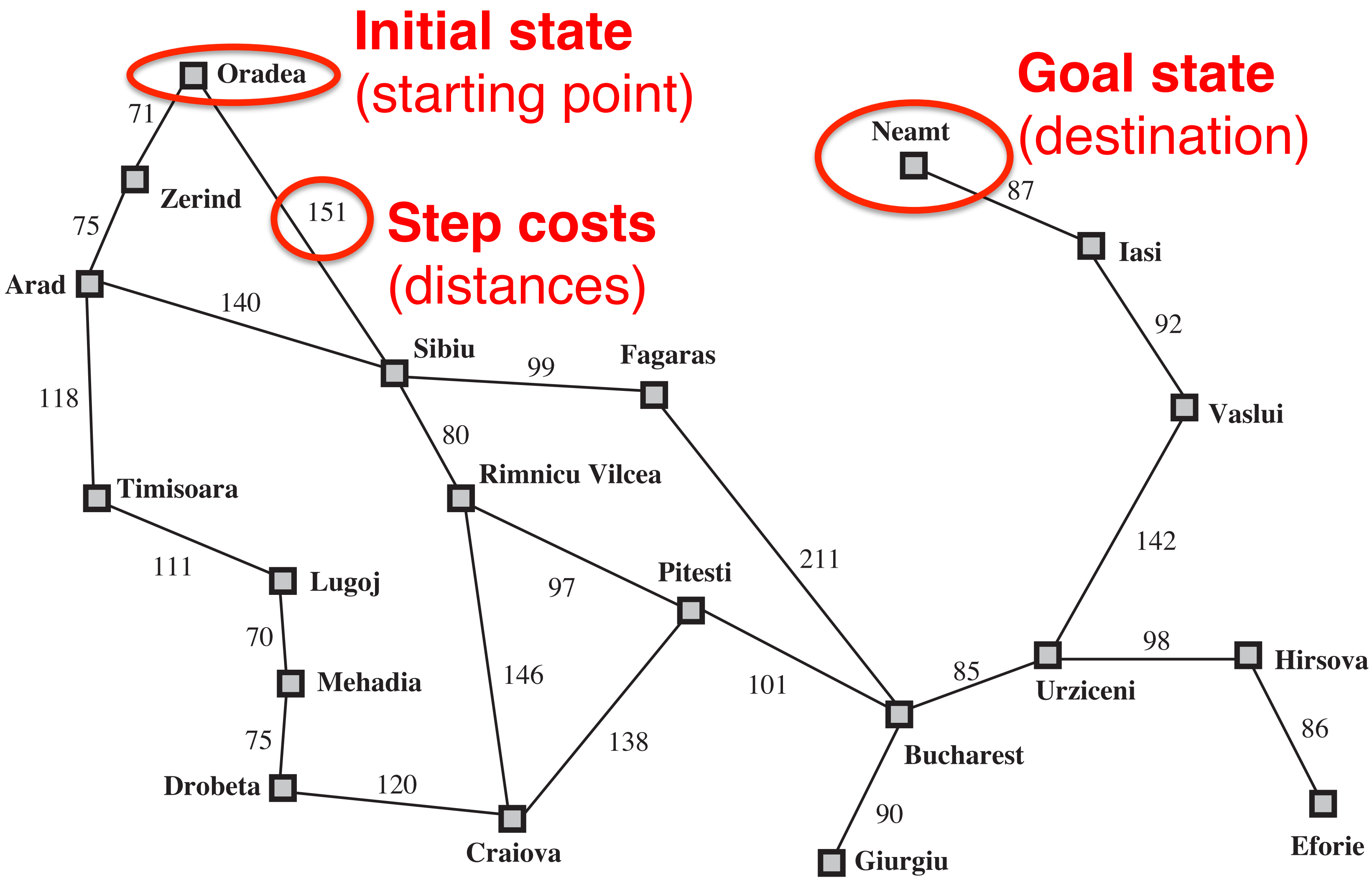
Nodes: states
Edges: actions
Edge weights:
cost of actions

Solution



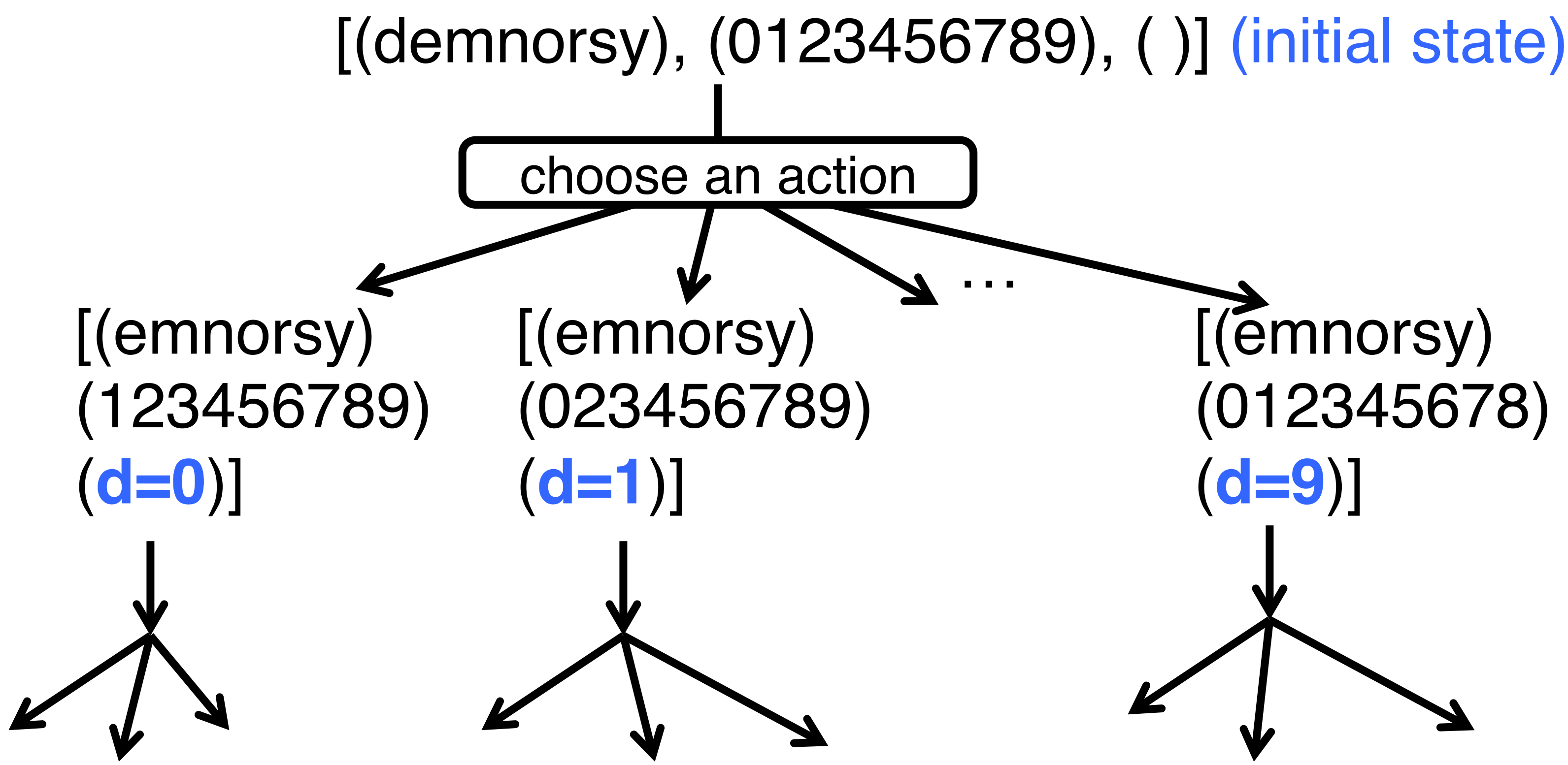
A single path from the initial state to a goal state.

Weighted state space graphs



Search

Search Tree



NB: If the *state space graph* has loops, the *search tree* may be infinite!

Problem solving with search

Initial state and operators *define* search tree

We need a **search algorithm** to *build* (the relevant parts of) the search tree.

NB: In code, build/represent only what is needed

- Do NOT generate the entire search tree
- Do NOT save everything generated
- Generate states incrementally
- Forget anything not needed for future

The size of the search tree

If there are *b* possible actions at each node:
(*b* = **branching factor)**

At depth 1, there are *b* nodes.

At depth 2, there are $b*b = b^2$ nodes.

…

At depth d, there are b^d nodes.

The size of a search tree with **depth *d*** and **branching factor *b*** is $O(b^d)$

Reducing the size of the search tree

What is the branching factor of 8-queens?

If queens can be placed anywhere: b=64
Size of tree: 64x63x…x57 = 178,462,987,637,760

If *n*-th queen is placed in *n*-th row: b= 8
Size of tree: 8^8 = 16,777,216

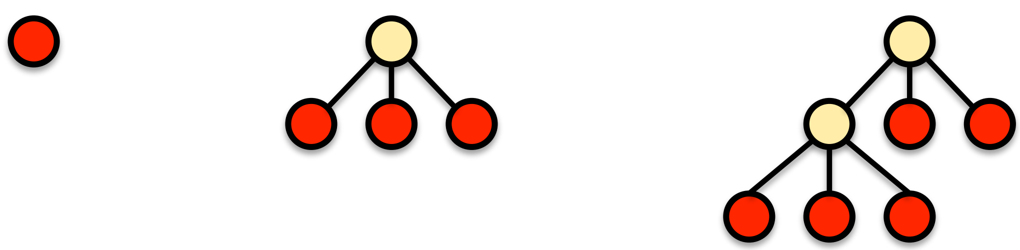
When possible, it can help to impose a specific order on the actions in advance.

Exploring the search tree

Start with the **root** (= the initial state).
It may not be possible to store/build the entire tree.

Leaf nodes (frontier) = unvisited nodes

Visiting a node: test whether it's a goal.
If not, expand it (find all its children).



Representing nodes of the search tree

n.STATE:
the corresponding state in the state space

n.PARENT:
pointer to the parent node in the search tree

n.ACTION:
the action which gets from parent to here

n.PATH-COST:
the total cost from the initial state to here

Expanding leaf nodes

Generating all children of a node in the search tree

```
Expand(Node N):
 Children = new List();
 For every Action a:
   child = apply(a, N)
   if child != null:
      Children.add(child)
 Return Children;
```

Traversing the tree

We need an ordered list of the leaf nodes we have not expanded yet **(= the queue)**
NB: The difference between search algorithms lies in *how they sort the queue*

We may also want a list of the states we have explored already **(= the explored set)**
This allows us to search on the state graph

Generic (tree) search function

```
SEARCH(Problem P, QueuingFunction QF):
  local: n   /*current node*/
         q   /*queue of nodes to explore*/
q ← new List(InitialState(P));
Loop:
    if q == () return failure;
    n ← pop(q);
    if n solves P return n; /*Goal test*/
    q ← QF(q, expand(n));  /*Expansion*/
end
```

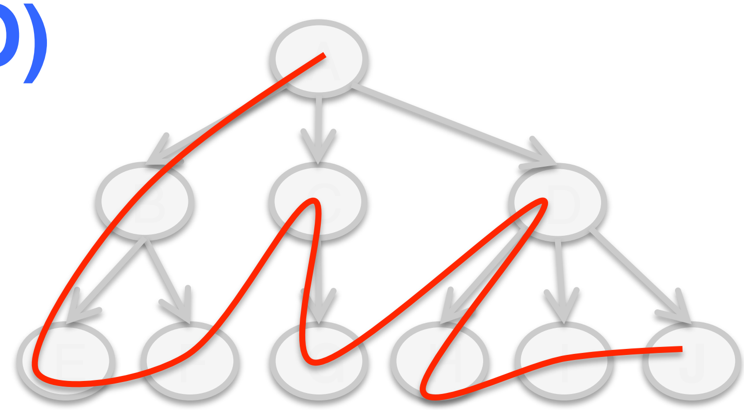
Uninformed (blind) search

The queuing function defines the search order

Depth-first search (LIFO)

Expand deepest node first

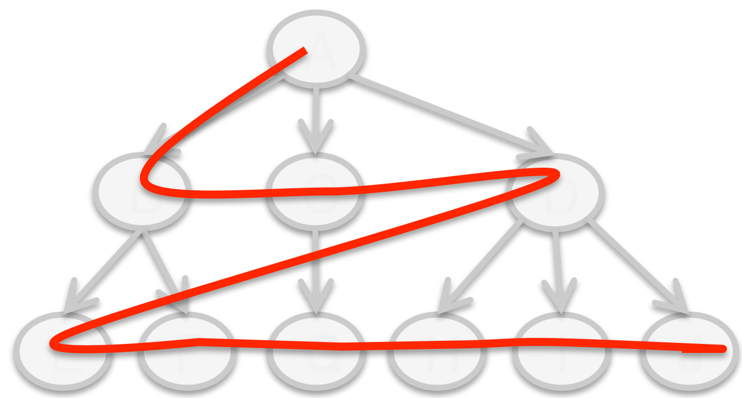
```
QF(old, new):
    Append(new, old)
```



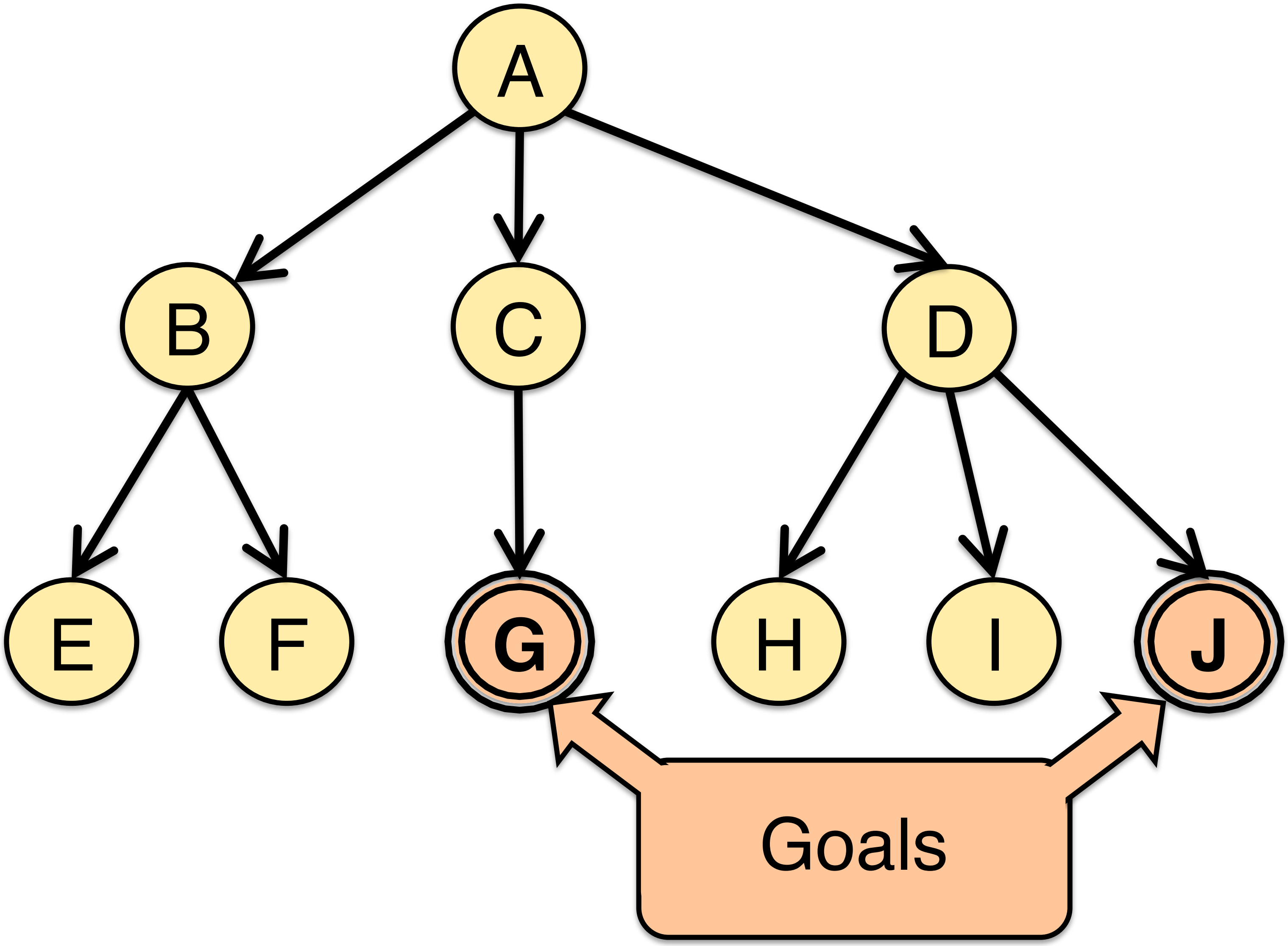
Breadth-first (FIFO)

Expand nodes level by level

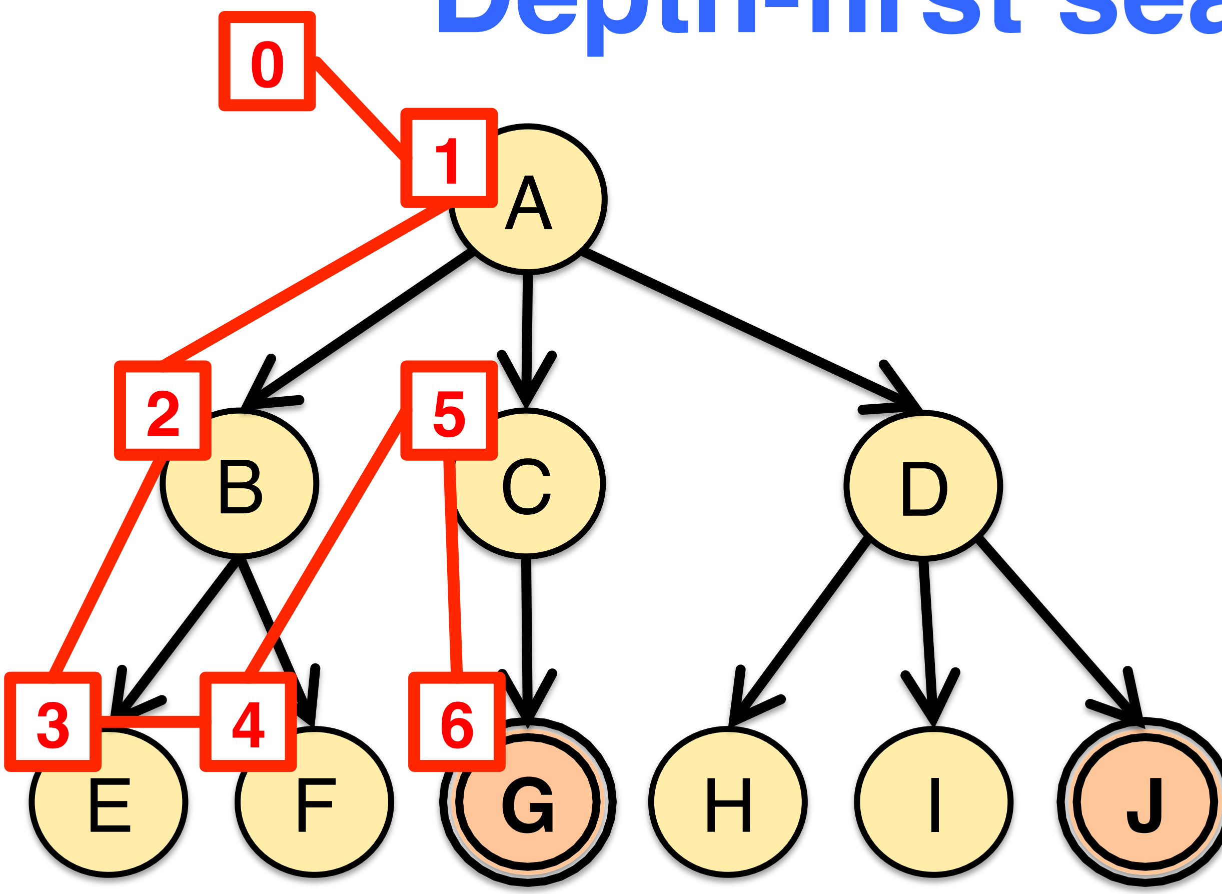
```
QF(old, new):
    Append(old, new);
```



Sample Search Tree

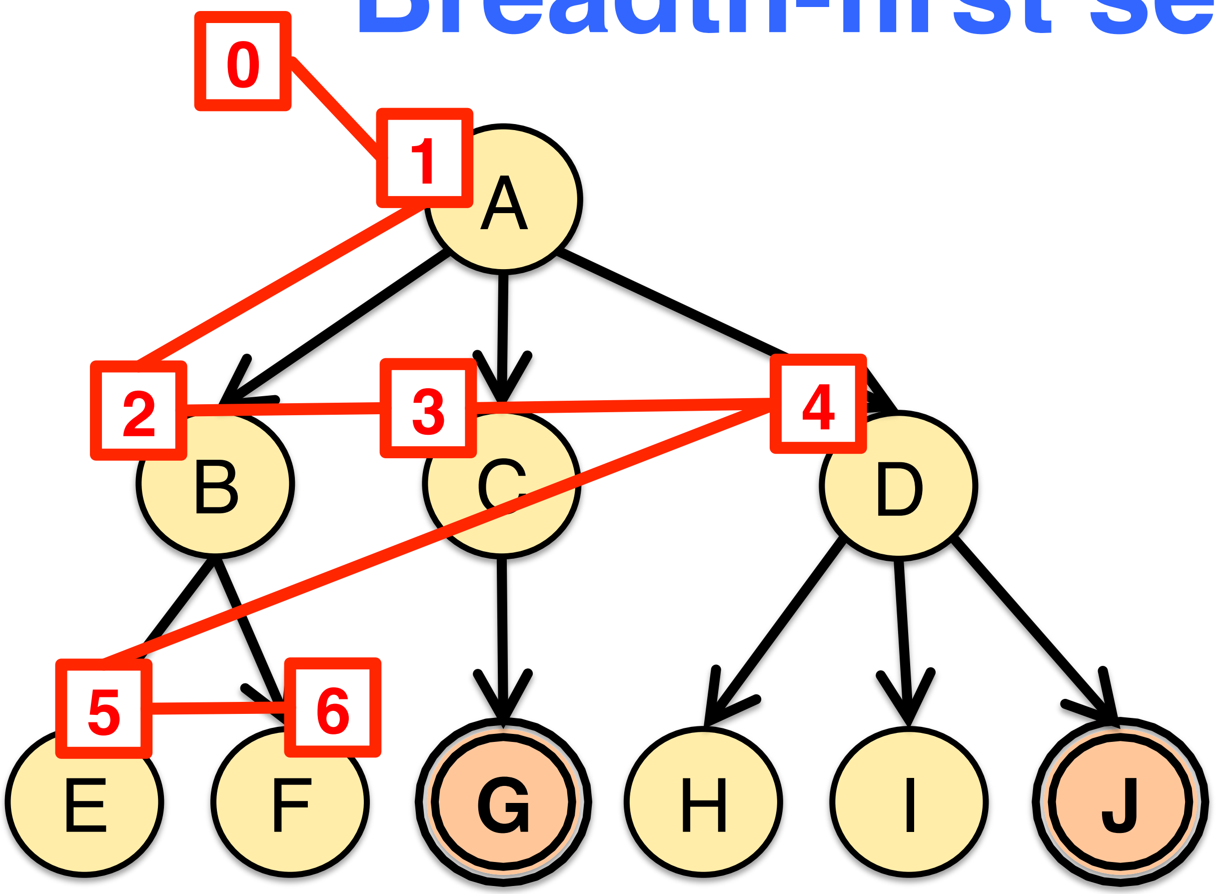


Depth-first search



	node current	*queue afterwards*
0.	-	(**A**)
1.	***A***	(**B C D**)
2.	***B***	(**E F** C D)
3.	***E***	(F C D)
4.	***F***	(C D)
5.	***C***	(**G** D)
6.	***G***	(D)

Breadth-first search



node current	*queue afterwards*
-	(**A**)
1. A	(**B C D**)
2. B	(C D **E F**)
3. C	(D E F **G**)
4. D	(E F G **H I J**)
5. E	(F G H I J)

…

Graph search

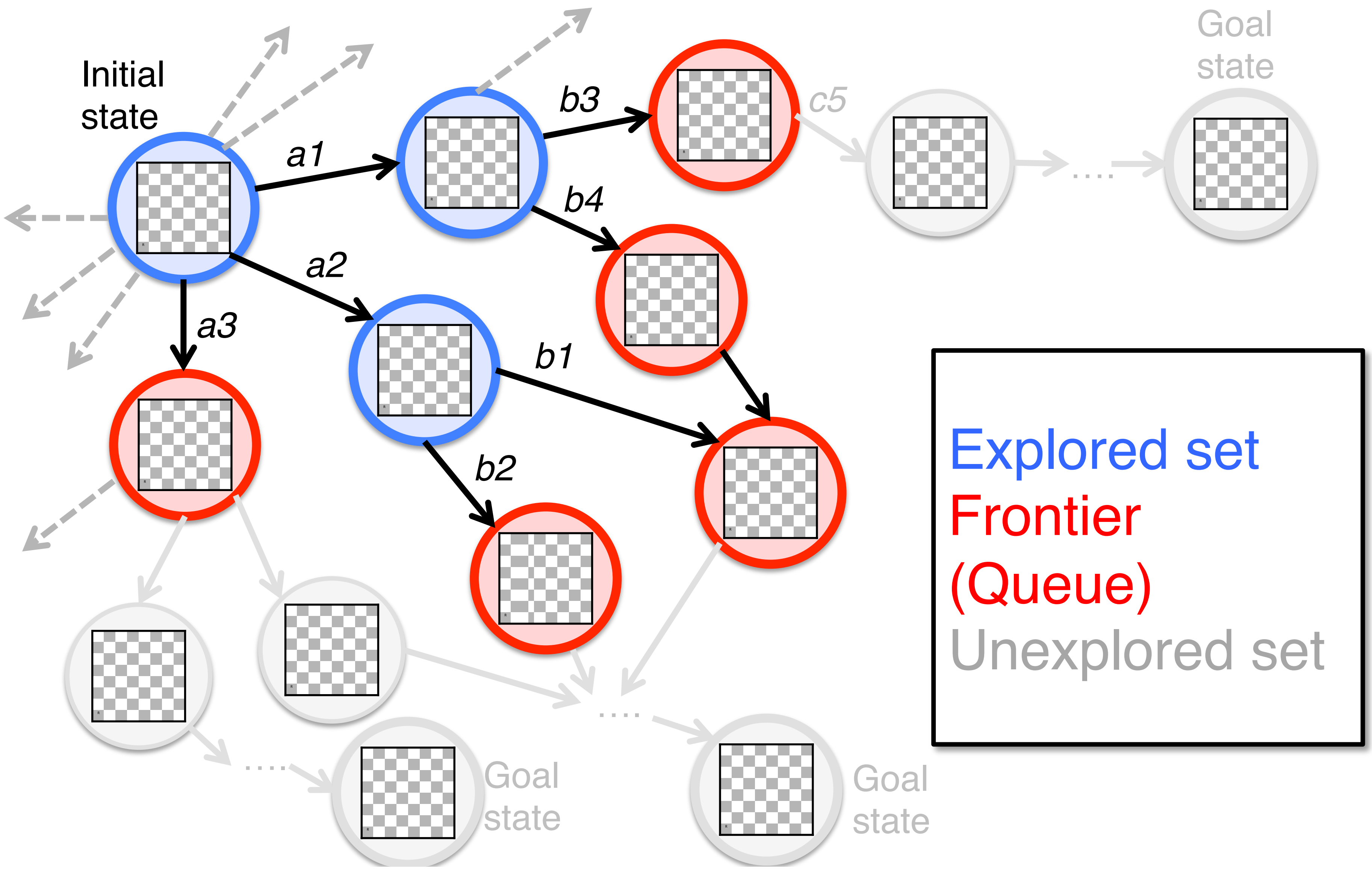
The size of the search tree

Each node in the search tree corresponds to a single sequence of actions from the root.

If a state in the state space can be reached through *n* different sequences of actions, it corresponds to *n* nodes in the search tree.

If the state space graph has loops, the search tree may be infinite!

Graph search



Graph search

```
SEARCH(Problem P, Queuing Function QF):
  local: n, q,  e; /* e= explored nodes */
  e ← new List();
  q ← new List(Initial_State(P));
  Loop:
    if q == () return failure;
    n ← pop(q);
    if n solves P return n;
    add n.STATE to e;
    for m in Expand(n):
       if m not in e or q: q ← QF(q, m);
end
```

Comparing search algorithms

Complexity of search algorithms

Time complexity: How long does it take to find a solution?

Space complexity: How much memory does it take to find a solution?

Properties of search algorithms

A search algorithm is **complete**
if it will find any goal whenever one exists.

A search algorithm is **optimal**
if it will find the cheapest goal.

Time complexity: how long does it take to find a solution?

Space complexity: how much memory does it take to find a solution?

Breadth-first search

Breadth-first search is **complete**, but only **optimal** if each action has the same cost
(it will return the shortest [shallowest] solution)

Time complexity: $O(b^d)$
If the shallowest goal is at depth *d*, breadth-first will visit all nodes up to depth *d*.

Space complexity: $O(b^d)$
The queue is of size $O(b^d)$

Tree-search DFS

m = maximal depth of search tree.

Only **complete** if *m* is finite.

(may try to wander down infinite branches)

DFS is **not optimal.**

Time complexity: $O(b^m)$

(may need to visit all nodes in search tree)

Space complexity: $O(bm)$

(only stores one branch of the search tree)

Graph-search DFS

Complete if the search space is finite.
DFS is **not optimal.**

Time complexity: bounded by size of search space.

Space complexity: bounded by size of search space.

To conclude…

Today's key concepts

Problem solving as search:

Solution = a finite sequence of actions

State graphs and search trees

Which one is bigger/better to search?

Systematic (blind) search algorithms

Breadth-first vs. depth-first; properties?

Your tasks

- **Reading**: Ch. 3.1-3.4 (only relevant parts)
- **Compass quiz**: Up at 2pm