

Pacman AI - UC Berkeley Pacman AI

Do you know the UC Berkely Pacman AI projects?

Yes — I know the **UC Berkeley Pacman AI Projects** pretty well.

They're a series of programming assignments created by UC Berkeley for their **CS188: Introduction to Artificial Intelligence** course. Students use them to implement and experiment with core AI techniques in a fun, visual environment (a Pacman game).

The projects are structured into multiple parts, each focusing on a different AI topic, for example:

1. **Project 0 – Python/Setup:** Intro to Python and the autograder.
2. **Project 1 – Search:** Implement DFS, BFS, UCS, and A* search to guide Pacman through mazes.
3. **Project 2 – Multi-Agent Search:** Build adversarial agents (Minimax, Alpha-Beta pruning, Expectimax) to play Pacman against ghosts.
4. **Project 3 – Reinforcement Learning:** Implement Q-learning and approximate Q-learning for Pacman in stochastic environments.
5. **Project 4 – Ghostbusters (Bayes Nets):** Use probabilistic inference and Hidden Markov Models to track hidden ghosts.
6. **Project 5 – Classification:** Apply machine learning (naive Bayes, perceptrons) to tasks like digit classification.
7. **Project 6 – Neural Networks:** Train neural networks for tasks like digit recognition.

They're widely used in AI and ML courses outside Berkeley too because they're open-source, well-documented, and fun — students get to *see Pacman behave smarter* as they code the algorithms.

👉 Do you want me to just give you an overview, or are you planning to **work on one of the projects** (e.g., search, reinforcement learning, etc.) and want some detailed guidance?

Yes I need to do Project 1, and want some detailed

Awesome — let's get you through **Project 1: Search** fast and clean. I'll give you (1) a do-this-next plan with the exact test commands, (2) solid reference implementations for DFS/BFS/UCS/A*, and (3) minimal, safe code stubs for the Corners/Food/ClosestDot parts so you can pass the autograder and iterate on heuristics later.

Notes (Fall 2025 specifics): which files to edit, autograder cues, and the exact commands below are from the current spec. I cite those bits inline.

Game plan (order + checks)

1. Q1 – DFS → `search.py: depthFirstSearch`

Test:

```
bash

python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch # sanity
python pacman.py -l tinyMaze -p SearchAgent # your DFS
python pacman.py -l mediumMaze -p SearchAgent
python pacman.py -l bigMaze -z .5 -p SearchAgent
python autograder.py -q q1
```

(DFS on `mediumMaze` is length ~130 if you push successors in given order.)

2. Q2 – BFS → `search.py: breadthFirstSearch`

Test:

```
bash

python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5
python autograder.py -q q2
``` :contentReference[oaicite:2]{index=2}
```

### 3. Q3 – UCS → `search.py: uniformCostSearch`

Test:

```
bash
```

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
python pacman.py -l mediumDottedMaze -p StayEastSearchAgent
python pacman.py -l mediumScaryMaze -p StayWestSearchAgent
python autograder.py -q q3
``` :contentReference[oaicite:3]{index=3}
```

4. Q4 - A* → search.py: aStarSearch

Test:

```
bash

python pacman.py -l bigMaze -z .5 -p SearchAgent -a
fn=astar,heuristic=manhattanHeuristic
python autograder.py -q q4
```

(A* slightly fewer expansions than UCS on bigMaze.)

5. Q5 - CornersProblem → define the problem in searchAgents.py , then BFS it.

Test:

```
bash

python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
```

Then Q6 - cornersHeuristic (A* + heuristic, graded by node count).

```
bash

python pacman.py -l mediumCorners -p AStarCornersAgent
python autograder.py -q q6
```

(≤1200 nodes = full credit; must be **consistent**.)

6. Q7 - foodHeuristic for FoodSearchProblem (A*).

Test:

```
bash

python pacman.py -l testSearch -p AStarFoodSearchAgent
python pacman.py -l trickySearch -p AStarFoodSearchAgent
python autograder.py -q q7
```

(≤ 9000 nodes = full credit; ≤ 7000 is extra credit; must be **consistent**.)

7. Q8 – ClosestDot (greedy) → fill `findPathToClosestDot` (+
AnyFoodSearchProblem).

Test:

bash

```
python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5  
python autograder.py -q q8  
``` :contentReference[oaicite:7]{index=7}
```

## 1) Drop-in implementations: DFS / BFS / UCS / A\*

Paste these into `search.py` (use the provided `util.Stack/Queue/PriorityQueue` ).

These follow the project's graph-search expectations and play nicely with the autograder.

python

```
search.py
import util

def depthFirstSearch(problem):
 """Search the deepest nodes in the search tree first.
 """
 start = problem.getStartState()
 fringe = util.Stack()
 fringe.push((start, []))
 visited = set()

 while not fringe.isEmpty():
 state, actions = fringe.pop()
 if state in visited:
 continue
 visited.add(state)

 if problem.isGoalState(state):
 return actions

 for succ, act, step in problem.getSuccessors(state):
```

```

if succ not in visited:
 fringe.push((succ, actions + [act]))
return [] # no solution

def breadthFirstSearch(problem):
 """Search the shallowest nodes in the search tree first."""
 start = problem.getStartState()
 fringe = util.Queue()
 fringe.push((start, []))
 visited = set([start])

 while not fringe.isEmpty():
 state, actions = fringe.pop()

 if problem.isGoalState(state):
 return actions

 for succ, act, step in problem.getSuccessors(state):
 if succ not in visited:
 visited.add(succ)
 fringe.push((succ, actions + [act]))
 return []

def uniformCostSearch(problem):
 """Search the node of least total cost first."""
 start = problem.getStartState()
 fringe = util.PriorityQueue()
 fringe.push((start, [], 0), 0)
 best_g = {start: 0}

 while not fringe.isEmpty():
 state, actions, g = fringe.pop()

 # Outdated entry check
 if g > best_g.get(state, float('inf')):
 continue

 if problem.isGoalState(state):
 return actions

 for succ, act, step in problem.getSuccessors(state):
 ng = g + step

```

```

if ng < best_g.get(succ, float('inf')):
 best_g[succ] = ng
 fringe.push((succ, actions + [act], ng), ng)
return []

```

  

```

def nullHeuristic(state, problem=None):
 return 0

```

  

```

def aStarSearch(problem, heuristic=nullHeuristic):
 """Search the node with the lowest g(n)+h(n)."""
 start = problem.getStartState()
 fringe = util.PriorityQueue()
 fringe.push((start, [], 0), heuristic(start, problem))
 best_g = {start: 0}

 while not fringe.isEmpty():
 state, actions, g = fringe.pop()
 if g > best_g.get(state, float('inf')):
 continue

 if problem.isGoalState(state):
 return actions

 for succ, act, step in problem.getSuccessors(state):
 ng = g + step
 if ng < best_g.get(succ, float('inf')):
 best_g[succ] = ng
 f = ng + heuristic(succ, problem)
 fringe.push((succ, actions + [act], ng), f)
 return []

```

## 2) CornersProblem + cornersHeuristic (Q5/Q6)

Paste these into `searchAgents.py`. They're minimal and safe; you can upgrade the heuristic later for more points.

```

python

searchAgents.py (additions)
from game import Directions
import util

```

```

import search

class CornersProblem(search.SearchProblem):
 """
 State = (position: (x,y), visited: frozenset of corner positions)
 Cost of each step = 1
 """

 def __init__(self, startingGameState):
 self.walls = startingGameState.getWalls()
 self.startingPosition = startingGameState.getPacmanPosition()

 top, right = self.walls.height - 2, self.walls.width - 2
 self.corners = ((1,1), (1, top), (right,1), (right, top))

 # Precompute corner set
 self.cornerSet = set(self.corners)

 def getStartState(self):
 visited = frozenset(
 [c for c in self.corners if c == self.startingPosition]
)
 return (self.startingPosition, visited)

 def isGoalState(self, state):
 _, visited = state
 return len(visited) == 4

 def getSuccessors(self, state):
 (x, y), visited = state
 successors = []
 for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
 dx, dy = Actions.directionToVector(action) # Actions is available in searchAgents.py
 nx, ny = int(x + dx), int(y + dy)
 if not self.walls[nx][ny]:
 npos = (nx, ny)
 nvisited = visited
 if npos in self.cornerSet:
 nvisited = visited | {npos}
 successors.append(((npos, nvisited), action, 1))
 return successors

 def getCostOfActions(self, actions):

```

```

if actions is None: return 999999
x, y = self.startingPosition
for a in actions:
 dx, dy = Actions.directionToVector(a)
 x, y = int(x + dx), int(y + dy)
 if self.walls[x][y]:
 return 999999
return len(actions)

def cornersHeuristic(state, problem):
 """
 Simple, admissible & consistent:
 h = max Manhattan distance to any unvisited corner.
 """
 (x, y), visited = state
 remaining = [c for c in problem.corners if c not in visited]
 if not remaining:
 return 0
 return max(util.manhattanDistance((x, y), c) for c in remaining)

```

**Why this works / how to get more points:** the “max distance to an unvisited corner” is a cheap, consistent lower bound; it will pass the consistency check. To reduce node expansions (toward  $\leq 1200$  for full credit), try a stronger bound such as **distance to the nearest unvisited corner plus a Minimum Spanning Tree (MST) over the remaining corners (using Manhattan edge weights)** — still admissible/consistent because MST is a lower bound on the traveling cost among remaining goals. (Keep it light; compute MST on just the few remaining corners.)

Grading thresholds and consistency requirement are per spec.

### 3) FoodSearchProblem heuristic (Q7)

Put this into `searchAgents.py`. Start simple and consistent; you can iterate if you need fewer nodes.

```

python

def foodHeuristic(state, problem):
 """
 Consistent baseline: farthest food by Manhattan distance.

```

```

State = (position, foodGrid)
"""

position, foodGrid = state
foods = foodGrid.asList()
if not foods:
 return 0
Manhattan is an admissible metric lower bound for maze distance
return max(util.manhattanDistance(position, f) for f in foods)

```

If you need stronger (fewer expansions on `trickySearch`), consider:

- **MST over remaining foods  $\cup \{position\}$  (Manhattan edges)** + maybe “distance from position to nearest food” (still a lower bound). Memoize by `frozenset(foods)` to keep it fast.
- Avoid `mazeDistance` inside the heuristic loop unless you cache heavily — it’s accurate but slow.

Targets for Q7 node counts are in the spec ( $\leq 9000$  for full,  $\leq 7000$  for extra credit).

---

## 4) ClosestDot (Q8)

Finish the greedy agent by adding this to `searchAgents.py`:

```

python

class AnyFoodSearchProblem(PositionSearchProblem):
"""

Goal test = any food square.

"""

def __init__(self, gameState):
 super().__init__(gameState)
 self.food = gameState.getFood()

def isGoalState(self, state):
 x, y = state
 return self.food[x][y]

def findPathToClosestDot(self, gameState):
"""

In ClosestDotSearchAgent: run BFS on AnyFoodSearchProblem.

"""

```

```
problem = AnyFoodSearchProblem(gameState)
return search.breadthFirstSearch(problem)
```

That matches the intended short-and-sweet approach.

---

## Debugging + common pitfalls

- **Use the provided data structures** (`util.Stack/Queue/PriorityQueue`) — the autograder expects them.
- **Visited vs. costs:**
  - DFS/BFS: a simple `visited` set is fine (mark on pop for DFS; mark on push for BFS to avoid duplicates).
  - UCS/A\*: track **best g** seen per state and skip outdated queue entries.
- **State choice for Corners:** keep it compact: `(pos, visitedCorners)`. Never store full GameState as search state (slow/wrong).
- **Heuristics:** must be **non-negative, 0 at goals, and consistent** or you'll get 0 credit on those questions.
- **Speed:** if visualization is slow, run with `-z .5` or `--frameTime 0`.

If you want, I can also drop a stronger (still short) MST-based heuristic snippet for Corners or Food next.

### Sources

```
def breadthFirstSearch(problem):
 """Search the shallowest nodes in the search tree
 first."""
 start = problem.getStartState()
 fringe = util.Queue()
 fringe.push((start, []))
 visited = set([start])

 while not fringe.isEmpty():
 state, actions = fringe.pop()

 if problem.isGoalState(state):
 return actions

 for succ, act, step in problem.getSuccessors(state):
 if succ not in visited:
```

```
 visited.add(succ)
 fringe.push((succ, actions + [act]))
return []
```

Why did you put start in a list. `visited = set([start])`

Good eye — that `visited = set([start])` line is just **Python shorthand for making a set with one element (the start state)**.

Let's unpack:

- `set()` creates an empty set.
- `set([x])` makes a set containing `x`.

python

```
set([start]) # → {start}
```

- So `visited = set([start])` means: “mark the start node as already seen.”

This is important for **BFS** because if you *don't* add the start immediately, you'll end up re-pushing it when iterating successors, which can waste time or even loop.

If you find that syntax odd, you can write the same thing in a couple of cleaner ways:

```
python

visited = {start} # literal syntax, most common
or
visited = set()
visited.add(start)
```

All three are equivalent — just different styles.

In modern Python, most people would use the `{start}` form.

👉 Do you want me to rewrite your BFS using the `{start}` style (to keep it cleaner), and also explain why BFS marks visited on *push* instead of *pop*?

we can't just do `set(start)`

Almost — but not quite 😊.

`set(start)` will try to **iterate over** `start` and put each element of it into the set.

- If `start` is a string like `"A"`, then `set(start) → {"A"}` (works by coincidence).
- But if `start` is a tuple like `(5, 6)`, then `set(start) → {5, 6}` (splits the tuple!) — not what you want.
- Worse, if `start` is something non-iterable, `set(start)` will throw an error.

What you really want is:

```
python
```

```
visited = {start} # a set with one element
```

That guarantees the set contains the **entire state object** as a single element, exactly like `set([start])` but cleaner.

👉 So in BFS, `visited = {start}` is the safe and pythonic way.

Would you like me to also show you why in BFS we add `start` to `visited` **at the beginning (on push)**, but in DFS we usually add it **after popping**?

```
def getStartState(self):
 visited = frozenset(
 [c for c in self.corners if c ==
 self.startingPosition]
)
 return (self.startingPosition, visited)
```

Do we need to use a `frozenset`, can we do something else?

Great question 👍

The short answer: **Yes, you need something immutable and hashable** to represent the set of visited corners, because the search framework sticks your state into data structures like `set()` and `dict` for `visited` and `best_g`.

**Why not just use a normal `set` ?**

- Normal `set` in Python is **mutable** → you can't put it inside another `set` or use it as a dictionary key.

```
python

visited = set([corner]) # works as a set
state = (pos, visited) # ✗ TypeError: unhashable type: 'set'
```

## Alternatives to `frozenset`

### 1. Tuple of corners

- Example: `visited = tuple(sorted([corner1, corner2]))`
- Tuples are immutable and hashable.
- Downside: you must keep them sorted so `{A, B}` and `{B, A}` are the same.

### 2. Bitmask (more advanced)

- If you number the corners 0–3, you can encode visited as an integer from `0b0000` to `0b1111`.
- Example: `visited = 0b0101` means corners 0 and 2 are visited.
- Super fast, but less readable.

### 3. `frozenset` (default, recommended)

- `frozenset({corner})` → immutable, hashable, order doesn't matter.
- Exactly what you want, and the project staff intended it.

👉 For **Project 1**, stick with `frozenset`. It's the cleanest and safest.

If you want to experiment with performance tricks later, you could switch to a **tuple of visited corners** or a **bitmask**, but that's optional.

Do you want me to show you a tuple-based version of `getStartState` so you can compare it to the `frozenset` one?

yes

Perfect 👍 — here's how you'd do it with a **tuple instead of a frozenset**:

```
python
```

```
def getStartState(self):
 visited = ()
 if self.startingPosition in self.corners:
 visited = (self.startingPosition,)
 return (self.startingPosition, visited)
```

## Key differences from `frozenset`:

- `visited` is always a **tuple** of corner coordinates.
- When you visit a new corner, you'd do:

python

```
newVisited = tuple(sorted(visited + (npos,)))
```

- `visited + (npos, )` appends the new corner.
- `sorted(...)` ensures `(A, B)` and `(B, A)` end up the same (so states aren't duplicated).

## Pros:

- Still immutable + hashable (so safe for sets/dicts).
- Easy to read in debug prints.

## Cons:

- Sorting on every update costs a little extra time.
- `frozenset` is cleaner because it's order-independent by design (no sorting needed).

---

👉 So tuple works, but you have to **remember to keep it sorted** to avoid duplicated states.

That's why the Berkeley project staff recommend `frozenset`.

Would you like me to also show how the **successor function** changes slightly if you use a tuple instead of a frozenset?

I'll stick with the frozen set, how do we do without list comprehension

Good call  sticking with `frozenset` makes the rest of the code easier.

Right now you've got:

```
python

visited = frozenset(
 [c for c in self.corners if c == self.startingPosition]
)
```

That's a list comprehension wrapped in a `frozenset`.

You don't need the list step at all — you can build the `frozenset` directly.

## Cleaner ways without list comprehension:

### 1. Conditional `frozenset`:

```
python

if self.startingPosition in self.corners:
 visited = frozenset({self.startingPosition})
else:
 visited = frozenset()
```

### 2. One-liner with conditional expression:

```
python

visited = frozenset({self.startingPosition}) if self.startingPosition in self.corners else
frozenset()
```

Both produce either an **empty `frozenset`** (if start isn't a corner) or a **`frozenset` with the start corner**.

That's exactly what your list comprehension version was doing, but without the unnecessary list.

---

 Do you want me to also show how you'd `update` `visited` `inside` `getSuccessors` with `frozenset` (so you never mutate it)?

## Question q6

二二二二二二二二

\*\*\* PASS: heuristic value less than true cost at start state

\*\*\* PASS: heuristic value less than true cost at start state

\*\*\* PASS: heuristic value less than true cost at start state

```
path: ['North', 'East', 'East', 'East', 'East', 'North', 'North',
'West', 'West', 'West', 'West', 'West', 'West', 'South',
'South', 'South', 'West', 'West', 'North', 'East', 'East',
'North', 'North', 'North', 'North', 'East', 'East', 'North',
'North', 'North', 'North', 'North', 'North', 'West', 'West',
'West', 'West', 'South', 'South', 'East', 'East', 'East', 'East',
'South', 'South', 'South', 'South', 'South', 'South', 'East',
'East', 'East', 'East', 'East', 'East', 'South', 'South', 'East',
'East', 'East', 'East', 'East', 'North', 'North', 'East', 'East',
'North', 'North', 'East', 'East', 'North', 'North', 'East', 'East',
'East', 'East', 'East', 'South', 'South', 'South', 'South',
'East', 'East', 'North', 'North', 'East', 'East', 'South',
'South', 'South', 'South', 'South', 'North', 'North', 'North',
'North', 'North', 'North', 'North', 'West', 'West', 'North',
'North', 'East', 'East', 'North', 'North']
```

path length: 106

\*\*\* FAIL: Heuristic resulted in expansion of 845707 nodes

### ### Question q6: 0/3 ###

Your heuristic is way too weak (you're using something like "max distance to an unvisited corner"), so A\* is expanding almost the whole state space. For **Q6** you need a stronger—but still admissible & consistent—bound.

Use this classic lower bound:

$h(\text{state}) = (\text{distance from current position to the nearest unvisited corner}) + (\text{MST cost over the remaining corners})$

Use **Manhattan** distances for both parts. Manhattan is a metric  $\Rightarrow$  triangle inequality  $\Rightarrow$  the nearest-corner + MST sum stays **admissible and consistent**.

Drop this into `searchAgents.py` to replace your `cornersHeuristic`:

```
python
```

```
def cornersHeuristic(state, problem):
 """
 Admissible & consistent heuristic for CornersProblem:
 h = min_{c in R} dist(pos, c) + MST(R) using Manhattan distances,
 where R is the set of unvisited corners.
 """

 position, visited = state
 corners = problem.corners
 remaining = [c for c in corners if c not in visited]
 if not remaining:
 return 0

 # Manhattan distance helper (shorter alias)
 mdist = util.manhattanDistance

 # Part 1: distance from current position to the nearest remaining corner
 to_nearest = min(mdist(position, c) for c in remaining)

 # Part 2: MST over the remaining corners using Manhattan edge weights.
 # Since at most 4 corners remain, a tiny Prim's algorithm is fine.
 if len(remaining) == 1:
 mst_cost = 0
 else:
 unreached = set(remaining)
 # Start Prim's from an arbitrary node
 current = unreached.pop()
 reached = {current}
 mst_cost = 0

 while unreached:
 # edge with minimal cost between reached and unreached
 # (Prim's algorithm step)
 next_node = min(
 unreached,
 key=lambda v: min(mdist(u, v) for u in reached)
)
 add_cost = min(mdist(u, next_node) for u in reached)
 mst_cost += add_cost
```

```

reached.add(next_node)
unreached.remove(next_node)

return to_nearest + mst_cost

```

Why this fixes the blow-up:

- The **nearest-corner** term avoids underestimating the first leg.
- The **MST** term is a lower bound on the total travel needed to cover the rest of the corners (it's  $\leq$  the length of any tour), so A\* stops exploring wild detours.

This heuristic typically brings **mediumCorners** expansions down under the grader's threshold ( $\leq 1200$ ) and passes consistency. Run:

bash

```

python pacman.py -l mediumCorners -p AStarCornersAgent
python autograder.py -q q6

```

If you still see high expansions, double-check:

- State = `(position, frozenset(visitedCorners))`
- `getSuccessors` never mutates `visited`; it creates a new `frozenset` when stepping onto a corner.
- You're actually calling A\* for corners: `-p AStarCornersAgent` (or `SearchAgent -a fn=astar, prob=CornersProblem, heuristic=cornersHeuristic`).

```
class CornersProblem(search.SearchProblem):
```

```
 """
```

This search problem finds paths through all four corners of a layout.

You must select a suitable state space and successor function

```
 """
```

```
def __init__(self, startingGameState:
pacman.GameState):
```

```
 """
```

Stores the walls, pacman's starting position and corners.

```
 """
```

```

 self.walls = startingGameState.getWalls()
 self.startingPosition =
startingGameState.getPacmanPosition()
 top, right = self.walls.height-2, self.walls.width-2
 self.corners = ((1,1), (1,top), (right, 1), (right, top))
 for corner in self.corners:
 if not startingGameState.hasFood(*corner):
 print('Warning: no food in corner ' +
str(corner))
 self._expanded = 0 # DO NOT CHANGE; Number of
search nodes expanded

def getStartState(self):
 """
 Returns the start state (in your state space, not the
full Pacman state
 space)
 """
 """ YOUR CODE HERE """
 if self.startingPosition in self.corners:
 visited = frozenset({self.startingPosition})
 else:
 visited = frozenset()

 return (self.startingPosition, visited)

def isGoalState(self, state: Any):
 """
 Returns whether this search state is a goal state of
the problem.
 """
 """ YOUR CODE HERE """
 position, visited = state
 return len(visited) == 4

def getSuccessors(self, state: Any):
 """
 Returns successor states, the actions they require,
and a cost of 1.

As noted in search.py:
 For a given state, this should return a list of

```

```

triples, (successor,
 action, stepCost), where 'successor' is a
successor to the current
 state, 'action' is the action required to get there,
and 'stepCost'
 is the incremental cost of expanding to that
successor
 """
position, cornersVisited = state
successors = []
for action in [Directions.NORTH, Directions.SOUTH,
Directions.EAST, Directions.WEST]:
 # Add a successor state to the successor list if
the action is legal
 # Here's a code snippet for figuring out whether
a new position hits a wall:
 x,y = position
 dx, dy = Actions.directionToVector(action)
 nextx, nexty = int(x + dx), int(y + dy)
 hitsWall = self.walls[nextx][nexty]
 if hitsWall:
 continue
 nextPos = (nextx, nexty)

 if nextPos in self.corners:
 nextVisited = cornersVisited |
frozenset([nextPos])
 else:
 nextVisited = cornersVisited
 successors.append(((nextPos, nextVisited),
action, 1))

 """ YOUR CODE HERE """
 self._expanded += 1 # DO NOT CHANGE
 return successors

def getCostOfActions(self, actions):
 """
 Returns the cost of a particular sequence of
actions. If those actions
 include an illegal move, return 999999. This is
implemented for you.

```

```
"""
 if actions == None: return 999999
 x,y= self.startingPosition
 for action in actions:
 dx, dy = Actions.directionToVector(action)
 x, y = int(x + dx), int(y + dy)
 if self.walls[x][y]: return 999999
 return len(actions)
```

```
def cornersHeuristic(state: Any, problem:
CornersProblem):
```

```
"""
```

A heuristic for the CornersProblem that you defined.

state: The current search state  
(a data structure you chose in your search problem)

problem: The CornersProblem instance for this layout.

This function should always return a number that is a lower bound on the shortest path from the state to a goal of the problem; i.e. it should be admissible (as well as consistent).

```
"""
```

```
corners = problem.corners # These are the corner coordinates
```

```
walls = problem.walls # These are the walls of the maze, as a Grid (game.py)
```

```
"*** YOUR CODE HERE ***"
```

```
position, visited = state
remaining = [c for c in corners if c not in visited]
if not remaining:
 return 0
```

```
Manhattan distance helper (shorter alias)
mdist = util.manhattanDistance
```

```

Part 1: distance from current position to the
nearest remaining corner
 to_nearest = min(mdist(position, c) for c in
remaining)

Part 2: MST over the remaining corners using
Manhattan edge weights.
Since at most 4 corners remain, a tiny Prim's
algorithm is fine.
if len(remaining) == 1:
 mst_cost = 0
else:
 unreached = set(remaining)
 # Start Prim's from an arbitrary node
 current = unreached.pop()
 reached = {current}
 mst_cost = 0

 while unreached:
 # edge with minimal cost between reached and
unreached
 # (Prim's algorithm step)
 next_node = min(
 unreached,
 key=lambda v: min(mdist(u, v) for u in
reached)
)
 add_cost = min(mdist(u, next_node) for u in
reached)
 mst_cost += add_cost
 reached.add(next_node)
 unreached.remove(next_node)

 return to_nearest + mst_cost

def aStarSearch(problem: SearchProblem,
heuristic=nullHeuristic):
 """Search the node that has the lowest combined
cost and heuristic first."""
 *** YOUR CODE HERE ***
 start = problem.getStartState()
 fringe = util.PriorityQueue()

```

```

bestCost = {start: 0}
Heuristic h(n) estimates the cost from current node
to the goal
fringe.push((start, [], 0), heuristic(start, problem))
while not fringe.isEmpty():
 state, path, cost = fringe.pop()

 if state in bestCost and cost > bestCost[state]:
 continue

 bestCost[state] = cost

 if problem.isGoalState(state):
 return path

 for (nextState, action, stepCost) in
problem.getSuccessors(state):
 newCost = cost + stepCost
 nextPath = path + [action]
 nextHeuristic = newCost + heuristic(nextState,
problem) # Get next estimated cost

 if nextState not in bestCost or newCost <
bestCost[nextState]:
 fringe.push((nextState, nextPath, newCost),
nextHeuristic)
return []

```

asherb@AsherRazer:~/Programs/CS6364/Project01\_Se  
arch\$ python autograder.py -q q6  
Note: due to dependencies, the following tests will be  
run: q4 q6  
Starting on 9-17 at 0:11:39

#### Question q4

---

```

*** PASS: test_cases/q4/astar_0.test
*** solution: ['Right', 'Down', 'Down']
*** expanded_states: ['A', 'B', 'D', 'C', 'G']
*** PASS: test_cases/q4/astar_1_graph_heuristic.test
*** solution: ['0', '0', '2']
*** expanded_states: ['S', 'A', 'D', 'C']

```

```
*** PASS: test_cases/q4/astar_2_manhattan.test
*** pacman layout: mediumMaze
*** solution length: 68
*** nodes expanded: 223
*** PASS: test_cases/q4/astar_3_goalAtDequeue.test
*** solution: ['1:A->B', '0:B->C', '0:C->G']
*** expanded_states: ['A', 'B', 'C']
*** PASS: test_cases/q4/graph_backtrack.test
*** solution: ['1:A->C', '0:C->G']
*** expanded_states: ['A', 'B', 'C', 'D']
*** PASS: test_cases/q4/graph_manypaths.test
*** solution: ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
*** expanded_states: ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']
```

### ### Question q4: 3/3 ###

### Question q6

\*\*\* PASS: heuristic value less than true cost at start state

\*\*\* PASS: heuristic value less than true cost at start state

\*\*\* PASS: heuristic value less than true cost at start state

```
path: ['North', 'East', 'East', 'East', 'East', 'North', 'North',
'West', 'West', 'West', 'West', 'North', 'North', 'North',
'North', 'North', 'North', 'North', 'North', 'West', 'West',
'West', 'West', 'South', 'South', 'East', 'East', 'East', 'East',
'South', 'South', 'South', 'South', 'South', 'South', 'South', 'West',
'West', 'South', 'South', 'South', 'West', 'West', 'East',
'East', 'North', 'North', 'North', 'East', 'East', 'East', 'East',
'East', 'East', 'East', 'East', 'South', 'South', 'East', 'East',
'East', 'East', 'East', 'North', 'North', 'East', 'East', 'North',
'North', 'East', 'East', 'North', 'North', 'East', 'East', 'East',
'East', 'South', 'South', 'South', 'South', 'South', 'South', 'East',
'North', 'North', 'East', 'East', 'South', 'South', 'South', 'South',
'South', 'South', 'North', 'North', 'North', 'North', 'North', 'North',
'North', 'North', 'West', 'West', 'North', 'North', 'North', 'East',
'East', 'North', 'North']
```

```
path length: 106
*** FAIL: Heuristic resulted in expansion of 264810
nodes
```

### Question q6: 0/3 ###

Finished at 0:11:42

Provisional grades

=====

Question q4: 3/3

Question q6: 0/3

-----  
Total: 3/6

Your grades are NOT yet registered. To register your grades, make sure to follow your instructor's guidelines to receive credit on your project.

Nice work—your A\* core looks fine. The blow-up in **q6** is almost surely the heuristic being too loose for that maze. Even `nearest + MST` with **Manhattan** edges can still expand tons of nodes in corner mazes because hallways/walls make Manhattan badly underestimate.

Fix: switch the heuristic to use **true maze distances** (shortest path lengths through walls) with **memoization**, and keep the same structure (nearest leg + MST over remaining corners). This stays **admissible & consistent** and dramatically cuts expansions.

### 1) Small tweak to `CornersProblem.__init__`

Store the starting game state so the heuristic can call `mazeDistance`:

python

```
def __init__(self, startingGameState: pacman.GameState):
 self.walls = startingGameState.getWalls()
 self.startingPosition = startingGameState.getPacmanPosition()
 self.startingGameState = startingGameState # ← add this
```

```

top, right = self.walls.height-2, self.walls.width-2
self.corners = ((1,1), (1,top), (right, 1), (right, top))
for corner in self.corners:
 if not startingGameState.hasFood(*corner):
 print('Warning: no food in corner ' + str(corner))
self._expanded = 0

```

## 2) Drop-in cornersHeuristic (maze-distance + MST + cache)

Put this into the same file (below your imports). It uses the project's provided `mazeDistance` utility (already in `searchAgents.py`) and caches distances so it's fast.

python

```

---- Heuristic helpers (top-level in searchAgents.py) ----
_distance_cache = {} # key: (p1, p2), value: maze distance (int)

def _cached_maze_dist(p1, p2, gameState):
 """Symmetric cached maze distance between two (x,y) points."""
 if p1 > p2:
 p1, p2 = p2, p1
 key = (p1, p2)
 d = _distance_cache.get(key)
 if d is not None:
 return d
 # mazeDistance is provided by the project; it runs a BFS through walls
 d = mazeDistance(p1, p2, gameState)
 _distance_cache[key] = d
 return d

def _mst_cost(points, gameState):
 """Prim's MST over 'points' with edge weight = maze distance (cached)."""
 if len(points) <= 1:
 return 0
 unreached = set(points)
 current = unreached.pop()
 reached = {current}
 total = 0
 while unreached:
 # pick cheapest edge from reached to an unreached point
 next_node, best = None, None
 for r in reached:
 for v in unreached:

```

```

w = _cached_maze_dist(r, v, gameState)
if (best is None) or (w < best):
 best = w
 next_node = v
 total += best
 reached.add(next_node)
 unreached.remove(next_node)
return total

```

Now replace your `cornersHeuristic` with:

python

```

def cornersHeuristic(state, problem: CornersProblem):
 """
 Strong, consistent heuristic for CornersProblem:
 h = min_{c in R} mazeDist(pos, c) + MST(R) with maze distances,
 where R = remaining (unvisited) corners.
 """

 position, visited = state
 remaining = [c for c in problem.corners if c not in visited]
 if not remaining:
 return 0

 # Part 1: cost to get from current position to the nearest remaining corner
 to_nearest = min(_cached_maze_dist(position, c, problem.startingGameState)
 for c in remaining)

 # Part 2: MST over remaining corners using real maze distances
 mst_cost = _mst_cost(remaining, problem.startingGameState)

 return to_nearest + mst_cost

```

### 3) Tiny cleanups (optional, but nice)

- In `getSuccessors`, you can write `cornersVisited | frozenset({nextPos})` instead of `frozenset([nextPos])`.
- Make sure `from game import Directions, Actions` is present at the top of the file.

### 4) Re-run

bash

```
python pacman.py -l mediumCorners -p AStarCornersAgent
python autograder.py -q q6
```

You should now see **node expansions drop well under the grader threshold** and get full credit for q6. If anything still looks off, send me the new expansions count and I'll fine-tune further.

```
def foodHeuristic(state: Tuple[Tuple, List[List]],
problem: FoodSearchProblem):
 """
 Your heuristic for the FoodSearchProblem goes here.
 """
```

This heuristic must be consistent to ensure correctness. First, try to come up with an admissible heuristic; almost all admissible heuristics will be consistent as well.

If using A\* ever finds a solution that is worse uniform cost search finds, your heuristic is *\*not\** consistent, and probably not admissible! On the other hand, inadmissible or inconsistent heuristics may find optimal solutions, so be careful.

The state is a tuple ( pacmanPosition, foodGrid ) where foodGrid is a Grid (see game.py) of either True or False. You can call foodGrid.asList() to get a list of food coordinates instead.

If you want access to info like walls, capsules, etc., you can query the problem. For example, problem.walls gives you a Grid of where the walls are.

If you want to *\*store\** information to be reused in other calls to the

heuristic, there is a dictionary called problem.heuristicInfo that you can use. For example, if you only want to count the walls once and store that value, try: problem.heuristicInfo['wallCount'] = problem.walls.count()

Subsequent calls to this heuristic can access problem.heuristicInfo['wallCount']

```

 """
 position, foodGrid = state
 """ YOUR CODE HERE """
 foodList = foodGrid.asList()
 if len(foodList) == 0:
 return 0
 distances = []
 for food in foodList:
 d = util.manhattanDistance(position, food)
 distances.append(d)
 return max(distances)

def aStarSearch(problem: SearchProblem,
heuristic=nullHeuristic):
 """Search the node that has the lowest combined cost and heuristic first."""
 """ YOUR CODE HERE """
 start = problem.getStartState()
 fringe = util.PriorityQueue()
 bestCost = {start: 0}
 # Heuristic h(n) estimates the cost from current node to the goal
 fringe.push((start, [], 0), heuristic(start, problem))
 while not fringe.isEmpty():
 state, path, cost = fringe.pop()

 if state in bestCost and cost > bestCost[state]:
 continue

 if problem.isGoalState(state):
 return path

 for (nextState, action, stepCost) in
problem.getSuccessors(state):

```

```

 newCost = cost + stepCost
 nextPath = path + [action]
 nextHeuristic = newCost + heuristic(nextState,
problem) # Get next estimated cost

 if nextState not in bestCost or newCost <
bestCost[nextState]:
 bestCost[nextState] = newCost
 fringe.push((nextState, nextPath, newCost),
nextHeuristic)
 return []

```

asherb@AsherRazer:~/Programs/CS6364/Project01\_Search\$ python autograder.py -q q7  
Note: due to dependencies, the following tests will be run: q4 q7  
Starting on 9-17 at 12:07:03

#### Question q4

---

```

*** PASS: test_cases/q4/astar_0.test
*** solution: ['Right', 'Down', 'Down']
*** expanded_states: ['A', 'B', 'D', 'C', 'G']
*** PASS: test_cases/q4/astar_1_graph_heuristic.test
*** solution: ['0', '0', '2']
*** expanded_states: ['S', 'A', 'D', 'C']
*** PASS: test_cases/q4/astar_2_manhattan.test
*** pacman layout: mediumMaze
*** solution length: 68
*** nodes expanded: 221
*** PASS: test_cases/q4/astar_3_goalAtDequeue.test
*** solution: ['1:A->B', '0:B->C', '0:C->G']
*** expanded_states: ['A', 'B', 'C']
*** PASS: test_cases/q4/graph_backtrack.test
*** solution: ['1:A->C', '0:C->G']
*** expanded_states: ['A', 'B', 'C', 'D']
*** PASS: test_cases/q4/graph_manypaths.test
*** solution: ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
*** expanded_states: ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']

```

### Question q4: 3/3 ###

Question q7

=====

```
*** PASS: test_cases/q7/food_heuristic_1.test
*** PASS: test_cases/q7/food_heuristic_10.test
*** PASS: test_cases/q7/food_heuristic_11.test
*** PASS: test_cases/q7/food_heuristic_12.test
*** PASS: test_cases/q7/food_heuristic_13.test
*** PASS: test_cases/q7/food_heuristic_14.test
*** PASS: test_cases/q7/food_heuristic_15.test
*** PASS: test_cases/q7/food_heuristic_16.test
*** PASS: test_cases/q7/food_heuristic_17.test
*** PASS: test_cases/q7/food_heuristic_2.test
*** PASS: test_cases/q7/food_heuristic_3.test
*** PASS: test_cases/q7/food_heuristic_4.test
*** PASS: test_cases/q7/food_heuristic_5.test
*** PASS: test_cases/q7/food_heuristic_6.test
*** PASS: test_cases/q7/food_heuristic_7.test
*** PASS: test_cases/q7/food_heuristic_8.test
*** PASS: test_cases/q7/food_heuristic_9.test
*** FAIL: test_cases/q7/food_heuristic_grade_tricky.test
*** expanded nodes: 9551
*** thresholds: [15000, 12000, 9000, 7000]
```

### Question q7: 3/4 ###

Finished at 12:07:03

Provisional grades

=====

Question q4: 3/3

Question q7: 3/4

-----

Total: 6/7

Your grades are NOT yet registered. To register your grades, make sure to follow your instructor's guidelines to receive credit on your project.

It's not passing one testcase on the foodSearch, is my A\* wrong?

Your A\* looks fine. The failure on `food_heuristic_grade_tricky` is because the heuristic (max Manhattan to a food) is too weak—it causes extra expansions. Bump it to use **maze distances** (true shortest paths through walls) with caching. That alone usually drops expansions under 9000.

Here's a **drop-in replacement** that's still admissible/consistent and fast enough:

python

```
Put near the top of searchAgents.py (module-level)
_food_dist_cache = {} # key: (min(p,q), max(p,q)) -> int maze distance

def _cached_maze_distance(p, q, gameState):
 # canonicalize key (unordered)
 key = (p, q) if p <= q else (q, p)
 if key in _food_dist_cache:
 return _food_dist_cache[key]
 # mazeDistance is provided by the project; BFS through walls
 d = mazeDistance(p, q, gameState)
 _food_dist_cache[key] = d
 return d
```

Now your heuristic:

python

```
def foodHeuristic(state, problem: FoodSearchProblem):
 """
 Strong, consistent heuristic:
 h = max over foods F of mazeDistance(position, f).
 (Optionally take max with an approximate 'food diameter' for a bit more bite.)
 """

 position, foodGrid = state
 foods = foodGrid.asList()
 if not foods:
 return 0

 # 1) Lower bound: you must at least reach the farthest remaining food
```

```

farthest = 0
for f in foods:
 farthest = max(farthest, _cached_maze_distance(position, f,
problem.startingGameState))

2) Optional extra bite (still admissible): approximate "diameter" among foods
pick f1 as farthest from current position, then f2 farthest from f1
(good bound, cheap; comment out if you want simpler)
if len(foods) >= 2:
 # find f1 (already computed distances above)
 # re-run to capture which f gave the farthest value
 f1, best = None, -1
 for f in foods:
 d = _cached_maze_distance(position, f, problem.startingGameState)
 if d > best:
 best, f1 = d, f
 # find farthest from f1
 diam = 0
 for f in foods:
 diam = max(diam, _cached_maze_distance(f1, f, problem.startingGameState))
 # the path that eats everything must be at least the larger of these
 return max(farthest, diam)

return farthest

```

Why this works:

- `mazeDistance` respects walls, so it's a **tighter** lower bound than Manhattan.
- Taking the **max distance to any food** is admissible: you must at least reach that food.
- Adding the **approximate diameter** (farthest pair via farthest-from-farthest) is still a lower bound, so it stays admissible/consistent, and usually trims expansions further.

Run again:

bash

```

python pacman.py -l trickySearch -p AStarFoodSearchAgent
python autograder.py -q q7

```

If it's still close, you can replace the "approximate diameter" with the **exact** pairwise diameter (max over all pairs of foods) using the same cache; that's admissible and

often even fewer nodes, but slightly heavier.

No I want to No I think we should still use manhattanDistance, but there's somewhere we can improve the heuristic

Totally workable with just **Manhattan**—you just need a tighter lower bound than “max distance to one food.”

Use this classic TSP-style bound:

**$h = (\text{distance from Pacman to the nearest remaining food}) + (\text{MST cost over all remaining foods}),$**

all distances measured by **Manhattan**.

Why it helps:

- You must at least reach *some* food first → nearest-leg.
- To eat the rest, you must at least connect them all → MST is a lower bound on that tour.
- Manhattan is a metric (triangle inequality), so this is **admissible and (in this setting) consistent**.

Drop-in replacement:

python

```
def foodHeuristic(state, problem):
```

```
 """
```

Stronger Manhattan-only heuristic for FoodSearchProblem:

$h = \min_{f \in F} \text{manhattan}(pos, f) + \text{MST}(F)$  with Manhattan edges,  
where  $F$  is the set of remaining foods.

```
 """
```

```
 position, foodGrid = state
```

```
 foods = foodGrid.asList()
```

```
 if not foods:
```

```
 return 0
```

```
 if len(foods) == 1:
```

```
 fx, fy = foods[0]
```

```
 px, py = position
```

```

 return abs(px - fx) + abs(py - fy)

Manhattan helper
def md(a, b):
 return abs(a[0] - b[0]) + abs(a[1] - b[1])

Part 1: distance from current position to the nearest food
to_nearest = min(md(position, f) for f in foods)

Part 2: Prim's MST over the foods with Manhattan weights
unreached = set(foods)
current = unreached.pop()
reached = {current}
mst_cost = 0

while unreached:
 # pick the cheapest edge from any reached node to any unreached node
 best_w = None
 best_v = None
 for r in reached:
 # find nearest unreached to r
 # (tiny sets here; O(n^2) Prim's is fine)
 for v in unreached:
 w = md(r, v)
 if best_w is None or w < best_w:
 best_w, best_v = w, v
 mst_cost += best_w
 reached.add(best_v)
 unreached.remove(best_v)

return to_nearest + mst_cost

```

Optional micro-optimizations (if you want even fewer expansions/time):

- Cache MST by the **food set**: `key = frozenset(foods)`, store `mst_cost` in `problem.heuristicInfo`.
- Also cache nearest-leg if you like (not usually needed).

Run again on `trickySearch`. This Manhattan-only heuristic is much tighter and should bring expansions under the grader threshold while keeping your A\* exactly as-is.

Can you explain Q7 and what we just did. I know we used Prim's Algorithm on a minimum spanning tree, can you explain why did we use MST and how this works?

Great question — let's slow down and really unpack **Q7 (foodHeuristic for the FoodSearchProblem)** and why we pulled in **Prim's MST** to improve it.

---

## 1. What is Q7 asking?

The **FoodSearchProblem** = Pacman has to eat all the food dots on the grid.

- **State** = (Pacman's position, foodGrid)
- Goal = no food left
- Successors = moving Pacman one square at cost 1

We're asked to design a **heuristic** for A\* search. Requirements:

- **Admissible**: never overestimates the true cost to finish.
- **Consistent**: satisfies the triangle inequality so A\* works properly.

The better (tighter) the heuristic, the fewer nodes A\* will expand.

---

## 2. What we started with

The simplest admissible heuristic is:

python

```
h = max_{food f} ManhattanDistance(position, f)
```

Why admissible?

- You must at least get to the farthest food, so that's a lower bound.  
Problem: It ignores all the **other food** — it only accounts for one leg of the journey. That's why it blew up expansions on `trickySearch`.
- 

## 3. Why use an MST?

Think of eating all food as a **Traveling Salesman Problem (TSP)**:

- Start at Pacman's current position
- Visit every food dot at least once
- Minimize total path length

The **true cost** is essentially “the shortest tour covering all foods.”

That's hard to compute exactly (NP-hard).

But what's a guaranteed **lower bound** on that cost?

👉 The cost of a **Minimum Spanning Tree (MST)** connecting all the food dots.

Why?

- An MST connects all foods with the smallest possible “wiring cost.”
- Any actual Pacman tour must at least traverse those connections somehow.
- So  $\text{MST} \leq \text{optimal tour cost}$ .

Then we add the cost from Pacman's current position to *enter* the MST (nearest food).

So heuristic:

ini

$h = \text{distance}(\text{position}, \text{nearest food}) + \text{MST}(\text{all remaining foods})$

This captures **both**:

- The “first hop” from Pacman to a food.
- The “must-connect cost” of all foods to each other.

And because Manhattan distance obeys triangle inequality, this remains **admissible** and **consistent**.

## 4. Why Prim's algorithm?

We needed to actually compute the MST cost quickly.

- At most there are maybe ~10–20 foods in `trickySearch`.
- That's tiny, so  $O(n^2)$  Prim's (grow MST one node at a time) is totally fine.
- Each edge weight = Manhattan distance between foods.
- Summing those edges gives the MST cost.

## 5. Why does this help?

- **max distance** heuristic only “sees” one dot.
- **MST heuristic** “sees” the *whole structure* of the food distribution.
  - It knows you must traverse multiple edges.
  - It grows tighter the more scattered the food is.

So A\* avoids exploring silly detours because it knows “even in the best case, I still have to spend at least MST cost.”

That drops expansions way down under the autograder thresholds.

---

 In short:

We used MST because it's a **mathematically sound lower bound** on visiting all the foods, cheap to compute, and makes the heuristic much sharper without breaking admissibility/consistency.

---

Would you like me to also illustrate this with a **mini example grid** (say Pacman + 3 foods) and show how the MST bound is tighter than just farthest distance?