CS 188: Artificial Intelligence
Spring 2009

Lecture 2: Queue-Based Search
1/22/2008

John DeNero – UC Berkeley
Many slides from Dan Klein, Stuart Russell or Andrew Moore

Announcements

- Project 0: Python Tutorial
  - Posted online now
  - Due next Wednesday, Jan 28
  - There is a lab today from 1pm-3pm in Soda 275
  - The lab is optional, but the assignment is not
  - If you submit, you won't get an email yet
- Project 1: Search
  - Posted tonight
  - Due in two weeks: Wednesday, Feb 4
  - Start early and ask questions. It's longer than most!

More Announcements

- Section
  - Section starts Monday
  - Section 104 from 5pm - 6pm will be held in 9 Evans
  - Times and locations will be on the website shortly
- Office hours
  - My new office hours: Tues 3-4 and Wed 11-12
  - GSI office hours are (or will be) on the website

Today

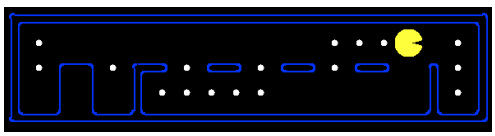
- Agents that Plan Ahead
- Search Problems
- Uninformed Search Methods (review for many)
  - Depth-First Search
  - Breadth-First Search
  - Uniform-Cost Search
- Heuristic Search Methods (new material)
  - Greedy Search

From Last Time: Reflex Agents

- Reflex agents:
  - Choose action based on current percept and memory
  - Do not consider future consequences of their actions
- Can a reflex agent be rational?
- How good was our agent from last class?
  - Reminder: ate food if it was there; avoided ghosts
  - Against random ghosts: won 31% of the time
  - On the original Pacman map: 5% win rate
  - Against reflex ghosts on small map: 3% win rate

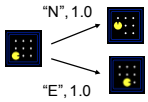
Goal Based Agents

- Goal-based agents:
  - Plan ahead
  - Make decisions based on (hypothesized) consequences of actions
  - Must have a model of how the world evolves in response to actions



[demo: plan fast/plan optimal]

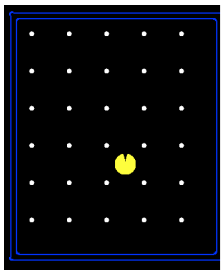
Search Problems

- A search problem consists of:
  - A state space
  - A successor function 

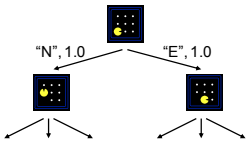
  - A start state and a goal test
- A solution is a sequence of actions (a plan) which transforms the start state to a goal state

How Big is the State Space?

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



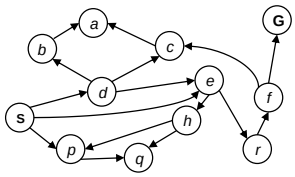
Search Trees



- 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

State Space Graphs

- For every search problem, there's a corresponding graph of the state space
- The successor function is represented by arcs
- We can rarely build this graph in memory



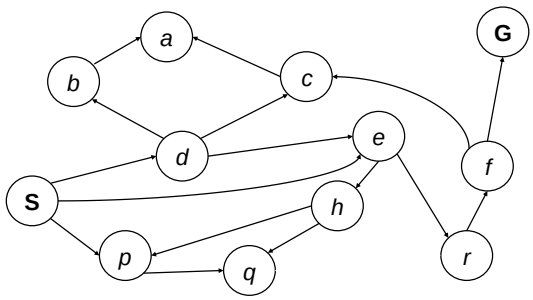
Laughably tiny search graph for a tiny search problem

General Tree Search

Detailed pseudocode is in the book!

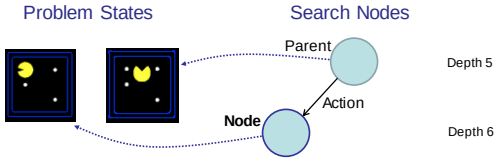
- Tree Search
  - Initialize the *root node* of the search tree with the *start state*
  - While there are unexpanded leaf nodes (fringe):
    - Choose a leaf node (strategy)
    - If the node contains a goal state: return the corresponding solution
    - Else: expand the node and add its children to the tree
- Important ideas:
  - Fringe
  - Expansion
- Strategy: which fringe nodes to explore?

Example: Tree Search

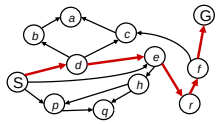


States vs. Nodes

- State space graphs have problem states
  - Represent an abstracted state of the world
  - Have successors, can be goal / non-goal, have multiple predecessors
- Search trees have search nodes
  - Represent a plan (path) which results in the node's state
  - Have a problem state and one parent, a path length, a depth & a cost
  - The same problem state may be in multiple search tree nodes

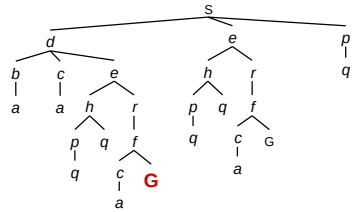


State Graphs vs Search Trees



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

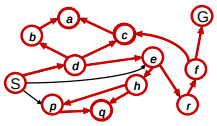
We almost always construct both on demand – and we construct as little as possible.

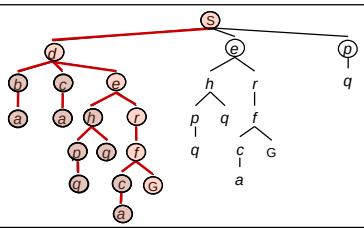


Review: Depth First Search

Strategy: expand deepest node first

Implementation: Fringe is a LIFO stack

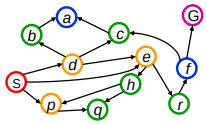


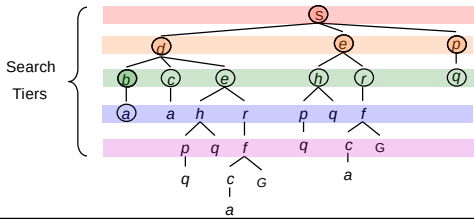


Review: Breadth First Search

Strategy: expand shallowest node first

Implementation: Fringe is a FIFO queue





Search Algorithm Properties

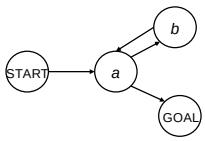
- Complete? Guaranteed to find a solution if one exists?
- Optimal? Guaranteed to find the least cost path?
- Time complexity?
- Space complexity?

Variables:

n	Number of states in the problem
b	The average branching factor B (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

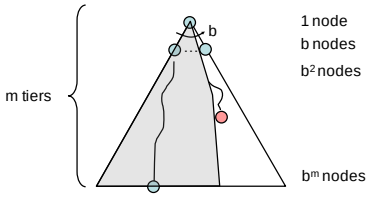
Algorithm		Complete	Optimal	Time	Space
DFS	Depth First Search	N	N	Infinite	Infinite



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

DFS

- With cycle checking, DFS is complete.

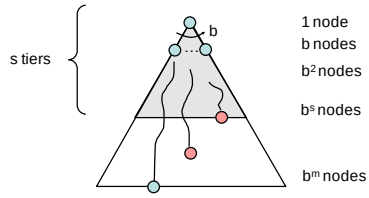


Algorithm		Complete	Optimal	Time	Space
DFS	w/ Path Checking	Y	N	$O(b^{m+1})$	$O(bm)$

- When is DFS optimal?

BFS

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

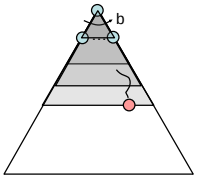


- When is BFS optimal?

Iterative Deepening

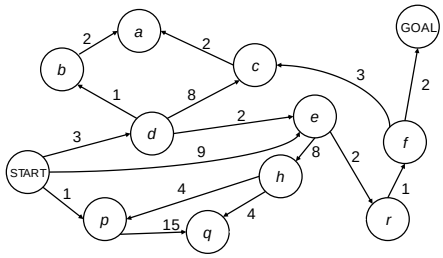
Iterative deepening uses DFS as a subroutine:

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	$O(b^{m+1})$	$O(bm)$
BFS		Y	N*	$O(b^{s+1})$	$O(b^s)$
ID		Y	N*	$O(b^{s+1})$	$O(bs)$

Costs on Actions

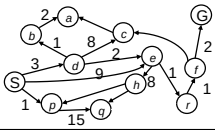


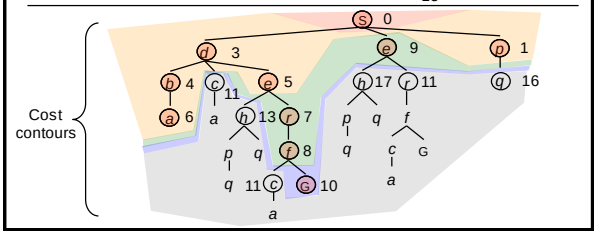
Notice that BFS finds the shortest path in terms of number of transitions. It does not find the least-cost path.
We will quickly cover an algorithm which does find the least-cost path.

Uniform Cost Search

Expand cheapest node first:

Fringe is a priority queue





Priority Queue Refresher

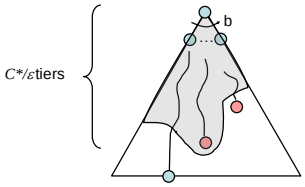
- A priority queue is a data structure in which you can insert and retrieve (key, value) pairs with the following operations:

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

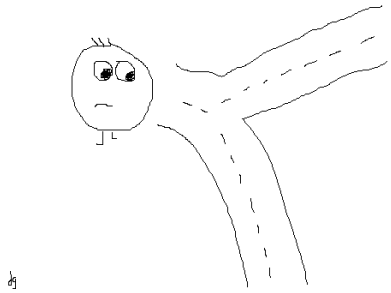
Uniform Cost Search

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



You can read more about uniform cost search's failure in the book, or by asking us…

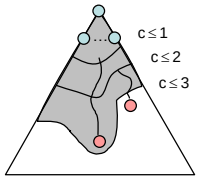
5 Minute Break

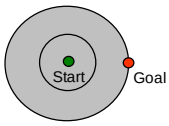


A Dan Gillick original

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

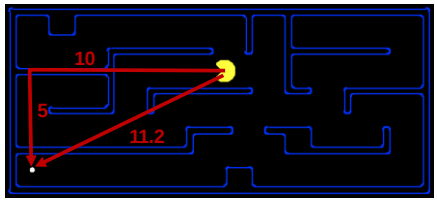




[demo: ucs contours]

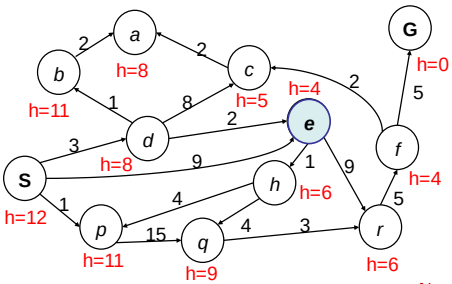
Search Heuristics

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



Best First / Greedy Search

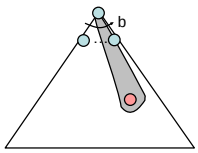
- Strategy: expand the closest node to the goal

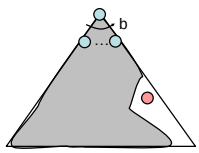


[demo: greedy]

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)

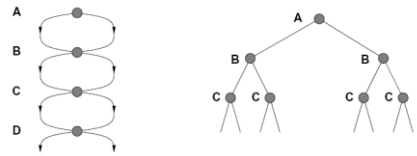




Search Gone Wrong?

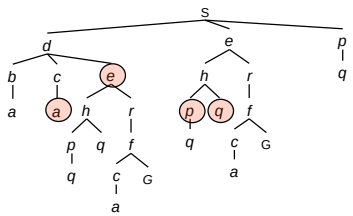
Extra Work?

- Failure to detect repeated states can cause exponentially more work (why?)



Graph Search

- In BFS, for example, we shouldn't bother expanding the circled nodes (why?)



Graph Search

- Very simple fix: never expand a state type twice

```
function GRAPH-SEARCH(problem, fringe) returns a solution, or failure
  closed ← an empty set
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
     if fringe is empty then return failure
     node ← REMOVE-FRONT(fringe)
     if GOAL-TEST(problem, STATE[node]) then return node
     if STATE[node] is not in closed then
        add STATE[node] to closed
        fringe ← INSERTALL(EXPAND(node, problem), fringe)
  end
```

- Can this wreck completeness? Why or why not?
- How about optimality? Why or why not?

Some Hints

- Graph search is almost always better than tree search (when not?)
- Implement your closed list as a dict or set!
- Nodes are conceptually paths, but better to represent with a state, cost, last action, and reference to the parent node