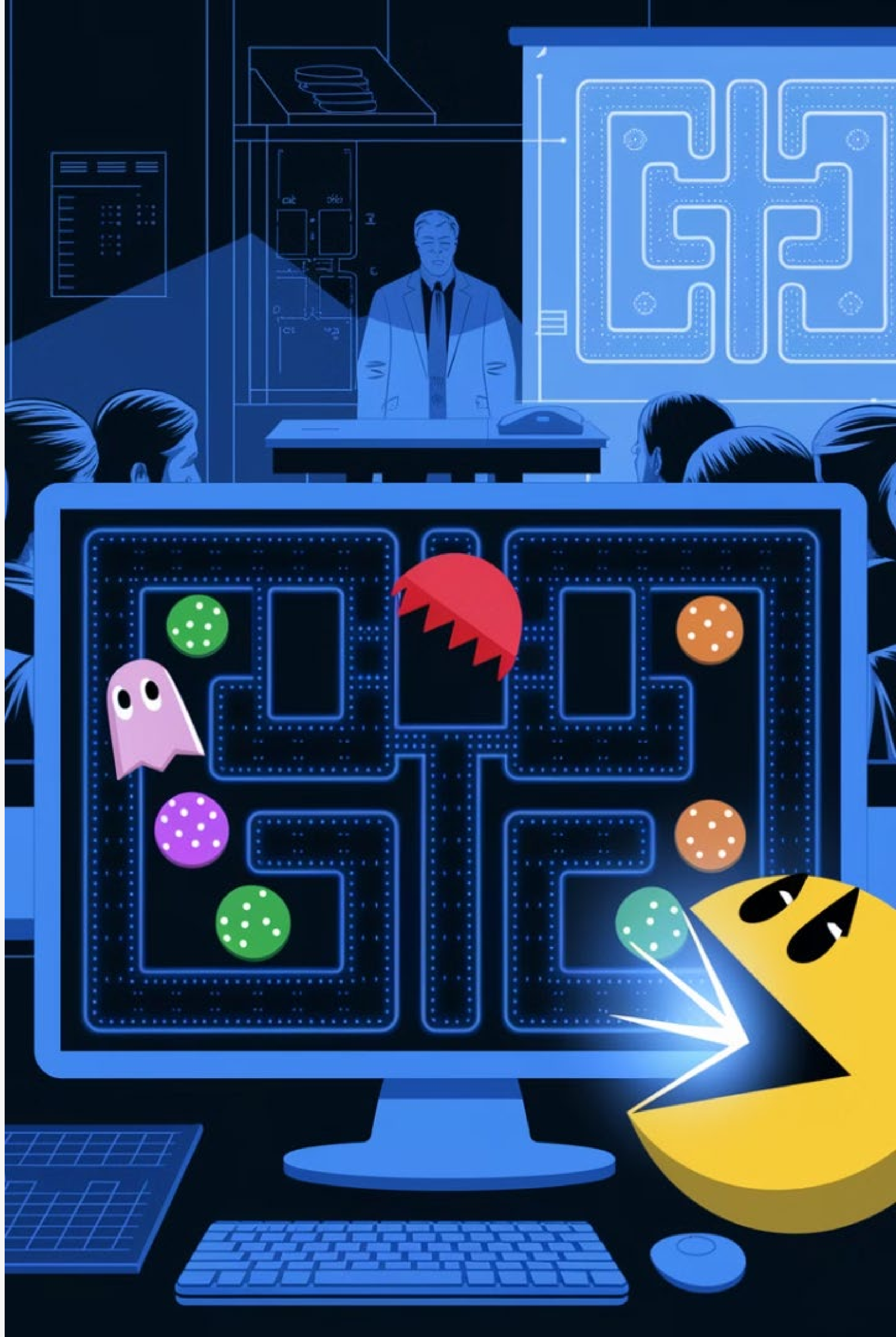
Artificial Intelligence Practical Session

The exercises completed in this session are designed to build a comprehensive understanding of AI—from basic search algorithms to sophisticated machine learning techniques—all while keeping students engaged with visual feedback and challenging problems that mirror real-world applications. It is based on UC Berkeley CS188 Intro to AI -- Course Materials.

Due to time constraints, labs will be guided/demonstrated by the professors.



Project Philosophy & Goals

Visualization

Projects allow students to directly visualize the results of their implemented AI techniques in action, providing immediate feedback and better intuition about abstract concepts.

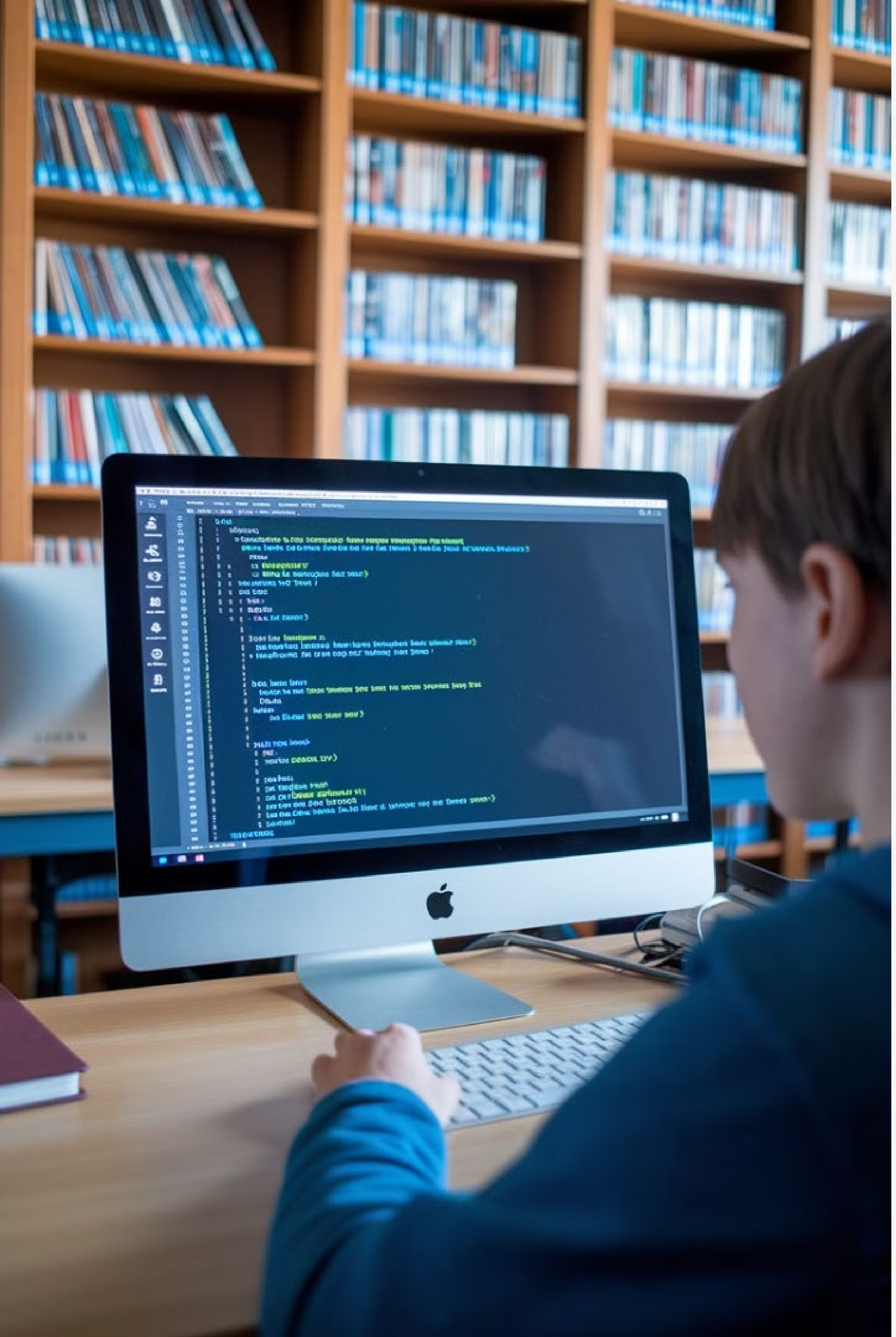
Minimal Scaffolding

Each project contains clear directions and code examples without excessive starter code, encouraging students to develop and understand their own solutions.

Real-World Challenge

Pac-Man presents a genuinely challenging environment that demands creative solutions, mirroring the complexity found in real-world AI applications.

These principles create an environment where students don't just implement algorithms—they develop intuition for how AI techniques work in practice, building skills that transfer to natural language processing, computer vision, robotics, and other cutting-edge fields.



Getting Started: UNIX/Python Tutorial

Python Fundamentals

Introduction to Python programming language basics with emphasis on features used throughout the Pac-Man projects, including data structures and object-oriented programming.

UNIX Environment

Essential UNIX commands and environment setup to efficiently develop, test and debug the AI implementations required for subsequent projects.

Development Workflow

Best practices for managing code, running experiments, and analyzing results in a systematic way that prepares students for the iterative nature of AI development.

This preliminary tutorial ensures all students have the technical foundation needed before diving into AI concepts. By standardizing on Python with no external dependencies, the course maintains accessibility while teaching practical programming skills that remain relevant in modern AI development.

Search Algorithms

1

Depth-First Search

Students implement this fundamental algorithm where Pac-Man explores as far as possible along each branch before backtracking, learning about stack-based frontiers and graph traversal.

2

Breadth-First Search

Implementation focuses on finding the shortest path in terms of moves, using queue-based frontiers to explore all neighbors at each depth level before proceeding.

3

Uniform Cost Search

Students extend their implementations to account for different movement costs, introducing priority queues and optimality guarantees.

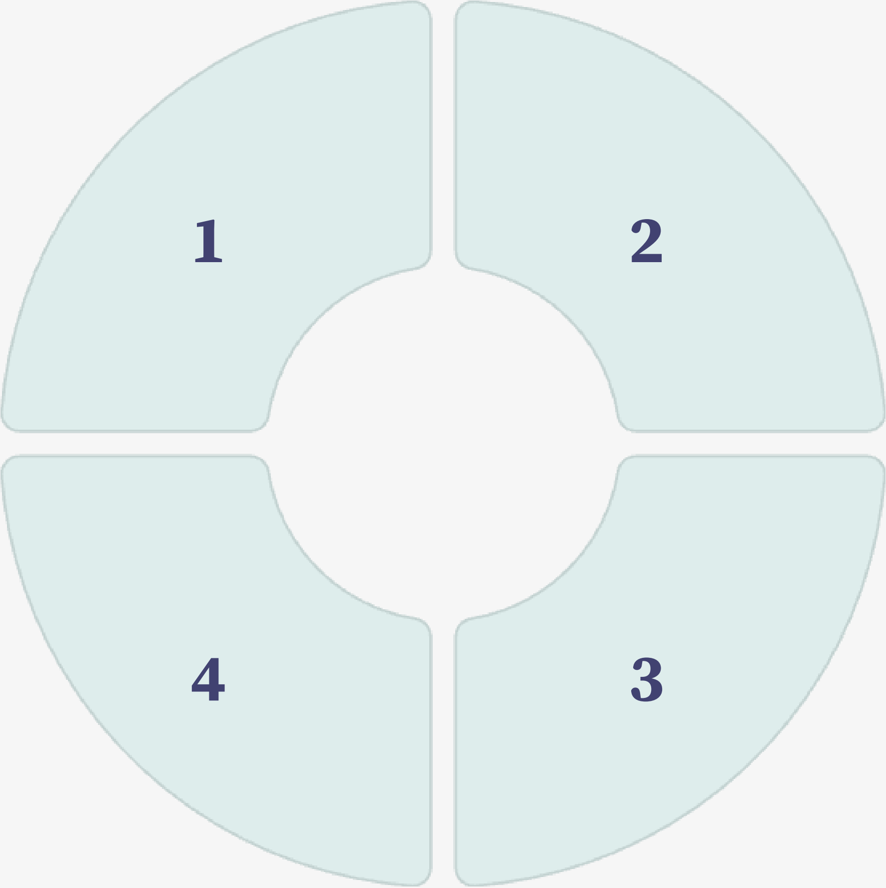
4

A* Search

The project culminates with implementing this informed search algorithm, requiring students to design admissible heuristics that significantly improve search efficiency.

Through these implementations, students watch Pac-Man navigate mazes of increasing complexity, concretely visualizing how different algorithms explore the state space. The traveling salesman problem variation challenges students to optimize dot-collection paths, introducing them to computationally intractable problems.

Multi-Agent Search



Adversarial Search

Students model Pac-Man and ghosts as competing agents, implementing minimax search strategies where Pac-Man maximizes score while ghosts minimize it.

Alpha-Beta Pruning

Implementation of this optimization technique dramatically improves the efficiency of minimax search by eliminating branches that won't influence the final decision.

Expectimax

Students develop algorithms for stochastic environments where ghosts move randomly, requiring probabilistic reasoning rather than assuming worst-case opponent behavior.

Evaluation Functions

The project challenges students to design and implement custom state evaluation functions that quantify the "goodness" of game positions when lookahead is limited.

This project bridges classical game theory with practical implementation, demonstrating how techniques from two-player games like chess apply to Pac-Man. Students gain intuition about pruning techniques and the tradeoffs between different models of opponent behavior, essential concepts in modern game AI and decision-making under uncertainty.

Reinforcement Learning

1 Value Iteration

Students implement this model-based algorithm that computes the optimal policy given a complete model of the environment, learning how Markov Decision Processes formalize sequential decision problems.

2 Q-Learning

Moving to model-free learning, students implement this temporal-difference algorithm where Pac-Man learns optimal behavior through trial-and-error interaction with the environment.

3 Approximate Q-Learning

Students extend their implementation to use feature-based representations, enabling learning in the full Pac-Man game where the state space is too large for tabular methods.

4 Crawling Robot

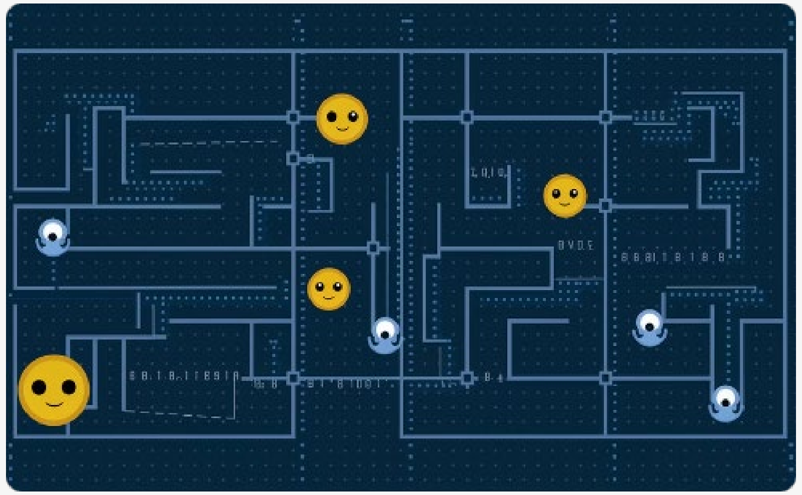
The project applies the same techniques to a simulated robot arm, demonstrating how reinforcement learning generalizes beyond games to physical control tasks.

This project demonstrates how reinforcement learning—the paradigm behind many recent AI breakthroughs—works in practice. Students witness agents starting with random behavior and gradually improving through experience, gaining insight into the exploration-exploitation tradeoff and the power of function approximation in complex environments.

Probabilistic Inference: Ghostbusters

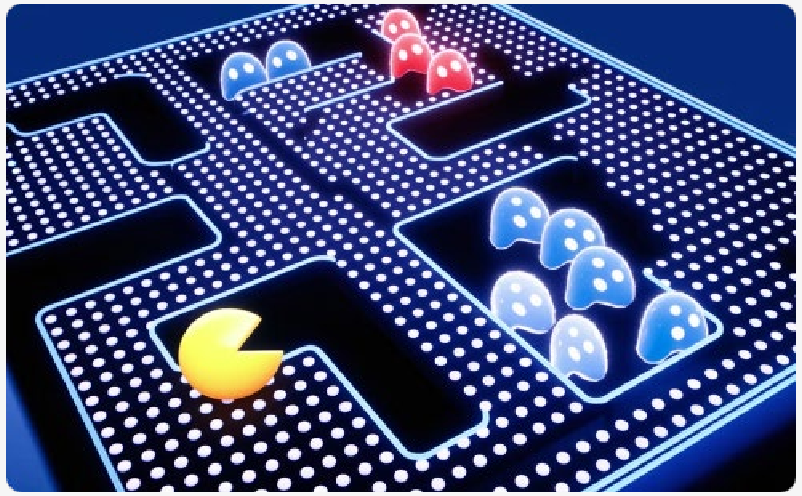
Hidden Markov Models

Students implement probabilistic inference in a hidden Markov model where Pac-Man must track invisible ghosts based on noisy sensor readings, learning about belief states and recursive state estimation.



Exact Inference

Implementation of the forward algorithm allows Pac-Man to maintain a precise probability distribution over ghost locations, demonstrating how Bayes' rule enables agents to reason under uncertainty.



Particle Filtering

Students implement this approximate inference method where beliefs are represented by samples, learning about the computational tradeoffs that enable inference in complex, real-world domains.

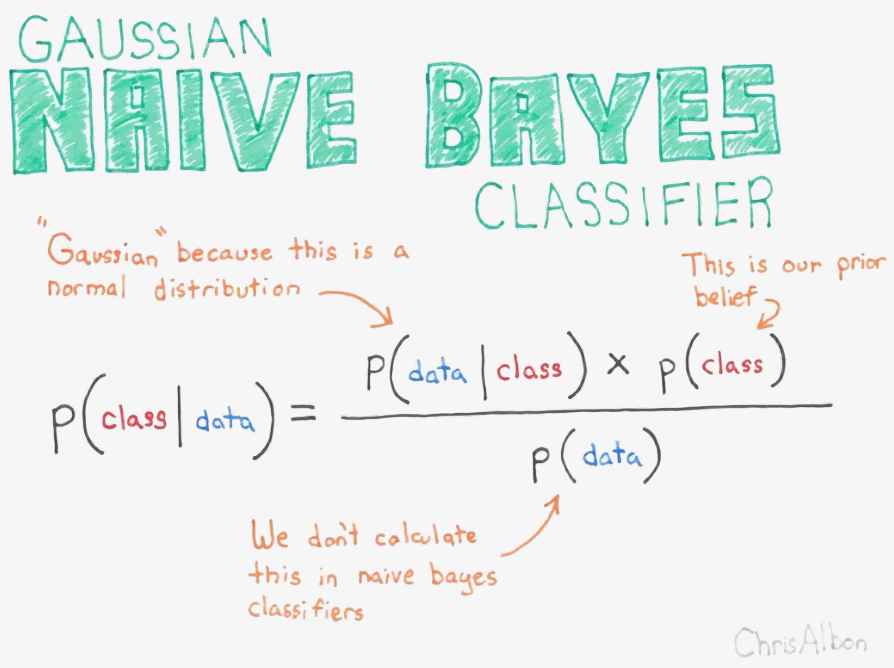
This project brings probabilistic models to life by showing how they enable intelligent decision-making with incomplete information. Students experience firsthand how representing uncertainty explicitly leads to more robust agent behavior—a key insight that underlies modern robotics, speech recognition, and other AI systems.

Machine Learning Classification

In this final project, students implement core machine learning algorithms for digit classification, including Naive Bayes (a generative probabilistic model), Perceptron (a discriminative online algorithm), and MIRA (a margin-based approach).

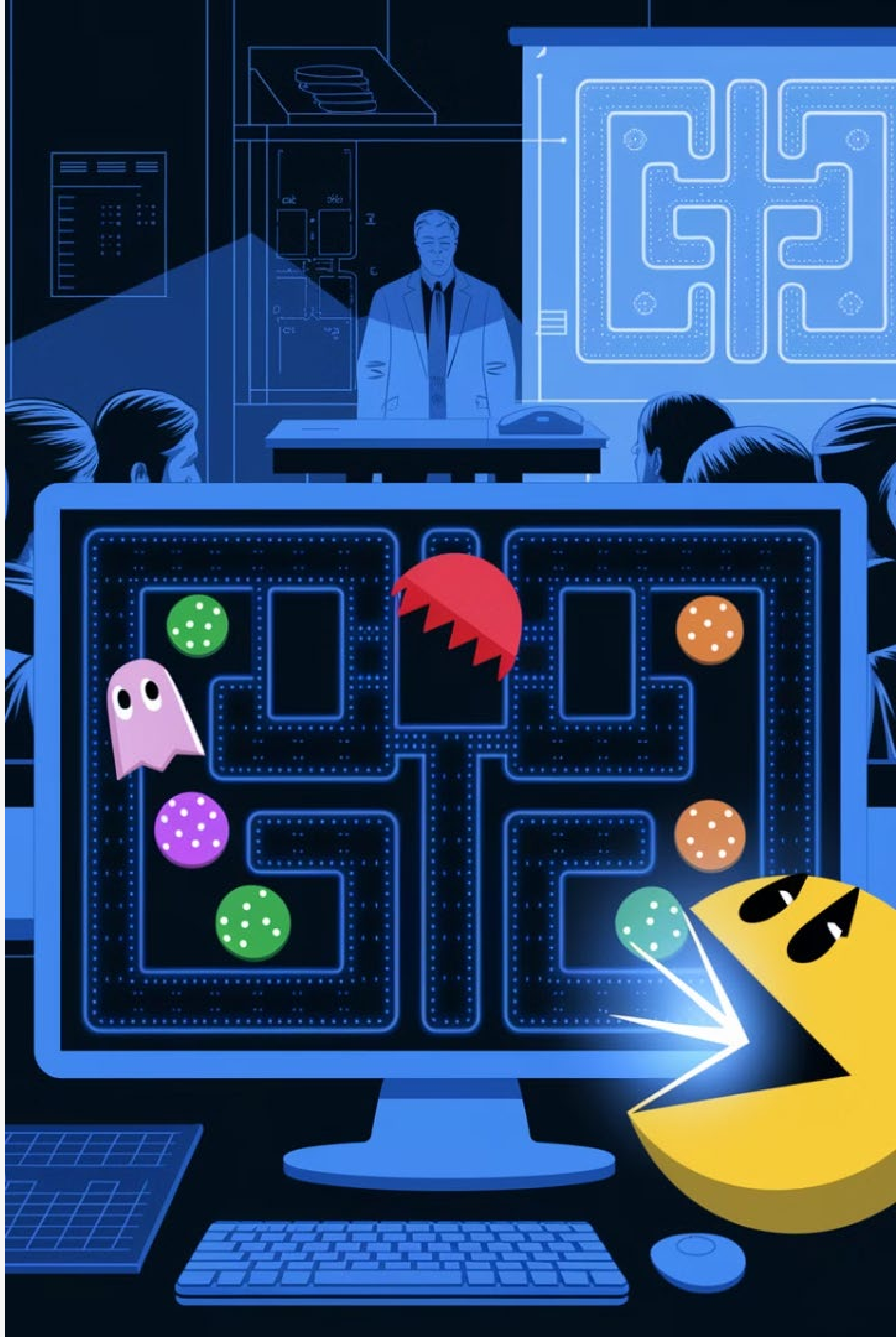
The culmination comes when students apply these techniques to create a behavioral cloning Pac-Man agent that learns by observing expert gameplay. This connects classification to decision-making and demonstrates how supervised learning enables agents to mimic human behavior—a technique increasingly used in applications from autonomous vehicles to virtual assistants.

Through these projects, students gain both theoretical understanding and practical implementation experience with fundamental AI techniques that power modern technological innovations.

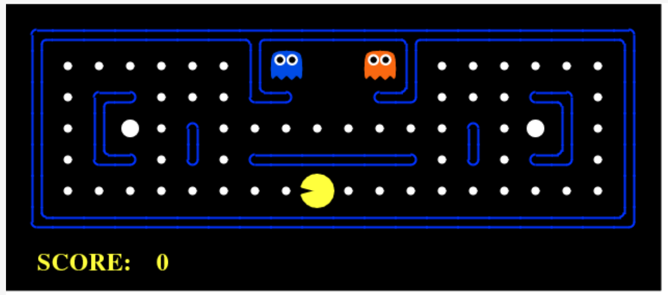


Artificial Intelligence Practical Session

Implementation



The Pacman project

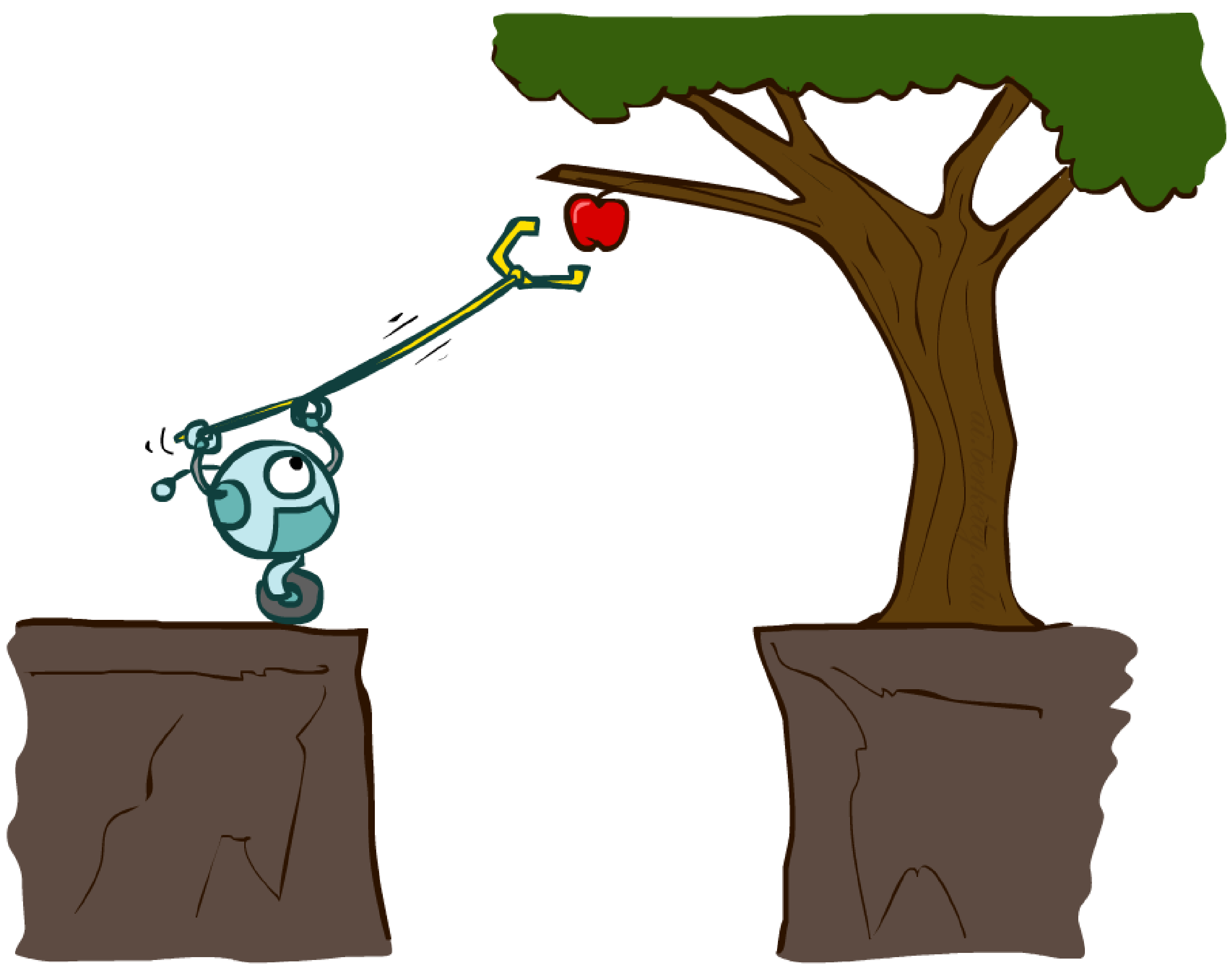


The Pac-Man projects were developed for UC Berkeley's introductory artificial intelligence course. It includes the different projects presented before:

- **Search**: Students implement depth-first, breadth-first, uniform cost, and A* search algorithms. These algorithms are used to solve navigation and traveling salesman problems in the Pacman world.
- **Multi-Agent Search**: Classic Pacman is modeled as both an adversarial and a stochastic search problem. Students implement multiagent minimax and expectimax algorithms, as well as designing evaluation functions.
- **Reinforcement Learning**: Students implement model-based and model-free reinforcement learning algorithms.
- **Ghostbusters**: Probabilistic inference in a hidden Markov model tracks the movement of hidden ghosts in the Pacman world. Students implement exact inference using the forward algorithm and approximate inference via particle filters.
- **Classification:** Students implement standard machine learning classification algorithms using Naive Bayes, Perceptron, and MIRA models to classify digits. Students extend this by implementing a behavioral cloning Pacman agent.

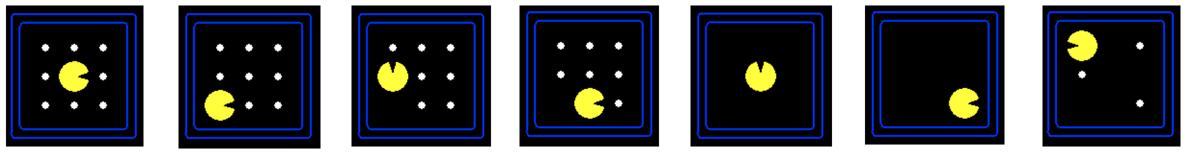
Search (I)

- Planning agents:
  - Ask “what if”
  - Decisions based on (hypothesized) consequences of actions
  - Must have a model of how the world evolves in response to actions
  - Must formulate a goal (test)
  - Consider how the world WOULD BE
- Optimal vs. complete planning

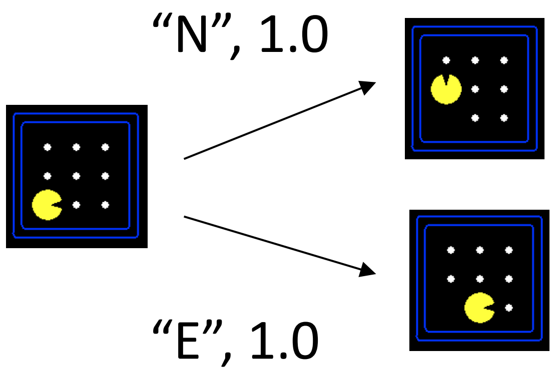


Search (II)

- A **search problem** consists of:
  - A state space



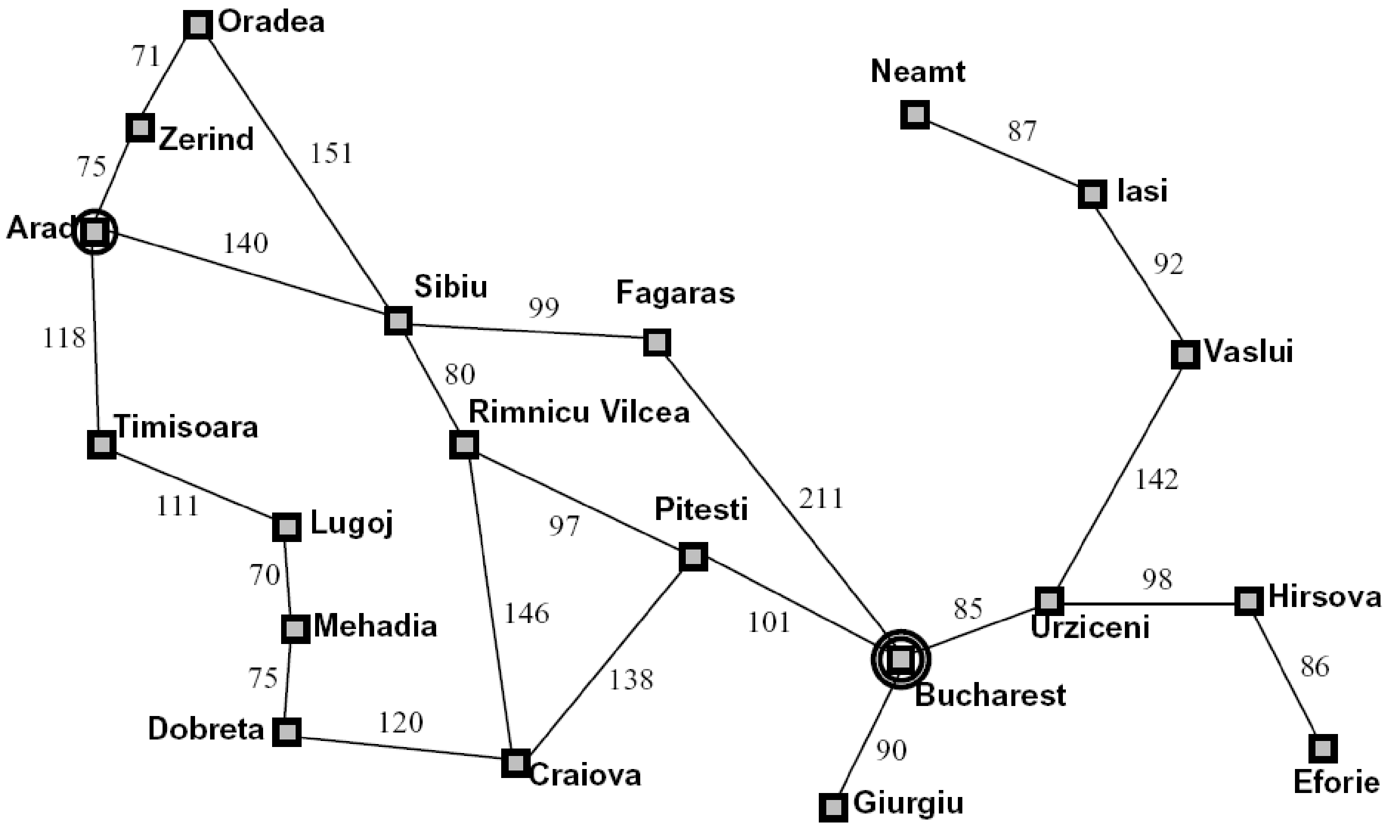
  - A successor function (with actions, costs)



  - 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

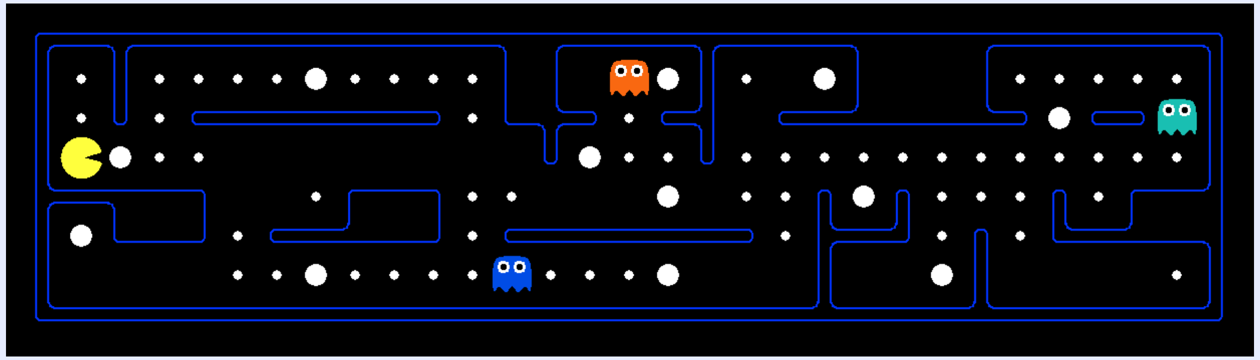
Search (III): Example: Traveling in Romania



- State space:
  - Cities
- Successor function:
  - Roads: Go to adjacent city with cost = distance
- Start state:
  - Arad
- Goal test:
  - Is state == Bucharest?
- Solution?

Search (IV): State Space

The **world state** includes every last detail of the environment



A **search state** keeps only the details needed for planning (abstraction)

- Problem: Pathing
  - States: (x,y) location
  - Actions: NSEW
  - Successor: update location only
  - Goal test: is (x,y)=END

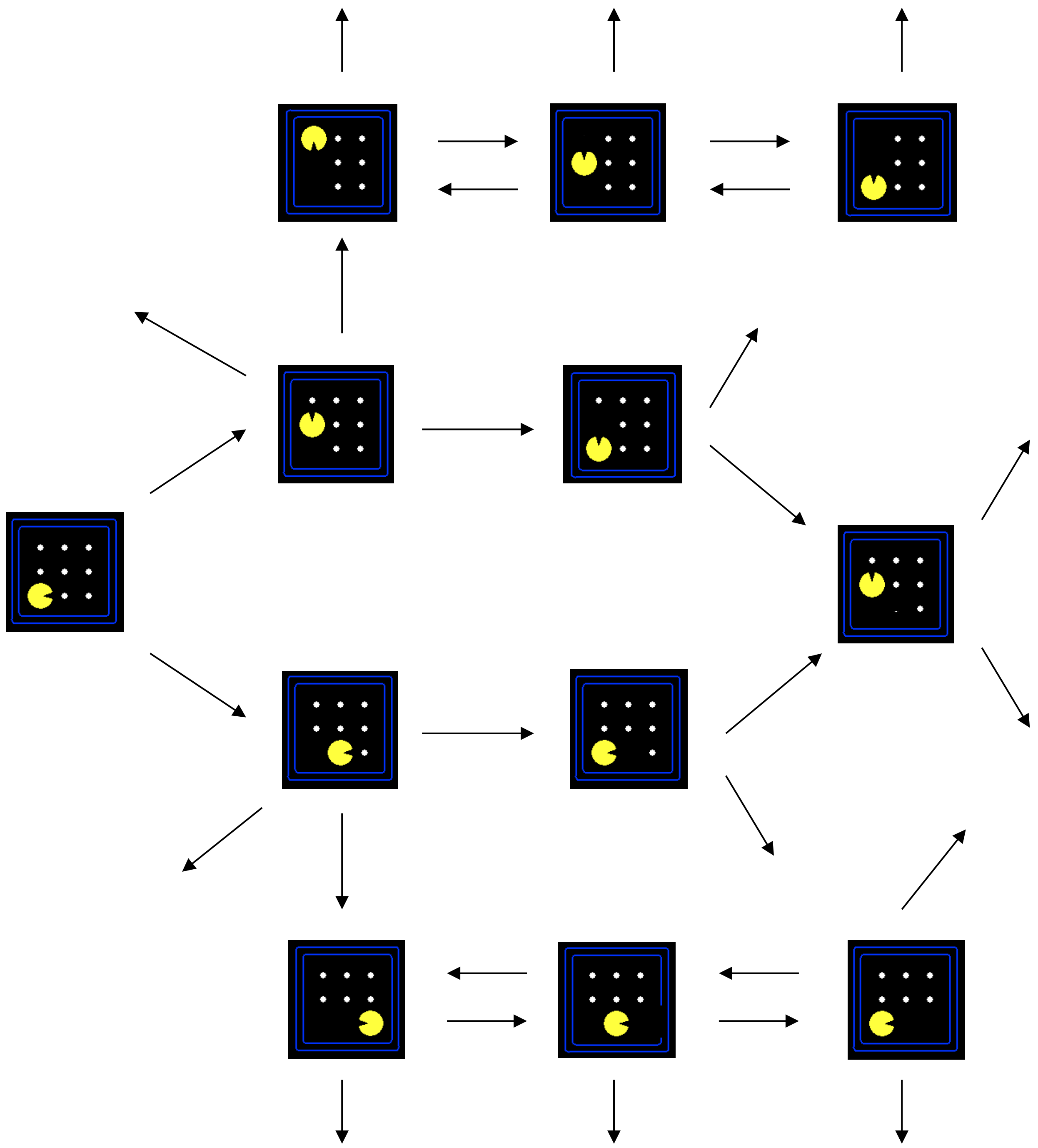
- Problem: Eat-All-Dots
  - States: {(x,y), dot booleans}
  - Actions: NSEW
  - Successor: update location and possibly a dot boolean
  - Goal test: dots all false

Search (V): State Space Graphs

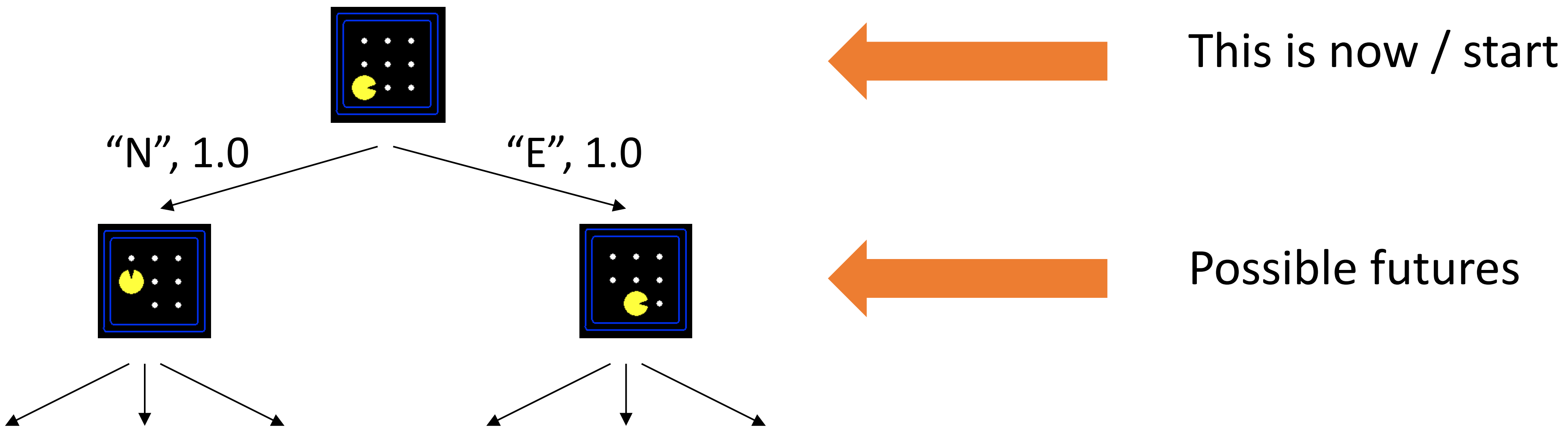
- State space graph: A mathematical representation of a search problem
  - Nodes are (abstracted) world configurations
  - Arcs represent successors (action results)
  - The goal test is a set of goal nodes (maybe only one)

- In a state space graph, each state occurs only once!

- We can rarely build this full graph in memory (it's too big), but it's a useful idea



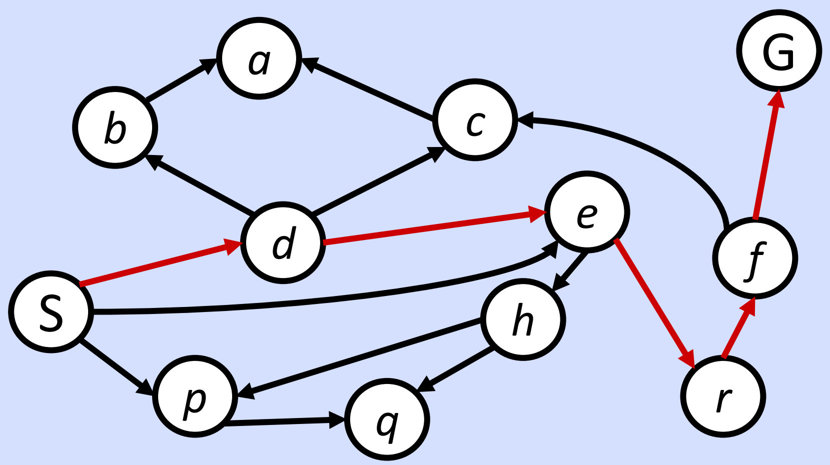
Search (VI): Search trees



- A search tree:
  - A "what if" tree of plans and their outcomes
  - The start state is the root node
  - Children correspond to successors
  - Nodes show states, but correspond to PLANS that achieve those states
  - **For most problems, we can never actually build the whole tree**

Search (VII): State Space Graphs vs. Search Trees

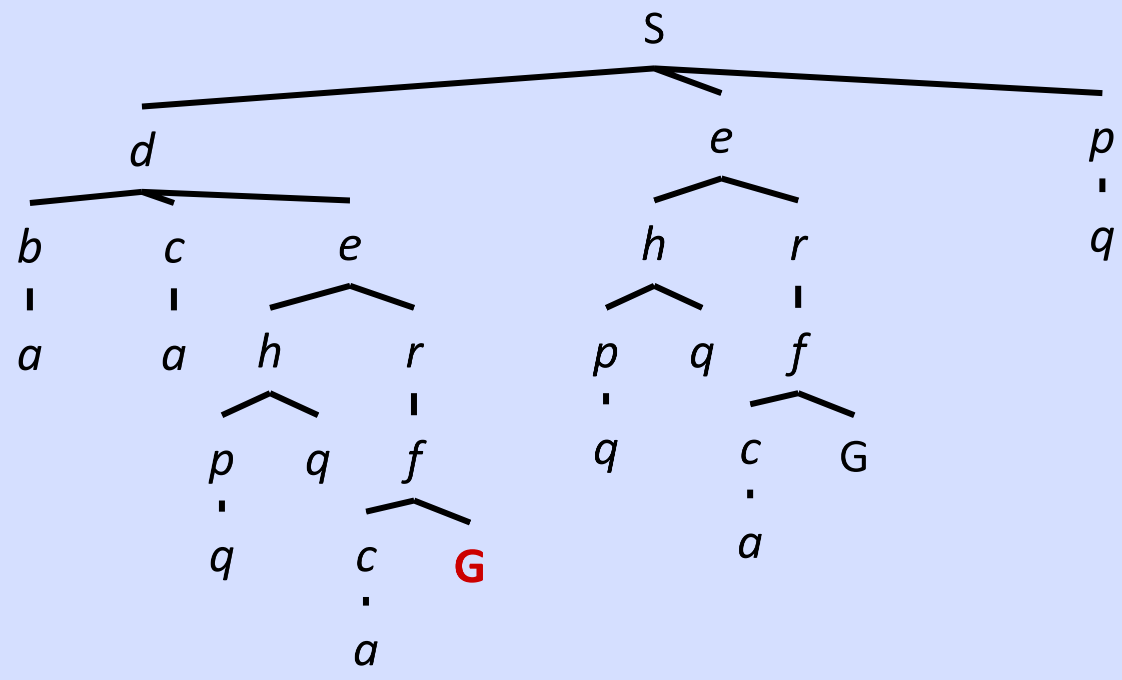
State Space Graph



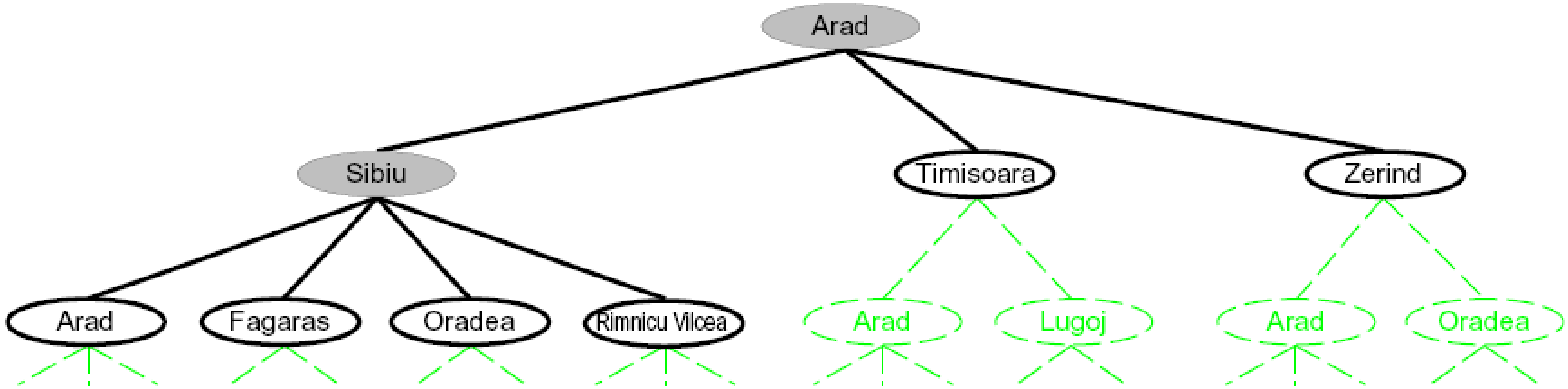
Each NODE in in the search tree is an entire PATH in the state space graph.

We construct both on demand – and we construct as little as possible.

Search Tree

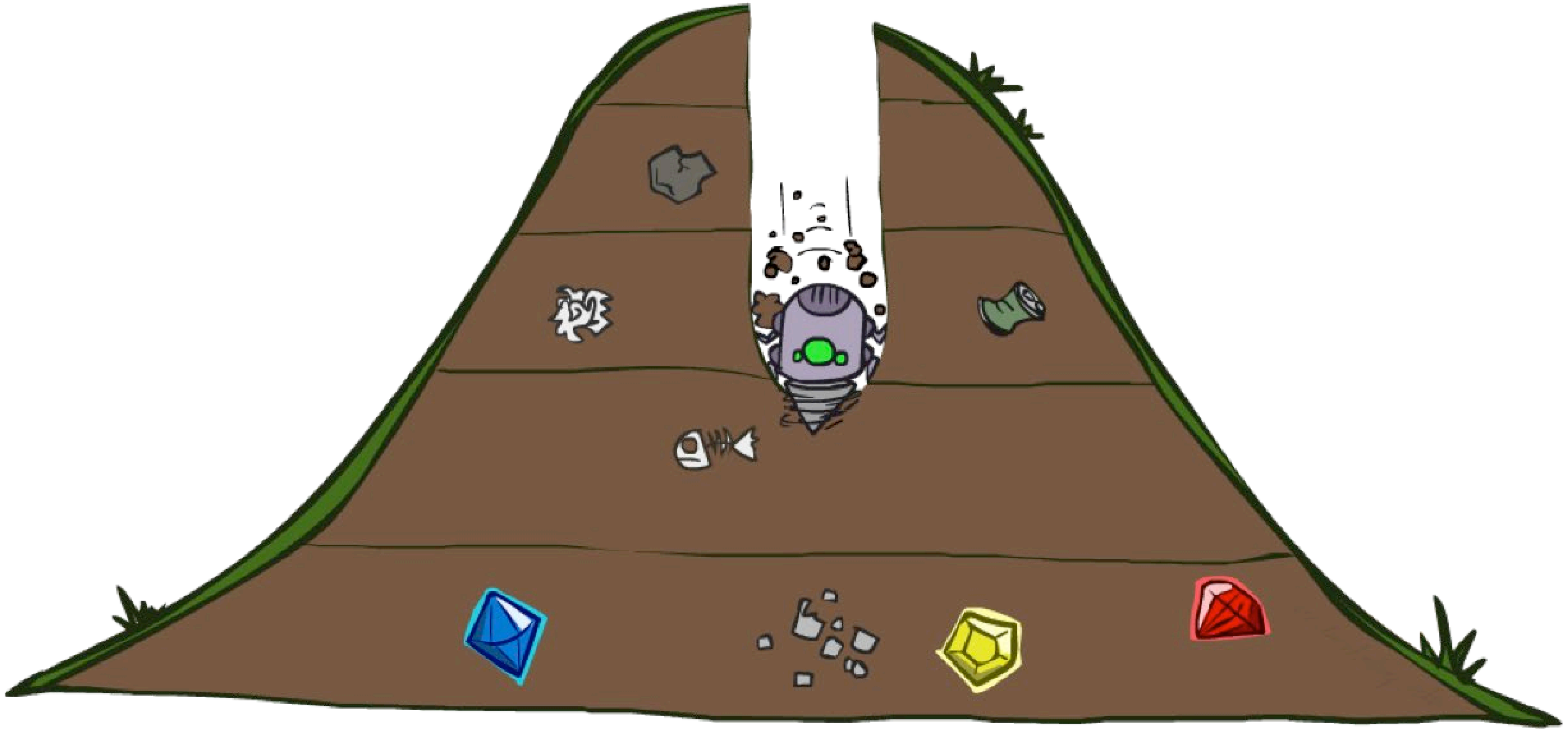


Search (VIII): Searching with a Search Tree



- Search:
  - Expand out potential plans (tree nodes)
  - Maintain a **fringe** of partial plans under consideration
  - Try to expand as few tree nodes as possible

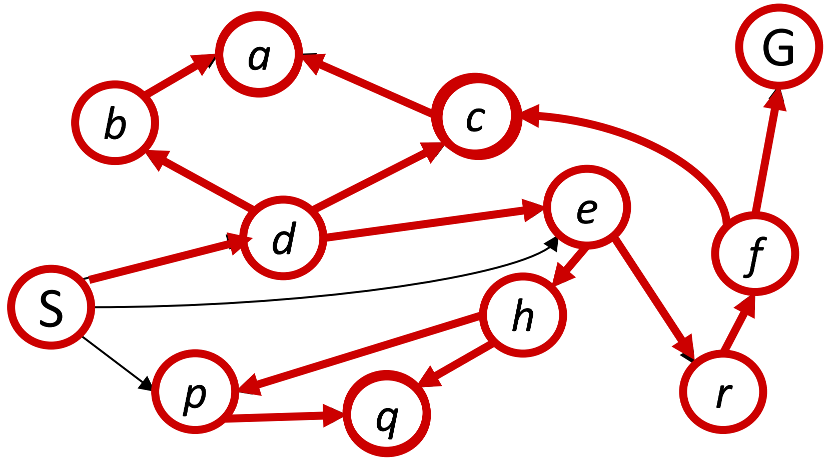
Search (IX): Depth-First Search

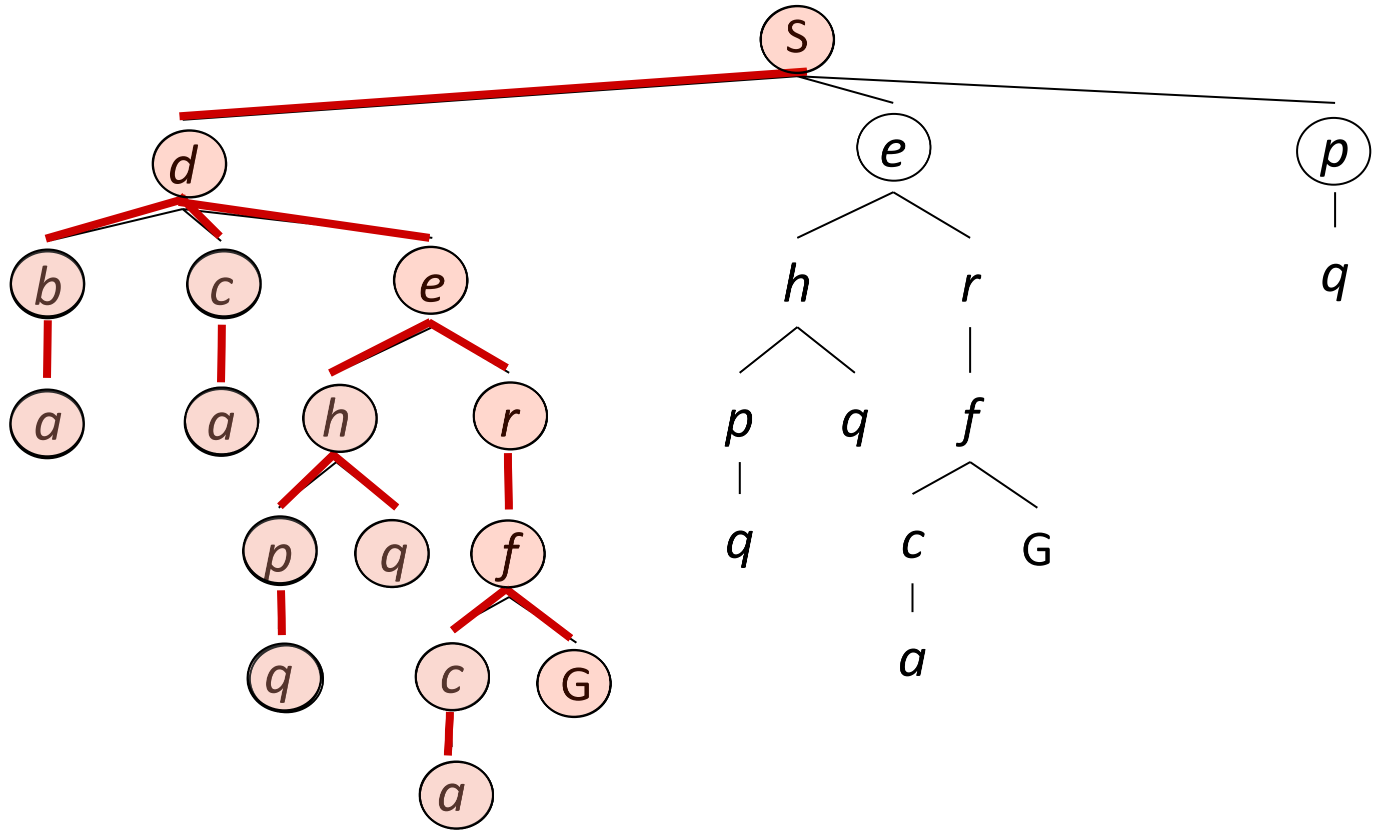


Search (X): Depth-First Search

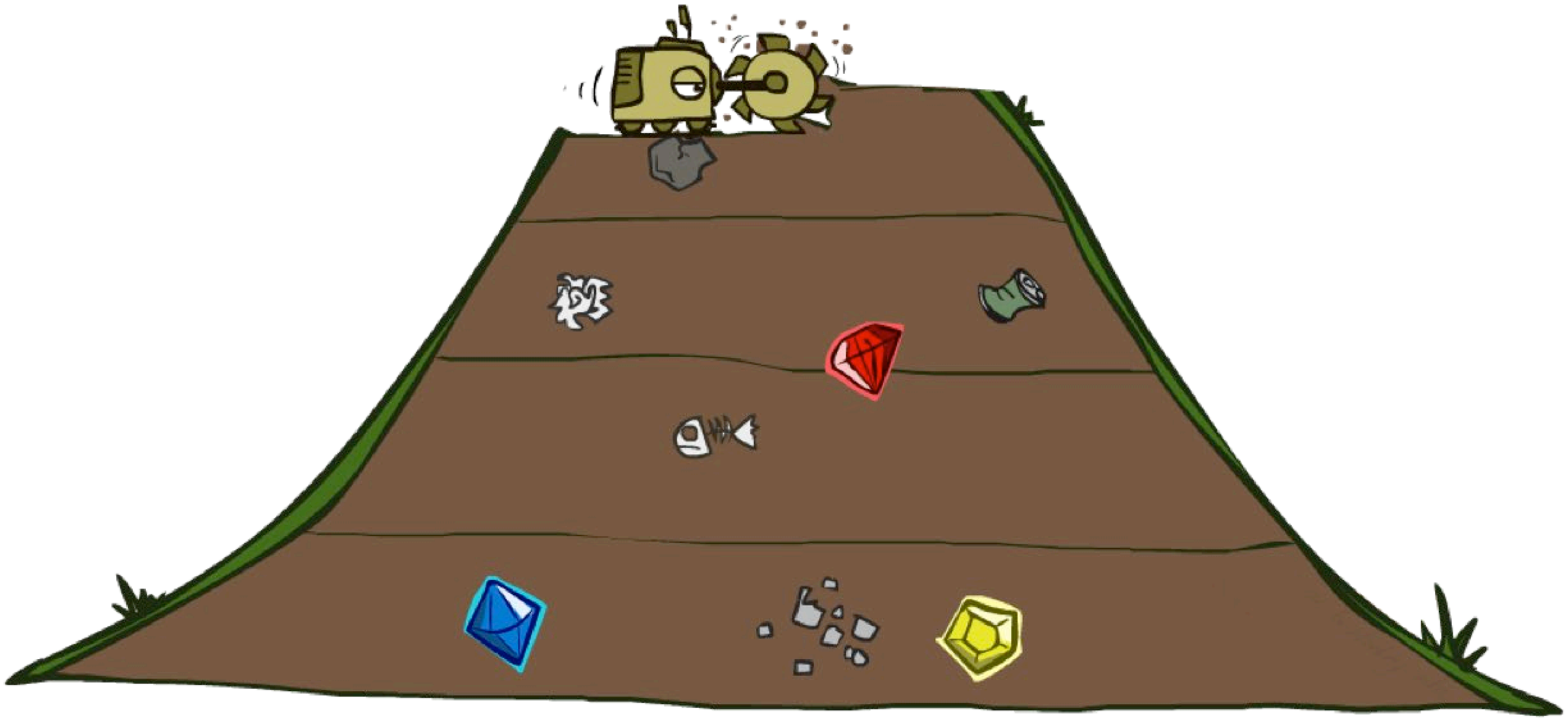
Strategy: expand a deepest node first

Implementation: Fringe is a LIFO stack





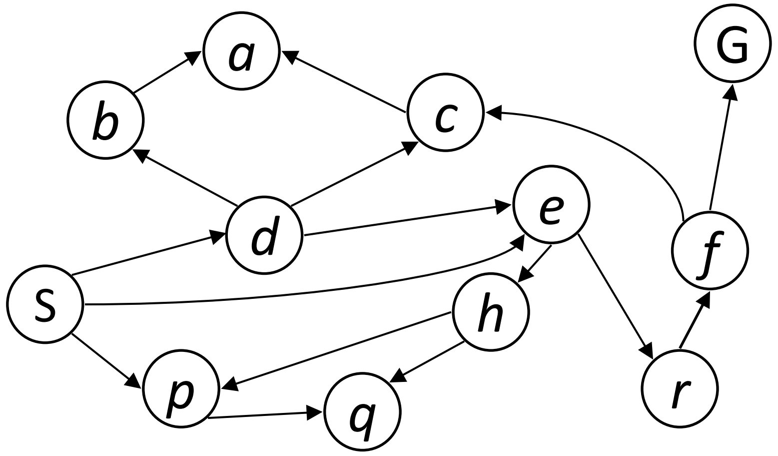
Search (X): Breadth-First Search



Search (XI): Breadth-First Search

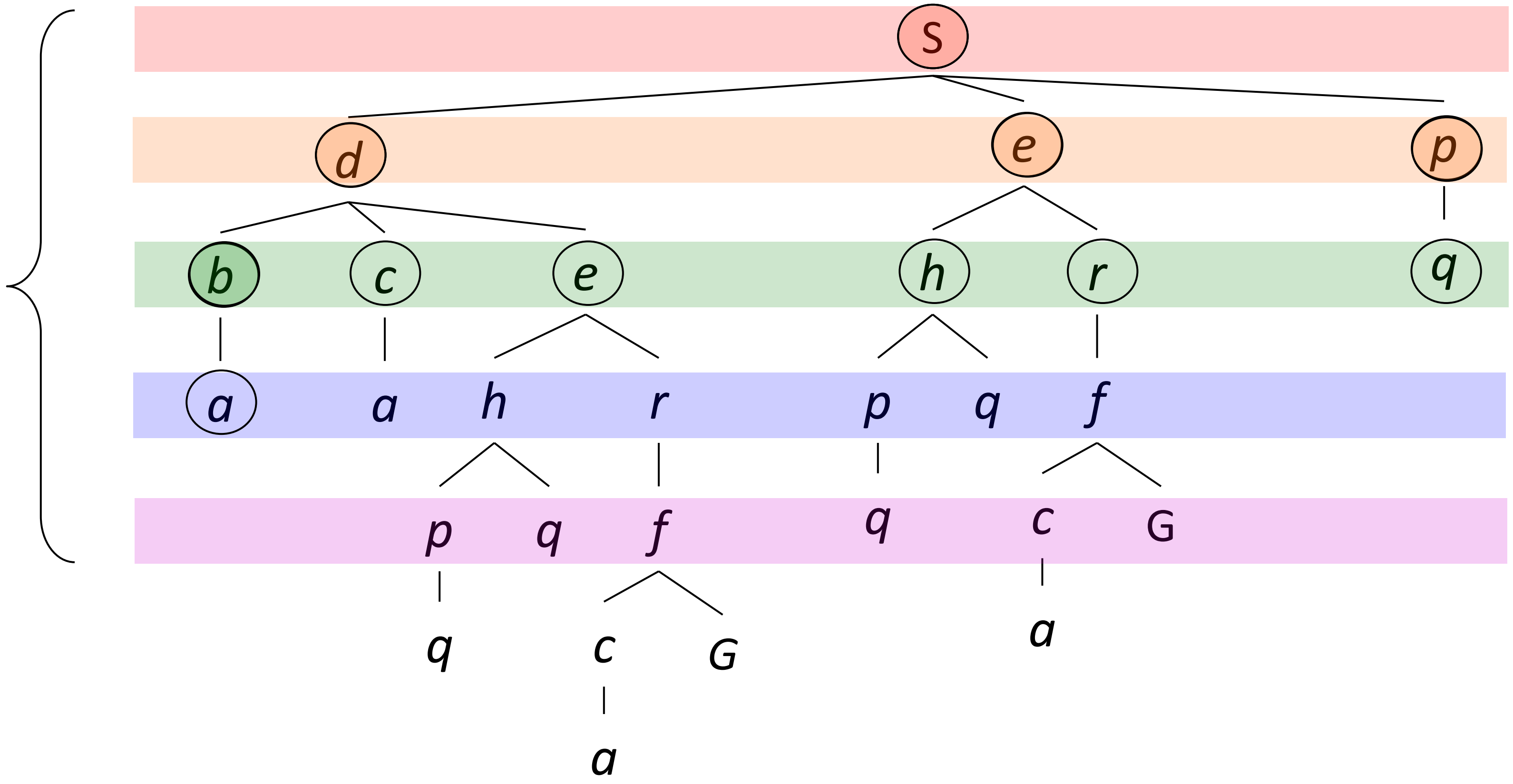
Strategy: expand a shallowest node first

Implementation: Fringe is a FIFO queue



Search

Tiers



Code for the implementation of the search algorithm

Download the code:

- `git clone https://github.com/felipexil/pacman.git`

Using python 2, test the game:

- `cd pacman/search/search`
- `python pacman.py`

Pacman lives in a shiny blue world of twisting corridors and tasty round treats. Navigating this world efficiently will be Pacman's first step in mastering his domain.

The simplest agent in searchAgents.py is called the GoWestAgent, which always goes West (a trivial reflex agent). This agent can occasionally win:

- `python pacman.py --layout testMaze --pacman GoWestAgent`

But, things get ugly for this agent when turning is required:

- `python pacman.py --layout tinyMaze --pacman GoWestAgent`

If Pacman gets stuck, you can exit the game by typing CTRL-c into your terminal.

Note that pacman.py supports a number of options that can each be expressed in a long way (e.g., --layout) or a short way (e.g., -l). You can see the list of all options and their default values via:

- `python pacman.py -h`

Finding a Fixed Food Dot using Depth First Search (I)

In searchAgents.py, you'll find a fully implemented SearchAgent, which plans out a path through Pacman's world and then executes that path step-by-step. The search algorithms for formulating a plan are not implemented -- that's your job.

First, test that the SearchAgent is working correctly by running:

- `python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch`

The command above tells the SearchAgent to use tinyMazeSearch as its search algorithm, which is implemented in search.py. Pacman should navigate the maze successfully.

All of your search functions need to return a list of actions that will lead the agent from the start to the goal. These actions all have to be legal moves (valid directions, no moving through walls).

Make sure to use the Stack, Queue and PriorityQueue data structures provided to you in util.py!

Implement the depth-first search (DFS) algorithm in the depthFirstSearch function in search.py. To make your algorithm complete, write the graph search version of DFS, which avoids expanding any already visited states.

Finding a Fixed Food Dot using Depth First Search (II)

ALGORITHM

```
procedure DFS(G,v):    /* G → Graph, v → starting vertice */
        let S be a stack
        S.push(v)
        while S is not empty
                v = S.pop()
                if v is not labeled as discovered:
                        label v as discovered
                        for all edges from v to w in G.adjacentEdges(v) do
                                S.push(w)
```

- `python pacman.py -l tinyMaze -p SearchAgent -a fn=bfs`
- `python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs`
- `python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=bfs`

Finding a Fixed Food Dot using Depth First Search (III)

```python
def depthFirstSearch(problem):
    # Needed data structures
    st = util.Stack()
    parent = dict()
    steps = list()
    have_visited = set()

    start_state = (problem.getStartState(), None, None)
    st.push(start_state)

    while (not st.isEmpty()):
        state = st.pop()
        have_visited.add(state[0])  # Visit the current state

        # Found goal state
        if (problem.isGoalState(state[0])):

            # Trace back the path and reverse it to get the correct sequence of steps
            while (state != start_state):
                steps.append(state[1])
                state = parent[state]

            steps.reverse()
            return steps

        # Expand successors
        successors = problem.getSuccessors(state[0])
        for successor in successors:
            if (successor[0] not in have_visited):
                parent[successor] = state
                st.push(successor)

    # Solution/Path does not exist
    return None
```

Finding a Fixed Food Dot using Breadth First Search

ALGORITHM

```
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 the 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)
```