Uninformed search

Lirong Xia

Rensselaer

Spring, 2017
Today’s schedule

➢ Rational agents

➢ Search problems
  • State space graph: modeling the problem
  • Search trees: scratch paper for solution

➢ Uninformed search
  • Depth first search (DFS) algorithm
  • Breadth first search (BFS) algorithm
You entered a maze in darkness
No map but you build one as you explore
Limited sight, only know which direction does not have a wall
  know nothing about enemies, traps, etc.
  you only see the exit when you step on it
Goal: write a walkthrough to minimize the cost of reaching the next level
How would you do it?
An agent is an entity that perceives and acts.

A rational agent selects actions that maximize its utility function.

Characteristics of the percepts, environment, and action space dictate techniques for selecting rational actions.
Example 1: Pacman as an Agent

Agent

Sensors

?  
Actuators

Environment

Percepts

Actions

SCORE: 18
When goal = search for something (no cost yet)
A search problem consists of:

- A state space
- A successor function (with actions, costs)
- 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.
State space graph: modeling the problem

- A directed weighted graph of all states
  - $a \rightarrow b$: $b$ is a successor of $a$
  - weight($a \rightarrow b$): the cost of traveling from $a$ to $b$

- Note: just for analysis, usually the state space graph is not fully built
What's in a State Space?

The world state specifies every last detail of the environment.

A search state keeps only the details needed (abstraction).

• Problem: Pathing
  • States: (x,y) location
  • Actions: NSEW
  • Successor: adjacent locations
  • Goal test: is (x,y) = END

• Problem: Eat-All-Dots
  • States: {(x,y), dot booleans}
  • Actions: NSEW
  • Successor: updated location and dot booleans
  • Goal test: dots all false
State Space Sizes?

- World state:
  - Agent positions: 120
  - Food count: 30
  - Ghost positions: 12
  - Agent facing: NSEW

- How many
  - World states?
    \[ 120 \times 2^{30} \times 12^2 \times 4 \]
  - States for pathing?
    120
  - States for eat-all-dots?
    \[ 120 \times 2^{30} \]
• **A search tree:**
  - 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
Space graph vs. search tree

- Nodes in state space graphs are problem states:
  - Represent an abstracted state of the world
  - Have successors, can be goal/non-goal, have multiple predecessors

- Nodes in search trees are plans:
  - Represent a plan (sequence of actions) which results in the node’s state
  - Have a problem state and one parent, a path length, a depth and a cost
  - The same problem state may be achieved by multiple search tree nodes

Problem States  Search Nodes

[Diagram showing state space graph and search tree nodes with problem states and search nodes, along with depth information for Depth 5 and Depth 6]
Uninformed search

- **Uninformed search**: given a state, we only know whether it is a goal state or not
- Cannot say one non-goal state looks better than another non-goal state
- Can only traverse state space blindly in hope of somehow hitting a goal state at some point
  - Also called **blind search**
  - Blind does **not** imply unsystematic!
Breadth-first search (search tree)
BFS

- Never expand a node whose state has been visited
- Fringe can be maintained as a First-In-First-Out (FIFO) queue (class Queue in util.py)
- Maintain a set of visited states
- fringe := \{node corresponding to initial state\}
- loop:
  - if fringe empty, declare failure
  - choose and remove the top node v from fringe
  - check if v’s state s is a goal state; if so, declare success
  - if v’s state has been visited before, skip
  - if not, expand v, insert resulting nodes into fringe
- This is the BFS you should implement in project 1
Properties of breadth-first search

- May expand more nodes than necessary
- BFS is complete: if a solution exists, one will be found
- BFS finds a shallowest solution
  - Not necessarily an optimal solution if the cost is non-uniform
- If every node has b successors (the branching factor), shallowest solution is at depth d, then fringe size will be at least $b^d$ at some point
  - This much space (and time) required 😞
Depth-first search
Properties of depth-first search

- **Not complete** (might cycle through non-goal states)
- If solution found, generally not optimal/shallowest
- If every node has \( b \) successors (the **branching factor**), and we search to at most depth \( m \), fringe is at most \( bm \)
  - Much better space requirement 😊
  - Saves even more space by recursion
- **Time**: still need to check every node
  - \( b^m + b^{m-1} + \ldots + 1 \) (for \( b>1 \), \( O(b^m) \))
  - **Inevitable** for uninformed search methods…
If we keep a set of visited stages

- Never add a visited state to the fringe
- This version of DFS is complete (avoid cycling)
- Space requirement can be as bad as BFS
DFS

- Never expand a node whose state has been visited
- Fringe can be maintained as a Last-In-First-Out (LIFO) queue (class Stack in util.py)
- Maintain a set of visited states
- fringe := \{node corresponding to initial state\}
- loop:
  - if fringe empty, declare failure
  - choose and remove the top node \( v \) from fringe
  - check if \( v \)’s state \( s \) is a goal state; if so, declare success
  - if \( v \)’s state has been visited before, skip
  - if not, expand \( v \), insert resulting nodes into fringe

- This is the DFS you should implement in project 1
You can start to work on Project 1 now

- Read the instructions on course website and the comments in search.py first

- Q1: DFS
  - LIFO

- Q2: BFS
  - FIFO

- Due in two weeks (Feb 3)

- Check util.py for LIFO and FIFO implementation

- Use piazza for Q/A
Dodging the bullets

- The auto-grader is very strict
  - 0 point for expanding more-than-needed states
  - no partial credit
- Hint 1: do not include `print "Start:"`, `problem.getStartState()` in your formal submission
  - comment out all debugging commands
- Hint 2: remember to check if a state has been visited before
- Hint 3: return a path from start to goal. You should pass the local test before submission (details and instructions on project 1 website)