CS 232: Artificial Intelligence

Spring 2024

Prof. Carolyn Anderson
Wellesley College

Reminders

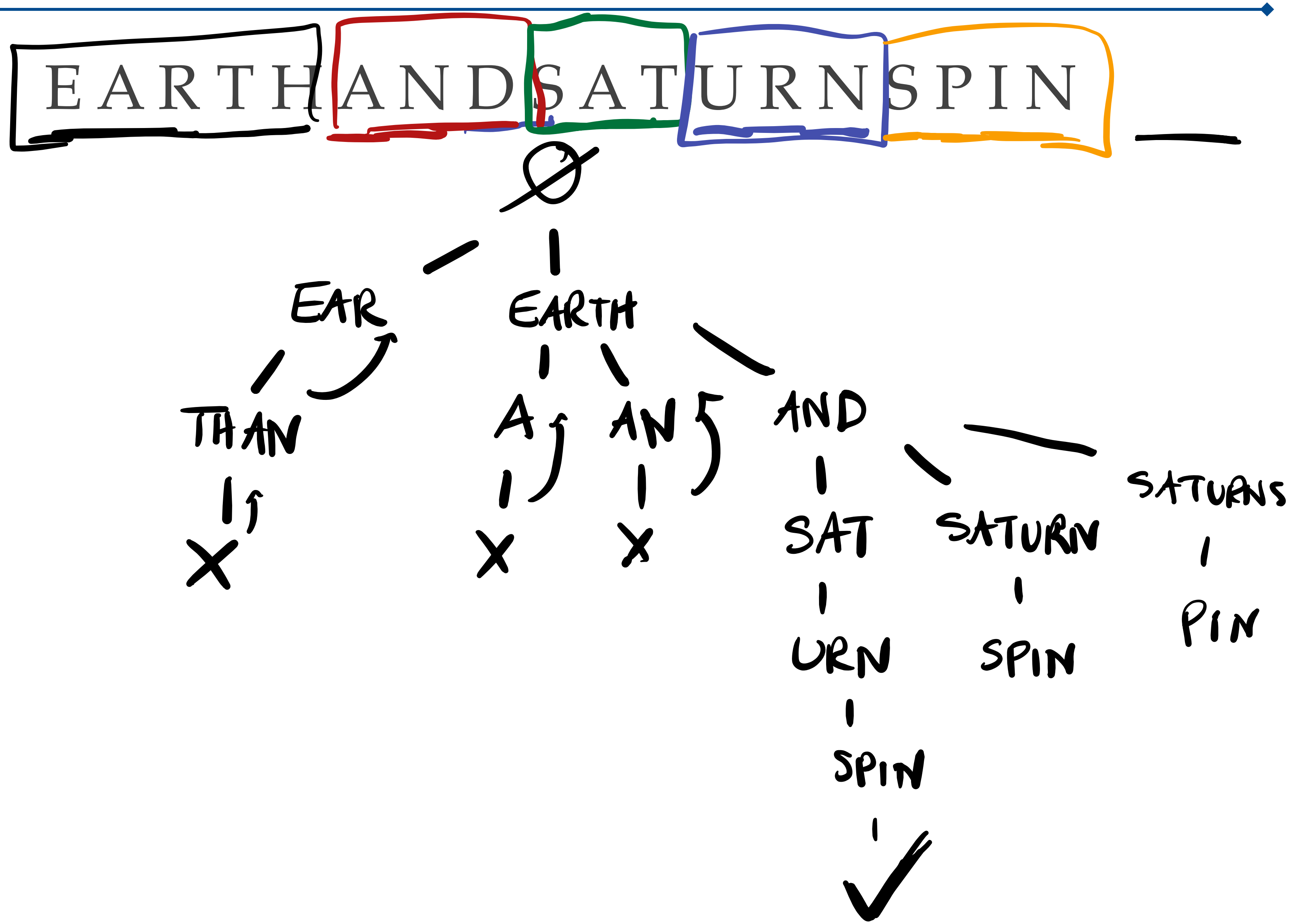
- Homework 2 is due Monday
- I have help hours Friday from 3:30-4:30pm
- Lyra has help hours Sunday from 4-6 on Zoom
- I have help hours Monday from 4-5:15
- Reading for next Tuesday: YLLATAILY Chapter 3-4

Recap

Evaluating Solvers

- **Completeness**: Is the algorithm guaranteed to find a solution when there is one?
- **Optimality**: Does the strategy find the optimal solution?
- **Time complexity**: How long does it take to find a solution?
- **Space complexity**: How much memory is needed to perform the search?

Backtracking Search Application



Search Algorithms

Basic search algorithms: *Tree Search*

Generalized algorithm to solve search problems

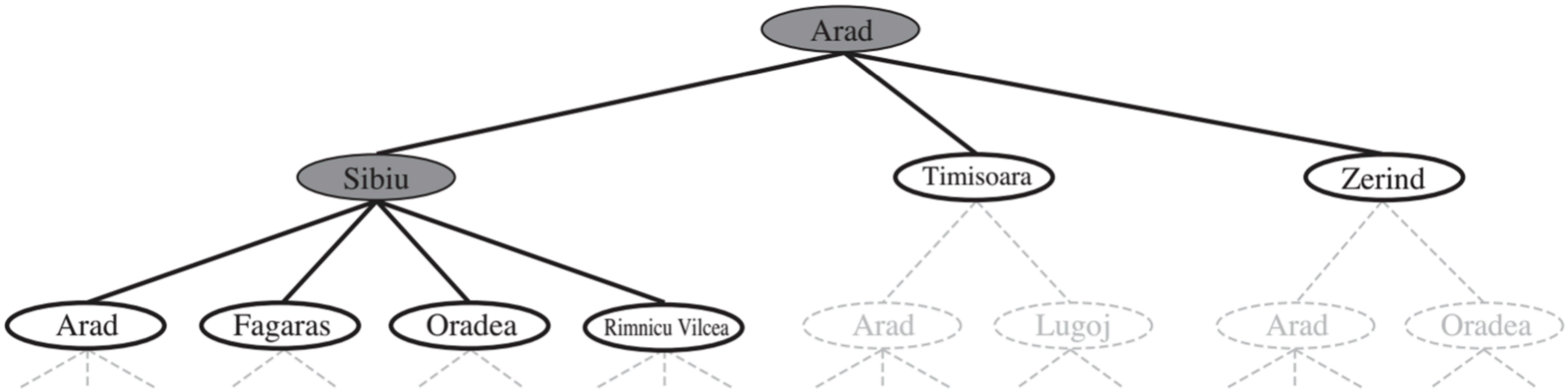
Enumerate in some order all possible paths from the initial state

Root = initial state

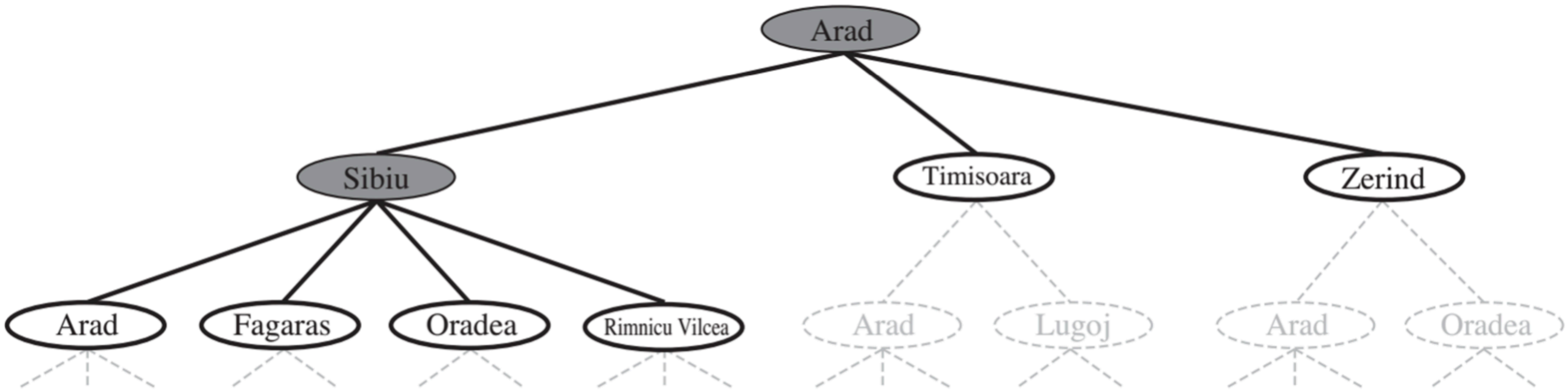
Nodes in search tree generated by the transition models

Treat different paths to the same node as distinct

Generalized tree search



Generalized tree search



The strategy determines search process!

function TREE-SEARCH(*problem, strategy*) return a solution or failure

Initialize frontier to the *initial state* of the *problem*

do

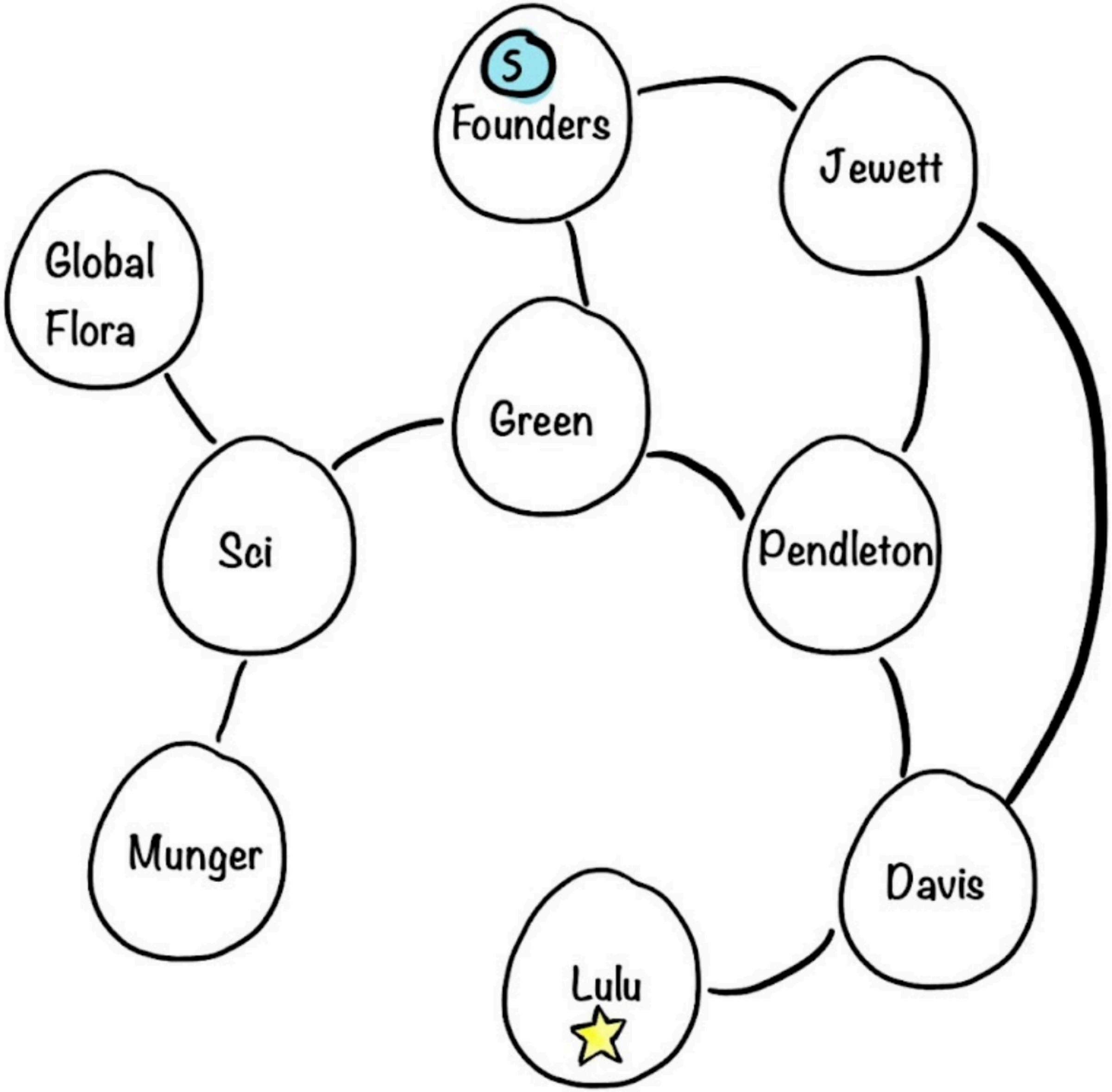
if the frontier is empty then return *failure*

choose leaf node for expansion according to strategy & remove from frontier

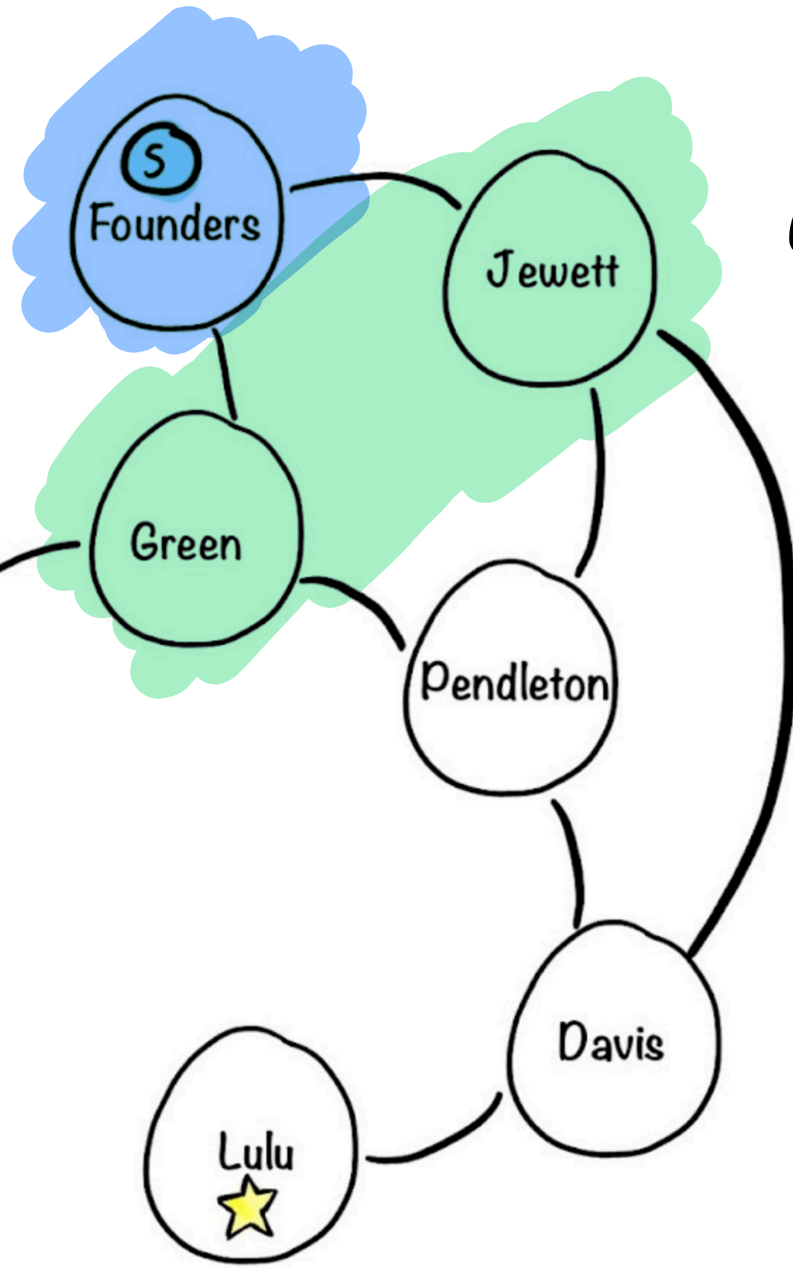
if node contains goal state then return *solution*

else expand the node and add resulting nodes to the frontier

Slides adapted from Chris Callison-Burch


S
Founders
Jewett
Global
Flora
Green
Sci
Pendleton
Munger
Davis
Lulu

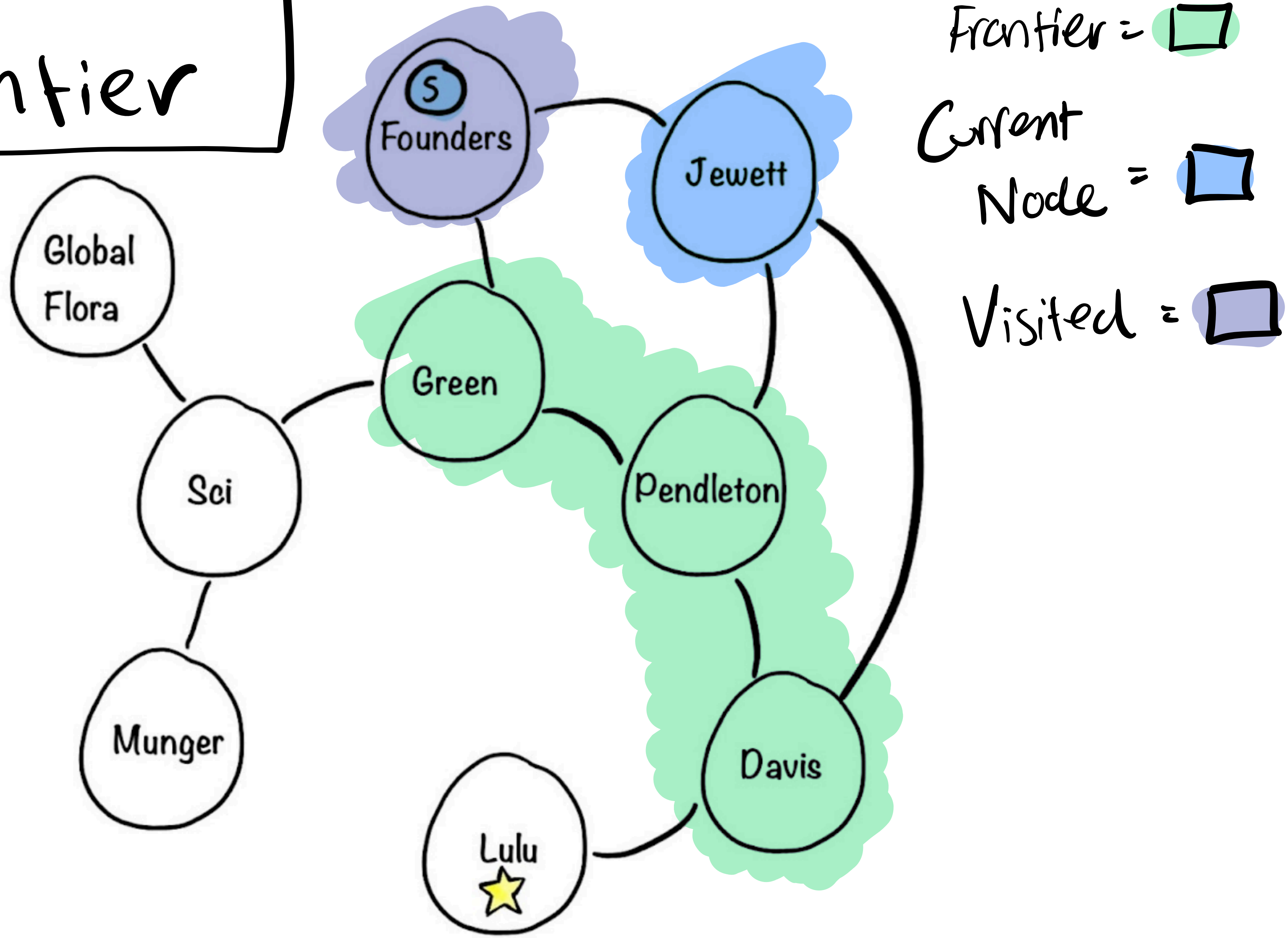
The Frontier



Frontier =

Current Node =

The Frontier



States Versus Nodes

A state is a physical configuration
a representation of the environment

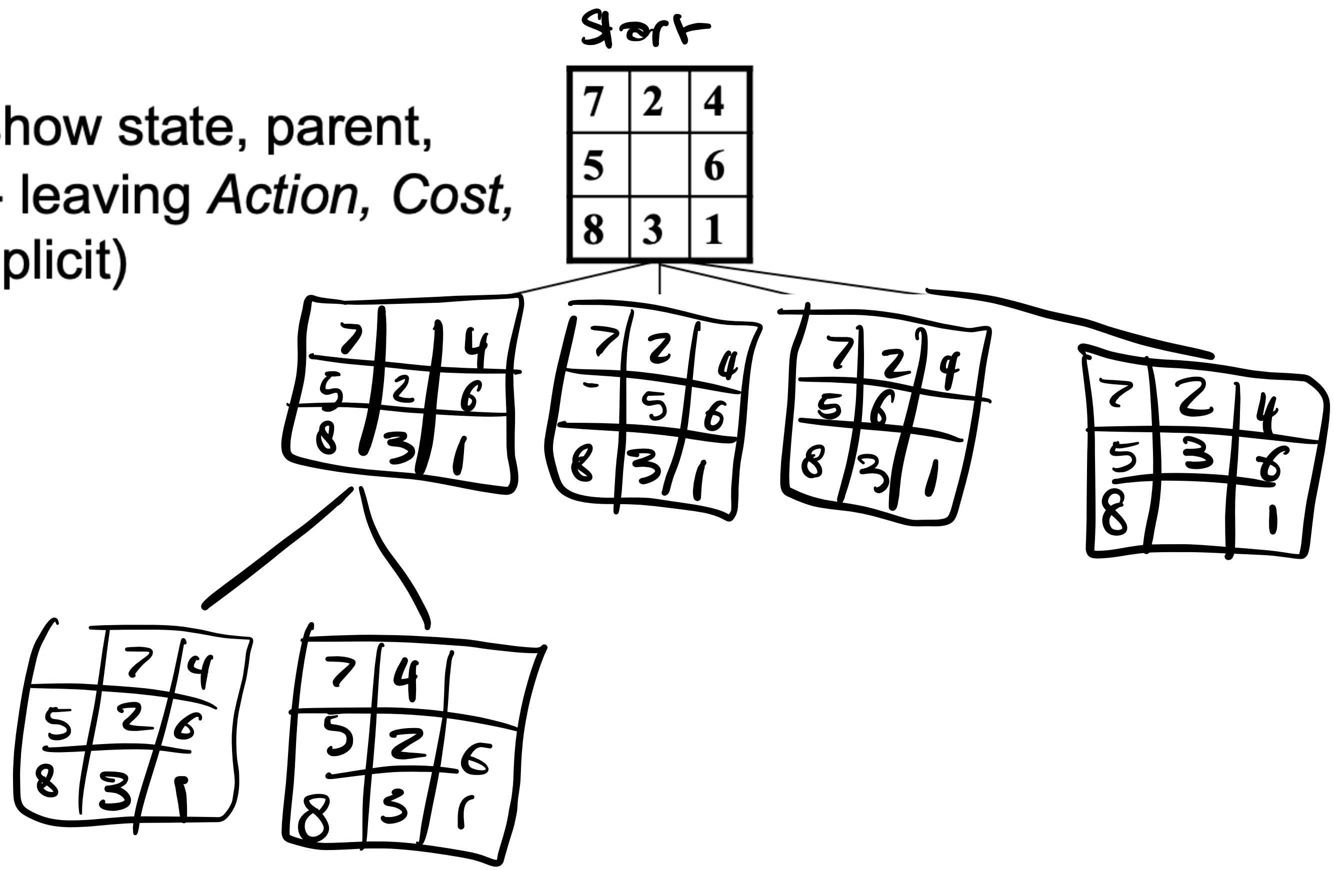
A node is a data structure

Node : < state, parent-node, children,
action, path-cost, depth >

States don't have cost or parents or depth

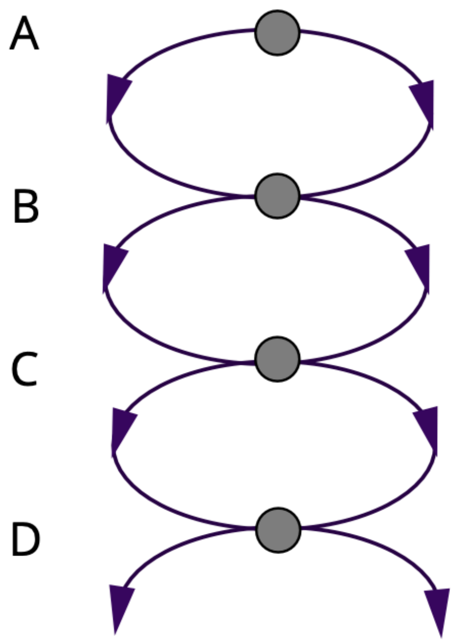
8-Puzzle *Search Tree*

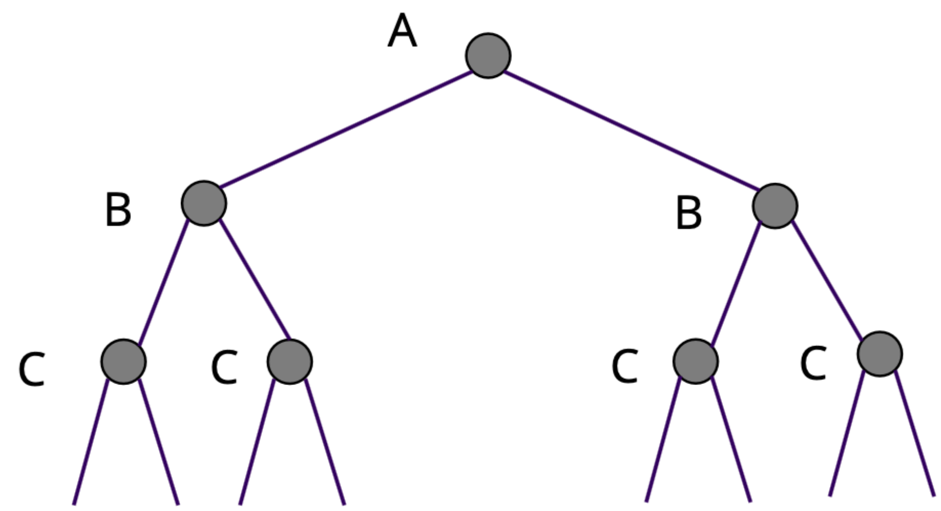
(Nodes show state, parent, children - leaving *Action, Cost, Depth* Implicit)



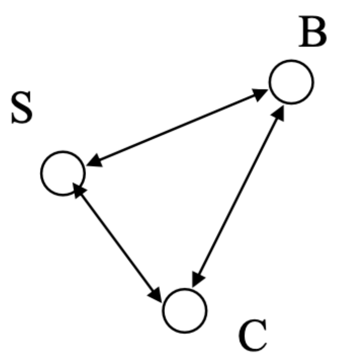
Problem: Repeated states

Failure to detect ***repeated states*** can turn a linear problem into an ***exponential*** one!

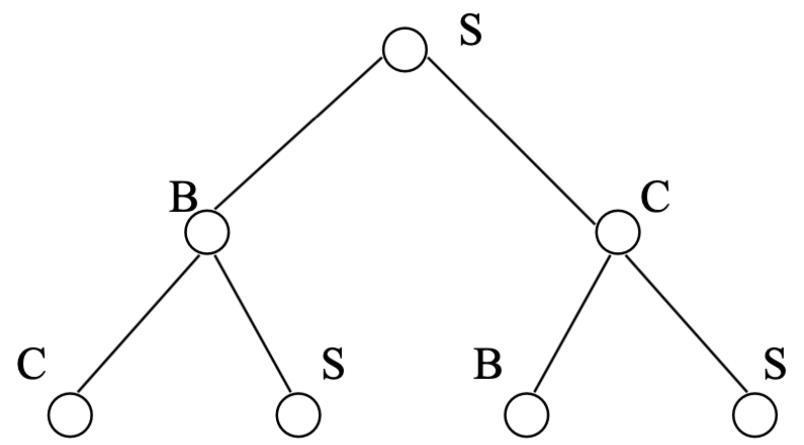




Solution: Graph Search!



State Space



Search Tree

Graph search

- Simple Mod from tree search: *Check to see if a node has been visited before adding to search queue*
  - must keep track of all possible states (can use a lot of memory)
  - e.g., 8-puzzle problem, we have 9!/2 ≈182K states

Graph Search vs Tree Search

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 nose 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 node to the explored set
expand the chosen node, adding the resulting nodes to the frontier
only if not in the frontier of explored set

Slides adapted from Chris Callison-Burch

Uninformed Search

Uninformed Search

Uses only information available in problem definition

Informally:

Uninformed search: All non-goal nodes in frontier look equally good

Informed search: Some non-goal nodes can be ranked above others.

Breadth-First Search

Breadth-first search

Idea:

- Expand *shallowest* unexpanded node

Implementation:

- *frontier* is FIFO (First-In-First-Out) Queue:
  - Put successors at the *end* of *frontier* successor list.

Breadth-first search

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure
```

Position within queue of new items determines search strategy

Breadth-first search

```
function EXPAND(problem, node) yields nodes
    s ← node.STATE
    for each action in problem.ACTIONS(s) do
        s′ ← problem.RESULT(s, action)
        cost ← node.PATH-COST + problem.ACTION-COST(s, action, s′)
        yield NODE(STATE=s′, PARENT=node, ACTION=action, PATH-COST=cost)
```

Node data structure contains variables like the state, a pointer to its parent node, the action that was used to create this state, and the path cost.

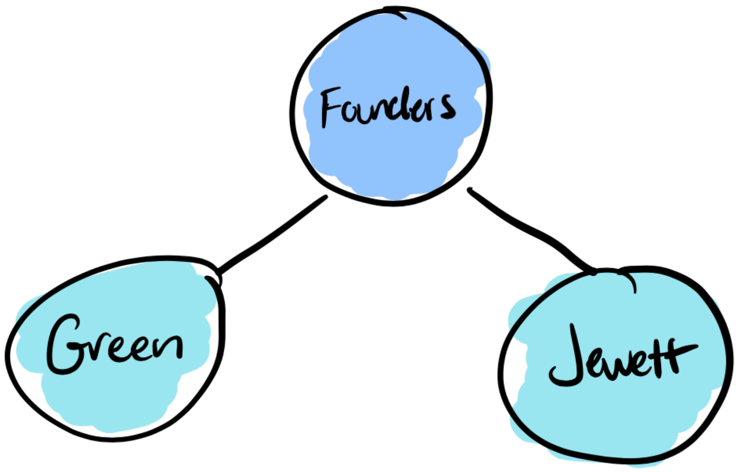
The Python yield keyword means that we don't have to pre-compute a list of all successors.

Slides adapted from Chris Callison-Burch

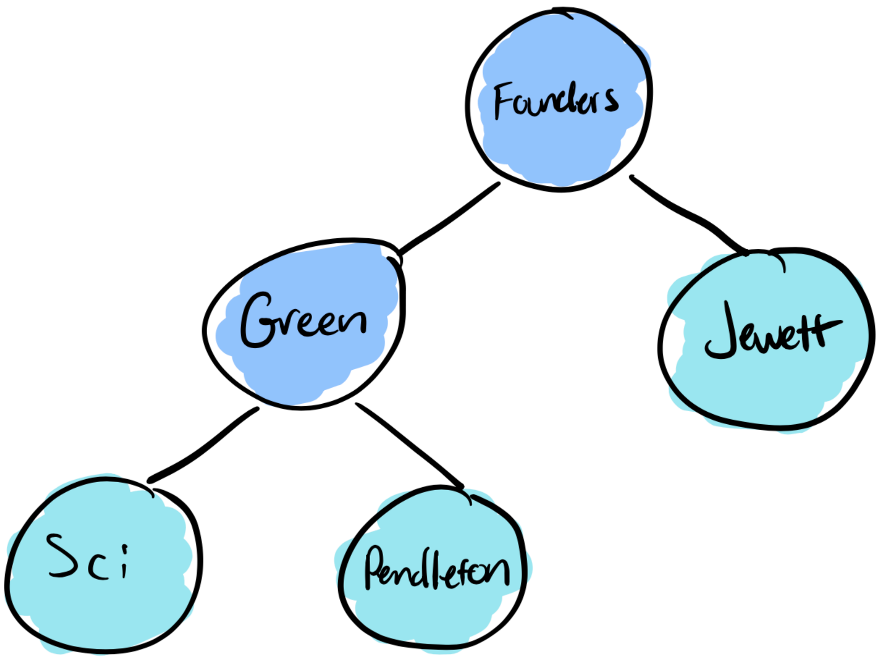
Breadth-first search

function BREADTH-FIRST-SEARCH(*problem*) **returns** a solution node or *failure*
    node ← NODE(*problem*.INITIAL)
    if *problem*.IS-GOAL(*node*.STATE) **then return** *node*
    frontier ← a FIFO queue, with *node* as an element
    reached ← {*problem*.INITIAL}
    while not IS-EMPTY(*frontier*) **do**
        node ← POP(*frontier*)
        for each *child* **in** EXPAND(*problem*, *node*) **do**
            s ← *child*.STATE
            if *problem*.IS-GOAL(*s*) **then return** *child*
            if *s* is not in *reached* **then**
                add *s* to *reached*
                add *child* to *frontier*
    return *failure*

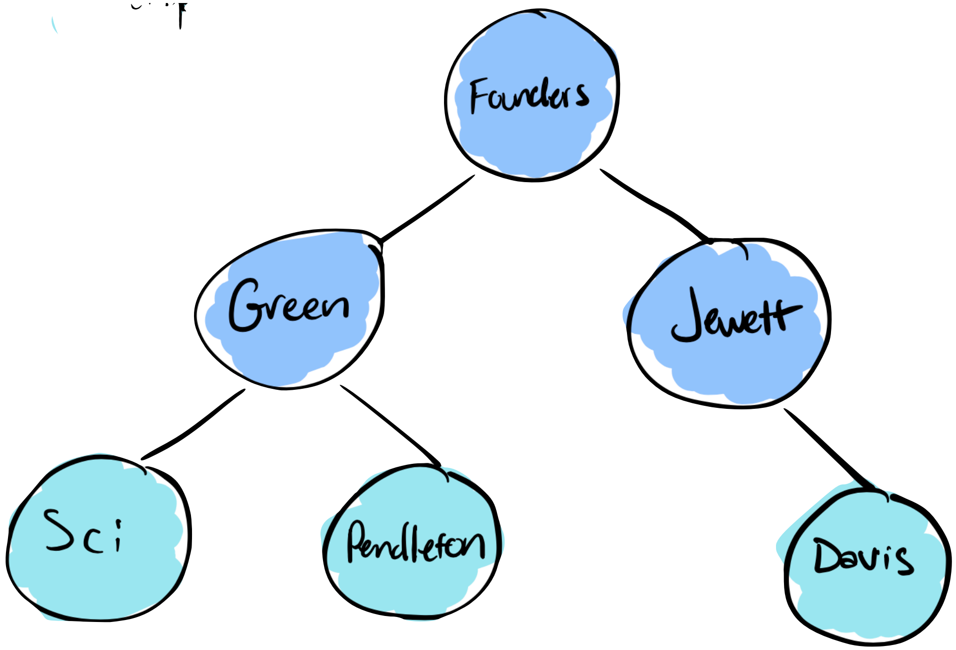
Subtle: *Node inserted into queue only after testing to see if it is a goal state*



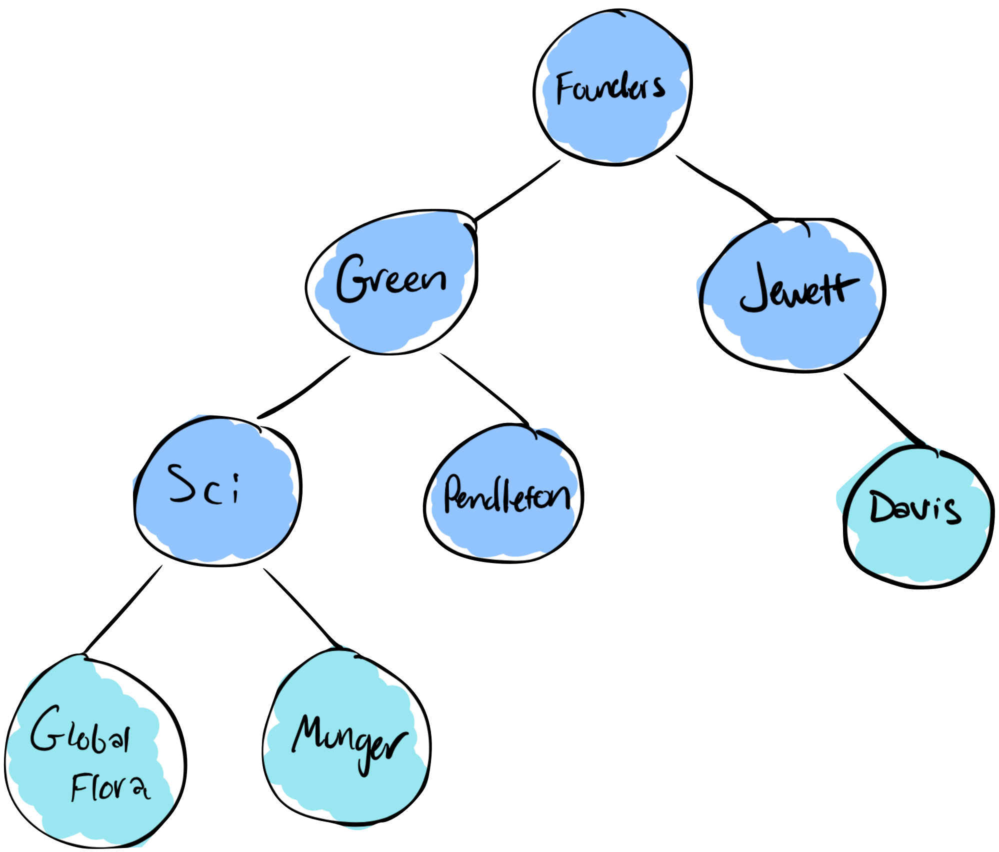
Frontier	Visited
Green	Founders
Jewett



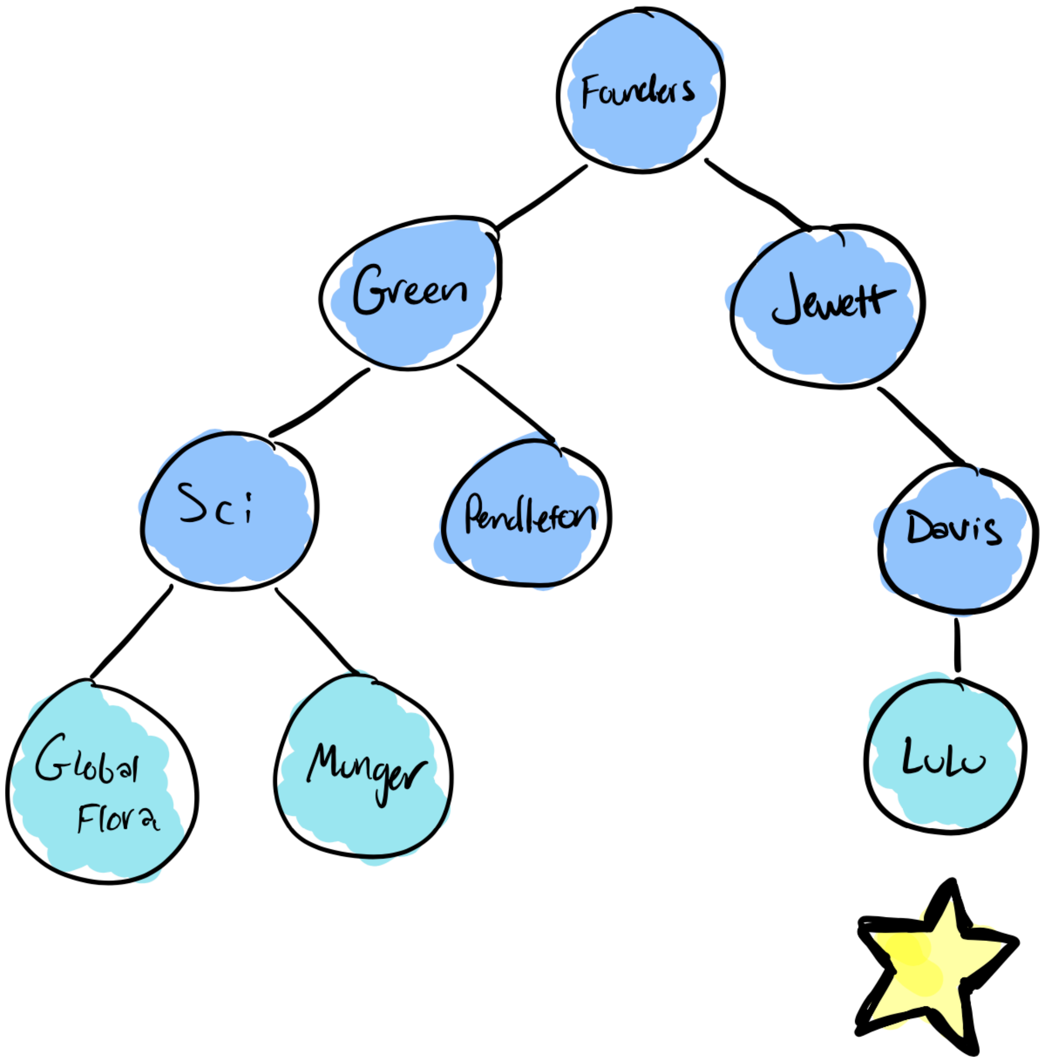
Frontier	Visited
~~Green~~	Founders
Jewett	Green
Pendleton	
Sci	



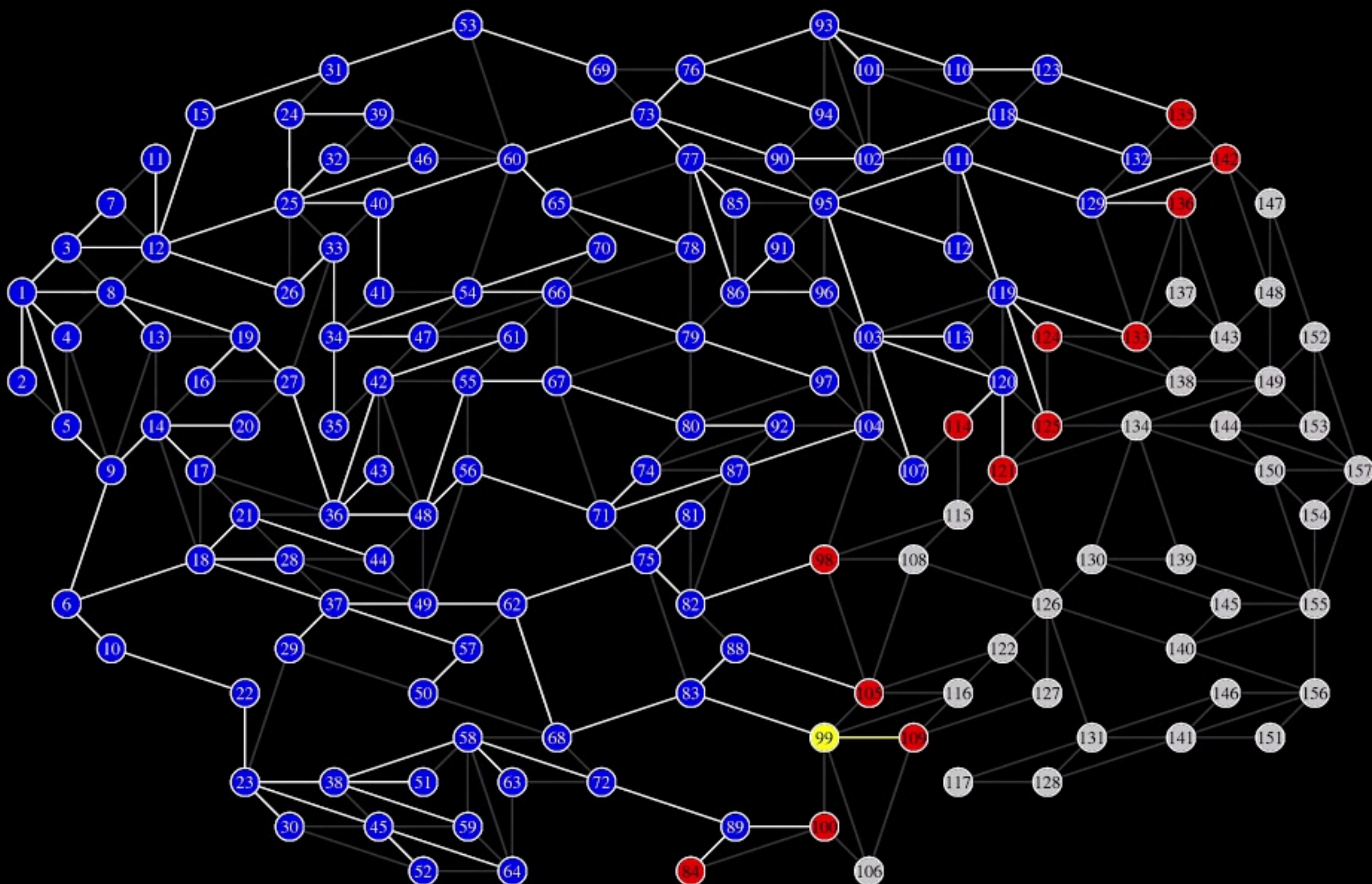
Frontier	Visited
~~Green~~	Founders
~~Jewett~~	Green
Pendleton	Jewett
Sci	
Davis	

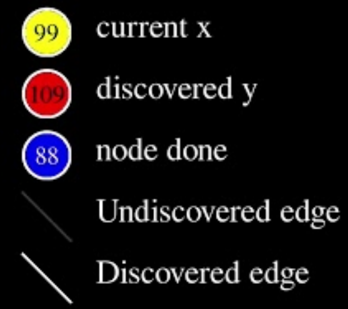


Frontier	Visited
~~Green~~	Founders
~~Jewett~~	Green
~~Pendleton~~	Jewett
~~Sci~~	Pendleton
Davis	Sci
Global Flora	
Munger	



Frontier	Visited
~~Green~~	Founders
~~Jewett~~	Green
~~Pendleton~~	Jewett
~~Sci~~	Pendleton
~~Davis~~	Sci
Global Flora	Davis
Munger	
LuLu	






```
bfs(x):
  make a new queue called q
  mark x visited
  push x onto q

  while q not empty:
    pop q into x
    for each y in x connections
    if y not visited:
      mark y visited
      push y onto q
```

Properties of breadth-first search

Complete? Yes

Optimal? Yes

Time Complexity? $O(b^d)$

Space Complexity? $O(b^d)$

b: maximum branching factor of search tree

d: depth of the least cost solution (shortest path to goal)

Exponential Space (and time) Is Not Good...

- Exponential complexity uninformed search problems *cannot* be solved for any but the smallest instances.
- *(Memory* requirements are a bigger problem than *execution* time.)

DEPTH	NODES	TIME	MEMORY
2	*110*	*0.11 milliseconds*	*107 kilobytes*
4	*11110*	*11 milliseconds*	*10.6 megabytes*
6	10^6	*1.1 seconds*	*1 gigabytes*
8	10^8	*2 minutes*	*103 gigabytes*
10	10^{10}	*3 hours*	*10 terabytes*
12	10^{12}	*13 days*	*1 petabytes*
14	10^{14}	*3.5 years*	*99 petabytles*

Assumes b=10, 1M nodes/sec, 1000 bytes/node

Depth-First Search

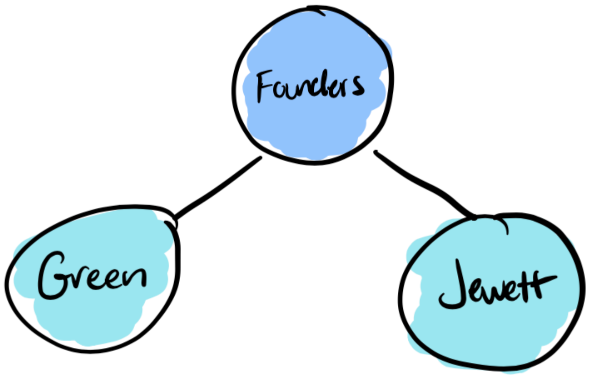
Depth-first search

Idea:

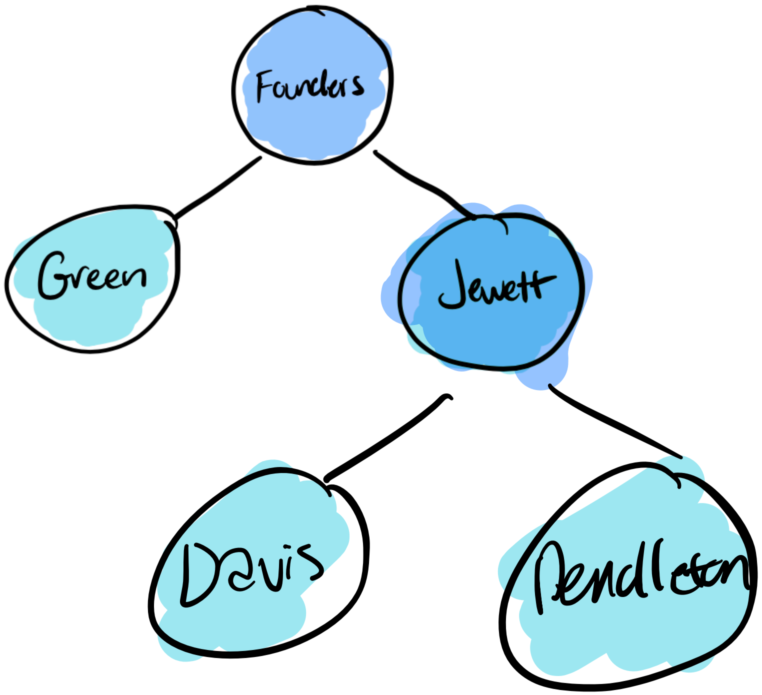
- Expand *deepest* unexpanded node

Implementation:

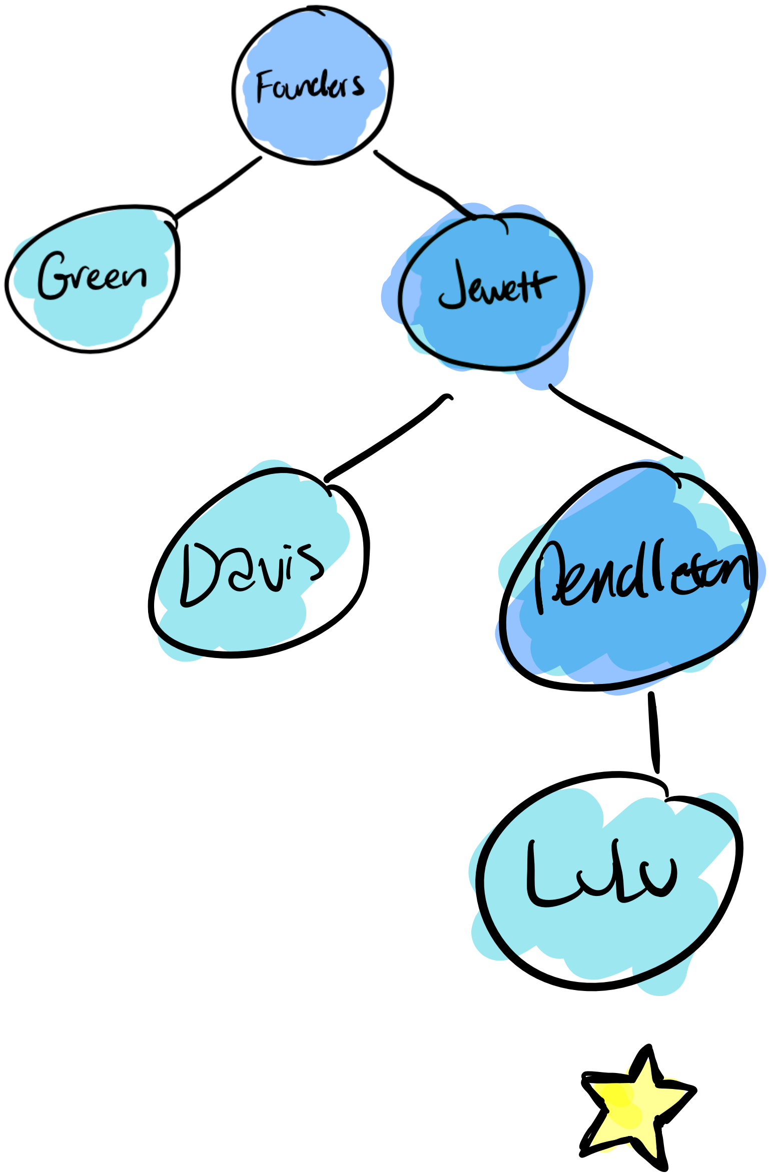
- *frontier* is LIFO (Last-In-First-Out) ~~Queue~~ Stack:
  - Put successors at the *front* of *frontier* successor list.



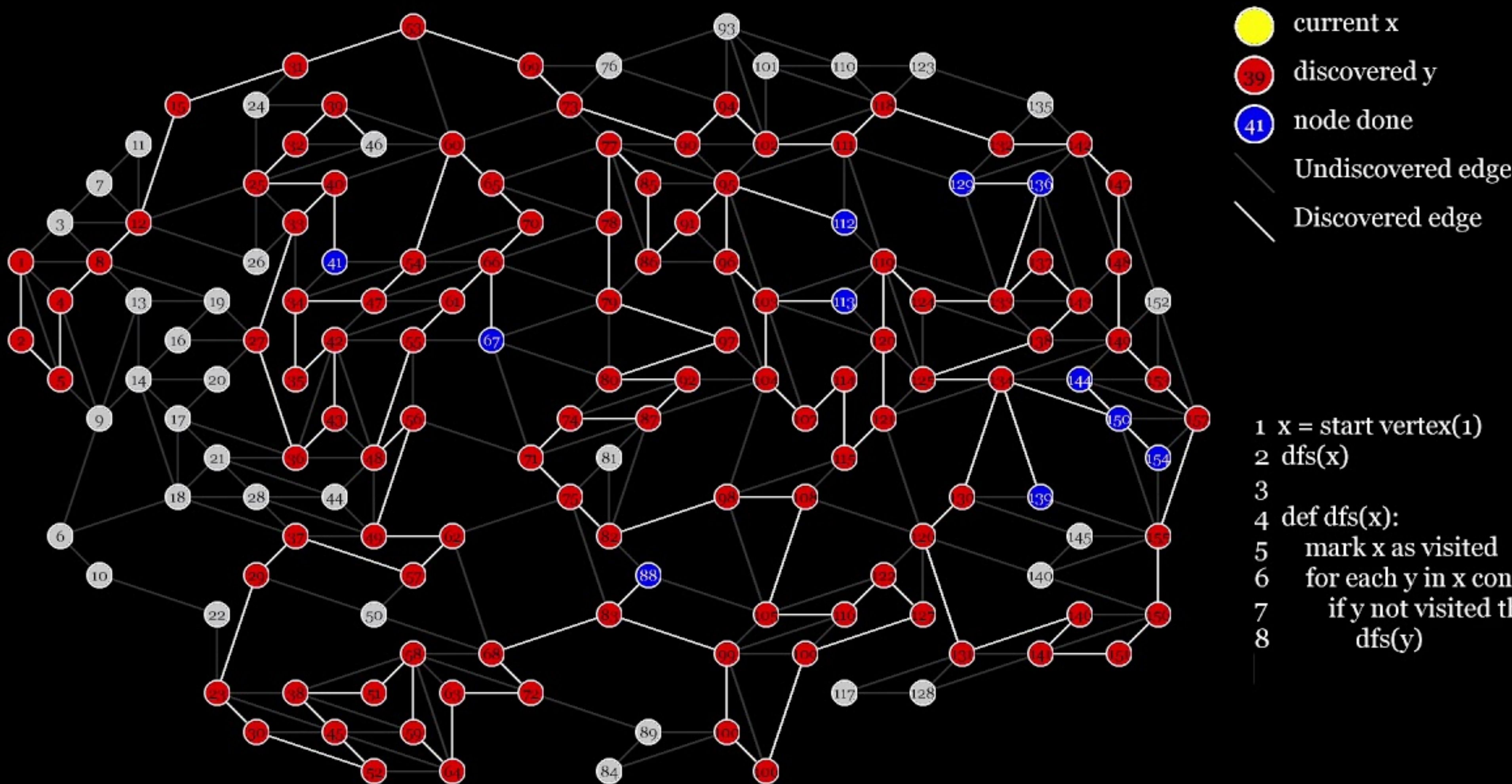
Frontier	Visited
Green	Founders
Jewett	



Frontier	Visited
Green	Founders
~~Jewett~~	Jewett
Davis	
Pendleton	



Frontier	Visited
Green	Founders
~~Jewett~~	Jewett
Davis	Pendleton
~~Pendleton~~	
Lulu	




```
x = start vertex(1)
dfs(x)

def dfs(x):
    mark x as visited
    for each y in x connections:
        if y not visited then
            dfs(y)
```

Properties of depth-first search

Complete? Yes*

Optimal? No

Time Complexity? $O(b^m)$

Space Complexity? $O(b*m)$

b = maximum branching factor (how many children?)

d = depth of least cost solution

m = maximum depth of the search tree

Slides adapted from Chris Callison-Burch

Depth-first vs Breadth-first

Use depth-first if

- *Space is restricted*
- There are many possible solutions with long paths and wrong paths are usually terminated quickly
- Search can be fine-tuned quickly

Use breadth-first if

- *Possible infinite paths*
- Some solutions have short paths
- Can quickly discard unlikely paths

Search Conundrum

Breadth-first

- ☑ Complete,
- ☑ Optimal
- ☒ *but* uses $O(b^d)$ space

Depth-first

- ☒ Not complete *unless m is bounded*
- ☒ Not optimal
- ☒ Uses $O(b^m)$ time; terrible if m >> d
- ☑ *but* only uses *O(**b*m) space***

Depth-limited search: A building block

Depth-First search *but with depth limit l.*

- i.e. nodes at depth l *have no successors.*
- No infinite-path problem!

If $l = d$ (by luck!), then optimal

- But:
  - **If $l < d$ then incomplete** ☹️
  - **If $l > d$ then not optimal** ☹️

Time complexity: $O(b^l)$

Space complexity: $O(bl)$ ☺️