# Uninformed Search

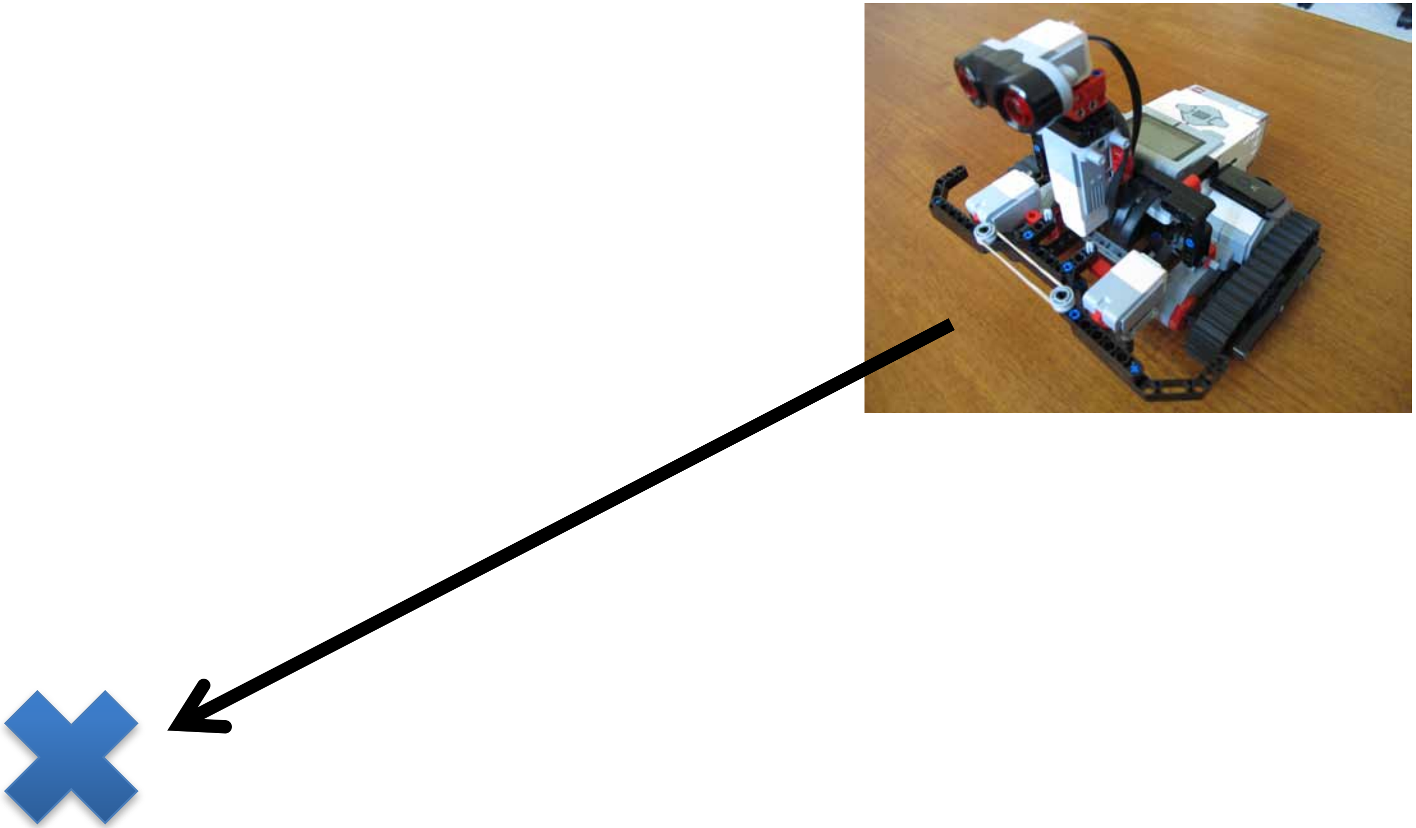
Dr. Demetrios Glinos
University of Central Florida

CAP4630 – Artificial Intelligence

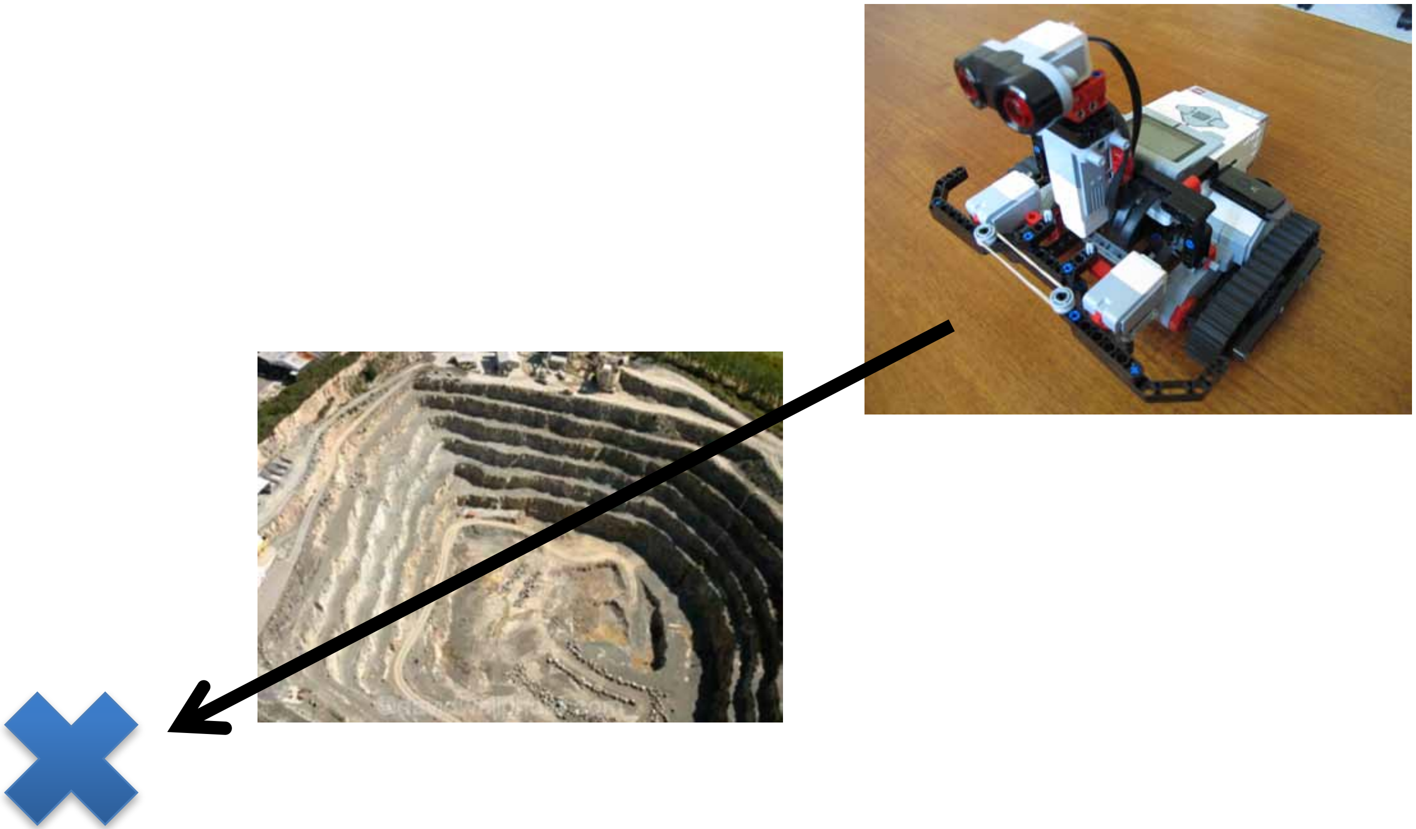
# Today

- Agents that Plan
- Search Problems
- State Space Graphs and Search Trees
- Uninformed Search Methods
  - Depth-First Search
  - Breadth-First Search
  - Uniform-Cost Search
- Searching Pac-Man Mazes

# Why Plan?

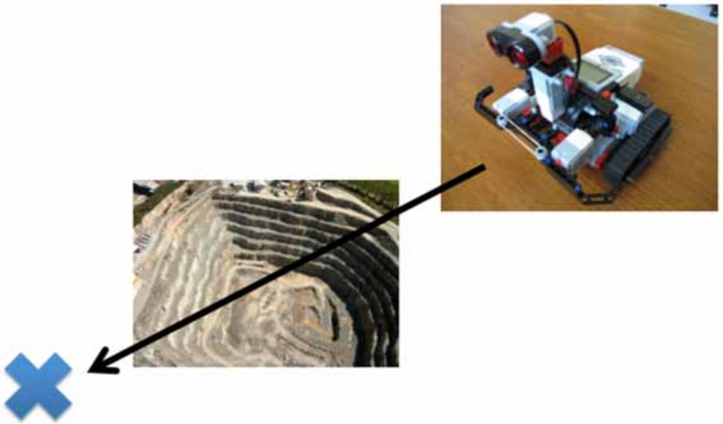


# Why Plan?



# Reflex Agents

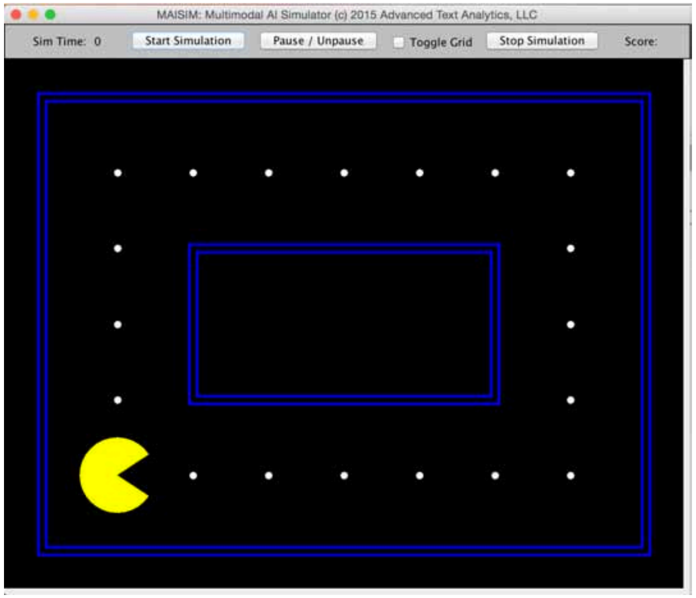
- Reflex agents base decisions on:
  - Current percepts
  - [ optionally ] What they currently know
    - Memory of past
    - Model of the current world state



  - Here, if current percept includes the pit, the agent may have a workaround

- *Reflex Agents do not consider the future consequences of their actions*
- *They look only at "what is", not "what might be"*

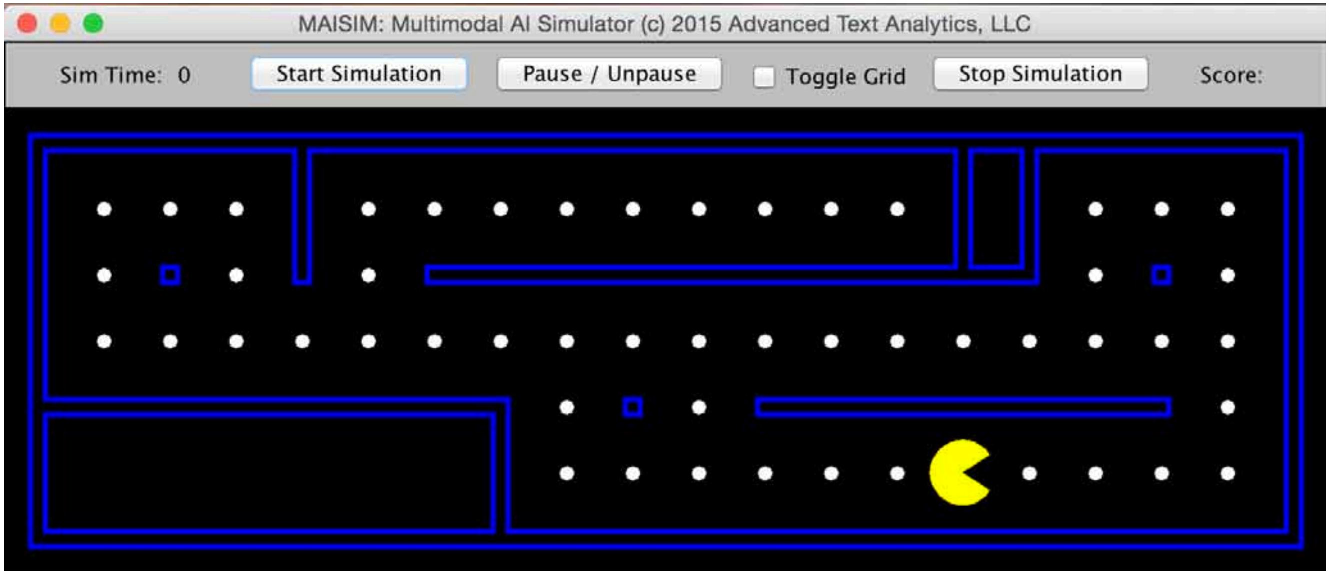
# Can a Reflex Agent be Rational?




*Q1: Is this rational behavior?*
*Q2: Could a genius do better?*

*demo: circuit*

# Can a Reflex Agent be Rational?

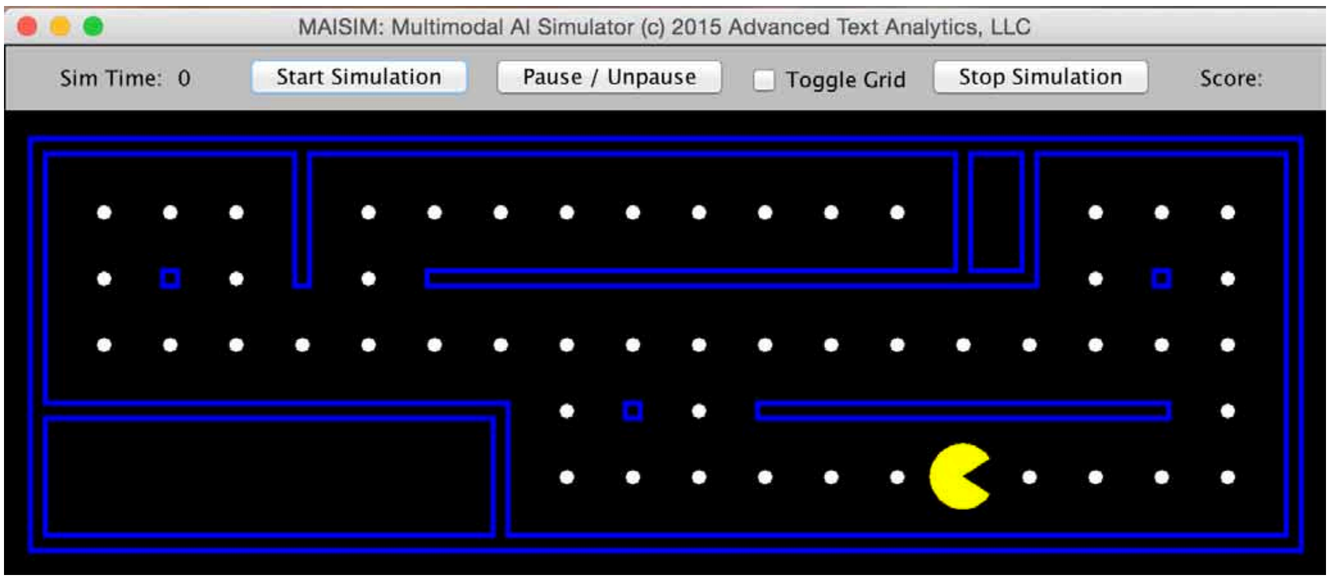



*Q1: What happened?*

*Q2: Is this rational behavior?*

*demo: simple*

# A More Complex Reflex Agent

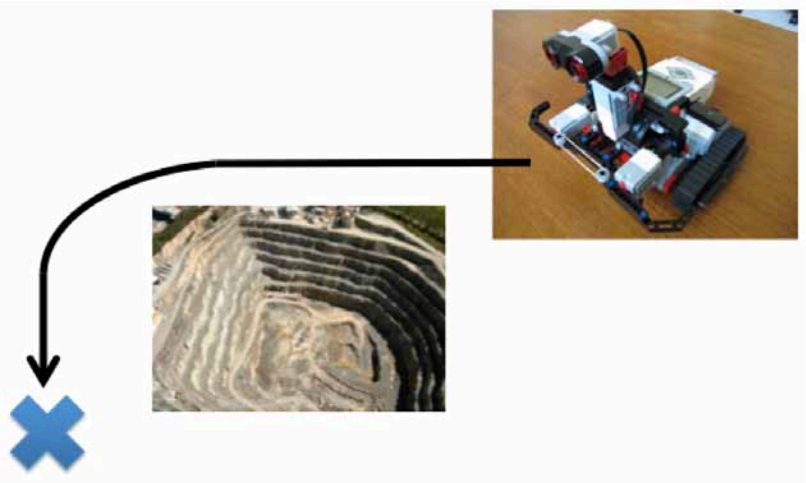



*demo: reflex*

*Note: This agent is also competitive on mazes with ghosts*

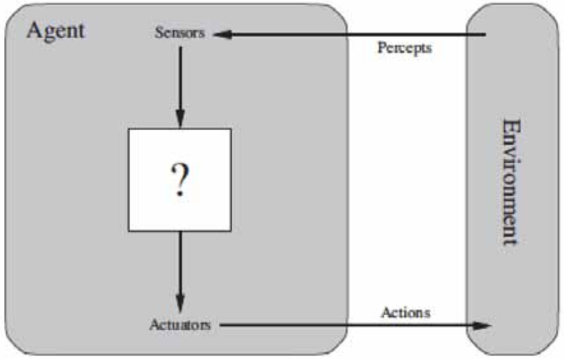
# Planning Agents

- Planning agents base decisions on:
  - Hypothesized consequences of their actions (*"what might be"*)
  - plus current percepts, memory, etc.



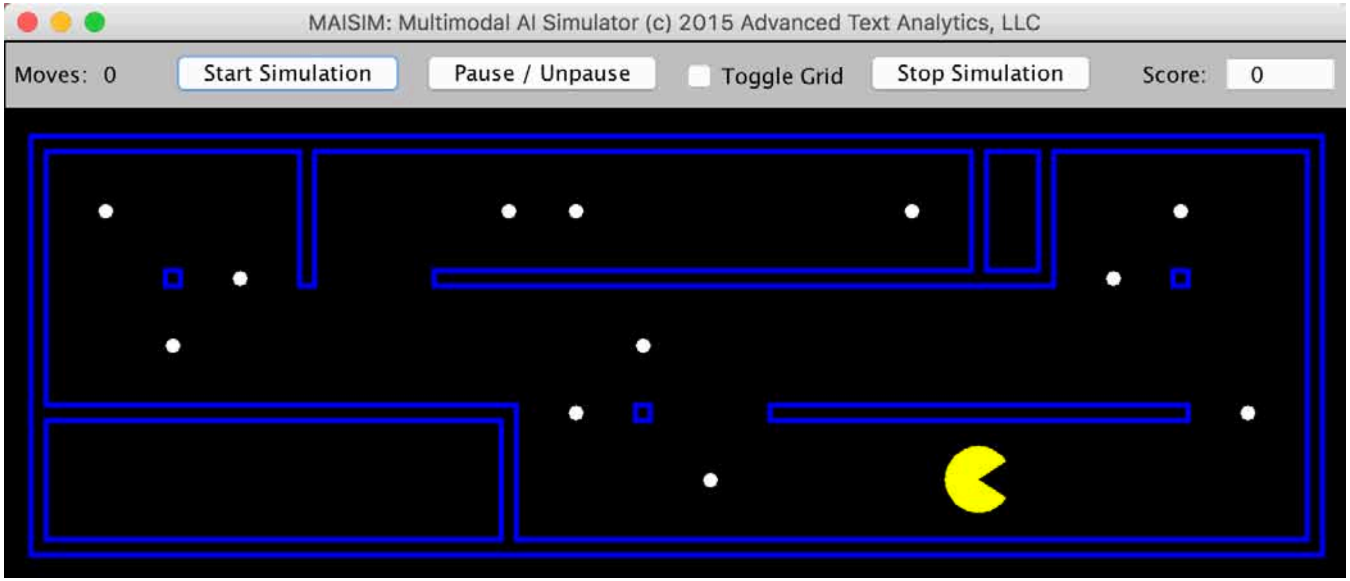
→ Must have a **model** of how the agent believes the world *responds* to its actions

→ Must have a **goal**




- Plans can be *complete* or *partial*
- Complete plans can be *optimal* or *suboptimal*
- Agent can be a *planning* or *replanning* agent

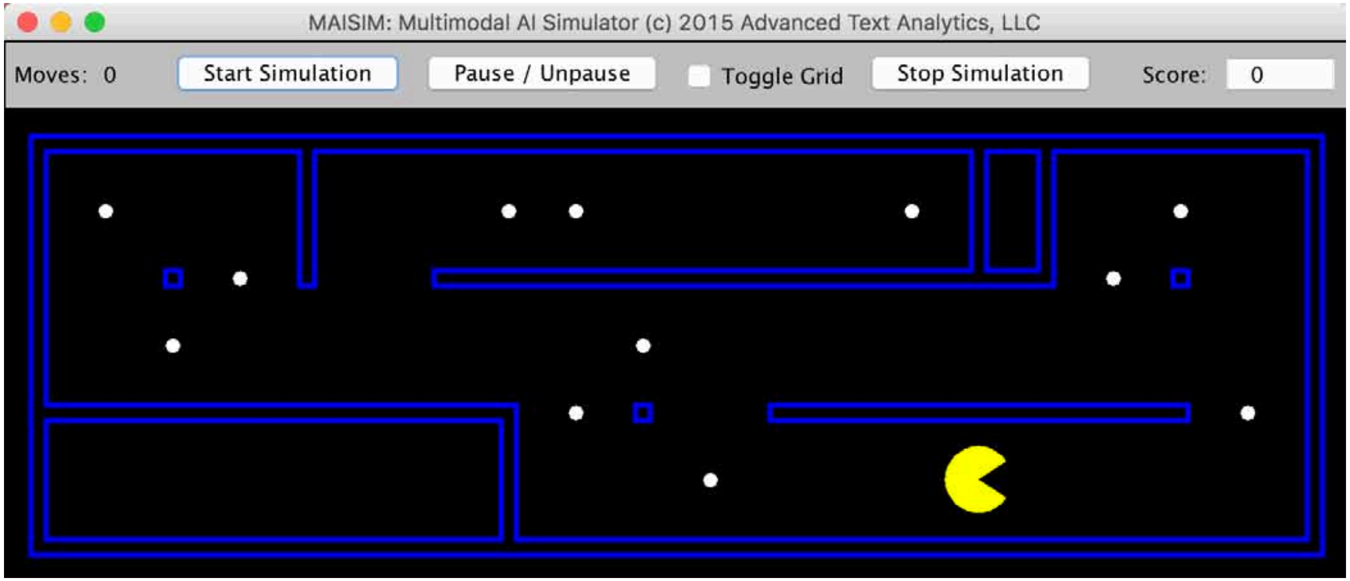
# Replanning Agent




*Q1: What is this agent's planning strategy?*
*Q2: Is the sum of the mini-plans complete?*
*Q3: Is the sum of the mini-plans optimal?*

*demo: replan*

# Optimal Planning Agent

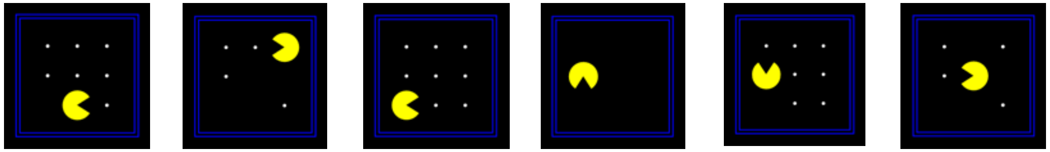



*Q1: Is this agent's plan complete?*
*Q2: Is it optimal?*

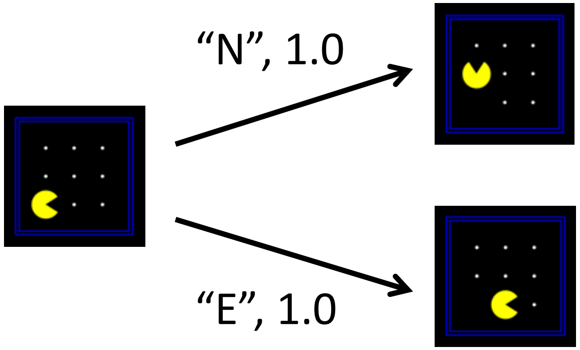
*demo: optimal*

# Search Problems

- A search problem consists of:
  - A state space

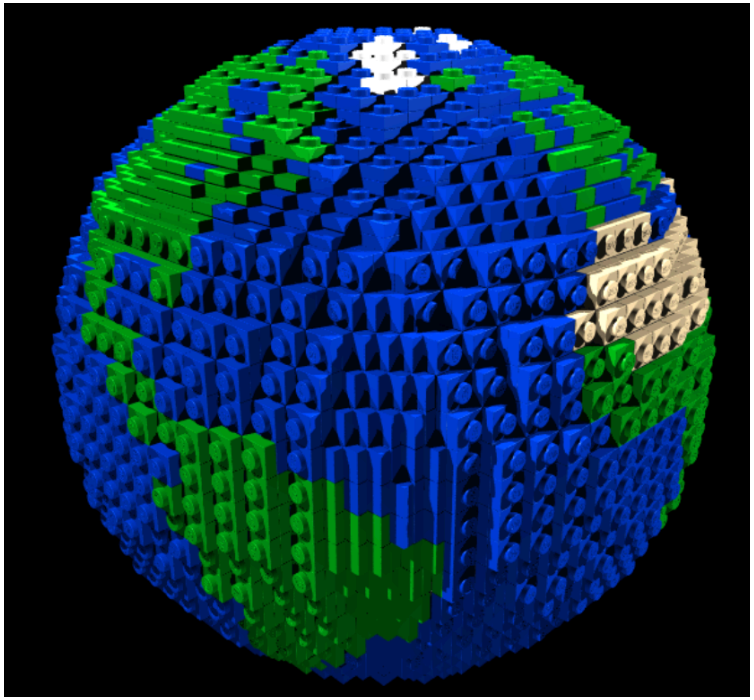


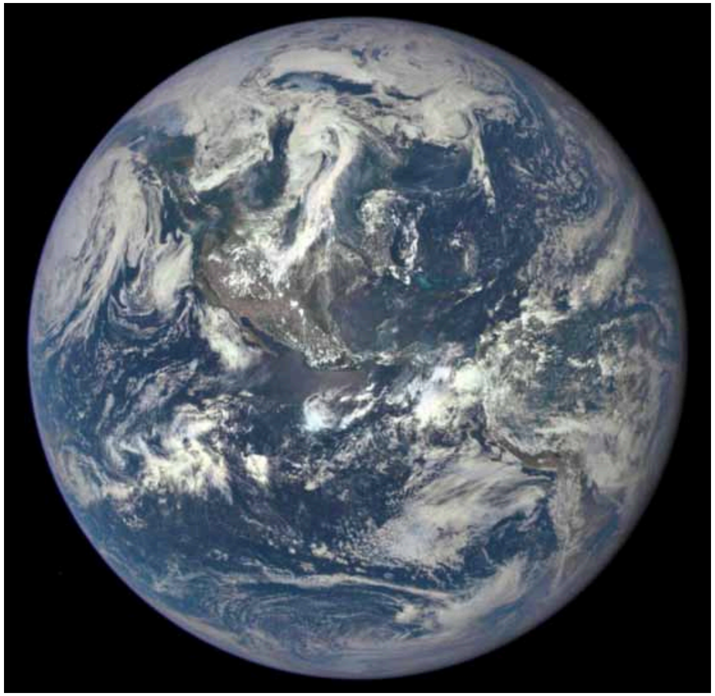
  - A successor function (with actions, costs)



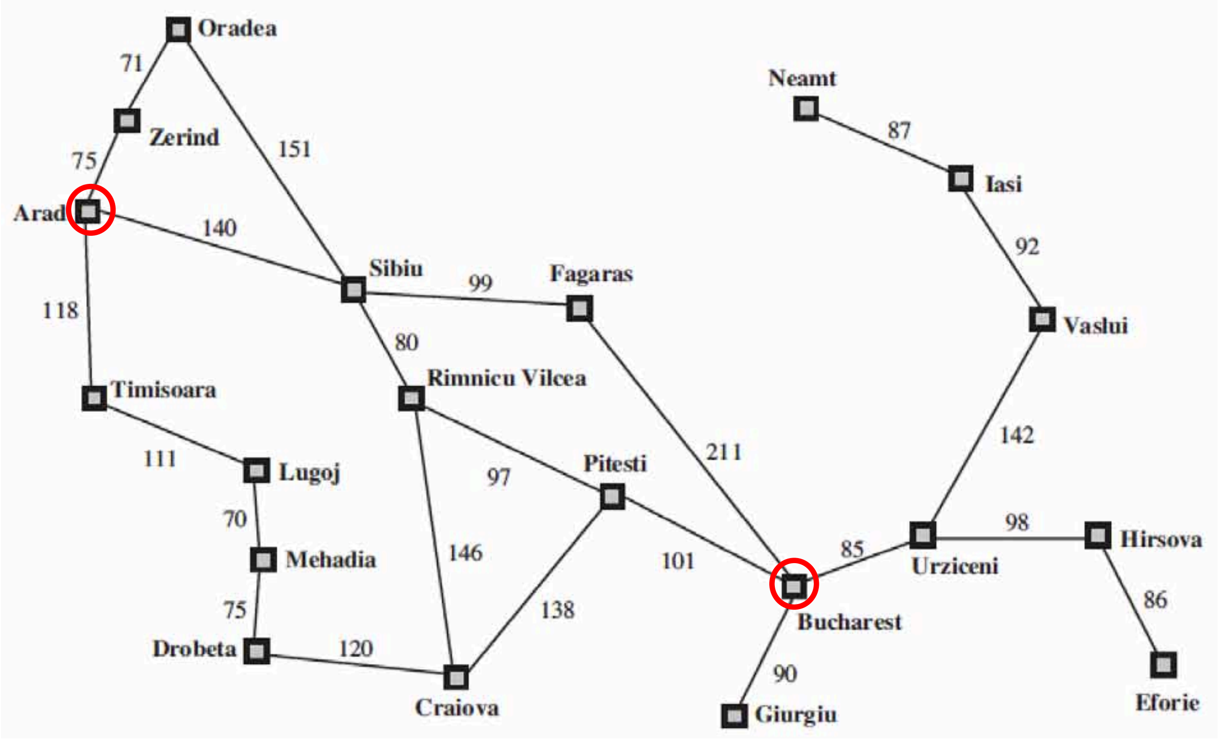

  - Start state
  - Goal test
- A solution is a *sequence of actions (the plan)* that tranforms the start state into a goal state

# Search Problems Involve Models





# Example Search Problem

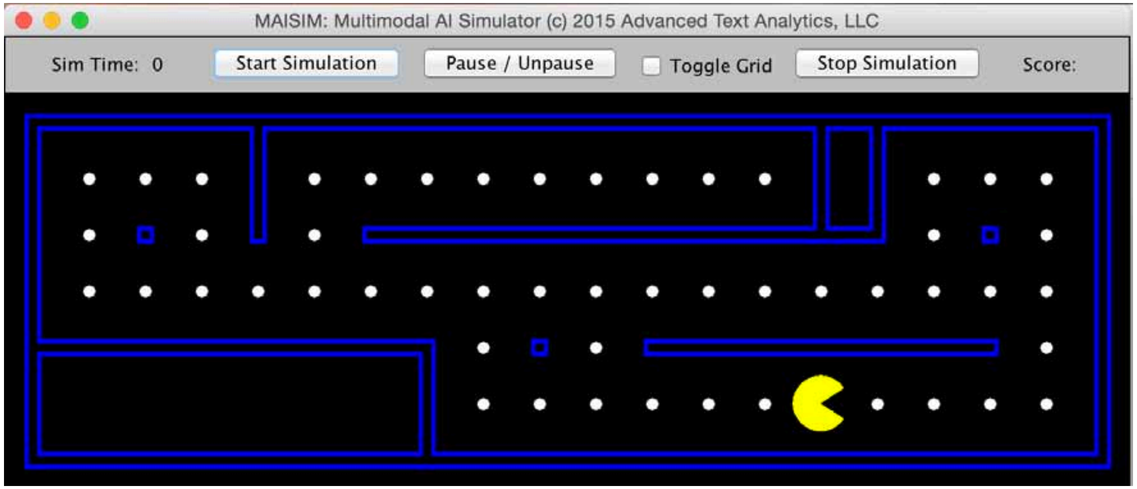



- The Search Problem: Traveling from Arad to Bucharest in Romania
- State space:
  - Cities
- Successor function:
  - Roads connections, with distances
- Start State:
  - Arad
- Goal test:
  - Are we in Bucharest?

# What to Include in a State Space

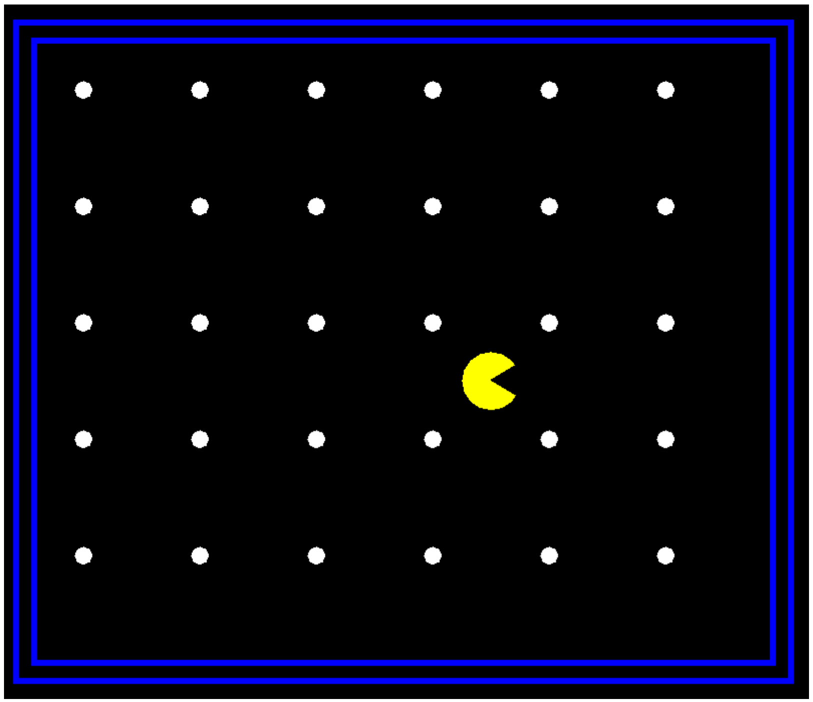
The world state includes every detail of the environment

A search state includes only the details needed for planning




- Problem: Pathing
  - States: (x,y) locations
  - Actions: NSEW
  - Successor function: update location only
  - Goal test: is (x,y) = GOAL ?
- Problem: Eat-All-Food
  - States: [ (x,y), booleans for food dots ]
  - Actions: NSEW
  - Successor function: update location and maybe also a food dot boolean
  - Goal test: all food dot booleans are false

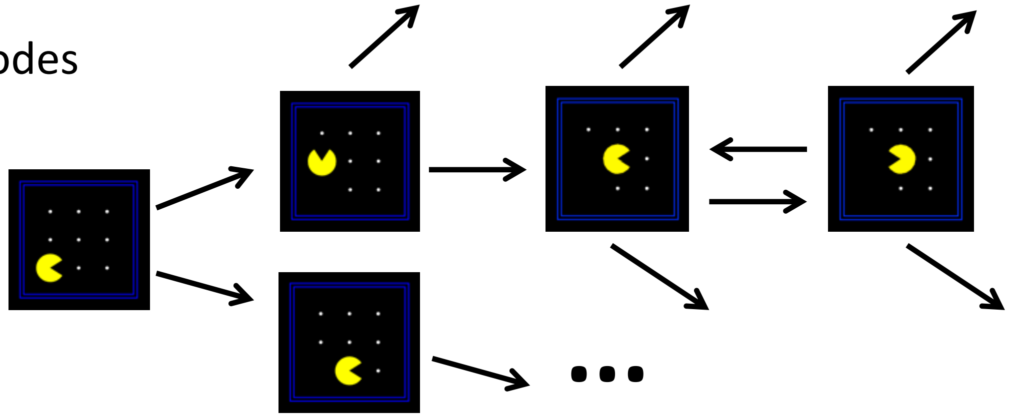
# How Large Can the State Space Be?



- The World:
  - 120 agent positions
  - 30 food pellets (dots)
  - 4 agent orientations (NSEW)
- Counting states:
  - World states: $(120)(2^{30})(4)$
  - States for Pathing: 120
  - States for eat-all-dots: $(120)(2^{30})$

# State Space Graphs

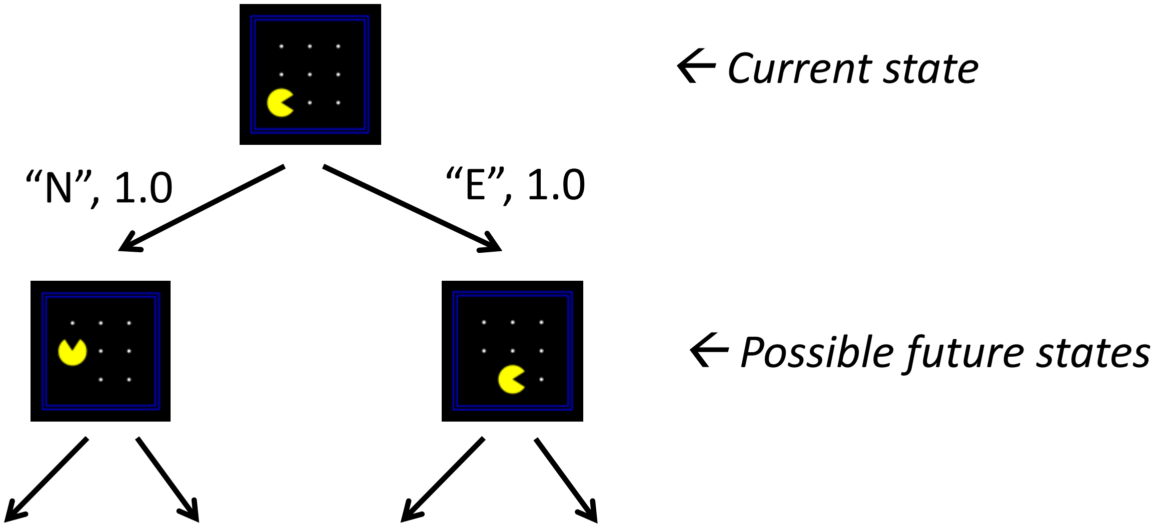
- We can use a **state space graph** to represent a search problem
  - Nodes are world states
  - Arcs show successors
  - Goal test is a set of nodes



- Each state appears only once in a state space graph
- Generally impractical to construct for real world problems

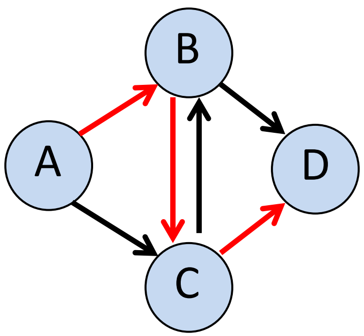
# Search Trees

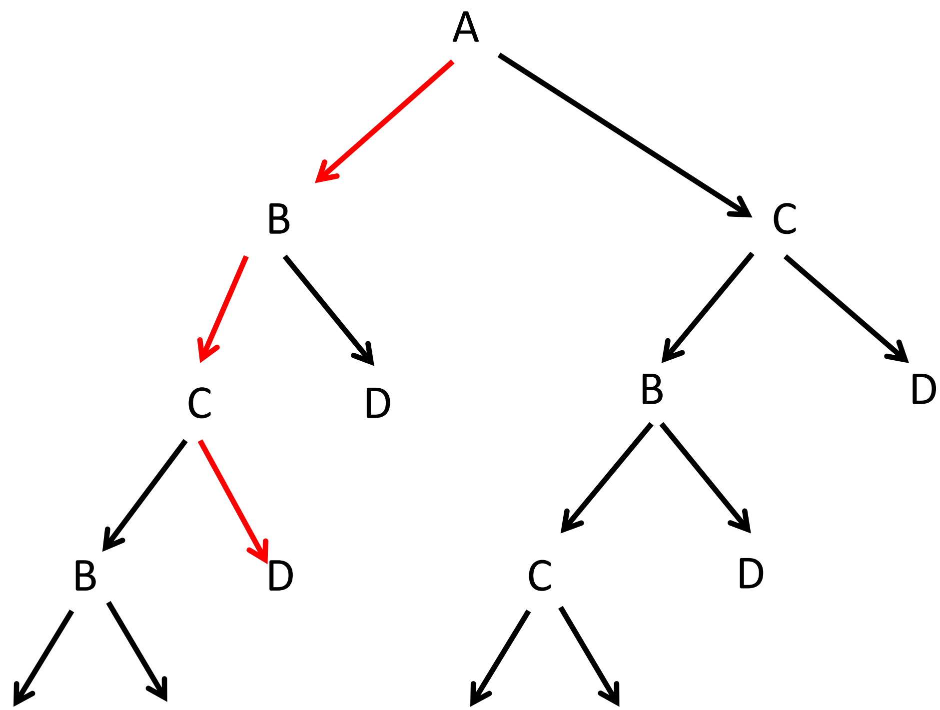
- A search tree represents possible actions and their outcomes (successor states)
- We use search trees to represent possible **plans** of action




- We can construct a tree from the corresponding graph
- Again, generally impractical to construct for real world problems

# Paths From Root are Plans



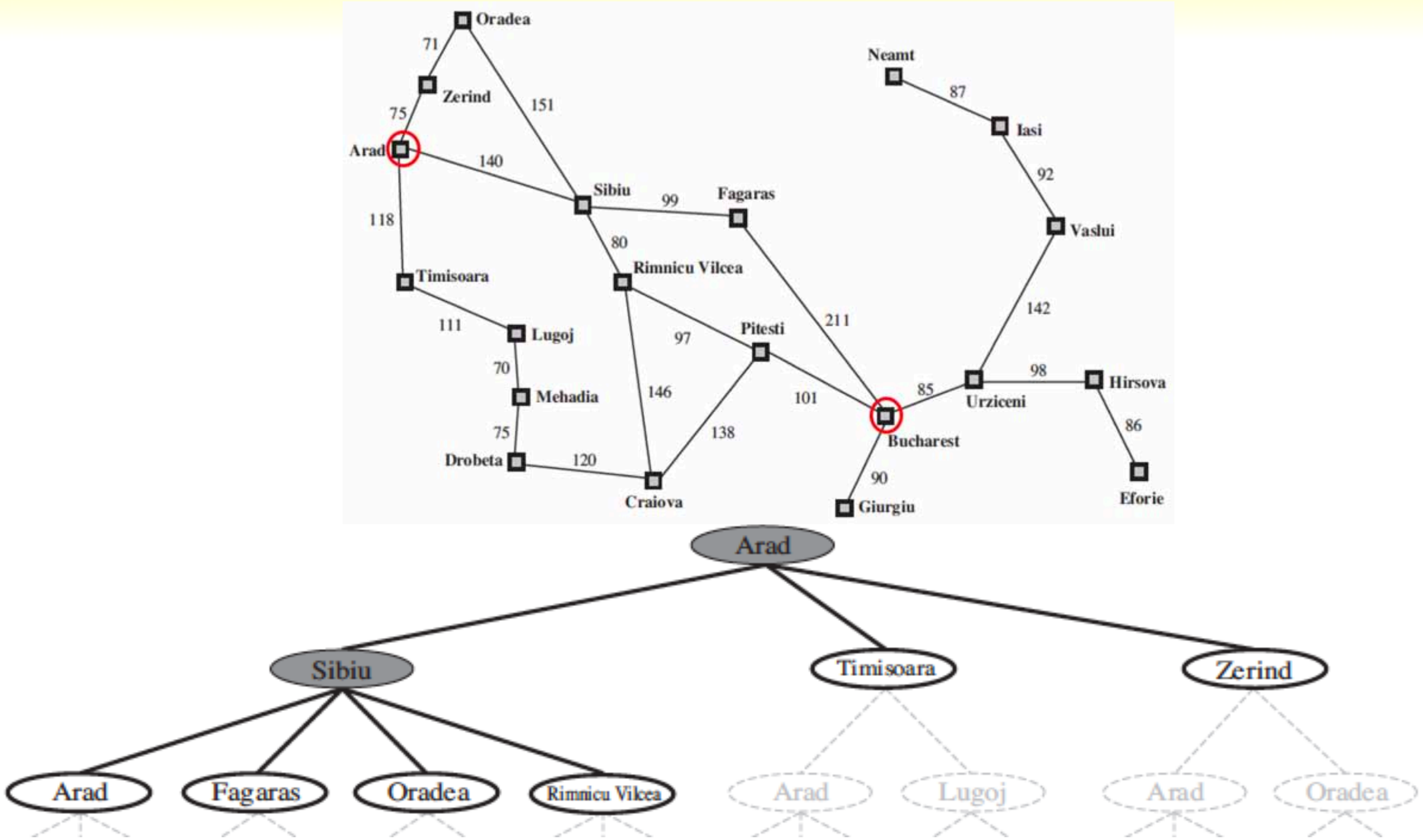

Nodes in tree show states, but represent plans from A (start) to D (goal)

Plans along red route:

( A )
( A, B )
( A, B, C )
( A, B, C, D )

Q: How large is this tree?

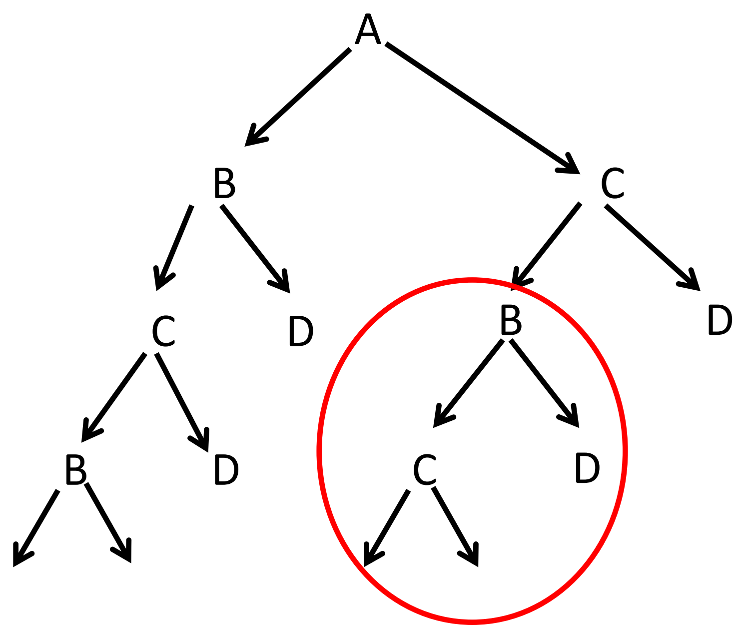
# Search Tree from Graph




- This search tree was constructed from the graph
- Arad appears 4 times in this tree
- Once we have been in Arad, there is nothing to be gained by visiting there again

# Tree Search Methods

- Tree search methods are all about
  - Building only what you need
  - Avoiding duplication by not revisiting states we have visited before



# General Tree Search Algorithm

```
function TREE-SEARCH( problem, strategy ) returns a solution, or failure
    initialize the fringe ("frontier") using the initial state of problem
    loop do
        if the fringe is empty, then return failure
        choose a node according to strategy and remove it from the fringe
        if the node contains a goal state then return the corresponding solution
        else
            expand the node, adding the resulting nodes to the fringe
    end
```

Note: This algorithm revisits previously visited nodes

# General Graph Search Algorithm

```
function GRAPH-SEARCH( problem, strategy ) returns a solution, or failure
    initialize the fringe using the initial state of problem and explored set to empty
    loop do
        if the fringe is empty, then return failure
        choose a node according to strategy and remove it from the fringe
        if the node contains a goal state then return the corresponding solution
        else
            add the node to the explored set,
            expand the node, adding the resulting nodes to the fringe, but only if
            they are not already on the fringe or the explored set
    end
```

*Notes:*

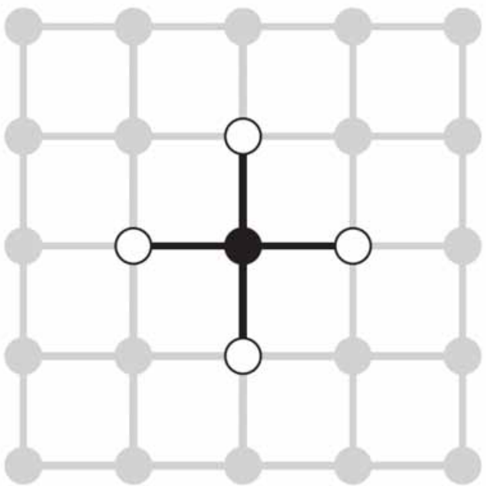
1. *The explored set is also sometimes called the "closed set"*
2. *Russell & Norvig call this algorithm "graph search" because it uses the explored set to avoid visiting a node more than once*

# Importance of Explored Set

- *"Those who cannot remember the past are condemned to repeat it."*
  -- George Santayana (1863-1952), philosopher, essayist, poet, and novelist

- Failure to recognize and ignore previously visited nodes (states) can cause a tractable problem to become intractable

- Consider a rectangular-grid problem (similar to Pac-Man, but without walls):

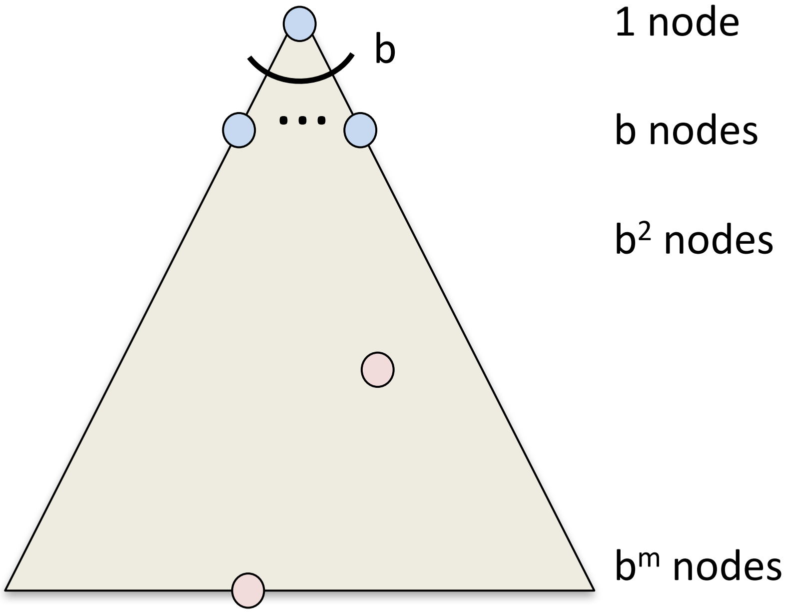


- Each node has 4 neighbors, except at borders
- Suppose we have a 43 x 43 grid and start in center
- A depth d search that includes repetition has $4^d$ nodes
  - for d = 20, this is over 1 trillion nodes
- But there are only $2d^2$ distinct states reachable in d steps
  - for d = 20, this is only 800 distinct nodes

# Search Algorithm Properties

Key properties of all search algorithms

- Complete: Is it guaranteed to find a solution if one exists?
- Optimal: Is it guaranteed to find the lowest cost solution?
- Time complexity?
- Space complexity?




Number of nodes in entire tree: $1 + b + b^2 + \ldots + b^m = O(b^m)$

where: b is the *branching factor*
m is the maximum depth

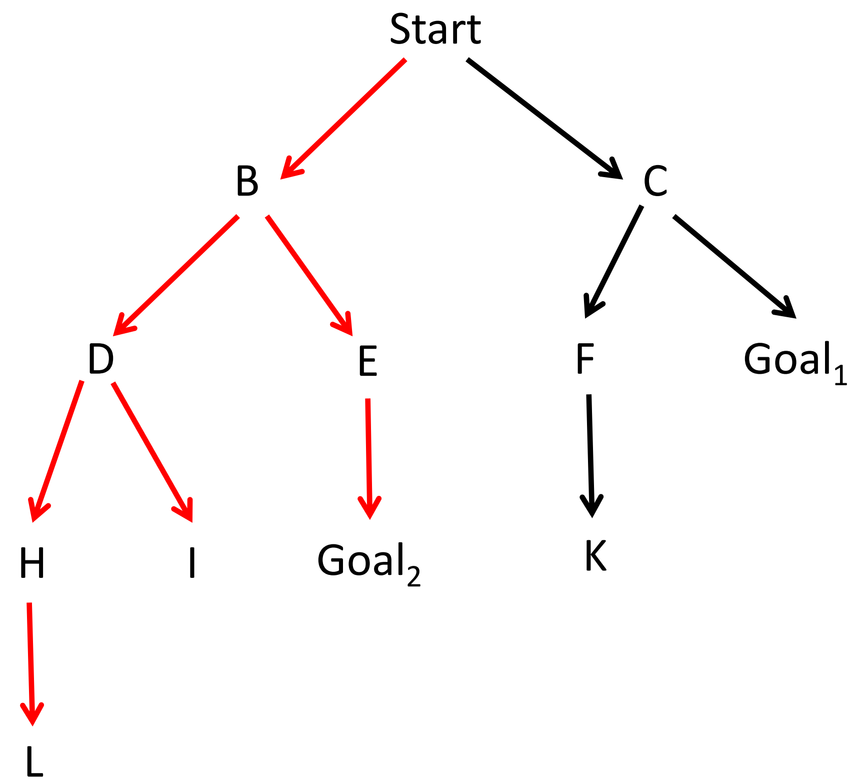
*Note: there can be multiple solutions at varying depths*

# Depth-First Search

**Strategy:** expand the deepest node first

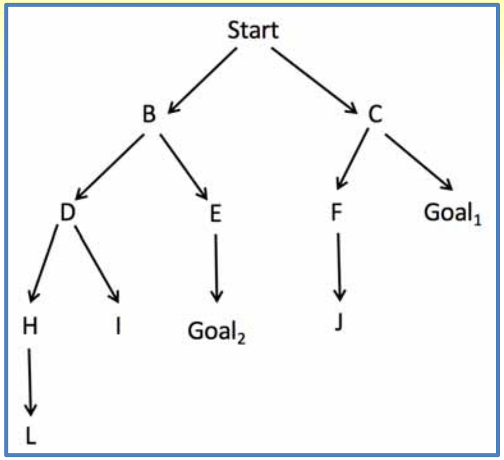
**Implementation:**

fringe is a LIFO (stack)




Assuming we add paths to fringe in reverse alphabetic order of the last state identifier, DFS will perform a left-first search (shown in red) to find a goal

# DFS Example 1



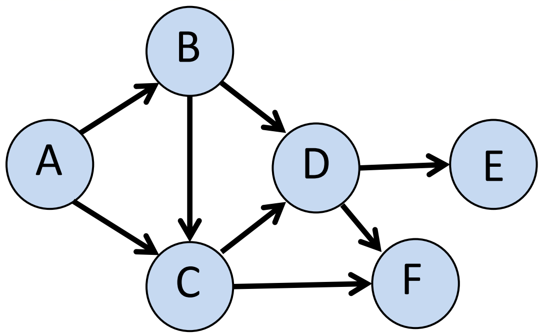

- Given graph at right
- Assume we desire a path from Start to any Goal
- Assume we add nodes in reverse alphabetic order
- Use DFS to find a solution path
- Show fringe and explored set at each step

| State/Action | Fringe contents (paths) | Explored set (states) |
|---|---|---|
| Initial state: | (Start) | { } |
| Expand (Start) | (Start,C) (Start,B) | Start |
| Expand (Start,B) | (Start,C) (Start,B,E) (Start,B,D) | Start, B |
| Expand (Start,B,D) | (Start,C) (Start,B,E) (Start,B,D,I) (Start,B,D,H) | Start, B, D |
| Expand (Start,B,D,H) | (Start,C) (Start,B,E) (Start,B,D,I) (Start,B,D,H,L) | Start, B, D, H |
| Expand (Start,B,D,H,L) | (Start,C) (Start,B,E) (Start,B,D,I) | Start, B, D, H, L |
| Expand (Start,B,D,I) | (Start,C) (Start,B,E) | Start, B, D, H, L , I |
| Expand (Start,B,E) | (Start,C) (Start,B,E,Goal$_2$) | Start, B, D, H, L , I, E, Goal$_2$ |
| Expand (Start,B,E,Goal$_2$) | Goal state reached | |

*Note: We use parentheses to indicate a comma-separated path of states*

# DFS Example 2

- Given graph at right
- Assume start at A and goal is F
- Assume we add nodes in reverse alphabetic order
- Use DFS to find solution path
- Show fringe and explored set at each step




| State/Action | Fringe contents (paths) | Explored set (states) |
|---|---|---|
| Initial state: | (A) | { } |
| Expand (A) | (A,C) (A,B) | A |
| Expand (A,B)* | (A,C) (A,B,D) | A, B |
| Expand (A,B,D) | (A,C) (A,B,D,F) (A,B,D,E) | A, B, D |
| Expand (A,B,D,E) | (A,C) (A,B,D,F) | A, B, D, E |
| Expand (A,B,D,F) | Goal reached -- final path is (A,B,D,F) | |

Notes: (1) Path (A,B,C) ignored at * since C already reached by (A,C)

(2) We "backtracked" from dead end at E

(3) Also, recall that DFS operates as a LIFO structure (i.e., a stack)

# Depth-First Search (DFS) Properties

Space complexity: O( bm )

- Look at size of fringe
- Solution path (size m) + unexpanded siblings of nodes along path (bm)

Time complexity: O( $b^m$ )

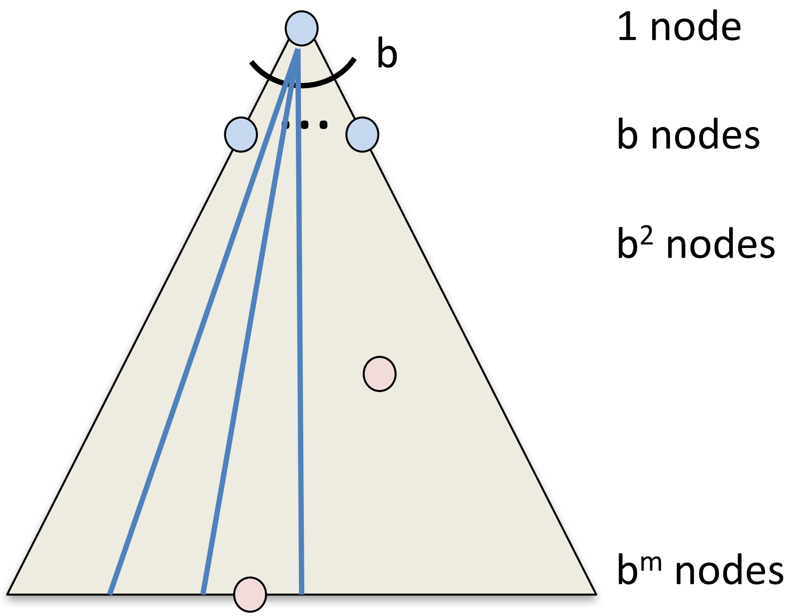
- Could process entire tree

Completeness:

- Only if m is finite (i.e., if we prevent cycles

Optimality:

- No. DFS finds "leftmost" solution



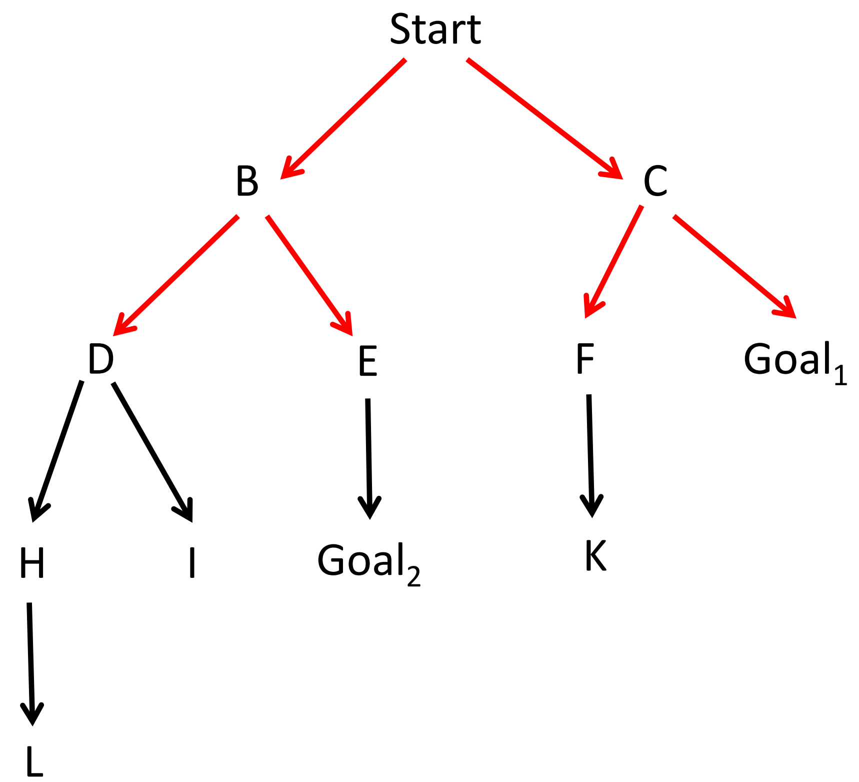

*DFS expands some left prefix of the tree*

# Breadth-First Search

Strategy: expand the shallowest node first

Implementation:

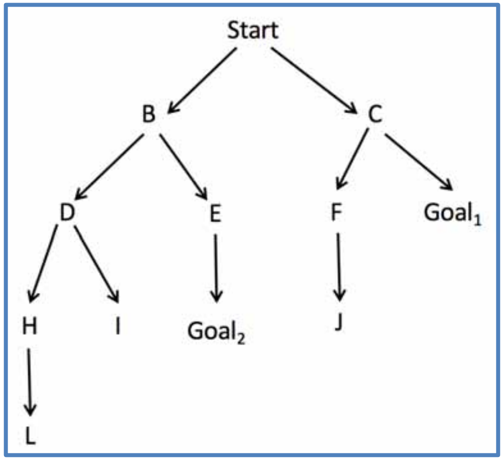
fringe is a FIFO (queue)




Assuming we add paths to fringe in reverse alphabetic order of the last state identifier, BFS will search the part of the tree shown in red to find a goal

# BFS Example 1

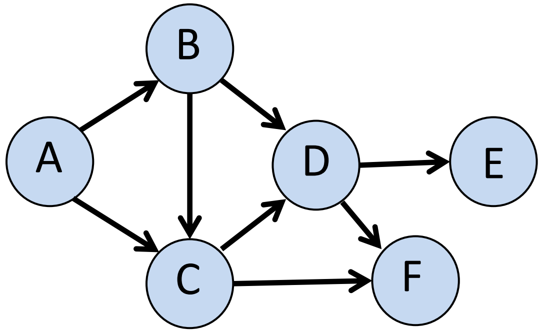
- Given graph at right
- Assume we desire a path from Start to any Goal
- Assume we add nodes in reverse alphabetic order
- Use BFS to find a solution path
- Show fringe and explored set at each step




| State/Action | Fringe contents (paths) | Explored set (states) |
|---|---|---|
| Initial state: | (Start) | { } |
| Expand (Start) | (Start,C) (Start,B) | Start |
| Expand (Start,C) | (Start,B) (Start,C,Goal$_1$) (Start,C,F) | Start, C |
| Expand (Start,B) | (Start,C,Goal$_1$) (Start,C,F) (Start,B,E) (Start,B,D) | Start, C, B |
| Expand (Start, C,Goal$_1$) | Goal state reached | |

# BFS Example 2

- Given graph at right
- Assume start at A and goal is F
- Assume we add nodes in reverse alphabetic order
- Use BFS to find solution path
- Show fringe and explored set at each step




| State/Action | Fringe contents (paths) | Explored set (states) |
|---|---|---|
| Initial state: | (A) | { } |
| Expand (A) | (A,C) (A,B) | A |
| Expand (A,C) | (A,B) (A,C,F) (A,C,D) | A, C |
| Expand (A,B)* | (A,C,F) (A,C,D) (A,B,D) | A, C, B |
| Expand (A,C,F) | Goal reached -- final path is (A,C,F) | |

Notes: (1) Path (A,B,C) ignored at * since C already in explored set

(2) Also, recall that BFS operates as a FIFO structure (i.e., a queue)

# Breadth-First Search (BFS) Properties

**Space complexity: $O( b^s )$**

- Dominated by last layer searched
- s = depth of shallowest solution

**Time complexity: $O( b^s )$**

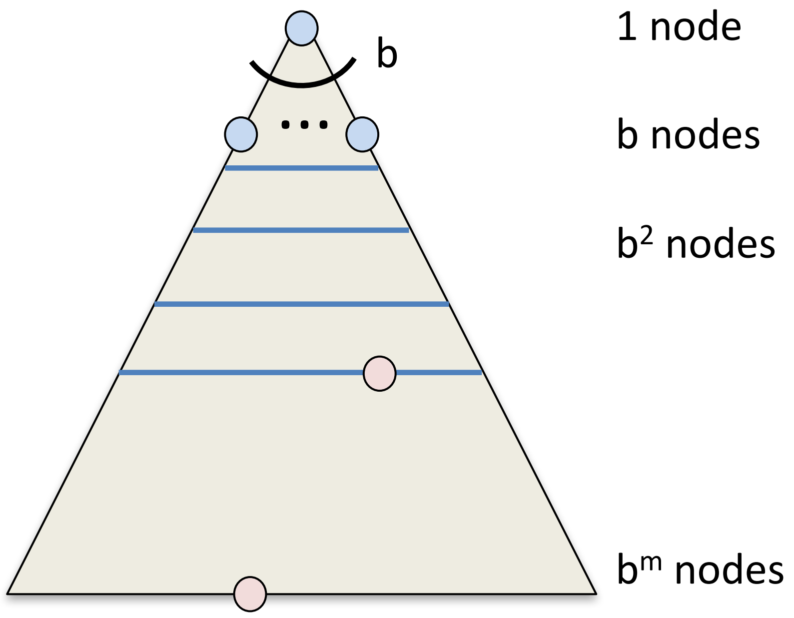
- same as for space

**Completeness:**

- Yes, since s must be finite if a solution exists

**Optimality:**

- Only if costs are all equal

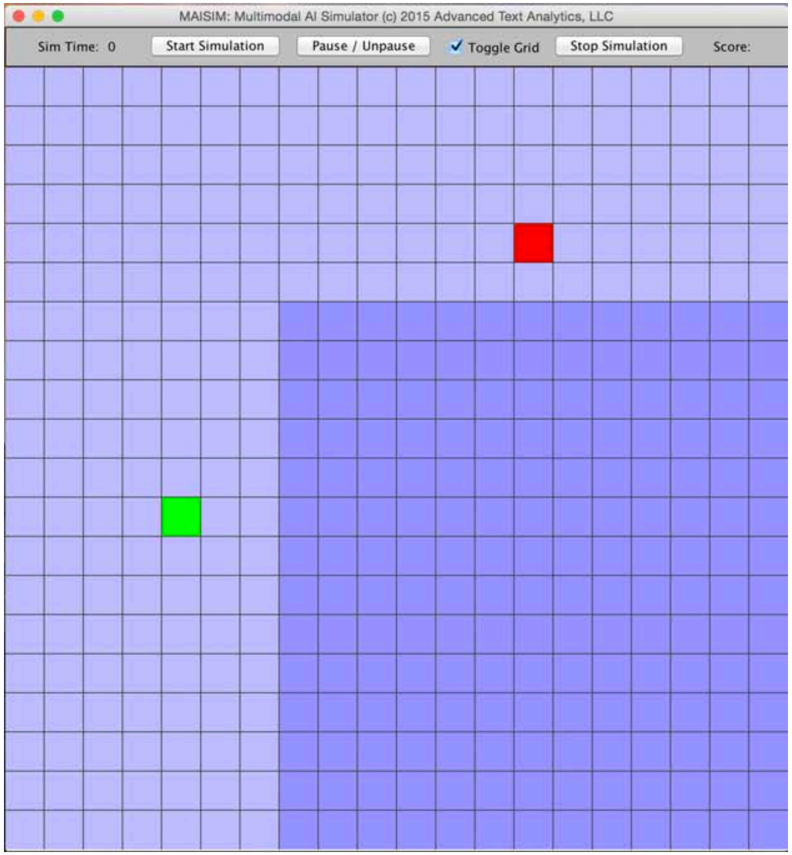



*BFS expands nodes layer-by-layer*

# Quiz: DFS v. BFS

Q1: When will BFS outperform DFS?

Q2: When will DFS outperform BFS?

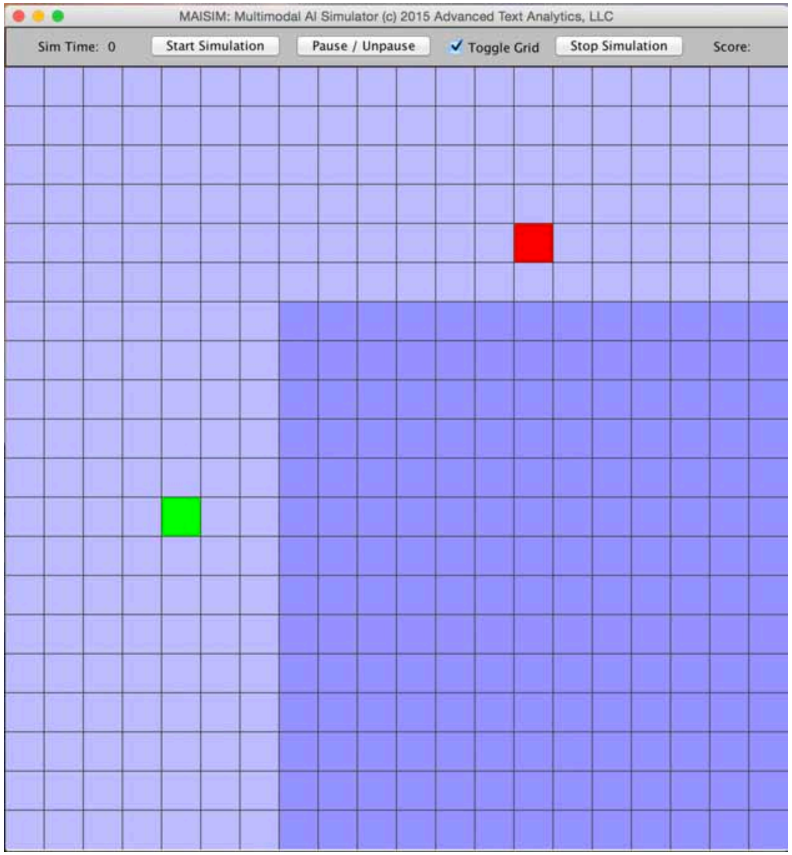


# Quiz: DFS v. BFS

Q1: When will BFS outperform DFS?

Q2: When will DFS outperform BFS?

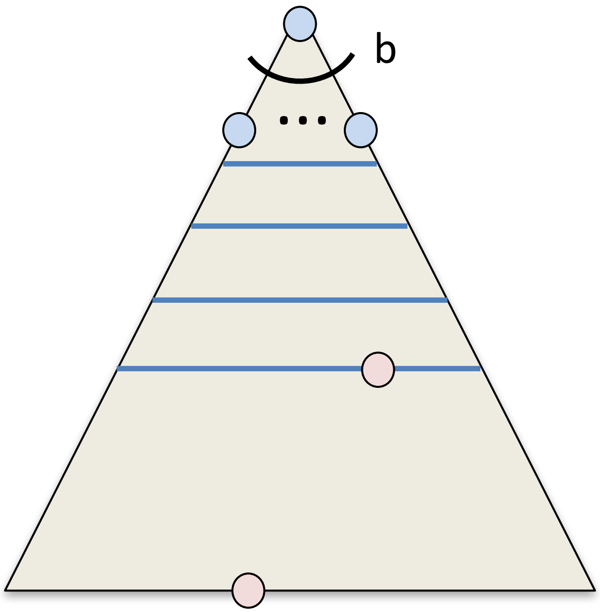
- Things to consider
  - branching factor
  - how deep is the search tree
  - how deep is a/the solution



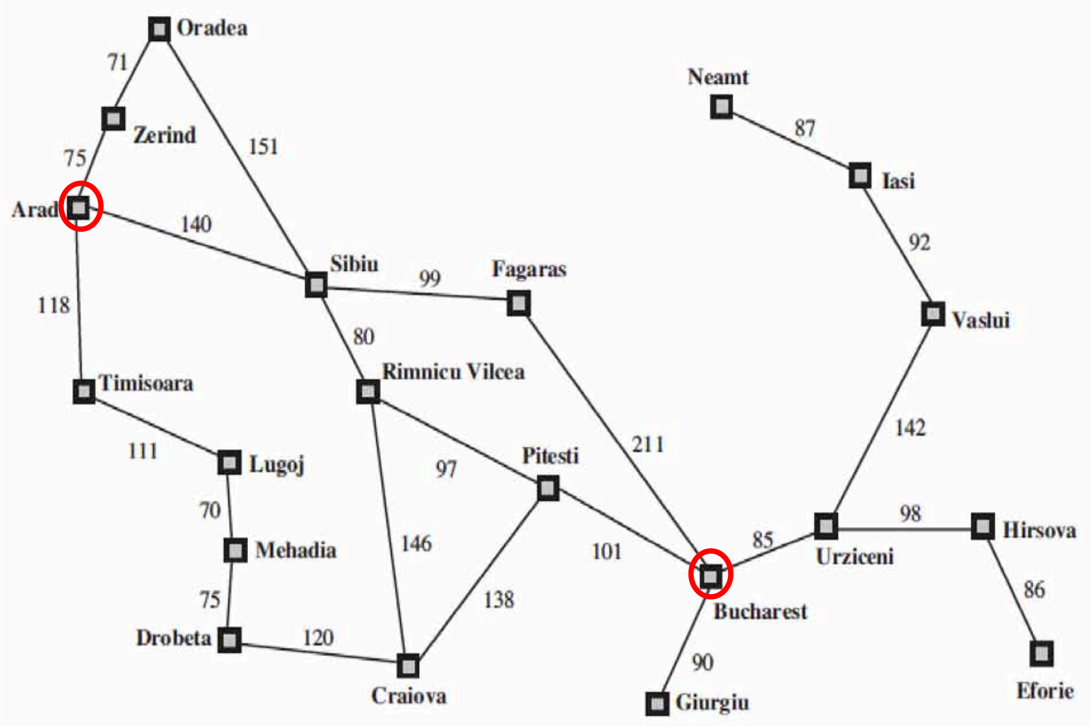

*demo: dfs, bfs*

# Iterative Deepening Search (IDS)

- Basic idea:
  - Combine DFS space advantage with BFS time / shallow-solution advantages
- How?
  - Run DFS with depth limit 1. If no solution...
  - Run DFS with depth limit 2. If no solution...
  - Keep deepening until solution found
- Performance
  - Asymptotically same as BFS: $O(\ b^s\ )$
  - Reason: most work at bottom level
  - Nodes at bottom row generated once
  - Those 1 level up are generated twice, etc.



# Cost-Sensitive Search




- BFS finds path with least number of nodes
- DFS finds “leftmost” solution path

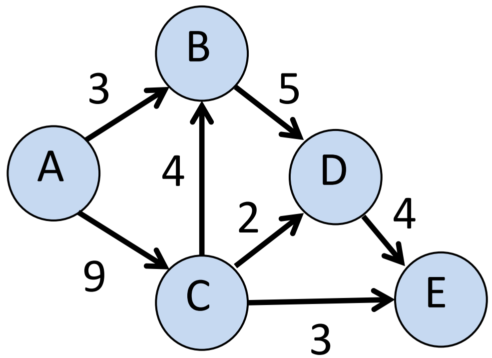
→ We want *least cost* path

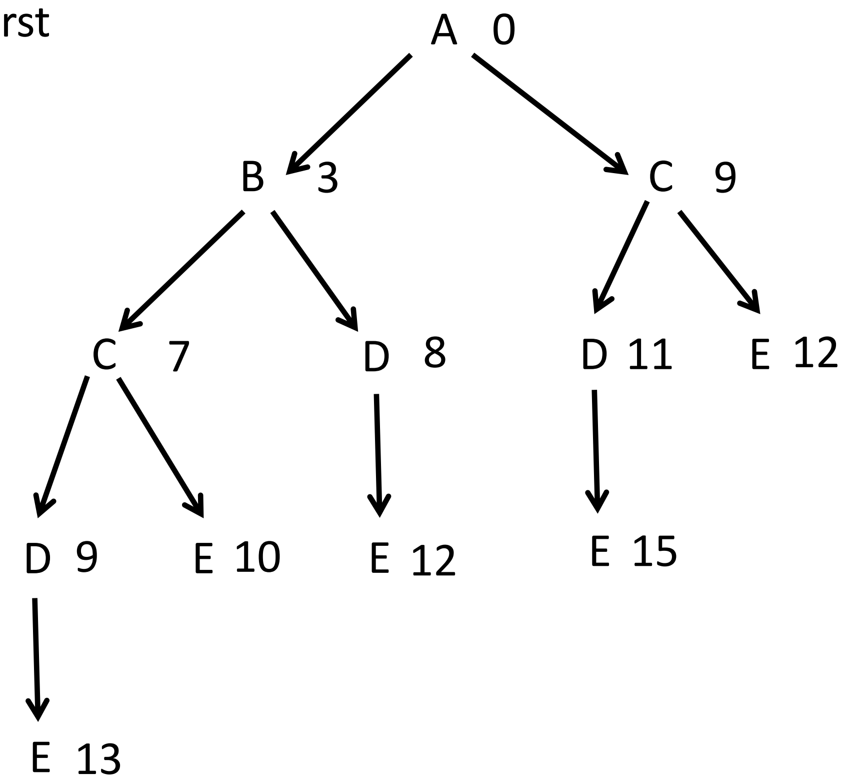
# Uniform Cost Search

Strategy: expand the cheapest node first

Implementation:

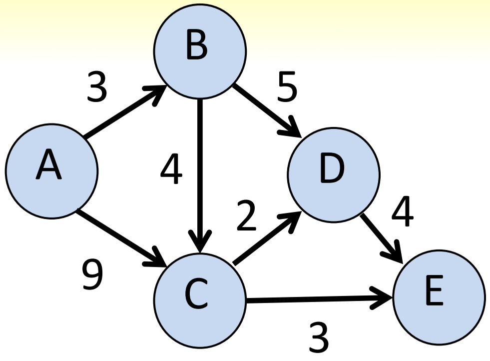
fringe is a priority queue




Assume the problem is to go from A to E and we add nodes in reverse alphabetic order

# UCS Example




- Given graph at right
- Assume start at A and goal is E
- Use UCS to find solution path
- Show fringe (but not explored set) at each step

| State/Action | Fringe contents (paths) | | | |
|---|---|---|---|---|
| Initial state: | [(A),0] | | | |
| Expand [(A),0] | [(A,C),9] | [(A,B),3] | | |
| Expand [(A,B),3] | [(A,C),9] | [(A,B,D),8] | [(A,B,C),7] | |
| Expand [(A,B,C),7] | [(A,C),9] | [(A,B,D),8] | [(A,B,C,E),10] | [(A,B,C,D),9] |
| Expand [(A,B,D),8] | [(A,C),9] | [(A,B,C,E),10] | [(A,B,C,D),9] | [(A,B,D,E),12] |
| Expand [(A,C),9] | [(A,B,C,E),10] | [(A,B,C,D),9] | [(A,B,D,E),12] | [(A,C,E),12] |
| Expand [(A,B,C,D),9] | [(A,B,C,E),10] | [(A,B,D,E),12] | [(A,C,E),12] | [(A,B,C,D,E),13] |
| Expand [(A,B,C,E),10] | Goal reached with cost 10 | | | |

Notes:

1. We do not declare goal reached when add a path to fringe, only when pull off of fringe
2. We use explored set (not shown) to reject paths to states that are longer than previously known paths to same states

# Uniform Cost Search (UCS) Properties

Space complexity: O( $b^{C^*/\varepsilon}$ )

- exponential in "effective depth"
- where C* = cost of solution
- b is average branching factor
- and ε = minimum arc cost

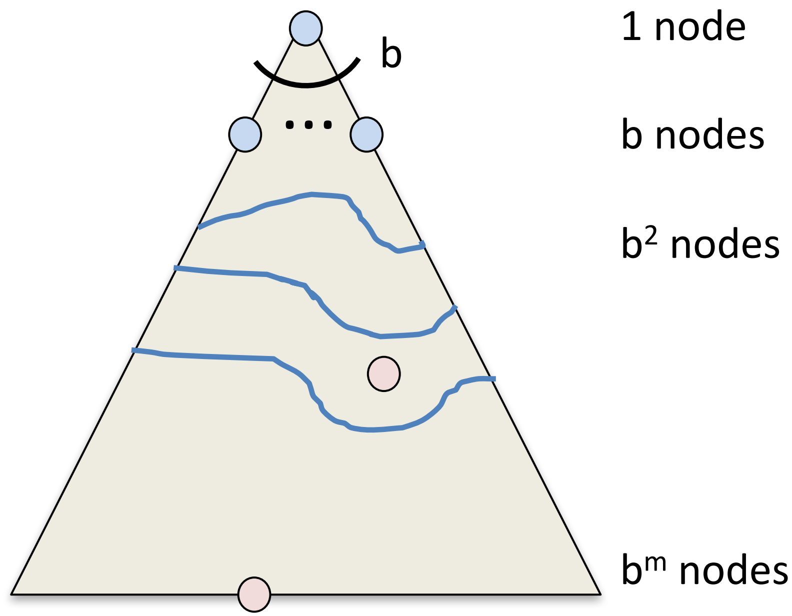
Time complexity: O( $b^{C^*/\varepsilon}$ )

- same as for space

Completeness:

Yes, if min arc cost is positive and best solution has a finite cost

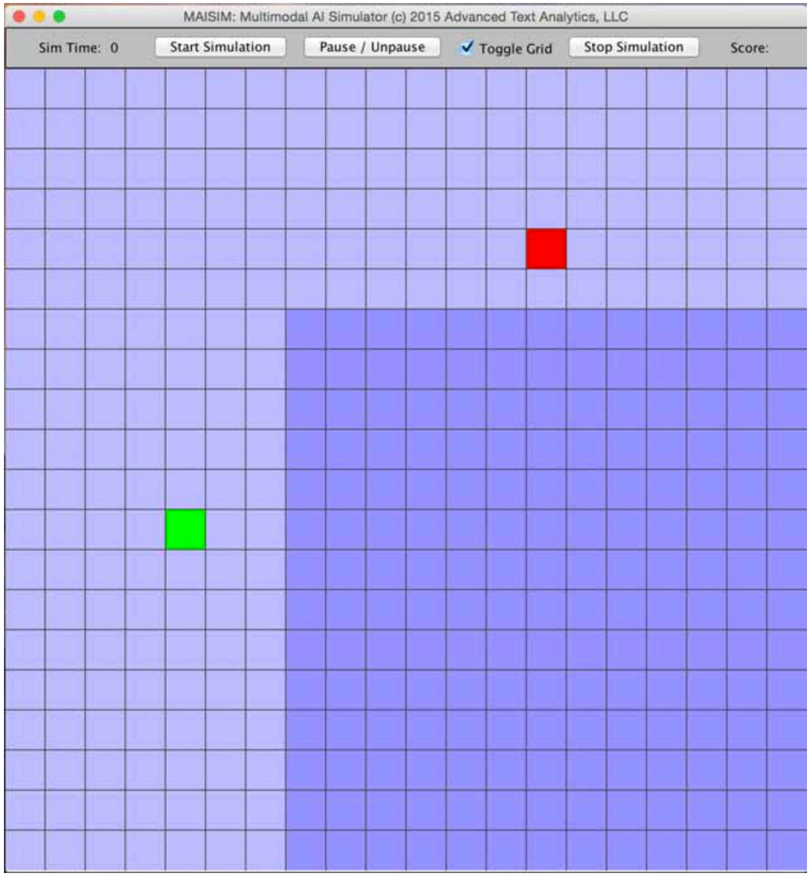
Optimality: Yes




*UCS expands nodes in "cost contours"*

# Uniform Cost Search in Action

Look for ways to improve the search

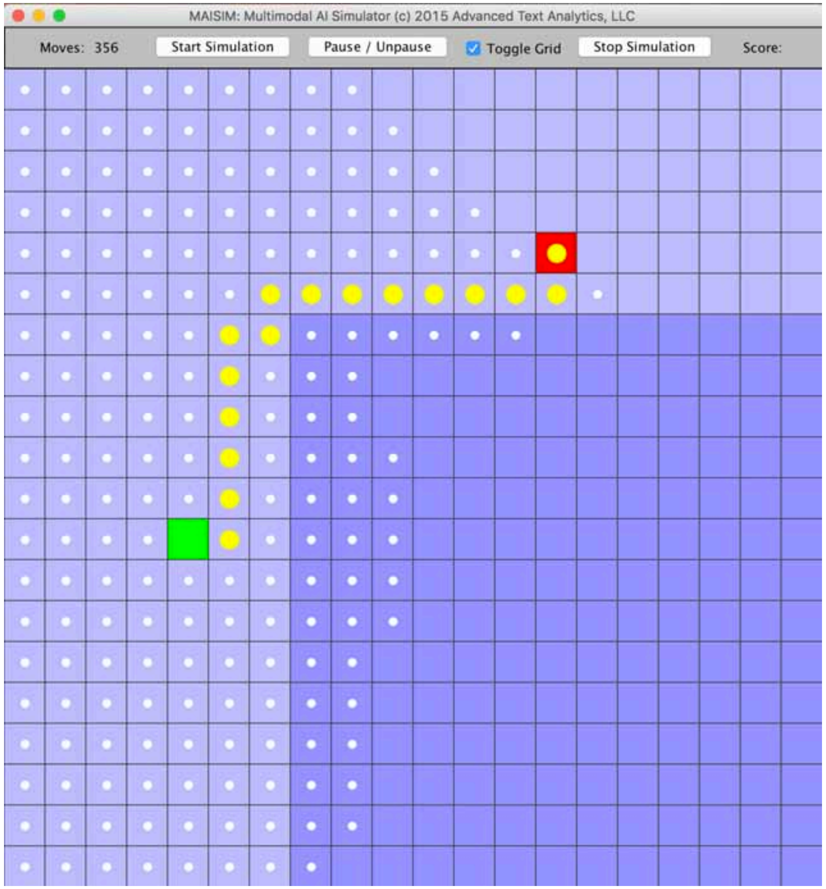



*demo: ucs*

# Issues with Uniform Cost Search

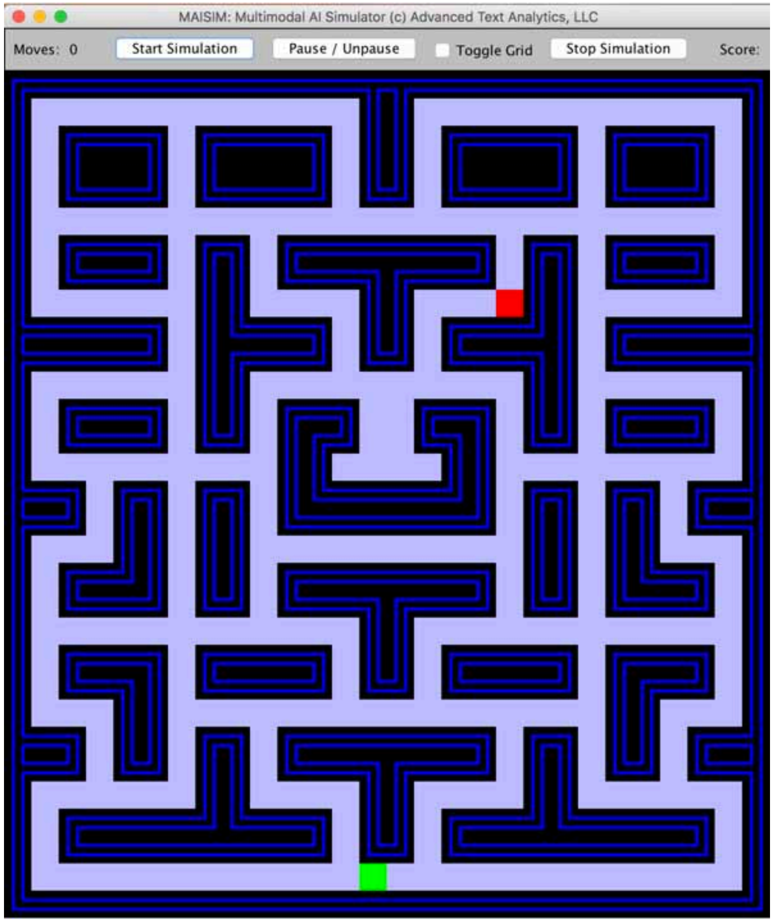
- Search is in all directions
- Information about goal is not used to guide the search

*( Heuristic search will help us with that )*



# Searching Pac-Man Mazes

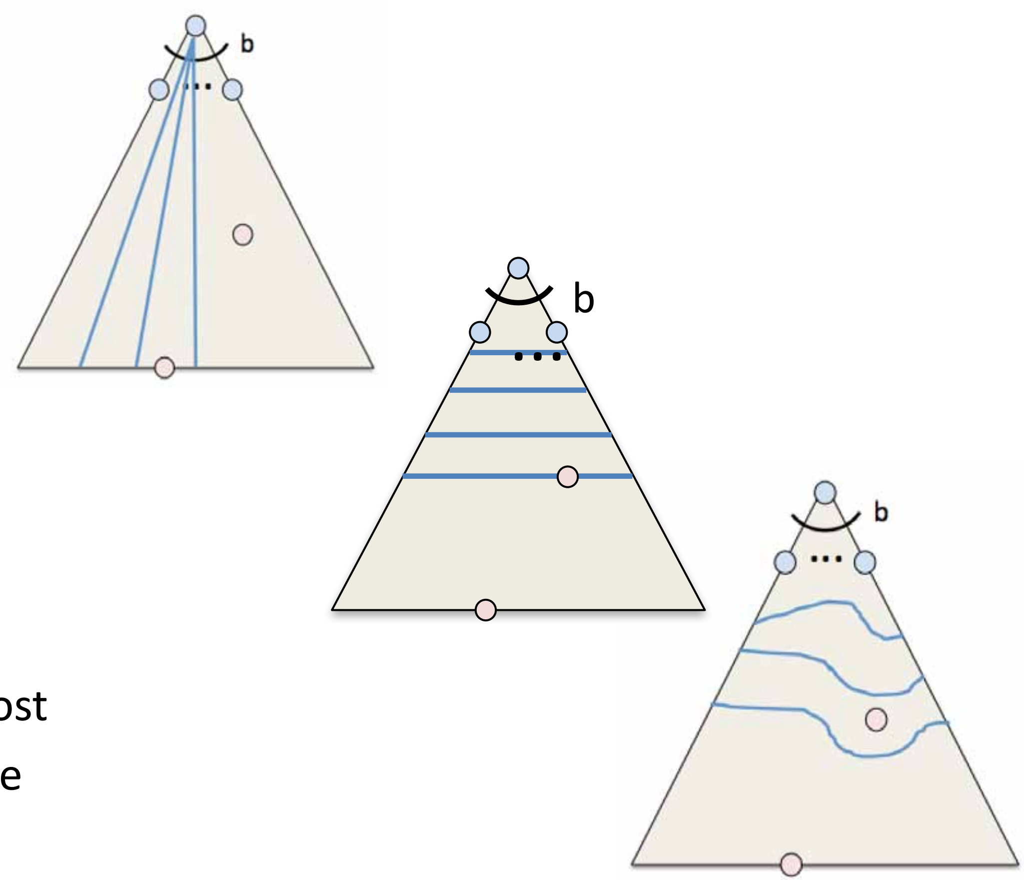
- Pac-Man mazes are characterized by
  - Finite search space
  - Low branch factor
- Nevertheless, we can pose problems that cannot be solved practically using
  - exhaustive search
  - basic uninformed search methods
  - e.g., Program 1
- Here, we illustrate searching a maze using
  - DFS
  - BFS




*demo: maze-dfs, maze-bfs*

# Uninformed Search Strategy Review

- Fringe can be a priority queue
  - Depth-first search
    - order by LIFO
    - can instead use a stack
  - Breadth-first search
    - order by FIFO
    - can instead use a queue
  - Uniform cost search
    - order based on lowest cost
    - must use a priority queue




Q: What about iterative-deepening search?

# Closing Thoughts

- All of these search methods use the same general graph search algorithm

- We ***can*** use a priority queue for all of the fringes
  - The only difference is the priority scheme:  LIFO, FIFO, min cost
  - But for BFS & DFS, can save log(n) overhead by using simpler queue or stack

- From an agent perspective, search operates over a model of the world
  - Agent first searches (simulates actions), and then acts

→ The search is only as good as the model

*( i.e., if the model does not account for ghosts, then Pac-Man is toast )*