6.01

Lecture 12: Graph Search

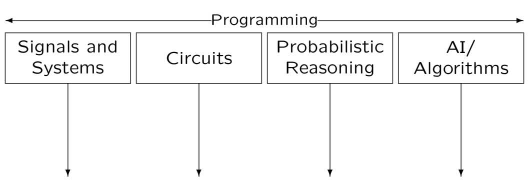
Notes

6.01: Introduction to EECS I

The **intellectual themes** in 6.01 are recurring themes in engineering:

- design of complex systems
- modeling and controlling physical systems
- augmenting physical systems with computation
- building systems that are robust to uncertainty

Approach: focus on **key concepts** to pursue **in depth**



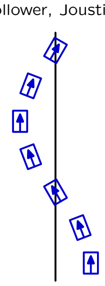
Notes

Module 1: Signals and Systems

Modeling and analyzing behavior of physical systems

Topics: Feedback Control Systems

Lab Exercises: Wall-finder, Wall-follower, Jousting



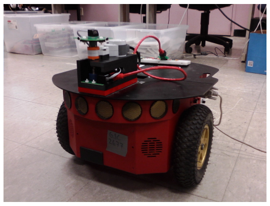
Notes

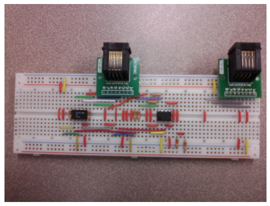
Module 2: Circuits

Designing, constructing, and analyzing physical systems

Topics: Resistive Networks, Op-Amps, Linearity and Equivalence

Lab Exercises: Design a new sensory modality for the robot





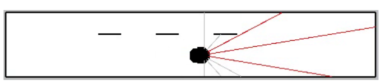
Notes

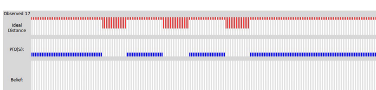
Module 3: Bayesian Reasoning

Modeling uncertainty and designing robust systems

Topics: Subjective Probability, Bayesian Inference

Lab Exercises: Localization and Parking





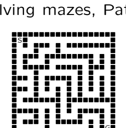
Notes

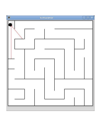
Module 4: Planning

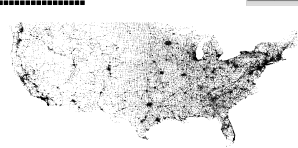
Augmenting physical systems with computation.

Topics: Graph Search

Lab Exercises: Solving mazes, Path planning on maps







Notes

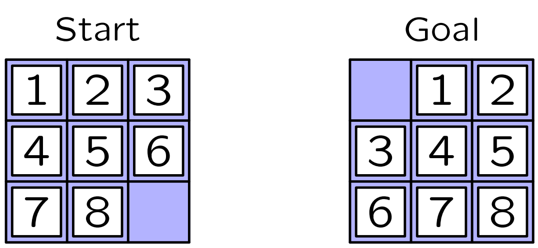
Graph Search (Path Planning)

What is a graph?

- Some set V of vertices
- Some collection E of edges connecting vertices

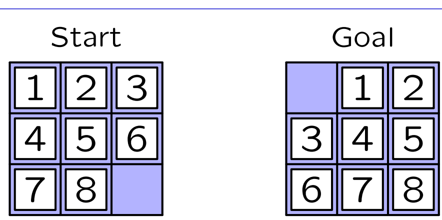
Notes

Example: 8-Puzzle



Notes

Check Yourself



How many different board configurations (states) exist?

1. $8^2 = 64$
2. $9^2 = 81$
3. $8! = 40,320$
4. $9! = 362,880$
5. None of the above

Notes

Graph Search

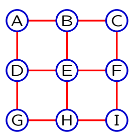
In this module:

- Develop algorithms to systematically "search" through a graph
- Analyze how well the algorithms perform
- Optimize the algorithms:
  - Find "better" paths (results)
  - Consider fewer cases (speed)
- Observe the algorithms at work in multiple contexts
  - Robot path-planning
  - Route Planning in USA
  - Language
  - Biology

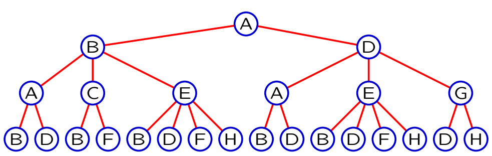
Notes

Example: Grid Search

Find path between 2 points on a rectangular grid.



Represent **all possible paths** from A with a **tree**:

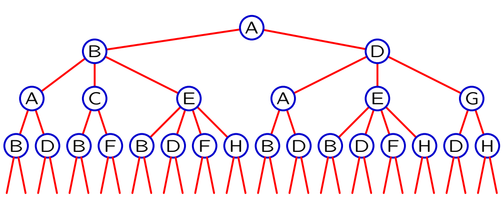


Notes

Problem?

Notice that there are infinitely many paths.

The tree is infinitely large!

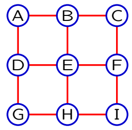


Strategy:
construct the tree **incrementally** while looking for a path

Notes

Python Representation of Grid

Represent the grid as instance of class `Grid`



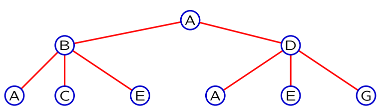

```
class Grid:
    def __init__(self, width, height, start, goal):
        self.width = width
        self.height = height
        self.start = start
        self.goal = goal

grid = Grid(3, 3, (0,0), (2,2))
```


Notes

Search Trees in Python

Represent each **node** in the tree as an instance of `SearchNode`



Note: no explicit representation for *entire* tree

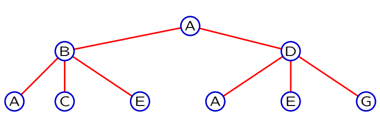
Issues:

- need to "grow" the tree as we search it
- need to reconstruct paths in tree

Notes

Search Trees in Python

Represent each **node** in the tree as an instance of `SearchNode`



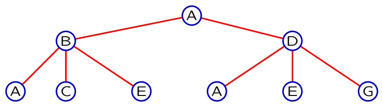

```
class SearchNode:
    def __init__(self, state, parent):
        self.state = state
        self.parent = parent

    def path(self):
        p = []
        node = self
        while node:
            p.append(node.state)
            node = node.parent
        return p[::-1]
```


Notes

Pathfinding Algorithm

Construct the tree and find a path to the goal.



Algorithm:

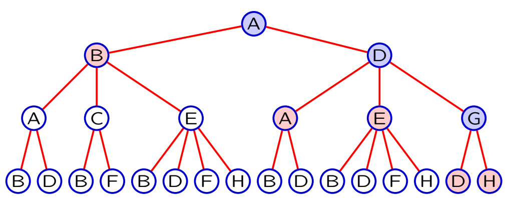
- Initialize **agenda** (list of nodes to consider)
- Repeat the following:
  - Remove one node from the agenda ("expand")
  - Add that node's successors to the agenda ("visit")

  until **goal is found** or **agenda is empty**
- Return resulting path

Notes

Order Matters!

Strategy: Replace last node in agenda by its successors



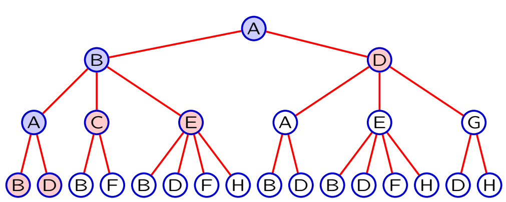
Agenda: ~~A~~ AB ~~AD~~ ADA ADE ~~ADG~~ ADGD ADGH

Depth-first Search

Notes

Order Matters!

Strategy: Replace first node in agenda by its successors



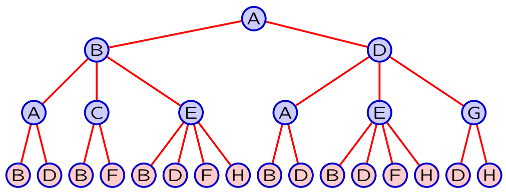
Agenda: ~~A~~ ~~AB~~ ~~ABA~~ ABAB ABAD ABC ABE AD

Still *Depth-first* Search

Notes

Order Matters!

Strategy: Remove first node and add its successors to end



Agenda: ~~A AB AD ABA ABC ABE ADA ADE ADG~~ ABAB ABAD ABCB ABCF ABEB ABED ABEF ABEH ADAB ADAD ADEB ADED ADEF ADEH ADGD ADGH

Breadth-first Search

Notes

Order Matters!

Depth-First Search (DFS):

- Push and Pop from same side of agenda
- Works down one branch of the tree before moving on to another branch

Breadth-First Search (BFS):

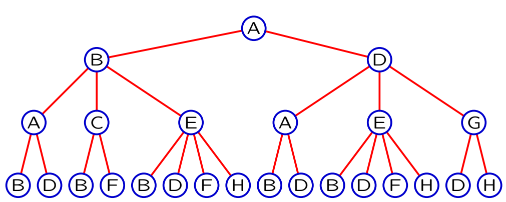
- Push and Pop from different sides of agenda
- Considers all paths of length n before considering paths of length $n+1$

Notes

Too Much Searching

Find path between 2 points on a rectangular grid.

Represent **all possible paths** with a **tree**:



But don't need to consider all nodes!

Notes

Pruning

"Prune" the tree to reduce the amount of work.

Pruning Strategy 1:
Don't consider any path the visits the same state twice.

Algorithm:

- Initialize **agenda** (list of nodes to consider)
- Repeat the following:
  - Remove one node from the agenda
  - Add each child (of that node) to the agenda **if its state is not in the parent's path**.

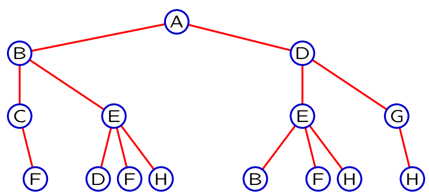
  until **goal is found** or **agenda is empty**
- Return resulting path

Notes

Pruning

"Prune" the tree to reduce the amount of work.

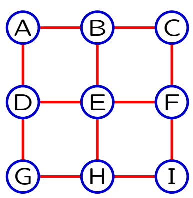
Pruning Strategy 1:
Don't consider any path that contains the same state twice.



Notes

Pruning

Under strategy 1, BFS in the 3x3 grid still visits 16 nodes...
but there are only 9 states!



We should be able to reduce the search even further.

Notes

Dynamic Programming

Basic idea behind dynamic programming:

- Break big problem into easy ones, solve and combine.
- Remember the solutions to the easy problems for later use.

Appropriate if problem has:

- *optimal substructure:* best solution is combination of optimal solutions to sub-problems
- *overlapping sub-problems:* same sub-problem occurs many times while solving overall problem

Notes

Dynamic Programming

As applies to search:
(Depends slightly on which algorithm we're using)

BFS: The shortest path $S \to X \to G$ is made up of the shortest path $S \to X$ and the shortest path $X \to G$.

DFS: A path $S \to X \to G$ is made up of a path $S \to X$ and a path $X \to G$.

The moral: once we have found a path $S \to X$, we don't need to spend time looking for other paths through X.

Said another way: Many paths that include $S \to X$, but don't need to recompute while exploring rest of path (*memoization*); and once have a satisfactory path $S \to X$, don't need to keep looking for others (*dynamic programming*).

Notes

Dynamic Programming

As applied to graph search: Don't consider any path that visits a state that you have already visited via some other path.

Need to remember which states we have visited to avoid visiting them again.

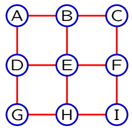
Algorithm:

- Initialize **visited set**
- Initialize **agenda** (list of nodes to consider)
- Repeat the following:
  - Remove one node from the agenda
  - Add each child (of that node) to the agenda **if its state is not already in the visited set**, and **add each of these new states to the visited set**

  until **goal is found** or **agenda is empty**
- Return resulting path

Notes

Check Yourself!



Consider a breadth-first search with dynamic programming, from A to I. How many states are visited?

1. 2
2. 4
3. 6
4. 8
5. 10

Notes

Search in `lib601`

`lib601` procedure called `search`, takes arguments:

- *successors*: function that takes a state and returns a list of successor states
- *start_state*: the state from which to start the search
- *goal_test*: a function that takes a state and returns `True` if that state satisfies the goal condition, and `False` otherwise
- *dfs*: boolean; if `True`, run a depth-first search; if `False`, run a breadth-first search

`search` returns a list of states from the root of the tree to the goal, or `None` if no path exists.

Notes

Casting Problems as Search Problems

Biggest issue is choice of **state**.

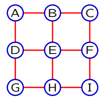
From the state, we must be able to:

- Determine successors
- Test for goal condition

We'll get a lot of practice with this during the labs and homeworks this week.

Notes

Example: Grid Search




```
class Grid:
    def __init__(self, width, height, start, goal):
        self.width = width
        self.height = height
        self.start = start
        self.goal = goal

grid = Grid(3, 3, (0,0), (2,2))

def grid_successors(state):
    r,c = state
    out = []
    for (dr,dc) in [(0,1),(1,0),(0,-1),(-1,0)]:
        if 0<=(r+dr)<grid.height and 0<=(c+dc)<grid.width:
            out.append((r+dr,c+dc))
    return out

result = search(grid_successors, grid.start, lambda x: x==grid.goal, False)
```


Notes

Recap

Developed two search algorithms:

- Breadth-first search
- Depth-first search

Discussed the benefits and drawbacks of each

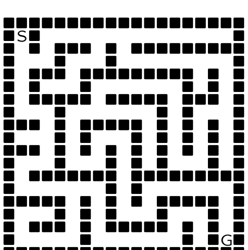
Developed two pruning rules:

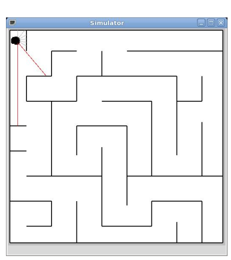
- Don't consider paths that revisit states
- Only consider the first path to a given state

Notes

Labs This Week

Software Lab: Solving Mazes
Design Lab: Robots in Mazes





Notes