CS 232: Artificial Intelligence

Fall 2023

Prof. Carolyn Anderson
Wellesley College

Reminders

- Homework 2 will be released today
- I have help hours Thursday from ~~4~~ 4:30 - 5:30pm
- Reading for next Tuesday: YLLATAILY Chapter 3-4
- Tutor help hours start next week!

YLLATAILY Chapters 1-2

Big Ideas

Rule-based programming

- Pro: we understand the rules the program is using
- Con: we have to write the rules

Supervised learning

- Pro: AI generates its own rules
- Con: hard to understand *why* it's doing what it's doing

Big Ideas

Signs of AI Doom:

- The problem is too hard
- The problem is not what we thought it was
- There are sneaky shortcuts
- The AI tried to learn from garbage data

Big Ideas

AI Weaknesses

- Remembering things
- Planning ahead
- Data- and computation-intensive

Example Tasks

inability to adapt

- Self-driving cars
- Recipe generation — how does it know what tastes good?
- Résumé screening — possible bias from training data
  — keywords: easy but not useful
- Cockroach farming
- Tic-tac-toe — sneaky shortcuts
- Image recognition — sneaky shortcuts + bad data
  learning weird correlations
- Joke generation
- Super Mario
- Writing news articles

Recap

Rational agents

Given a goal, an AI agent must decide what the best action to take is in order to reach this goal.

For complex tasks, this can mean:

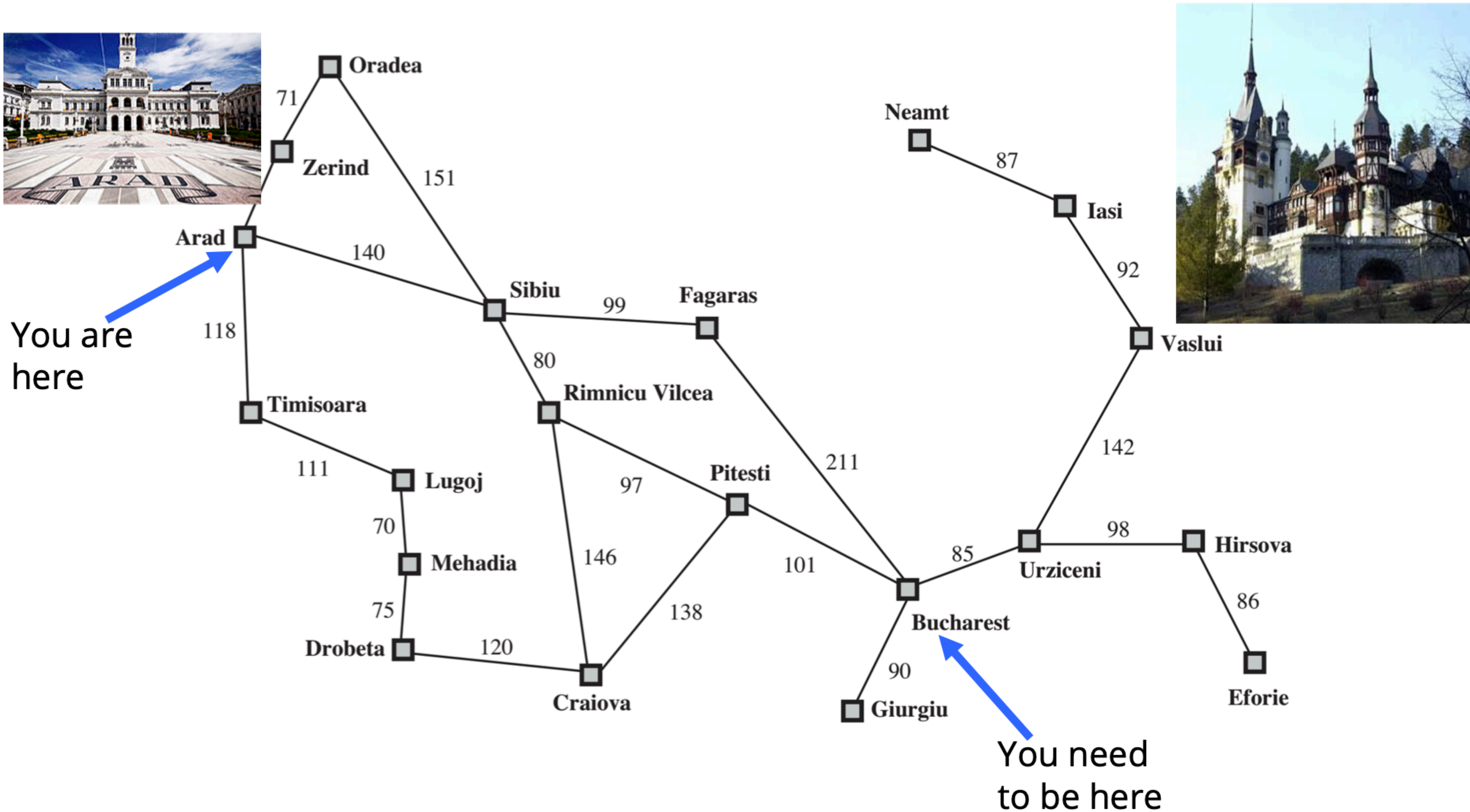
- gathering information
- coming up a set of possible actions
- weighing the best action
- acting
- updating and adapting based on changes to the environment

Agent Complexity

Problem-solving agent: capable of considering a sequence of actions that form a path to a goal state (planning ahead).

Search

Example search problem: Holiday in Romania



Holiday in Romania

On holiday in Romania; currently in Arad

- Flight leaves tomorrow from Bucharest

Formulate ***goal***

Be in Bucharest

goal state: Bucharest

Formulate ***search problem***

States: cities

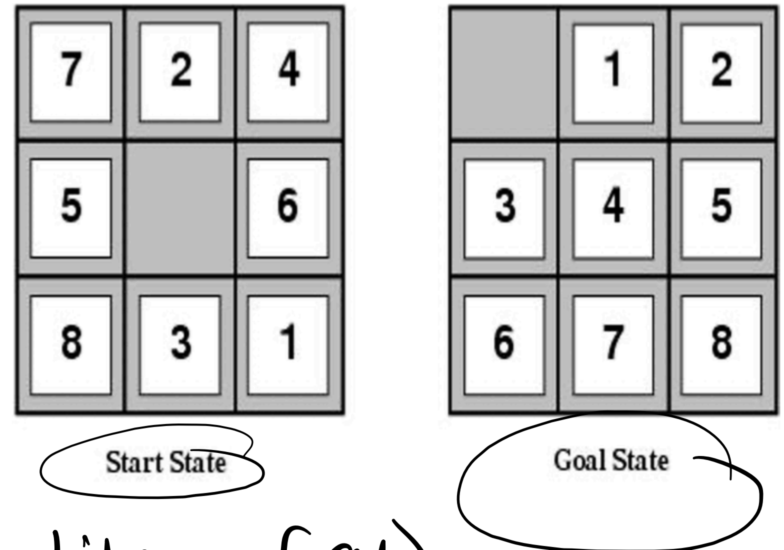
Actions: driving between cities

Cost: distance (between cities)

Find ***solution***

Sequence of cities

Example search problem: 8-puzzle



Formulate *goal*

state where tiles are in order (as shown on right)

Formulate *search problem*

states: ways to arrange tiles (9!)

Actions: move a tile: up, down, left, right

Cost: # of moves

Find *solution*

sequence of pieces moved w/ the direction they move

Search Algorithms

Basic search algorithms: *Tree Search*

Generalized algorithm to solve search problems

Enumerate in some order all possible paths from the initial state

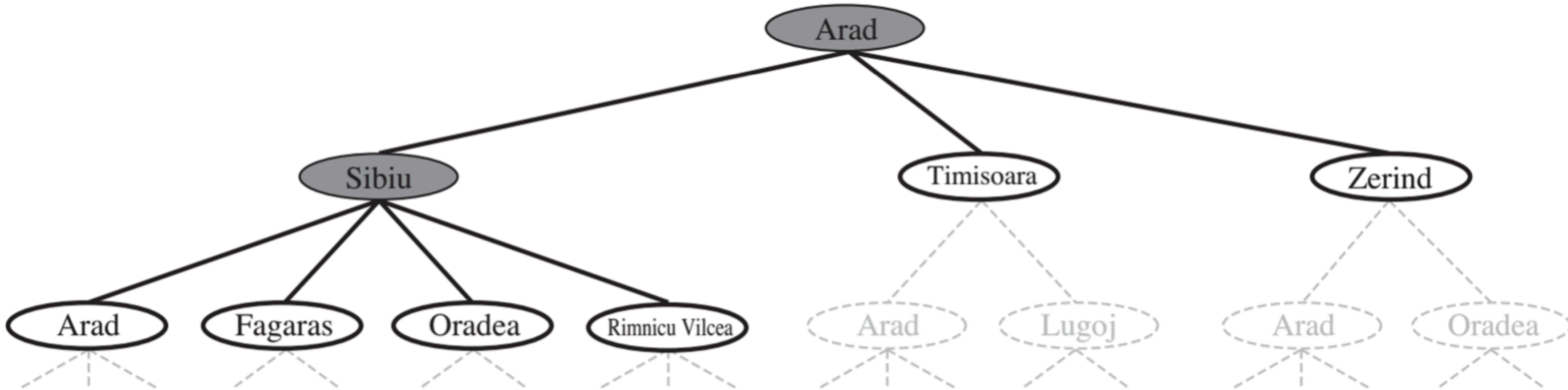
Search through explicit tree generation

Root: initial state

Nodes: states (generated through transition model)

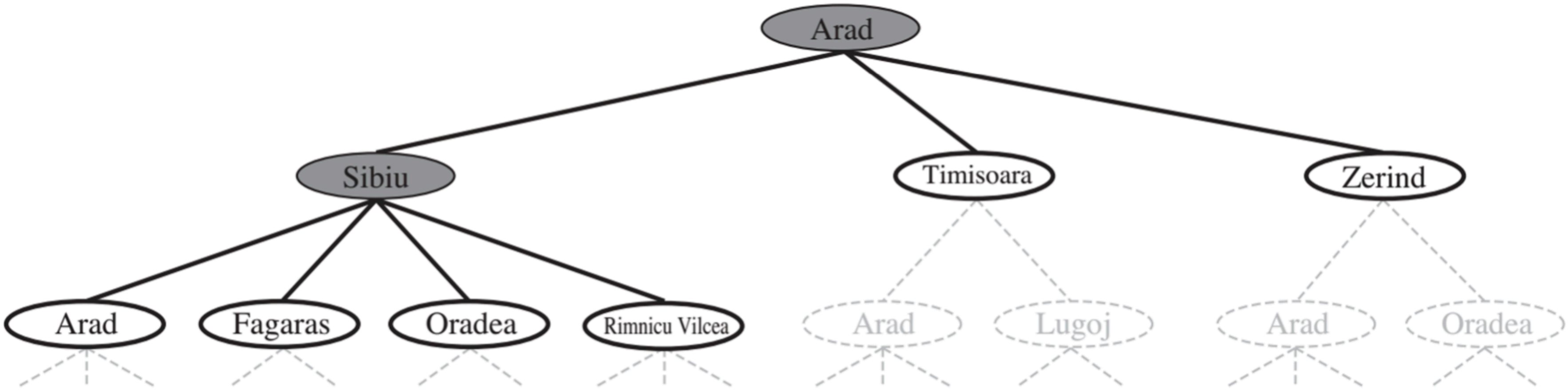
Tree search treats different paths to the same state (node) as distinct

Generalized tree search



initialize frontier to the start state
do
    if the frontier is empty then return failure
    choose the next node to expand according to strategy
    if node is goal, return solution
    else expand the node
        for each child:
            if child has not been visited, add to frontier

Generalized tree search



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

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

do

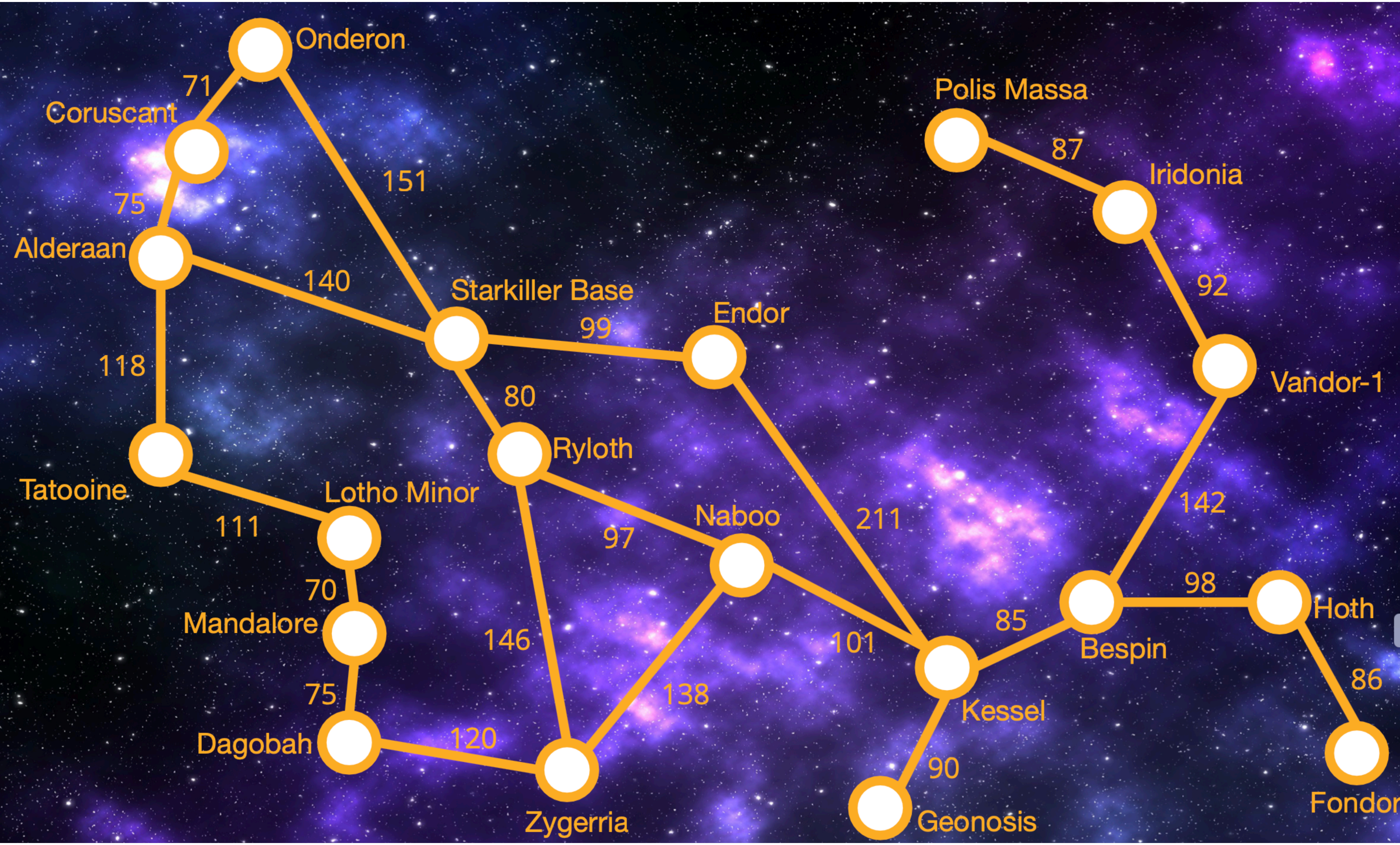
if the frontier is empty then return *failure*

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

if node contains goal state then return *solution*

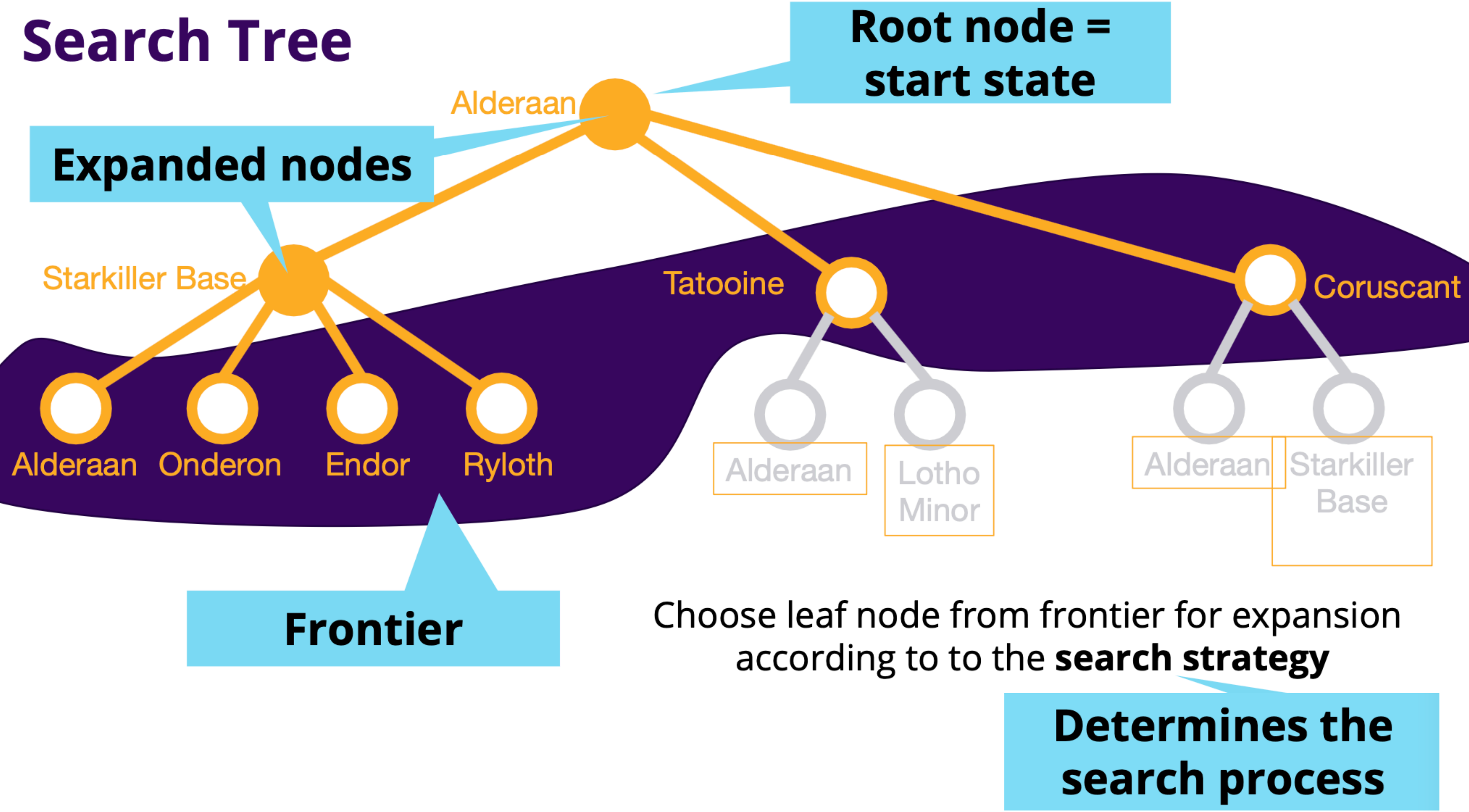
else expand the node and add resulting nodes to the frontier

The strategy determines search process!



Slides adapted from Chris Callison-Burch

Search Tree



Choose leaf node from frontier for expansion according to to the **search strategy**

States Versus Nodes

State: representation of a physical configuration of the environment

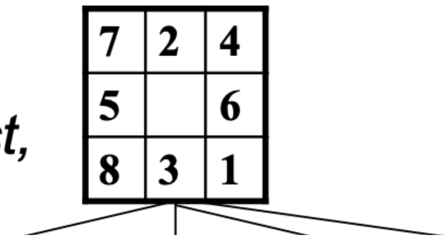
Node: a data structure with several fields:

< state, parent-node, children, action, cost, depth >

States don't have costs, parents, or depths!

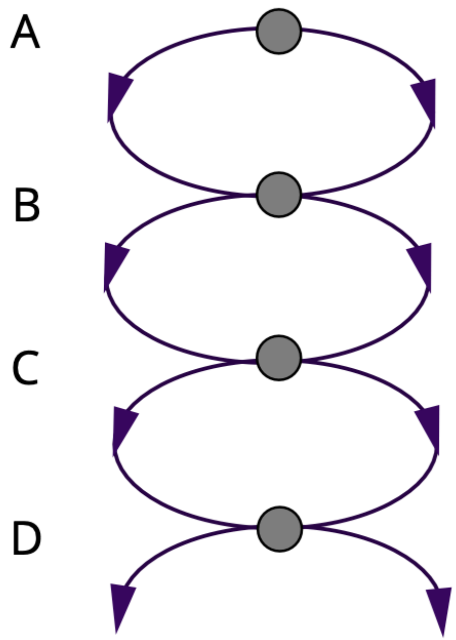
8-Puzzle *Search Tree*

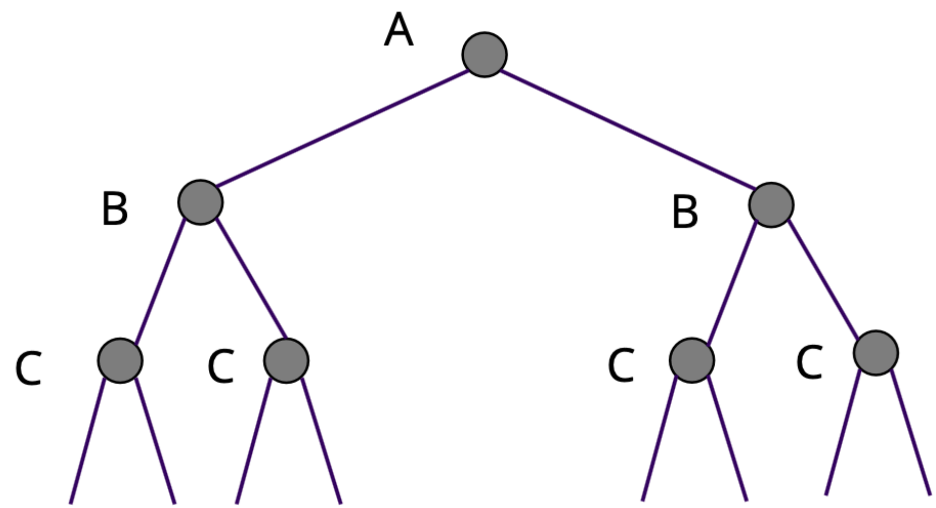
(Nodes show state, parent, children - leaving *Action, Cost, Depth* Implicit)



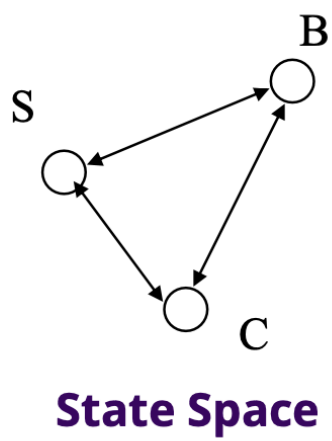
Problem: Repeated states

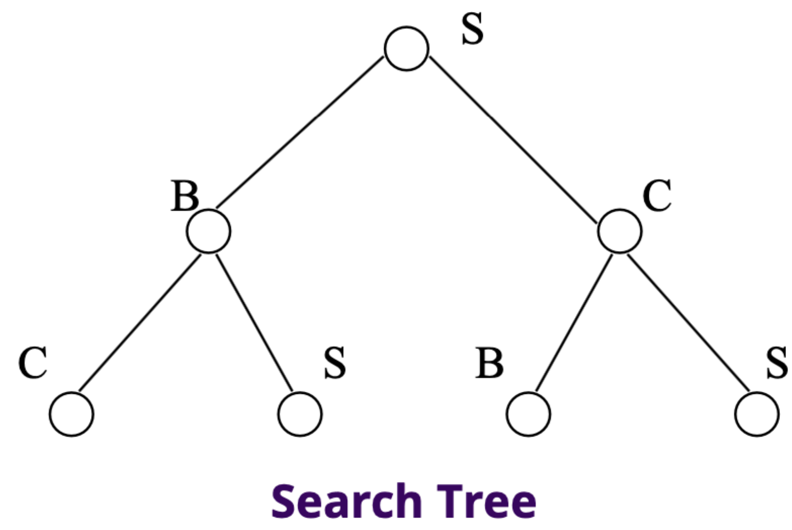
Failure to detect ***repeated states*** can turn a linear problem into an ***exponential*** one!





Solution: Graph Search!





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
  $s \leftarrow$ *node*.STATE
  for each *action* **in** *problem*.ACTIONS(s) **do**
    $s' \leftarrow$ *problem*.RESULT(s, *action*)
    cost $\leftarrow$ *node*.PATH-COST + *problem*.ACTION-COST(s, *action*, s')
    yield NODE(STATE=s', PARENT=*node*, ACTION=*action*, PATH-COST=*cost*)

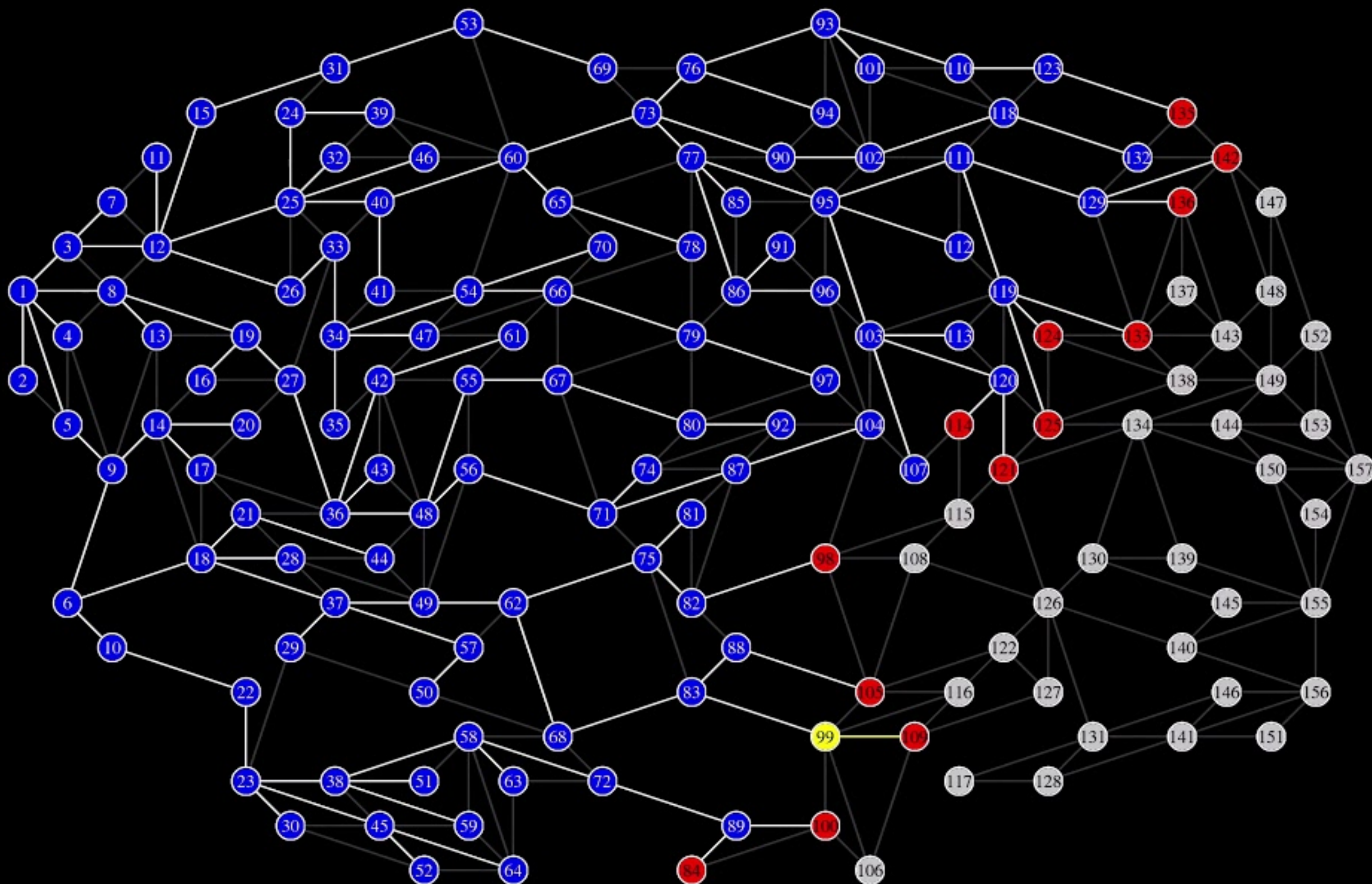
Node data structure contains variables like the state, a pointer to its parent node, the action that was used to create this state, and the path cost.

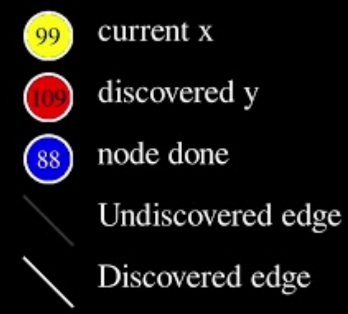
The Python yield keyword means that we don't have to pre-compute a list of all successors.

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*






```
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

b = branching factor

Complete? Yes (if b is finite)

Optimal? Yes if cost = 1 per step

d = depth

Time Complexity? $1 + b + b^2 + b^3 \ldots = O(b^d)$

Space Complexity? $O(b^d)$

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