Tree and Graph Search
Reading: AIAMA 3.1-3.4

- Problem Solving as State Space Search
- Example Problems
- Review Trees and Graphs
- Uniformed Search Strategies
Problem Solving Agents formulate problems by

- representing (model) the world as atomic states,
- defining an initial state that represents the initial condition of the world,
- defining a goal state that represents what they want the world to look like,
- and defining a function for allowable state transitions which map onto actions in the world.
function SIMPLE-PROBLEM-SOLVING-AGENT(\textit{percept}) \textbf{returns} an action

\begin{itemize}
  \item \textbf{persistent:} \textit{seq}, an action sequence, initially empty
  \item \textit{state}, some description of the current world state
  \item \textit{goal}, a goal, initially null
  \item \textit{problem}, a problem formulation
\end{itemize}

\textit{state} $\leftarrow$ UPDATE-STATE(\textit{state}, \textit{percept})

\textbf{if} \textit{seq} is empty \textbf{then}

\begin{itemize}
  \item \textit{goal} $\leftarrow$ FORMULATE-GOAL(\textit{state})
  \item \textit{problem} $\leftarrow$ FORMULATE-PROBLEM(\textit{state}, \textit{goal})
  \item \textit{seq} $\leftarrow$ SEARCH(\textit{problem})
  \item \textbf{if} \textit{seq} $=$ failure \textbf{then} \textbf{return} a null action
\end{itemize}

\textit{action} $\leftarrow$ FIRST(\textit{seq})

\textit{seq} $\leftarrow$ REST(\textit{seq})

\textbf{return} \textit{action}
Example: Sliding Tile Puzzle
Another Example: Peg Solitaire
State space terminology

The state space is the collection of the following:

- initial state
- actions
- transition model
- successors

The problem solving agent searches through this space to find a path from the initial to the goal state.
function TREE-SEARCH(problem) returns a solution, or failure
initialize the frontier using the initial state of problem
loop do
  if the frontier is empty then return failure
  choose a leaf node and remove it from the frontier
  if the node contains a goal state then return the corresponding solution
  expand the chosen node, adding the resulting nodes to the frontier
function GRAPH-SEARCH(problem) returns a solution, or failure
initialize the frontier using the initial state of problem
initialize the explored set to be empty
loop do
  if the frontier is empty then return failure
  choose a leaf node and remove it from the frontier
  if the node contains a goal state then return the corresponding solution
  add the node to the explored set
  expand the chosen node, adding the resulting nodes to the frontier
    only if not in the frontier or explored set
Data structures supporting search

We need a few data structures to implement the graph search algorithms.

- **Node structure**
  - the state description
  - a parent pointer or reference
  - the action applied to get from parent to this node
  - path cost, the cost of the path from the initial to this node

- **Function to return successor given state and action**

- **frontier queue (LIFO, FIFO, priority)**

- **explored set (dictionary or hash table)**
How to compare specific search algorithms

We evaluate and compare algorithms based on the following criteria

- Completeness - does it find a solution if one exists?
- Optimality - does the solution have the lowest possible path cost?
- Time Complexity - how long does it take to find the solution?
- Space Complexity - how much memory is needed during the search?

The complexity of the graph is summarized by the:

- branching factor, $b$
- depth of the closest goal, $d$
- maximum depth, $m$
Specific Graph Search Algorithms

Uninformed search strategies
- breadth-first
- uniform-cost
- depth-first
- depth-limited
- iterative deepening
- bidirectional
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure

node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
frontier ← a FIFO queue with node as the only element
explored ← an empty set
loop do
  if EMPTY?(frontier) then return failure
  node ← POP(frontier)  /* chooses the shallowest node in frontier */
  add node.STATE to explored
  for each action in problem.ACTIONS(node.STATE) do
    child ← CHILD-NODE(problem, node, action)
    if child.STATE is not in explored or frontier then
      if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
      frontier ← INSERT(child, frontier)
function Uniform-Cost-Search(problem) returns a solution, or failure

\[
\text{node} \leftarrow \text{a node with State = problem.INITIAL-STATE, Path-Cost = 0}
\]

\[
\text{frontier} \leftarrow \text{a priority queue ordered by Path-Cost, with node as the only element}
\]

\[
\text{explored} \leftarrow \text{an empty set}
\]

loop do

\[
\text{if EMPTY?(frontier) then return failure}
\]

\[
\text{node} \leftarrow \text{POP(frontier) /* chooses the lowest-cost node in frontier */}
\]

\[
\text{if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)}
\]

add node.STATE to explored

for each action in problem.ACTIONS(node.STATE) do

\[
\text{child} \leftarrow \text{CHILD-NODE(problem, node, action)}
\]

\[
\text{if child.STATE is not in explored or frontier then}
\]

\[
\text{frontier} \leftarrow \text{INSERT(child, frontier)}
\]

else if child.STATE is in frontier with higher Path-Cost then

\[
\text{replace that frontier node with child}
\]
function Depth-Limited-Search(problem, limit) returns a solution, or failure/cutoff
    return Recursive-DLS(Make-Node(problem.Initial-State), problem, limit)

function Recursive-DLS(node, problem, limit) returns a solution, or failure/cutoff
    if problem.Goal-Test(node.State) then return Solution(node)
    else if limit = 0 then return cutoff
    else
        cutoff-occurred? ← false
        for each action in problem.Actions(node.State) do
            child ← Child-Node(problem, node, action)
            result ← Recursive-DLS(child, problem, limit - 1)
            if result = cutoff then cutoff-occurred? ← true
            else if result ≠ failure then return result
        if cutoff-occurred? then return cutoff else return failure
function Iterative-Deepening-Search(problem) returns a solution, or failure
for depth = 0 to ∞ do
    result ← Depth-Limited-Search(problem, depth)
    if result ≠ cutoff then return result
Warmup

Consider the following graph with initial nodes A and goal node H. All edges have unit weight.

In what order are nodes expanded using:
1. breadth-first search
2. uniform cost search
3. depth-first search
4. depth-limited search with a max depth of 2
5. iterative-deepening search search

Assume nodes are considered and ties resolved using alphabetical order if needed.
Another example

Consider the same graph as the warmup, but consider the goal node to be G. All edges have unit weight except the one between D and E, which has a weight of 2.

In what order are nodes expanded using:
1. breadth-first search
2. uniform cost search
3. depth-first search
4. depth-limited search with a max depth of 2
5. iterative-deepening search
Next Actions

- Reading: Heuristic Search - AIAMA 3.5 and 3.6
- Take warmup before noon on Thursday 1/29.

Note:
- PS1 has been released. Due 8am 2/16 via Scholar.
- The project proposal is due on or before 5pm on 2/10. See webpage for details.