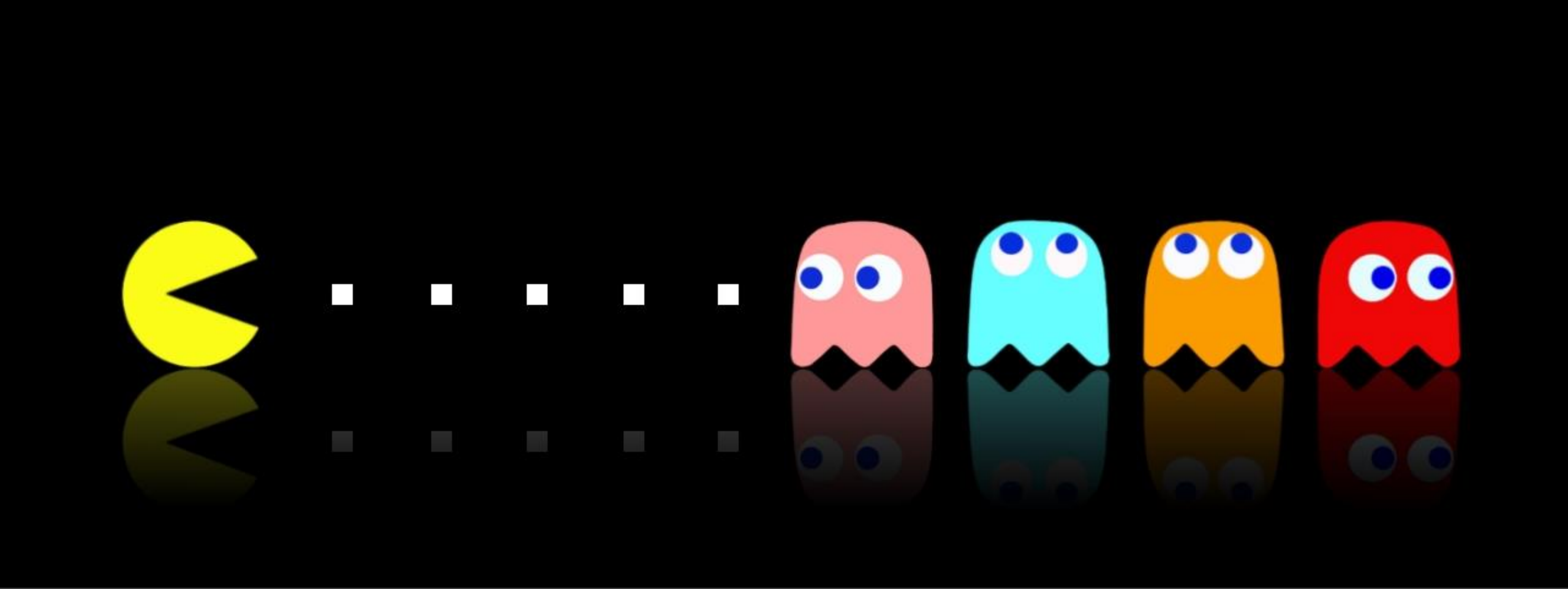
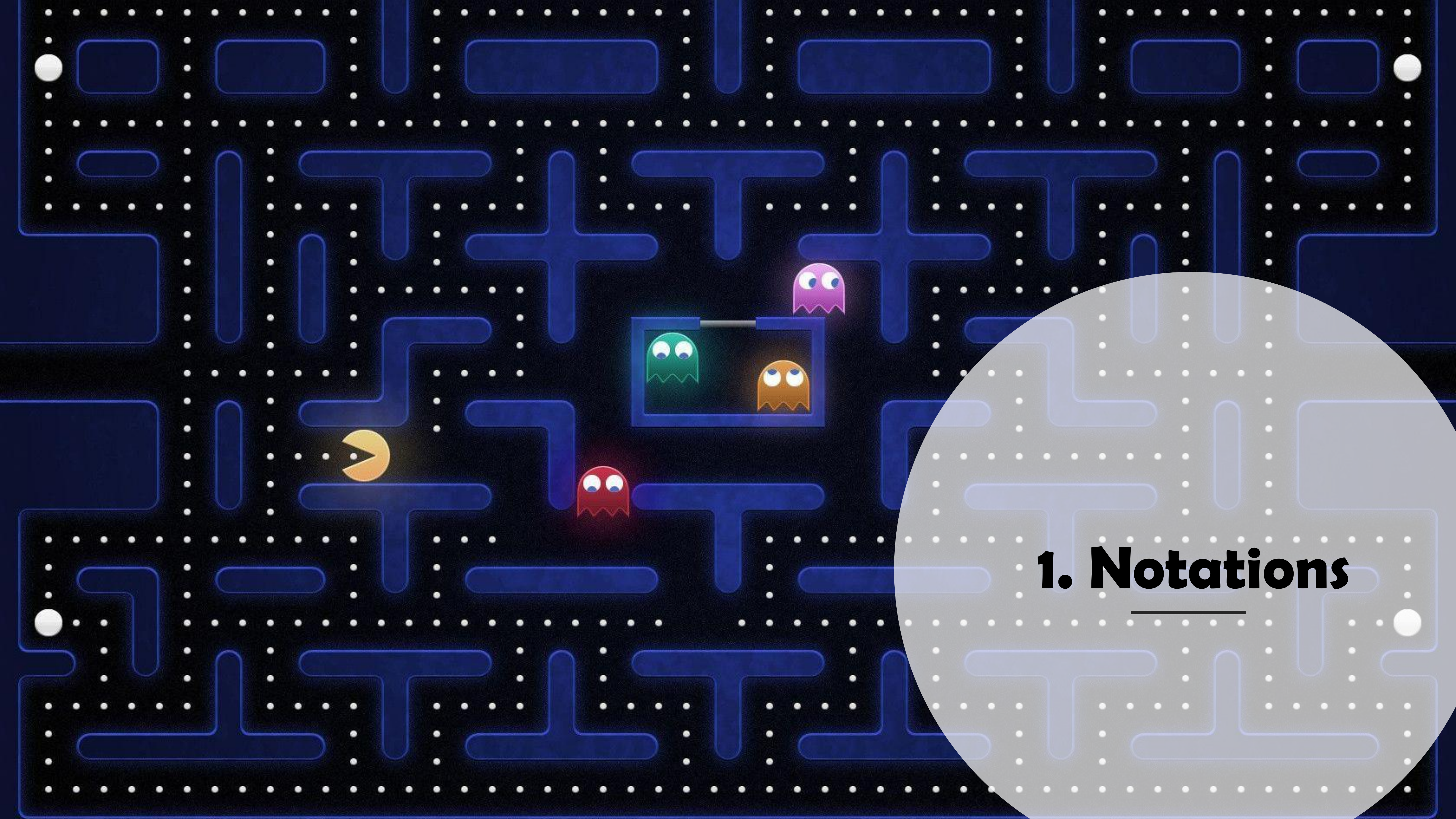
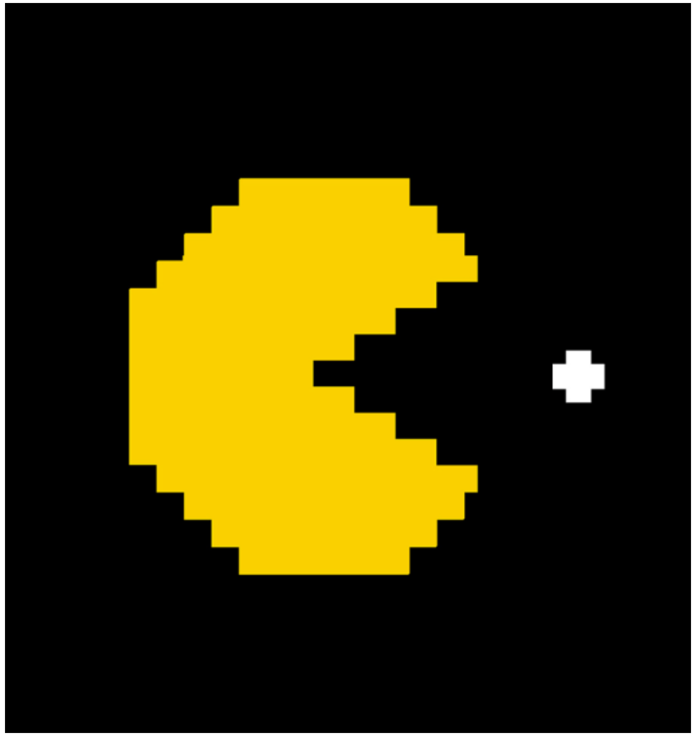


CS 3600 Project 1 Review

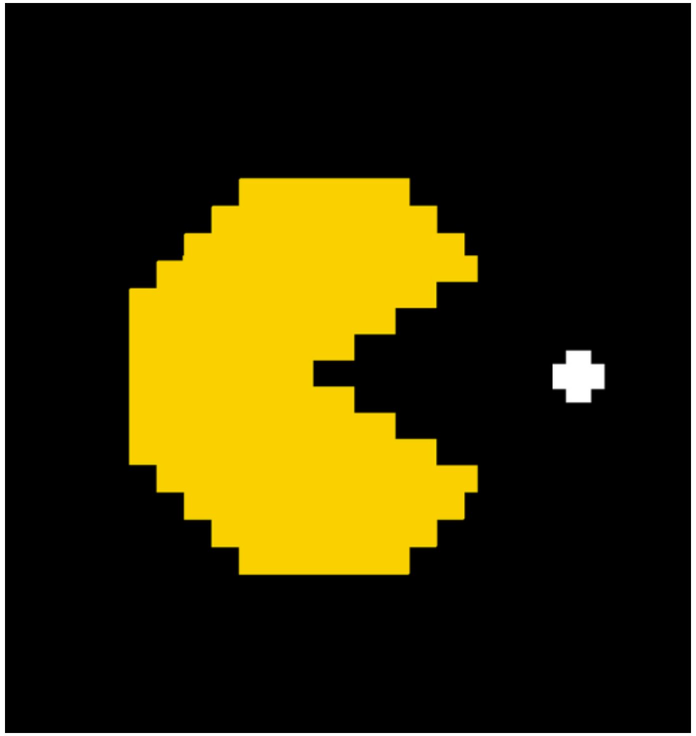


1. Notations



Notations: State

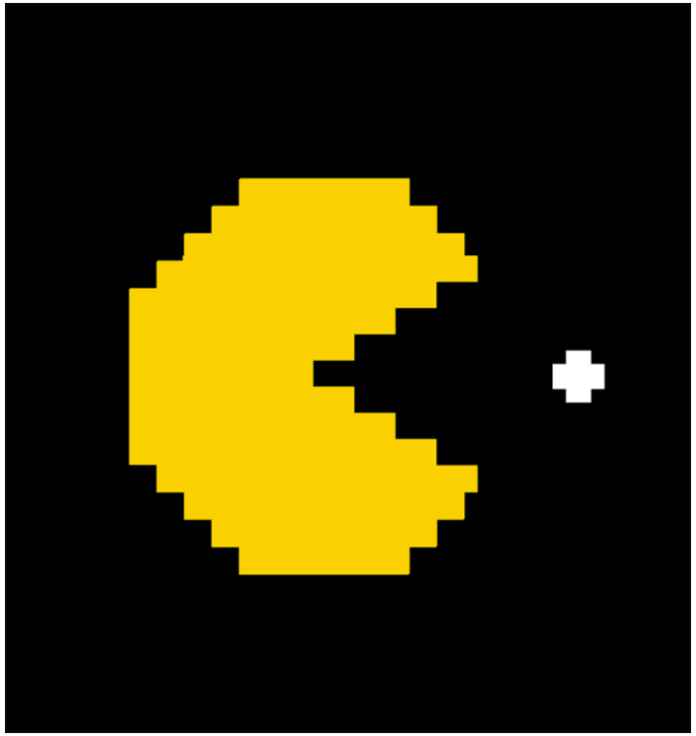
- Representation of where Pacman is on the map!
- Represented as ***<u>Euclidean</u>*** coordinates, or just a label (e.g. "A", "B",…)
- Will refine notion of "state" in later question of HW



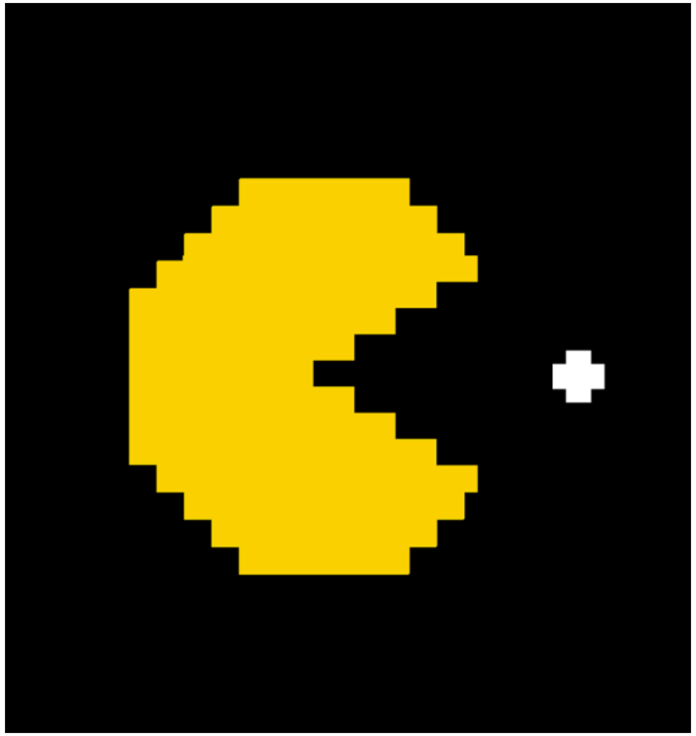
Notations: Successors

- The next possible states
- Tuple containing ***<u>three</u>*** items:
  - Current state
  - Action taken to reach it
  - Cost of action
- Call `getSuccessors()`

Notations: Path



- Sequences of actions returned from your search functions
- How to get from ***<u>start state</u>*** S to ***<u>goal state</u>*** G



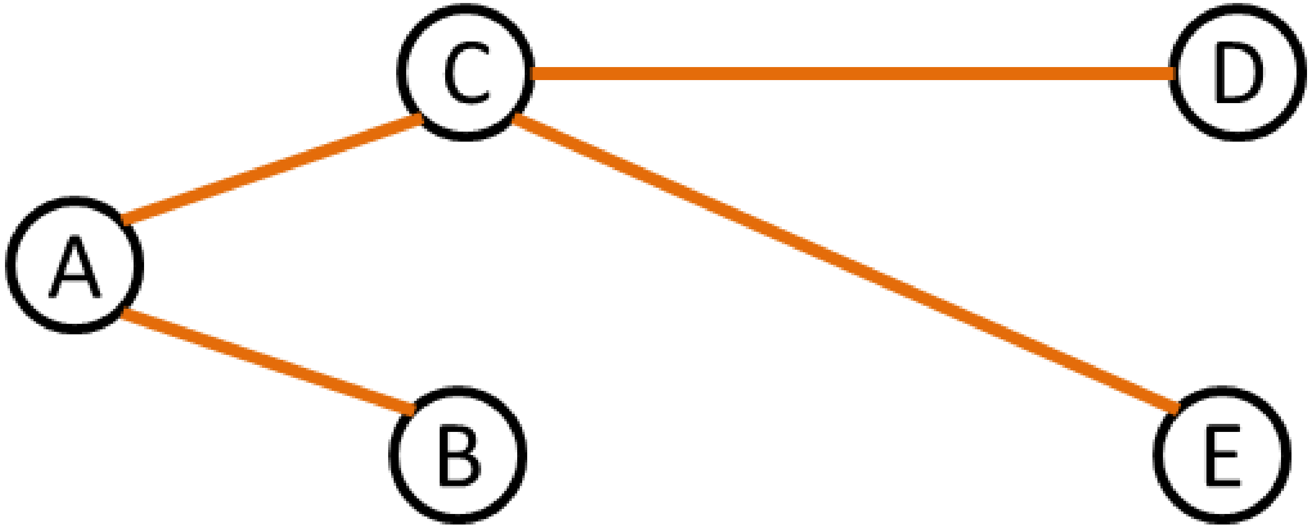
Notations: Goal

- State that you want to reach
- Final destination
- Check using **`isGoalState()`**

2. Graph Search Algorithms

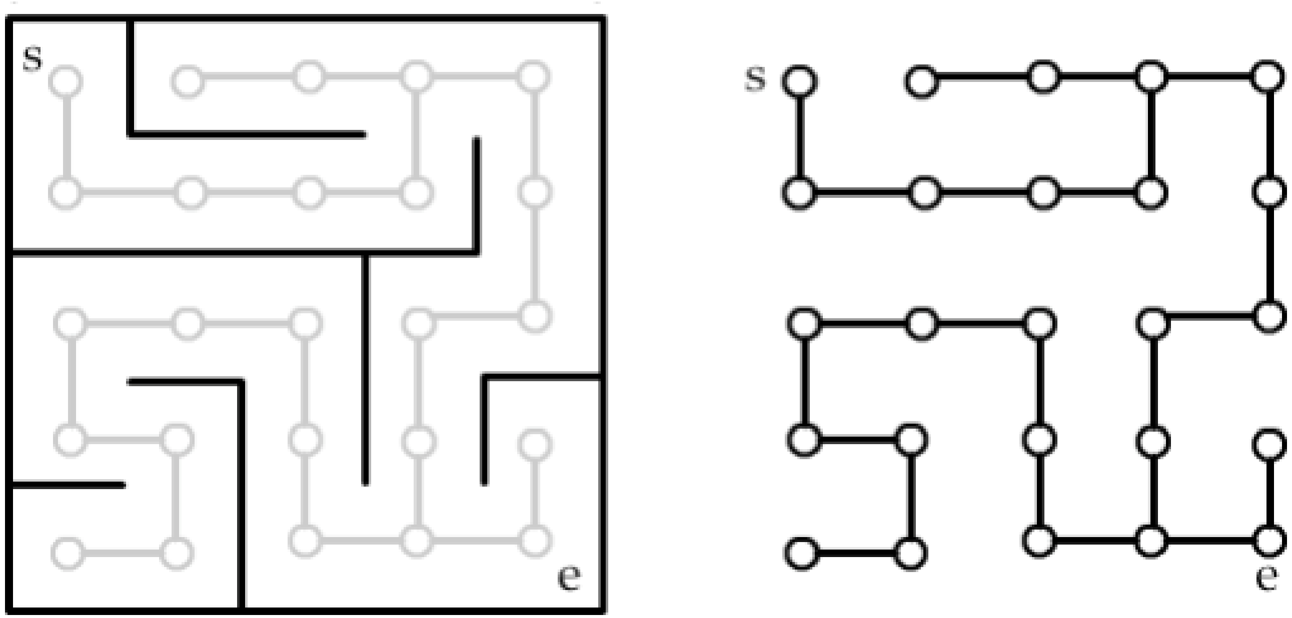
Graphs

- Graphs have **nodes** and edges



Representing grids as a graph

- Each cell becomes a node. Edges connect to adjacent cells.



Graph Searching Algorithms

- Begin at the start node and keep searching till we find the goal node
- Different algorithms → Different orders in which to search the nodes of the graph
- Open-list contains nodes that we've seen but haven't explored yet. Each iteration, we take a node off the open-list, and add its neighbors to the open-list.

Generic Search Algorithm

Algorithm 1 Generic Graph Search Algorithm

Input: *startState*, the state at which we begin our search.
Output: *path*, the sequence of actions to get from *startState* to a `goal state`

```
// Initialize variables
openList ⇐ dataStructure()          ▷ dataStructure ∈ {Stack, Queue, PriorityQueue}
closedList ⇐ set()                   ▷ Only explored nodes will go in here
metadata ⇐ createMetadata(startState, nil)
current ⇐ (startState, metadata)     ▷ Define current as the state plus metadata

// Search until goal state found.
while current.state is not a goal state do            ▷ Continue until goal reached
    p ⇐ current.state        ▷ Define p as the state we're interested in, the parent
    if p is not in closedList then                    ▷ If p is unexplored, explore it
        closedList.add(p)                             ▷ Mark p as explored.
        for s ∈ successors(p) do            ▷ Retrieve all successors for the parent
            s.metadata ⇐ createMetadata(s, p)   ▷ Create metadata using both p and s
            openList.add((s.state, s.metadata))  ▷ Add the successor onto the open list
    current ⇐ openList.next()           ▷ Fetch a new current from the openList

// Reconstruct path from startState to goal state
path ⇐ reconstructPath()            ▷ Reconstruct path to goal. See Section 2.4.
return path
```

Data Structures

Same code for all! Just need to change the open-list:

- **Queue** → Breadth-First Search (BFS)

- **Stack** → Depth-First Search (DFS)

- **Priority Queue** → Uniform Cost Search (UCS) and A*
  (Note: The priority functions are different for these two algorithms)

Metadata

We might need some metadata (~additional information) that we need to keep track of.

Wish to accomplish 2 things mainly:

1. Help keep track of the path taken to get to the goal
2. Allow priority of state to factor in for UCS and A*

Note:

- The metadata is problem-dependent
- Passing metadata in python can be easy – put everything in a tuple (including state), and store that tuple in the open-list.

1. Keeping track of path to goal

- **Full-Length Sequence of Actions:**
  - Simply keep a full-length list of actions taken to reach current state
  - So for each successor, the metadata will contain an entire sequence of actions to reach the parent, and a new action will be added to the end of that sequence to signify the action that takes you from s to that successor
  - *Once you complete the graph search, you'll have the full sequence of actions to reach the goal in current.metadata*
- **Parent Hash-map**
  - Note the action was taken to reach the successor, and we add this into the metadata.
  - When we begin exploring a state , we need to mark in a hashmap (*dict* in Python) the action taken to reach this state.
  - *Once we reach the goal state, we use the hashmap to back-track our way through the actions taken from each parent, untile we reach the startState*
- **Linked List**

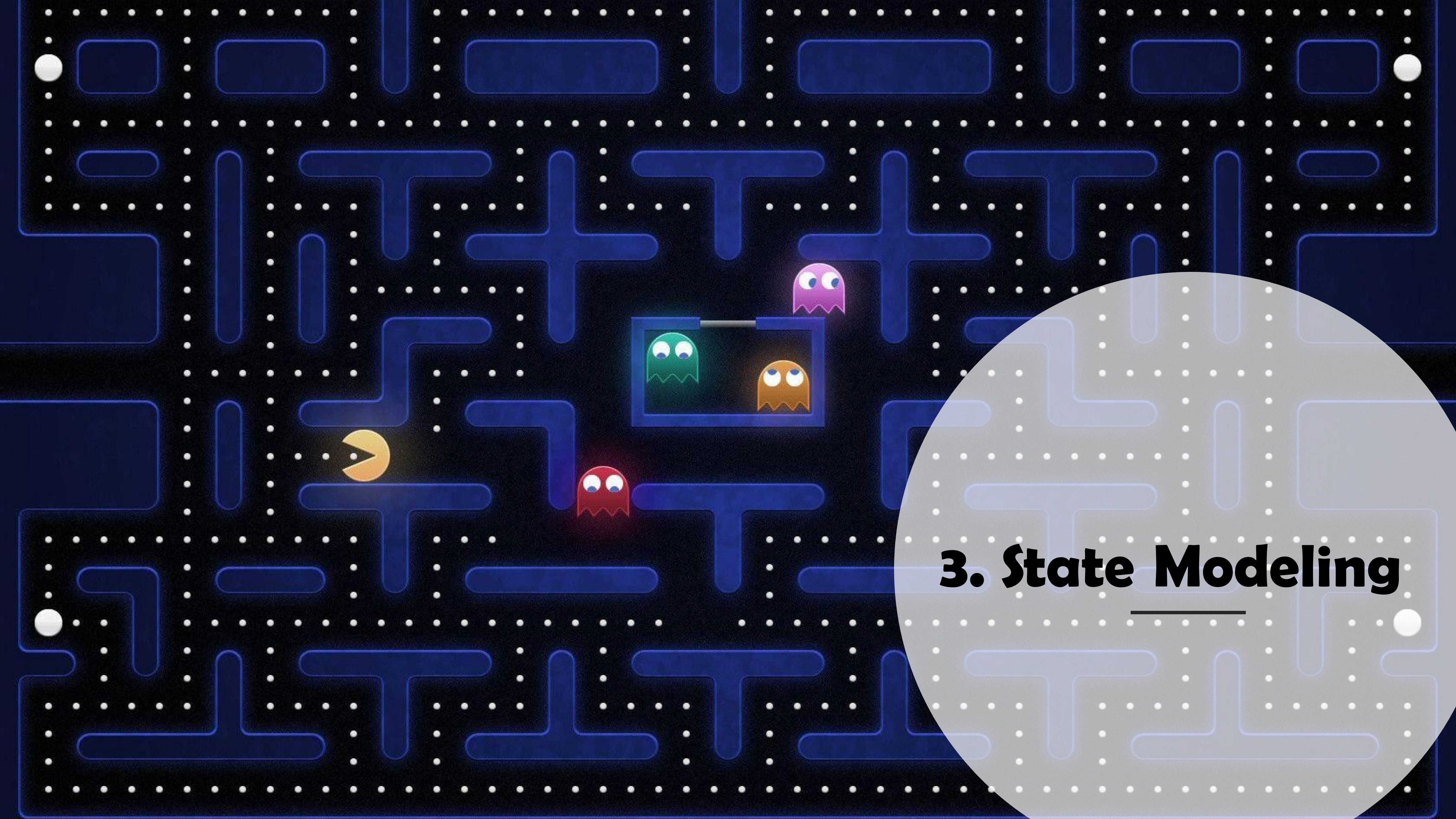
2. Determining Priority for A* and UCS

Priority needs to be passed around in the *metadata.*

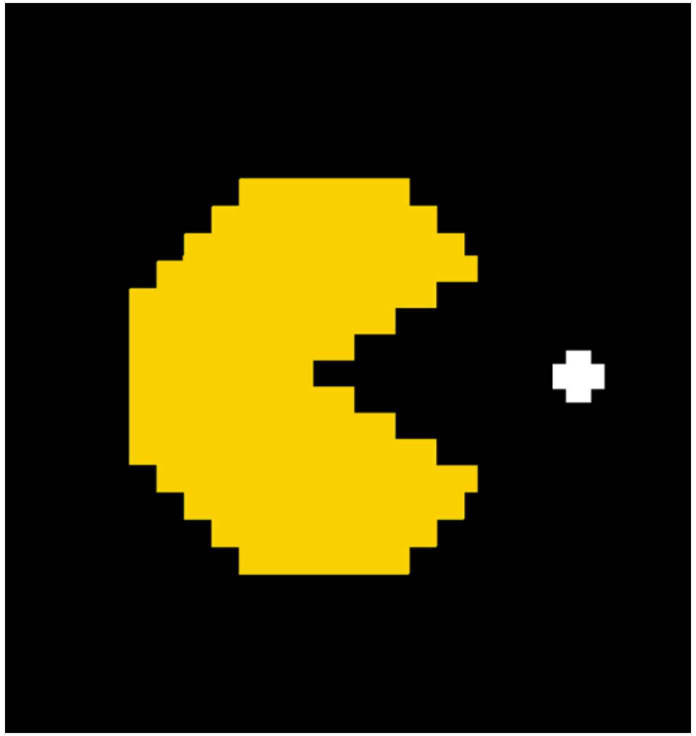
- **UCS**: priority(s) = g(s)
- **A***: priority(s) = g(s) + h(s)

g(s) tells us the cost of reaching state *s*. (therefore, of the ways that we've found to get to s so far, the cheapest of them is of cost *g(s)*)

h(s) <u>approximates</u> the remaining cost of reaching a goal state, starting from the current state *s*.

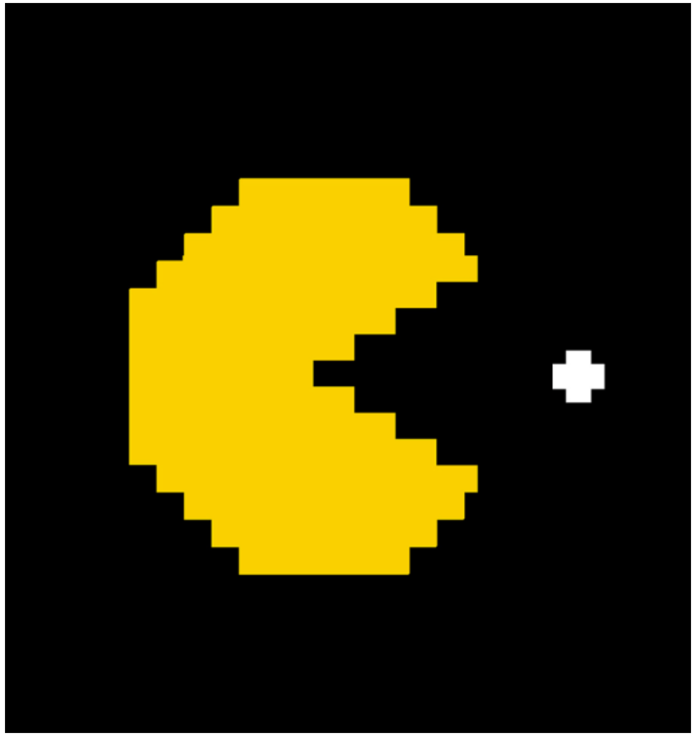


3. State Modeling



How do we model a search problem?

- **States**
- Successor Function
- Goal Function

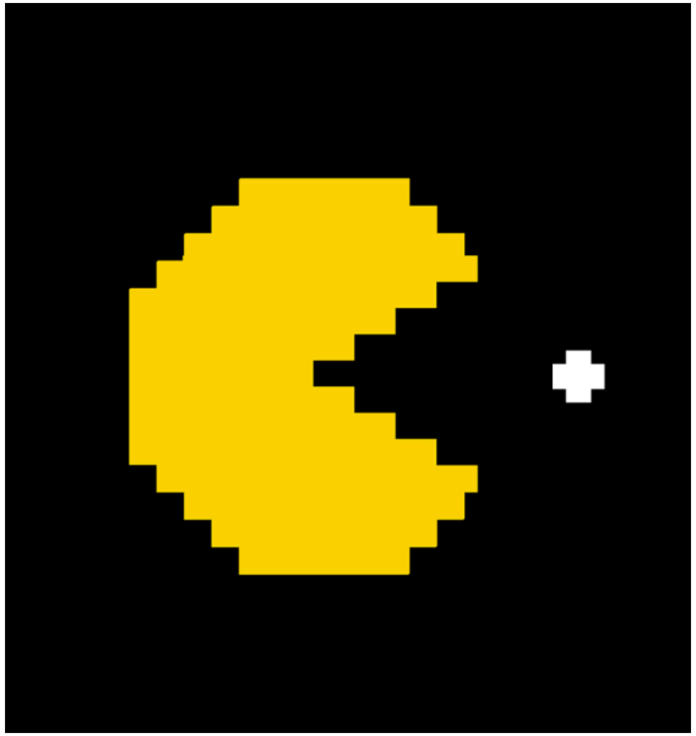


States:
Encoding Matters

- Only model what you need to solve the problem

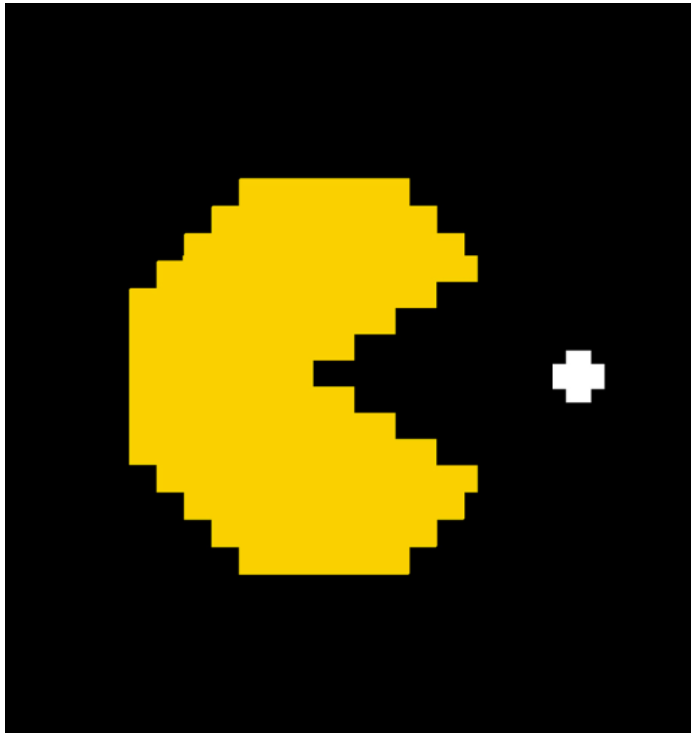
- What do we need to know about each state?
  - Its successors
  - If it is a goal state

- <u>Don't</u> try to model everything



How do we model a search problem?

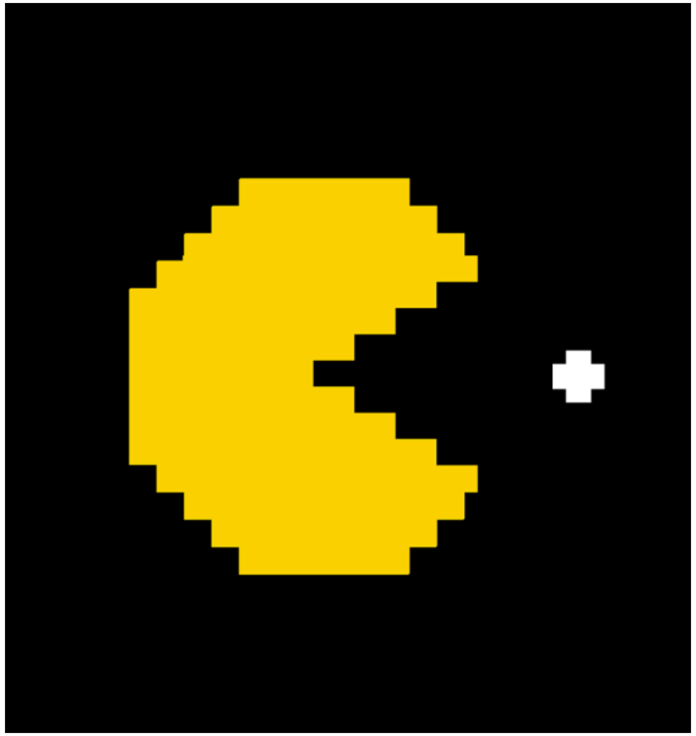
- States
- **Successor Function**
- Goal Function



Successor Function: Efficiency Matters

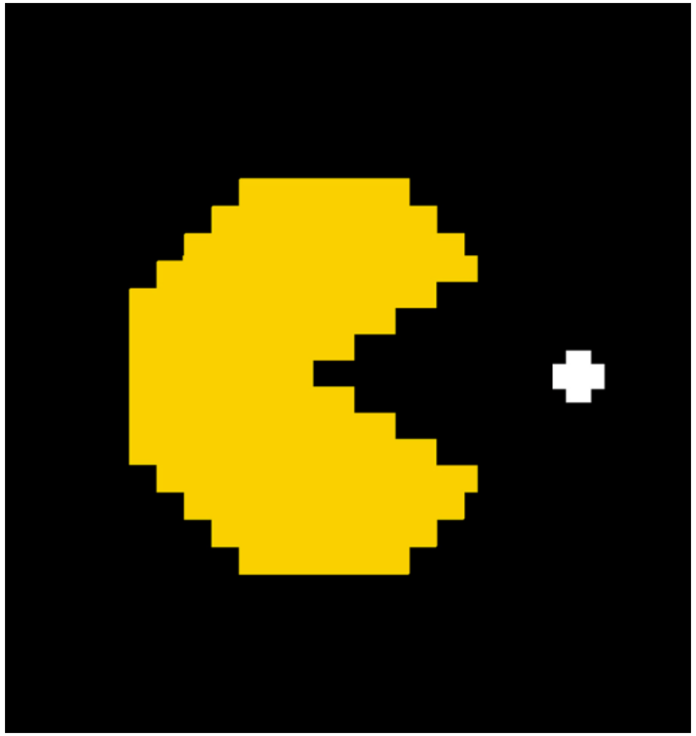
- How often is our successor function called?
  - Often, every time we add nodes to our open list

- Don't make this function slow to compute



How do we model a search problem?

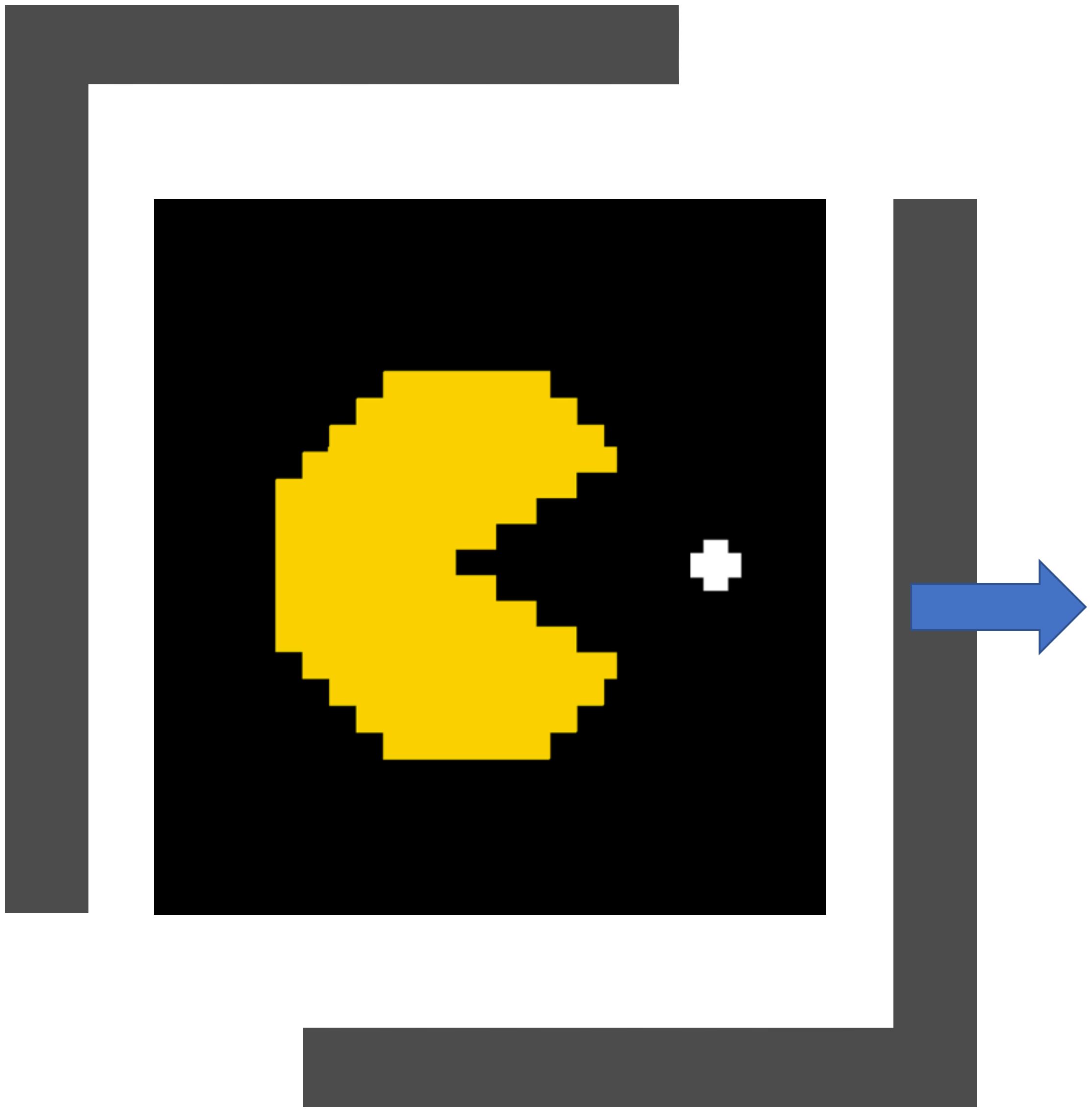
- States
- Successor Function
- **Goal Function**



Goal Function:
Efficiency Matters

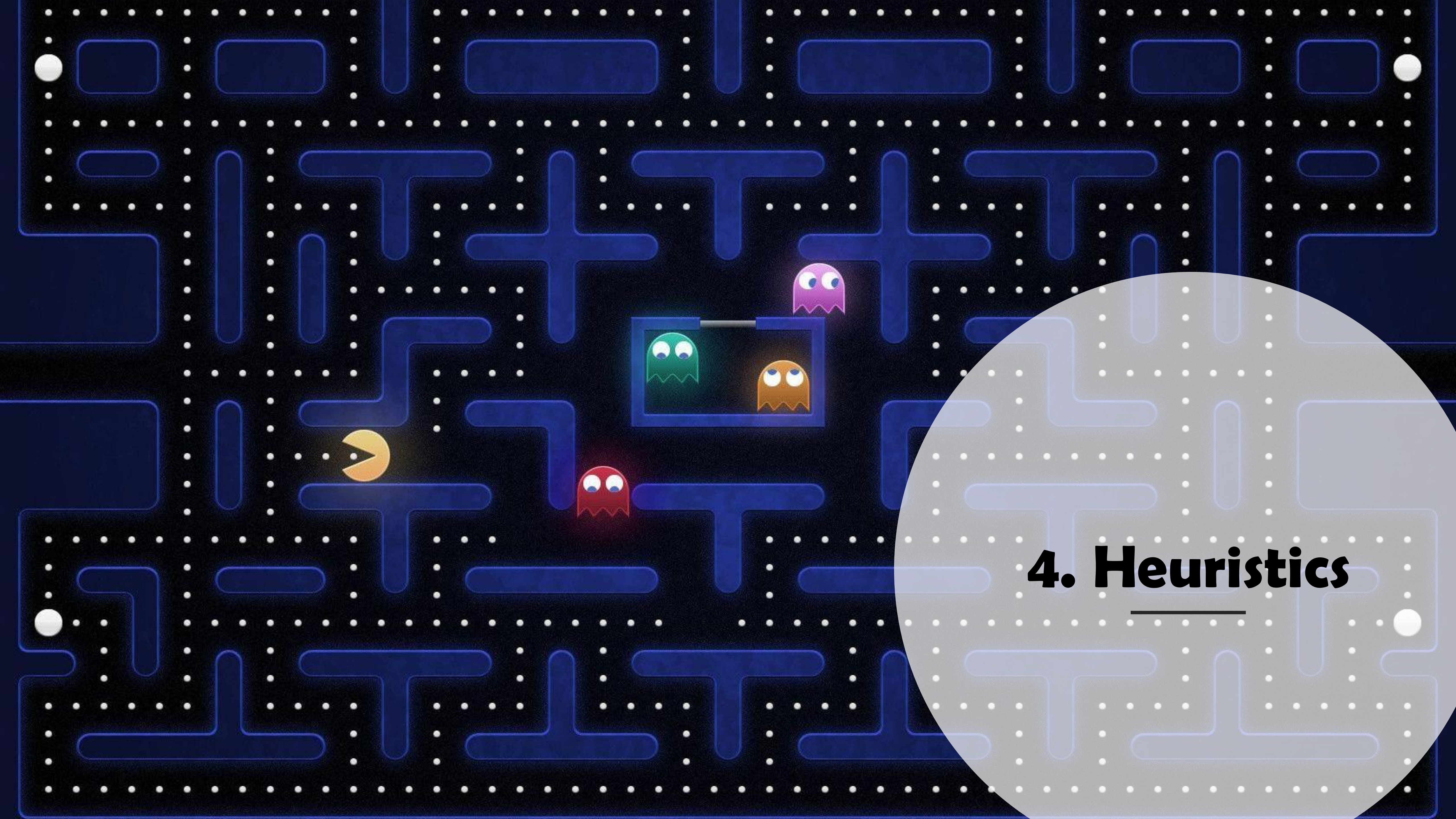
- How often is our goal function called?
    - Often, every time we remove from our open list

- Don't make this function slow to compute



Modeling Corners Problem

- States
  - Do we need to store the whole board?
  - Edit **`getStartState()`**
- Successor Function
  - Edit **`getSuccessors(state)`**
- Goal Function
  - Edit **`isGoalState(state)`**



4. Heuristics

<u>Heuristics: Definition!</u>

- **Quick Estimation / Rule of Thumb** for how **<u>close</u>** you are to the goal.
- When you are ***<u>AT the goal</u>***, the value of your heuristic should be **0**.
- Heuristics are **state specific**!!
- Give algorithms <u>intuitions</u> for focuses of optimization.

Heuristics: A* and UCS

- A* is designed to beat UCS because it is essentially **UCS + heuristics**.
- But only under certain conditions can A* beat UCS.

Heuristics: Good Heuristics

- **Admissibility**
- **Consistency**

- Also, a cute example to help y'all understand how heuristics are used in A*!

Admissible Heuristics

- **Never Overestimates!**
- Formal Definition:
  - Given a **true heuristic** represented as h*(s), then an admissible heuristic h(s) satisfies:

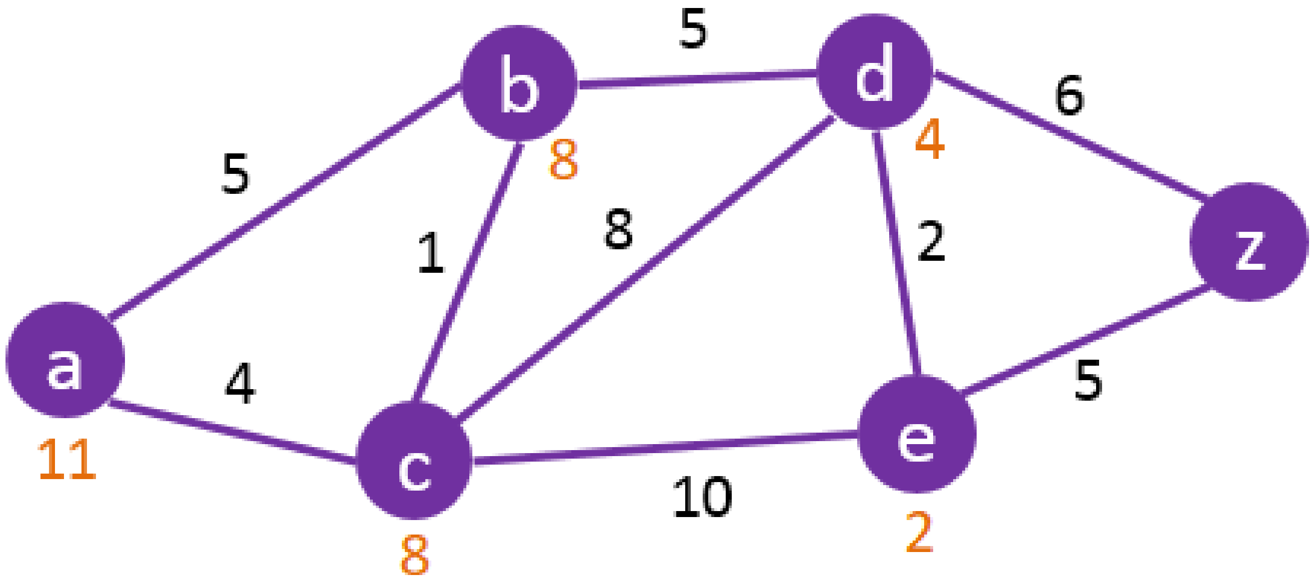
$$h(s) \le h^*(s) \text{ for all } s.$$

- Admissibility ensures the optimality of A*.

Consistent Heuristics

- The heuristic value should **decrease** when travelling from the parent to the child.
- *h(goal) = 0*
- *h(parent) - h(child) ≤ cost from parent to child*

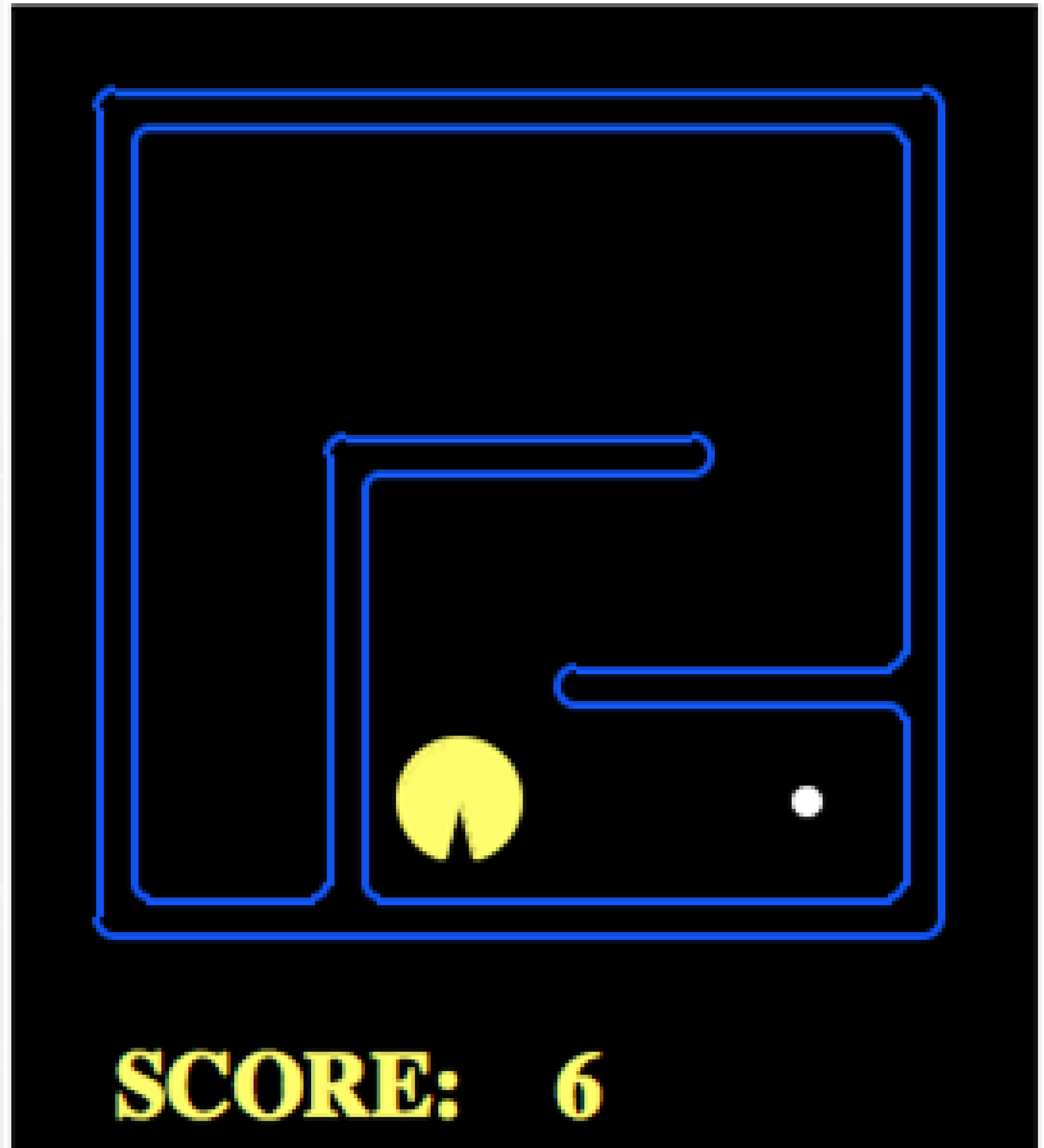
A* Example



Corners Heuristics

Goal

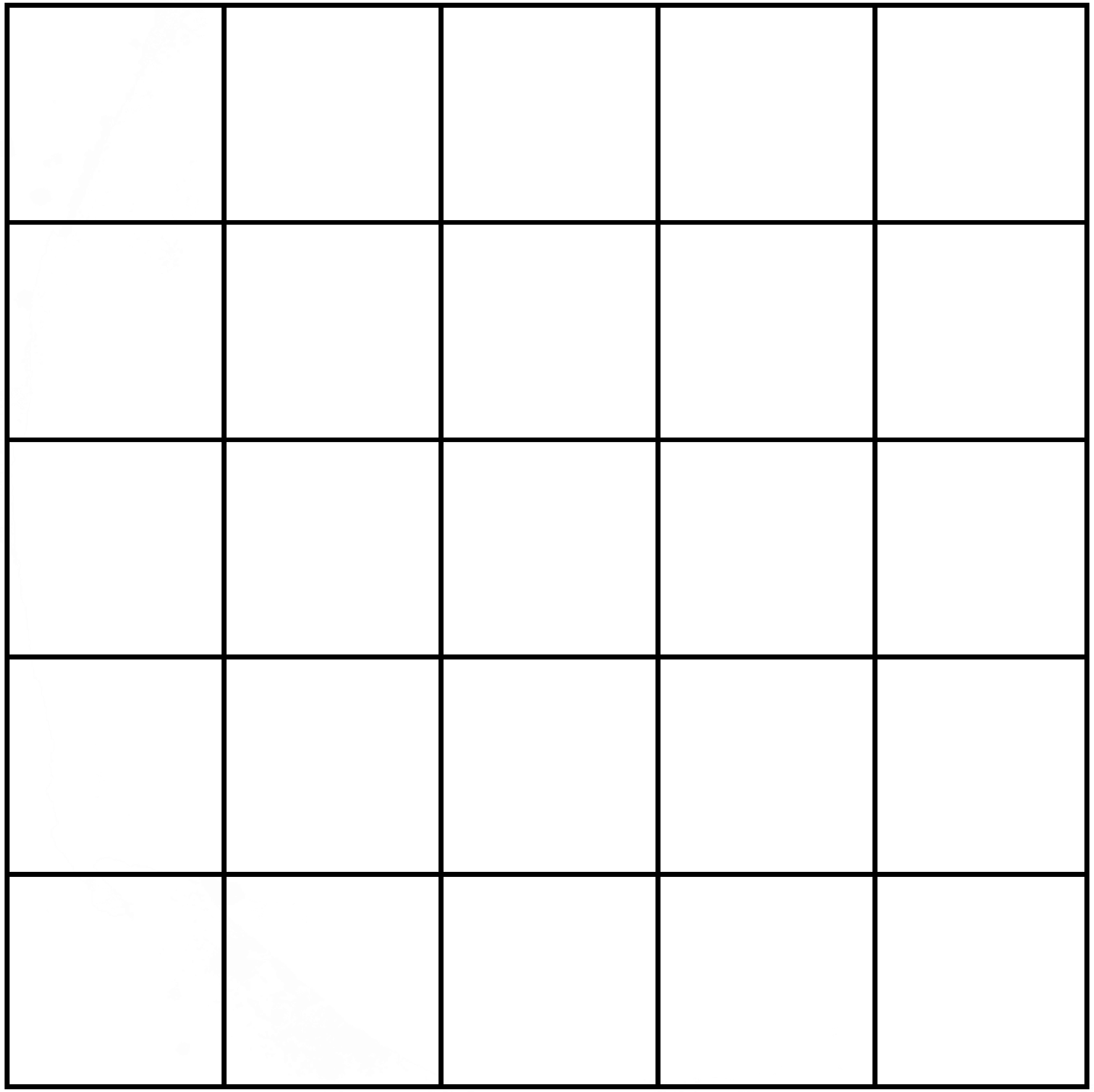
- Traverse all four corners as fast as possible
- TO EAT FOOD PELLETS BECAUSE SUSTENANCE IS IMPORTANT
- You can define the states to allow you to keep track of corners you have visited to help you calculate the heuristics values



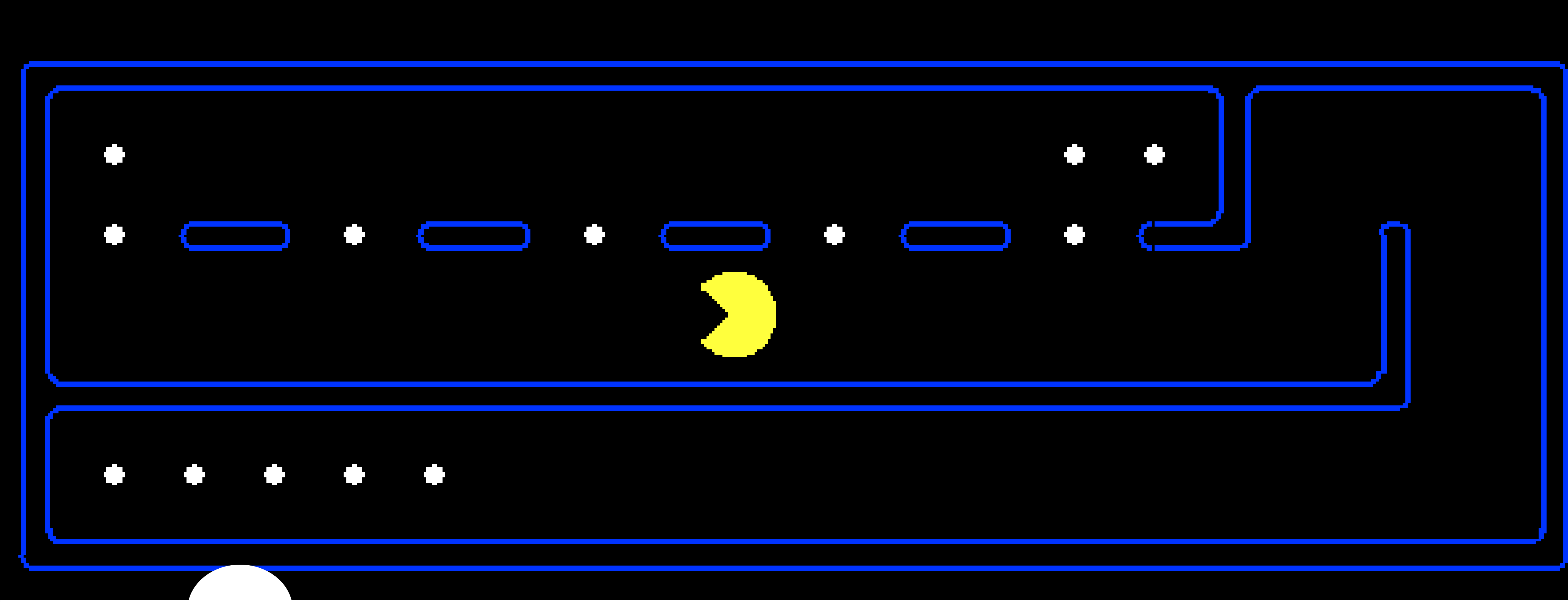
Some Common Approaches (and why they are wrong)

- Distance to the furthest pellet
- Distance to the closest pellet
  - Both approaches do not account for changing in the number of remaining pellets
- Sum of distances to all pellets

- Therefore, need to incorporate all possible information about all food pellets.

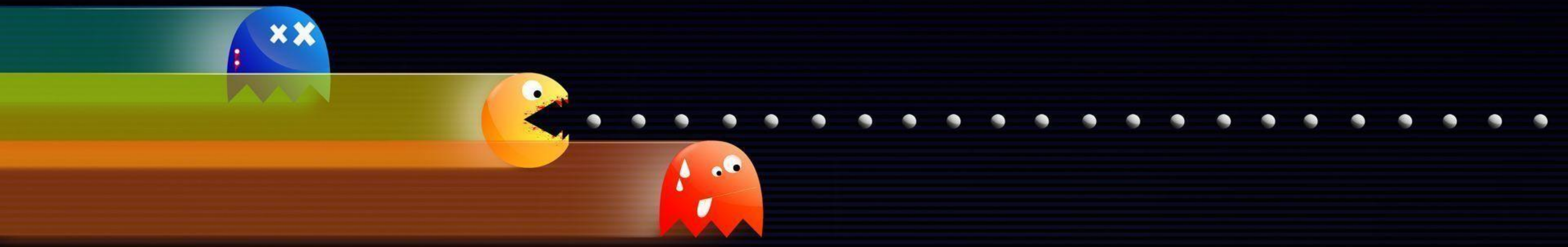


Food Heuristics



Hints

- Q6 is more statically defined (all pellets are placed at the corners)
- For Q7, the pellets are distributed across the whole grid
- Must come up with creative ways to incorporate all information



Questions?