# CS 188: Artificial Intelligence
## Spring 2010

Lecture 2: Queue-Based Search
1/21/2010

Pieter Abbeel – UC Berkeley
Many slides from Dan Klein

# Announcements

- Project 0: Python Tutorial
  - Out today. Due next week Thursday.
  - Lab sessions in 271 Soda:
    - Monday 2-3pm
    - Wednesday 4-5pm
  - The lab time is optional, but P0 itself is not
  - On submit, you should get email from the autograder
  - Potentially more lab sessions or office hours held in the lab --- track the announcements section on the webpage!
- Written 1: Search
  - Out today, also due next week Thursday.
- Sections starting next week, location: 285 Cory
  - Section 101: Tue 3-4pm
  - Section 104: Tue 4-5pm
  - Section 102: Wed 11-noon
  - Section 103: Wed noon-1pm

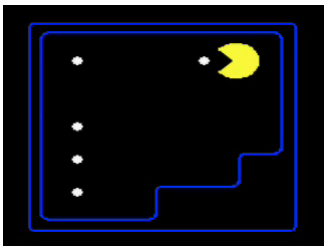
# Today

- Agents
- Search Problems
- Uninformed Search Methods (part review for some)
  - Depth-First Search
  - Breadth-First Search
  - Uniform-Cost Search
- Heuristic Search Methods (new for all)
  - Greedy Search

# Reminder

- Only a very small fraction of AI is about making computers play games intelligently
- Recall: computer vision, natural language, robotics, machine learning, computational biology, etc.
- That being said: games tend to provide relatively simple example settings which are great to illustrate concepts and learn about algorithms which underlie many areas of AI

# A reflex agent for pacman

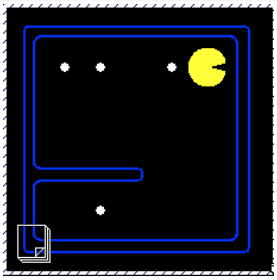


4 actions: move North, East, South or West

Reflex agent

- While(food left)
  - Sort the possible directions to move according to the amount of food in each direction
  - Go in the direction with the largest amount of food

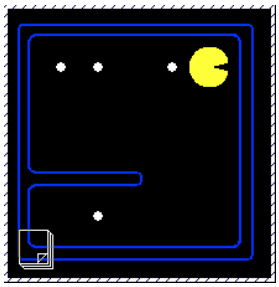
# A reflex agent for pacman (2)



Reflex agent

- While(food left)
  - Sort the possible directions to move according to the amount of food in each direction
  - Go in the direction with the largest amount of food

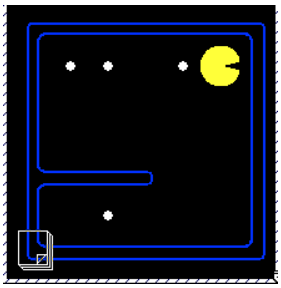
# A reflex agent for pacman (3)



Reflex agent

- While(food left)
  - Sort the possible directions to move according to the amount of food in each direction
  - Go in the direction with the largest amount of food
    - But, if other options are available, exclude the direction we just came from

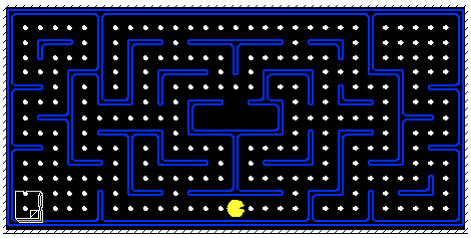
# A reflex agent for pacman (4)



Reflex agent

- While(food left)
  - If can keep going in the current direction, do so
  - Otherwise:
    - Sort directions according to the amount of food
    - Go in the direction with the largest amount of food
    - But, if other options are available, exclude the direction we just came from

# A reflex agent for pacman (5)



Reflex agent

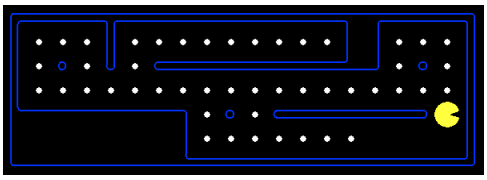
- While(food left)
  - If can keep going in the current direction, do so
  - Otherwise:
    - Sort directions according to the amount of food
    - Go in the direction with the largest amount of food
    - But, if other options are available, exclude the direction we just came from

# Reflex Agents

- Reflex agents:
  - Choose action based on current percept (and maybe memory)
  - May have memory or a model of the world's current state
  - Do not consider the future consequences of their actions
  - Act on how the world IS
- Can a reflex agent be rational?

# Goal Based Agents

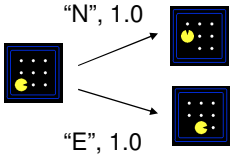
- Goal-based agents:
  - Plan ahead
  - Ask “what if”
  - Decisions based on (hypothesized) consequences of actions
  - Must have a model of how the world evolves in response to actions
  - Act on how the world WOULD BE



# Search Problems

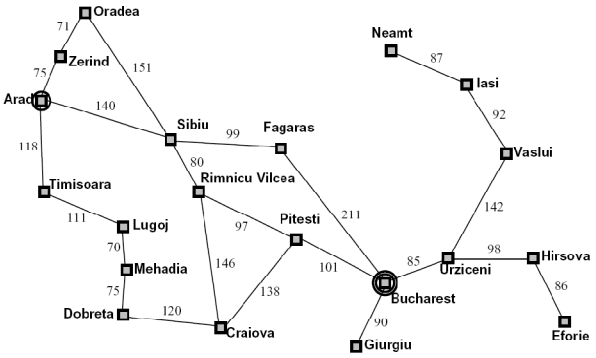
- A search problem consists of:
  - A state space

  - A successor function




  - A start state and a goal test
- A solution is a sequence of actions (a plan) which transforms the start state to a goal state

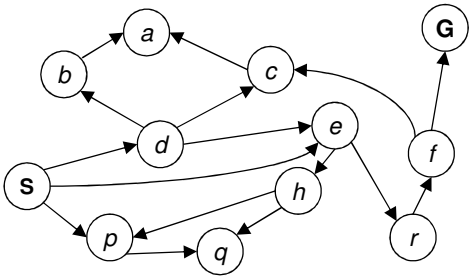
# Example: Romania




- State space:
  - Cities
- Successor function:
  - Go to adj city with cost = dist
- Start state:
  - Arad
- Goal test:
  - Is state == Bucharest?
- Solution?

# State Space Graphs

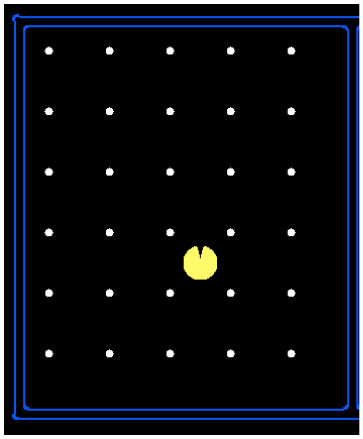
- State space graph: A mathematical representation of a search problem
  - For every search problem, there's a corresponding state space graph
  - The successor function is represented by arcs
- We can rarely build this graph in memory (so we don't)



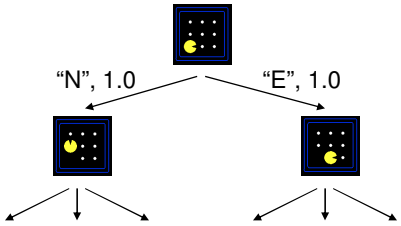

*Ridiculously tiny search graph for a tiny search problem*

# State Space Sizes?

- Search Problem:
  Eat all of the food
- Pacman positions:
  10 x 12 = 120
- Food count: 30

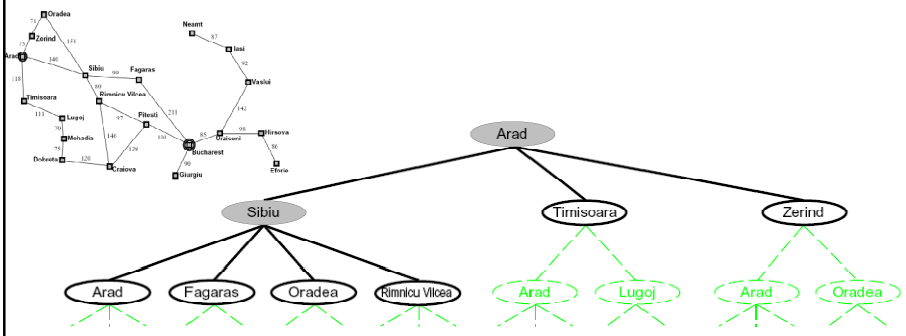


# Search Trees




- A search tree:
  - This is a "what if" tree of plans and outcomes
  - Start state at the root node
  - Children correspond to successors
  - Nodes contain states, correspond to PLANS to those states
  - For most problems, we can never actually build the whole tree

# Another Search Tree




- Search:
  - Expand out possible plans
  - Maintain a fringe of unexpanded plans
  - Try to expand as few tree nodes as possible

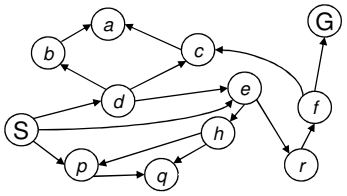
# General Tree Search

```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
   initialize the search tree using the initial state of problem
   loop do
      if there are no candidates for expansion then return failure
      choose a leaf node for expansion according to strategy
      if the node contains a goal state then return the corresponding solution
      else expand the node and add the resulting nodes to the search tree
   end
```

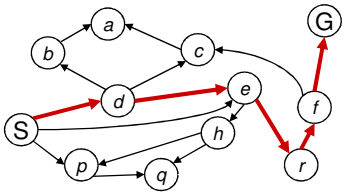
*Detailed pseudocode is in the book!*

- Important ideas:
  - Fringe
  - Expansion
  - Exploration strategy
- Main question: which fringe nodes to explore?

# Example: Tree Search

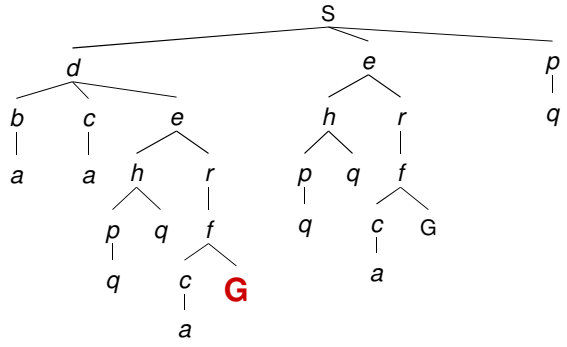



# State Graphs vs. Search Trees




*Each NODE in in the search tree is an entire PATH in the problem graph.*

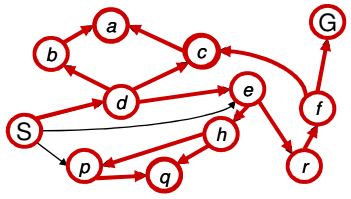
*We construct both on demand – and we construct as little as possible.*

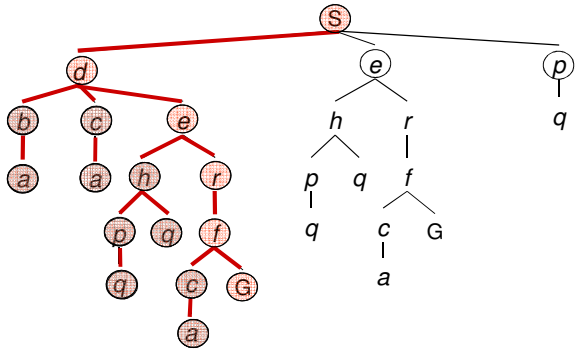


# Review: Depth First Search

*Strategy: expand deepest node first*

*Implementation: Fringe is a LIFO stack*

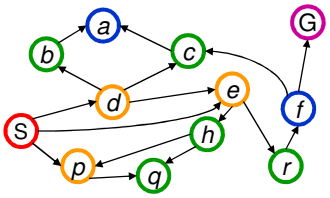


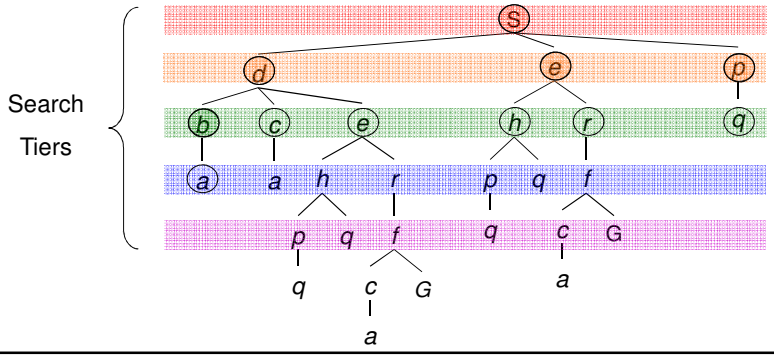




# Review: Breadth First Search

*Strategy: expand shallowest node first*

*Implementation: Fringe is a FIFO queue*



# Search Algorithm Properties

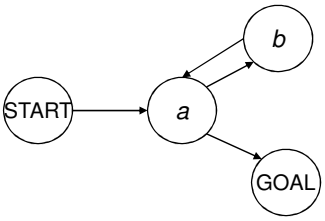
- Complete? Guaranteed to find a solution if one exists?
- Optimal? Guaranteed to find the least cost path?
- Time complexity?
- Space complexity?

Variables:

| | |
|---|---|
| $n$ | Number of states in the problem |
| $b$ | The average branching factor $B$<br>(the average number of successors) |
| $C^*$ | Cost of least cost solution |
| $s$ | Depth of the shallowest solution |
| $m$ | Max depth of the search tree |

# DFS

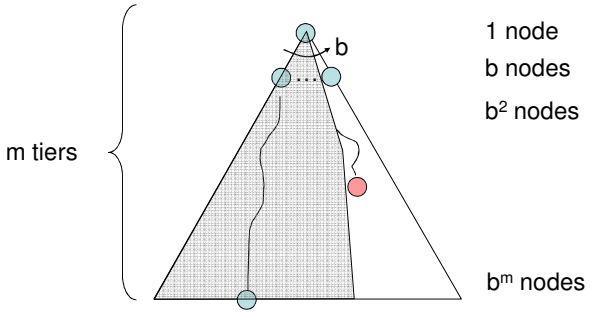
| Algorithm | | Complete | Optimal | Time | Space |
|---|---|---|---|---|---|
| DFS | Depth First Search | N | N | Infinite | Infinite |




- Infinite paths make DFS incomplete…
- How can we fix this?

# DFS

- With cycle checking, DFS is complete.*



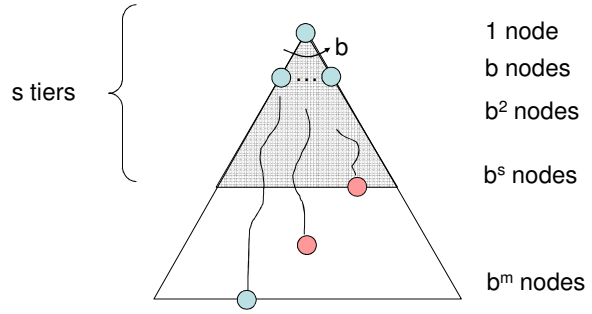

| Algorithm | | Complete | Optimal | Time | Space |
|---|---|---|---|---|---|
| DFS | w/ Path Checking | Y | N | $O(b^m)$ | $O(bm)$ |

- When is DFS optimal?

* Or graph search – next lecture.

# BFS

| Algorithm | | Complete | Optimal | Time | Space |
|---|---|---|---|---|---|
| DFS | w/ Path Checking | Y | N | $O(b^{m+1})$ | $O(bm)$ |
| BFS | | Y | N* | $O(b^{s+1})$ | $O(b^{s+1})$ |




- When is BFS optimal?

# Comparisons

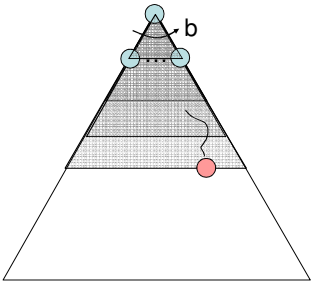
- When will BFS outperform DFS?
- When will DFS outperform BFS?

# Iterative Deepening

Iterative deepening uses DFS as a subroutine:

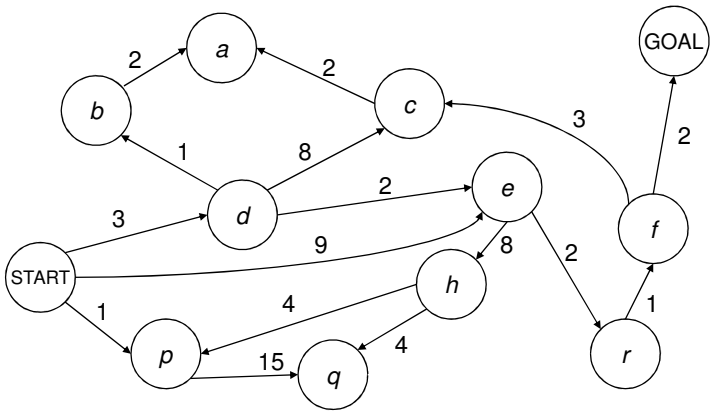
1. Do a DFS which only searches for paths of length 1 or less.
2. If "1" failed, do a DFS which only searches paths of length 2 or less.
3. If "2" failed, do a DFS which only searches paths of length 3 or less.

    ….and so on.




| Algorithm | | Complete | Optimal | Time | Space |
|---|---|---|---|---|---|
| DFS | w/ Path Checking | Y | N | $O(b^m)$ | $O(bm)$ |
| BFS | | Y | N* | $O(b^{s+1})$ | $O(b^{s+1})$ |
| ID | | Y | N* | $O(b^{s+1})$ | $O(bs)$ |

# Costs on Actions



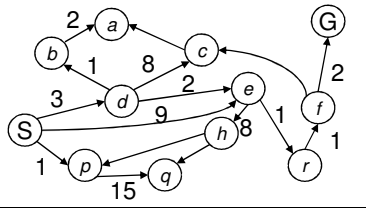

Notice that BFS finds the shortest path in terms of number of transitions. It does not find the least-cost path.

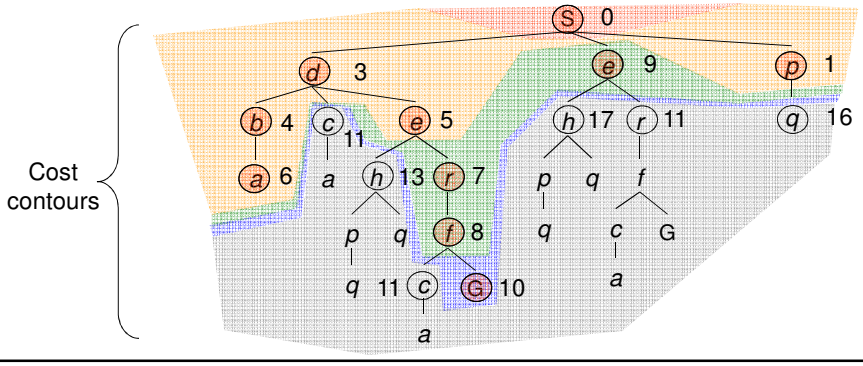
We will quickly cover an algorithm which does find the least-cost path.

# Uniform Cost Search

*Expand cheapest node first:*

*Fringe is a priority queue*




# Priority Queue Refresher

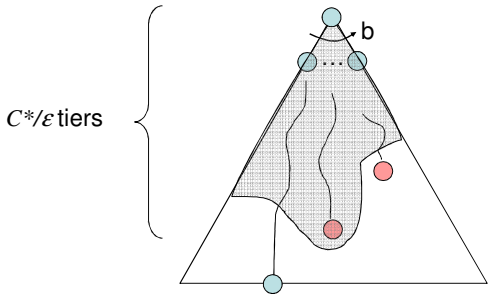
- A priority queue is a data structure in which you can insert and retrieve (key, value) pairs with the following operations:

| | |
|---|---|
| pq.push(key, value) | inserts *(key, value)* into the queue. |
| pq.pop() | returns the key with the lowest value, and removes it from the queue. |

- You can decrease a key's priority by pushing it again
- Unlike a regular queue, insertions aren't constant time, usually O(log *n*)
- We'll need priority queues for cost-sensitive search methods

# Uniform Cost Search

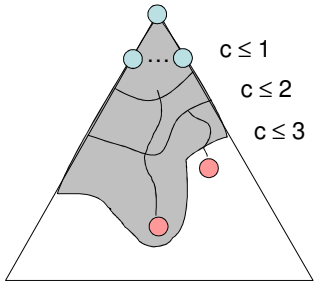
| Algorithm | | Complete | Optimal | Time | Space |
|---|---|---|---|---|---|
| DFS | w/ Path Checking | Y | N | $O(b^m)$ | $O(bm)$ |
| BFS | | Y | N | $O(b^{s+1})$ | $O(b^{s+1})$ |
| UCS | | Y* | Y | $O(b^{C^*/\varepsilon})$ | $O(b^{C^*/\varepsilon})$ |

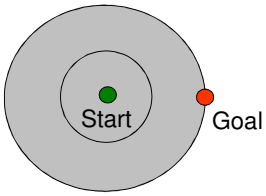



** UCS can fail if actions can get arbitrarily cheap*

# Uniform Cost Issues

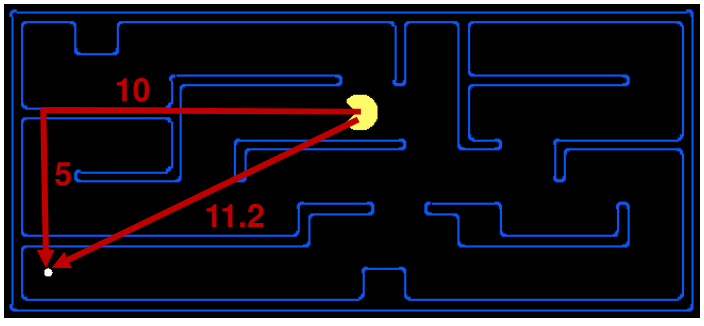
- Remember: explores increasing cost contours
- The good: UCS is complete and optimal!
- The bad:
  - Explores options in every "direction"
  - No information about goal location



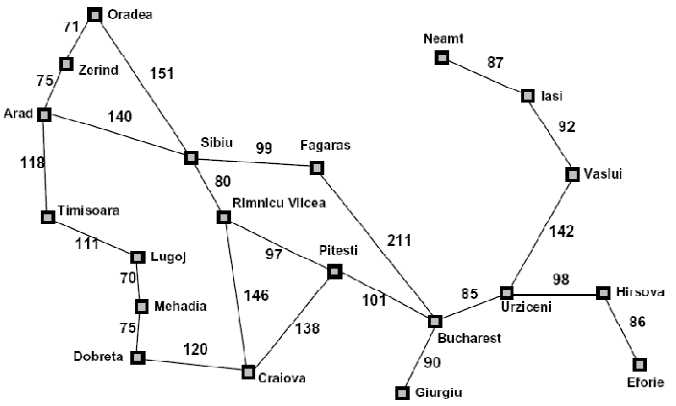




# Search Heuristics

- Any *estimate* of how close a state is to a goal
- Designed for a particular search problem
- Examples: Manhattan distance, Euclidean distance



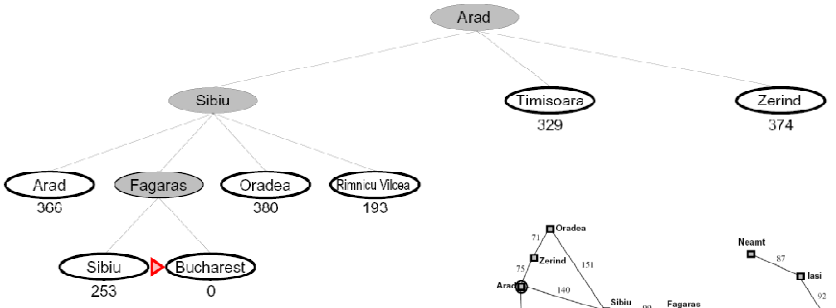
# Heuristics

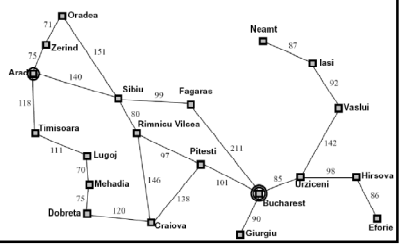



| Straight-line distance to Bucharest | |
|---|---|
| Arad | 366 |
| Bucharest | 0 |
| Craiova | 160 |
| Dobreta | 242 |
| Eforie | 161 |
| Fagaras | 178 |
| Giurgiu | 77 |
| Hirsova | 151 |
| Iasi | 226 |
| Lugoj | 244 |
| Mehadia | 241 |
| Neamt | 234 |
| Oradea | 380 |
| Pitesti | 98 |
| Rimnicu Vilcea | 193 |
| Sibiu | 253 |
| Timisoara | 329 |
| Urziceni | 80 |
| Vaslui | 199 |
| Zerind | 374 |

# Best First / Greedy Search

- Expand the node that seems closest…

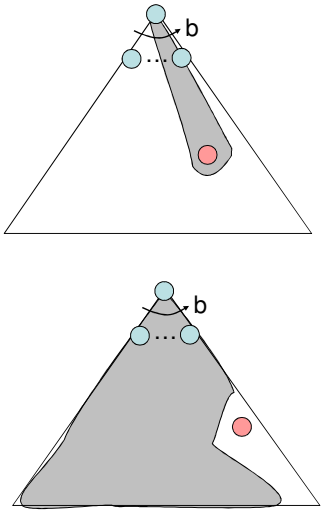


- What can go wrong?

# Best First / Greedy Search

- A common case:
  - Best-first takes you straight to the (wrong) goal
- Worst-case: like a badly-guided DFS in the worst case
  - Can explore everything
  - Can get stuck in loops if no cycle checking
- Like DFS in completeness (finite states w/ cycle checking)




- Can we leverage the heuristic information in a more sound way?

→A* search

We will cover that on Tuesday!