

COMP3270 Assignment 1: Search

Release date: 14 September 2026

Submission deadline: 2 October 2026, 23:59 Hong Kong time

Assignment weight: 10% of the course

Grading TA: Jingqi

Marks and Assessment

This is an **individual assignment**, marked out of **100** and worth **10%** of the course. Q0 is an ungraded warm-up.

Question	Code	Written answers	Total
Q0: Count the moves	0	0	0
Q1: Maze positions and successors	10	0	10
Q2: BFS-GSA	12	3	15
Q3: DFS-GSA	2	3	5
Q4: UCS-GSA	15	5	20
Q5: Collect all food	25	5	30
Q6: A*-GSA	15	5	20
Total	79	21	100

Code marks assess correctness and compliance with the stated algorithm and function contracts, including edge cases. Written-answer marks assess accurate observations and explanations that connect those observations to the algorithms. Public test counts are not mark allocations; passing the supplied cases alone does not establish full correctness. Errors in an earlier question do not automatically remove credit for otherwise correct later work: later implementations will be assessed independently where possible, and written answers are marked separately.

Academic Integrity: No AI

Do not use generative-AI tools for any part of this assignment. This includes interpreting questions, planning solutions, generating or completing code, debugging, and drafting or revising written answers. The rule applies to chatbots, coding agents and AI features built into editors. Disable AI assistance while working on A1.

Complete the work independently. Do not share code or written answers, copy another person's solutions, or offer your work for copying. You may use the supplied course materials and ordinary, non-AI documentation; acknowledge sources you use. Standard non-AI editor features are allowed.

Ask the teaching team or use Ed when you need clarification. Do not post your solution code publicly. You must understand your submission and may be asked to explain your code and reasoning.

Getting Started

You need Python **3.10 or later**, a plain-text code editor, and a terminal. The assignment uses only Python's standard library; you do not need to install any additional Python packages.

1. Extract the Files

Download `a_01.zip` and extract it. Open the extracted `a_01` folder, then its `student` folder. You should see `q0.py`–`q6.py`, `play.py`, `maps/`, `support/`, `test_cases/` and five `qN_answers.md` templates. Work in this extracted folder, not inside the ZIP preview. Keep the folders together because the question files import supplied helpers.

2. Open a Terminal in the Student Folder

- **Windows:** Open `student` in File Explorer. Click its address bar, type `powershell`, and press Enter. A PowerShell window opens in that folder.
- **macOS:** Open Terminal. Type `cd`, including the space, drag the extracted `student` folder from Finder into the Terminal window, and press Enter.

Enter the appropriate commands below, one at a time:

Check	Windows PowerShell	macOS Terminal
Python version	<code>python --version</code>	<code>python3 --version</code>
Correct folder	<code>dir q0.py</code>	<code>ls q0.py</code>

The first command should report Python 3.10 or later. The second should find `q0.py`. If it cannot find that file, return to the folder-opening step. If Python is unavailable, install it from python.org and reopen the terminal, or ask the teaching team for setup help.

The remaining examples use `python3`. On Windows, use `python` instead. Type only the command; do not type the `$` shown in example transcripts.

3. Edit and Save q0.py

Open your code editor, choose **File > Open**, and select `q0.py` from the extracted `student` folder. Find `def count_moves(path):` and read the comments below it. Replace the `raise NotImplementedError(...)` line inside that function with your own implementation, following Q0 below. Keep your function body indented by four spaces. Leave the supplied runner at the bottom of the file unchanged.

Save the file with **Ctrl+S on Windows** or **Command+S on macOS**. Keep the name `q0.py`, not `q0.py.txt`. The terminal runs the saved file on disk; changes still unsaved in the editor are not used.

4. Run Your Saved Code

In the terminal you opened in step 2, enter:

Windows	macOS
<code>python q0.py</code>	<code>python3 q0.py</code>

This runs your function on the small `corridor` example. Before you implement it, `NotImplementedError: Implement count_moves` is expected. After you implement it, inspect the result and compare it with the Q0 example below. Running the file does not automatically check every supplied test case.

5. Check It with the Tests

Run the same command with `--test`:

Windows	macOS
<code>python q0.py --test</code>	<code>python3 q0.py --test</code>

This compares your results with the supplied expected answers. For Q0, aim for **2/2 public checks passed**. If a check fails, read the first reported error or Expected/Actual difference, return to the relevant function, edit it, save it, and run the command again. A Python error identifies a file and line number; start by checking that location. Once Q0 works, continue with Q1.

Displaying Mazes and Replaying Searches

All commands run in the terminal. `--show-maze` displays the Q1 maze. `--replay` shows expansions and the returned path in Q2–Q4, and moves along the returned path in Q5–Q6. These displays help you inspect your result; they do not replace the tests.

Short options do the same thing: `-t` = `--test`, `-r` = `--replay`, `-m` = `--show-maze`, `-l` = `--list`, and `-h` = `--help`. For manual play, `-g` = `--goal`. Each program's help lists the options it supports.

A **maze** is the environment Pacman moves through. A **map file** in `maps/` defines its layout. A **test case** in `test_cases/qN/` names a map file and may provide extra input, such as Q0's path. Expected answers are stored separately in `_expected.json` files. Keep all these supplied files unchanged. You do not need to read or edit `support/`. Use `--help` for command options and `--list` (or `-l`) for test-case paths. Successful examples below require completed functions; excerpts omit other output. Public tests cover particular cases, not every possible implementation error: follow the stated algorithm and function contracts as well.

Completing Written Answers

Open the supplied `q2_answers.md` – `q6_answers.md` templates in your code editor. Fill in the blank responses and save each file with its existing name in `student/`. Write plain text; no special Markdown software is needed. The word limits apply to your responses, not the supplied prompts.

Files to Submit

Submit only the eleven provided answer files you complete:

- `q1.py`, `q2.py`, `q3.py`, `q4.py`, `q5.py`, `q6.py`
- `q2_answers.md`, `q3_answers.md`, `q4_answers.md`, `q5_answers.md`, `q6_answers.md`

Do not submit `maps/`, `support/`, `test_cases/`, autograder files, `play.py`, the assignment PDF, caches or other supplied files. Q0 is an ungraded warm-up and is not submitted. Keep the required filenames unchanged. Submit the individual files to **Assignment 1 on the COMP3270 Moodle course page**. Upload the eleven files listed above individually, without a ZIP archive. Check that all eleven files appear in your submission before the deadline.

Try Pacman Before Coding

Run the supplied game; it works before you implement any question:

```
python3 play.py
python3 play.py maps/corridor.json
```

The default `playground` has six food dots, junctions and loops; `corridor` is a tiny practice map. Use **arrows or W/A/S/D** to move, **R** to restart and **Q** to quit. Blocked moves do not count.

All maze displays use the same compact, monochrome layout. Each grid character is one cell: `●` is Pacman, `•` is food and `■` is a wall. The map filename appears above it. Row numbers are on the left; column digits are stacked above their cells, reading downwards when a column number has two or more digits. The coordinate labels sit outside the map.

Use `--list` (or `-l`) with any script to list accepted input paths. For a different map and goal, try:

```
python3 play.py maps/branching.json --goal any
```

Manual play collects **all** food by default, as in Q5/Q6. Use `--goal any` to stop at the first food, as in Q1–Q4. Weighted maps show entry costs and your accumulated cost. Maps named `unreachable` are deliberate no-solution examples; use Q to quit when you have explored what is reachable.

All scripts accept a short name, a JSON filename or a path relative to `student/`. For `q2.py`, `corridor`, `corridor.json` and `test_cases/q2/corridor.json` are equivalent; for `play.py`, use `corridor`, `corridor.json` or `maps/corridor.json`. Each question accepts test cases from its own `test_cases/qN/` folder. Forward slashes work on Windows and macOS.

Q0: Count the Moves (0 Marks)

1. Follow Pacman's Path

Run the small corridor (initial output excerpt, available before implementation):

```
$ python3 q0.py
Running Q0: corridor
Map: maps/corridor.json
012
0 ● .
Path: [(0, 0), (0, 1), (0, 2)]
```

Before implementation, this input is followed by `NotImplementedError: Implement count_moves`. That is expected: replace the placeholder to produce your result.

Each character is a cell: **●** is the start and **.** is food. Columns are labelled above; rows on the left. A state is Pacman's position, `(row, column)`. Follow the given path to the food.

The input is in `test_cases/q0/corridor.json`; it references `maps/corridor.json` and supplies the path. JSON writes coordinates as `[0, 1]`; the supplied loader converts them to Python tuples such as `(0, 1)`.

2. Implement One Function

In `q0.py`, replace the placeholder in `count_moves(path)` with your code. Return the number of moves in the nonempty path as an integer. The starting position is included. Work out the rule yourself.

3. Run Your Code

Save the file, then run:

```
python3 q0.py
python3 q0.py one_move
```

The first command uses the corridor; in `one_move`, the food is one move away. Look at the printed input and your result. You may add `print()` statements to debug.

4. Check Your Work

```
$ python3 q0.py --test
corridor: PASSED
one_move: PASSED
Q0: 2/2 public checks passed
All Q0 tests passed.
```

There is nothing to submit for Q0.

Q1: Represent the Maze

Implement the legal positions and moves that later searches will use. Moves go north, east, south or west and cost 1.

1. Inspect a Small Maze

Start with three cells. Add `--show-maze` to display the maze before calling your functions (maze excerpt):

```
$ python3 q1.py corridor --show-maze
Map: maps/corridor.json
012
0 ● .
```

The positions are `(0, 0)`, `(0, 1)` and `(0, 2)`: start, open cell, food. A later

`NotImplementedError` is expected while the functions are unfinished.

`q1.py` imports `WALL` (`■`), `PACMAN` (`●`), `FOOD` (`·`) and `EMPTY` (one space). Use these names in your code. Inputs use UTF-8. Every maze contains exactly one `PACMAN` and one or more `FOOD` cells. Pacman starts in a separate cell. The task is to reach **any one food cell**, not collect all food.

2. Q1(a): Check a Position

The upper-left cell is `(0, 0)`. Rows increase downwards; columns increase to the right.

Implement `is_valid_position(problem, state)` first. Both parameters are inputs; do not modify them. Return `True` for an in-bounds `EMPTY`, `PACMAN` or `FOOD` cell, including border cells. Return `False` for a wall or an out-of-bounds position.

`problem` is a supplied dictionary. Its `problem['grid']` value is a nonempty list of equal-length row strings. `problem['grid'][r][c]` is one cell. The `state` argument is a tuple of two integers, which may be negative or outside the maze. Check bounds before indexing: Python's negative indices access cells from the end of a sequence.

Check this part before implementing successors:

```
$ python3 q1.py corridor --test
corridor valid positions: PASSED
corridor successors: FAILED
```

Your first function has passed (1/2). The following error names the unfinished `successors` function and its line. This failure is expected because part (b) is unfinished. Run `python3 q1.py --test` next: aim for all seven position checks passing, or 7/14 overall.

3. Q1(b): Generate Legal Moves

At the starting position `(0, 0)` in the corridor, predict the legal moves. Then implement `successors(problem, state)` using your position-checking function. Here, `state` is a valid `(row, column)` tuple. Both parameters are inputs; do not modify them.

Return a **list of tuples** of the form `(action, next_state, step_cost)`. Include only legal moves, in **North, East, South, West** order. Use these exact action names. Each `next_state` is a coordinate tuple and each step costs 1.

Return an empty list if there are no legal moves. Moving onto the food is legal; your function describes moves and does not decide when search should stop.

4. Run, Check and Try a Larger Maze

Run the corridor first. After completing both functions, you should see:

```
$ python3 q1.py corridor
Running Q1: corridor
Start state: (0, 0)
Valid position: True
Successors: [('East', (0, 1), 1)]
```

Check the small test case, then all seven test cases:

```
python3 q1.py corridor --test
python3 q1.py --test
```

Your checkpoints are **2/2** for the corridor and **14/14** overall. The full suite checks walls, boundaries, negative coordinates, move order and cells with no legal moves. Q1 can pass completely before you begin BFS.

Now inspect the larger maze with branches and walls:

```
python3 q1.py playground --show-maze
```

For Q1, submit `q1.py`.

Q2: Find a Route Using BFS-GSA

Complete Q1 first: Q2 uses your `successors` function to find legal moves. If the Q1 checks fail, the tests will ask you to fix Q1 before checking BFS.

Complete `bfs_gsa` in `q2.py`. Keep its supplied optional parameters and default setup; these allow Q5 to reuse your search with a different problem definition. For search, use only `problem['start_state']`, `is_goal(problem, state)` and `successors(problem, state)`. The supplied `is_goal` returns `True` exactly when the state is any food cell, otherwise `False`. Treat each coordinate tuple as a whole state inside BFS; do not unpack it or access the grid.

Use the lecture's representation: a `deque` frontier of `(state, path)` pairs, an explored `set`, and an `expanded` list. For example, `((1, 1), [(1, 1)])` pairs a state with the path reaching it. The starter imports `deque`; `append` adds at the end and `popleft` removes the oldest entry.

1. Implement the Search

Use the lecture's **breadth-first graph-search algorithm (BFS-GSA)**. The explored set prevents repeated expansion. Follow this exact loop:

1. Begin with the initial state and its one-state path in the frontier. FIFO means first in, first out: remove the oldest entry.
2. Test for the goal when a state is removed from the frontier, before expanding it. Do not stop merely because a successor is a goal.
3. If the state is already in the explored set, skip it. Otherwise add it to the explored set and record its expansion.
4. For each successor, append its `(state, path)` pair in the supplied order, extending the current path by that successor. Include successors leading to explored states. Do not sort or filter successors; repeated states are skipped when removed.
5. If the frontier becomes empty without finding a goal, report that no route exists.

An **expanded state** is a non-goal state whose successors are examined. A dead end is still expanded. The removed goal and skipped duplicate entries are not expanded. FIFO order preserves the first-arriving path when equal-length routes meet.

Return a dictionary with exactly two keys:

- `'expanded'`: a list of coordinate tuples in expansion order, without duplicates.
- `'path'`: a list of coordinate tuples from start to goal, inclusive; `None` if no goal is reachable.

2. Run the Small Example

Save `q2.py`, then run the corridor (result excerpt):

```
$ python3 q2.py corridor
Solution found.
Moves: 2
Expanded states: 2
Path: [(0, 0), (0, 1), (0, 2)]
```

3. Check the Small Case, Then All Cases

```
python3 q2.py corridor --test
python3 q2.py multiple_food --test
python3 q2.py --test
```

Aim for 1/1 on each small case, then 7/7 overall. `multiple_food` checks that you test for a goal after removing a path, not when generating it. The grader checks values, not print formatting.

4. Follow BFS Through the Larger Maze

```
python3 q2.py playground --replay
```

Press Enter for the next expansion or `q` to quit. Two aligned grids separate the **unchanged maze** (start, food, walls and costs) from **search progress**:  marks earlier expansions and  the state just expanded. They appear side by side when space permits, or stacked in a narrow terminal. These are search steps, not Pacman's movements.

Below the grids, **Expanded so far** counts the expansions shown. The **Explored set** includes the just-expanded state and is listed in coordinate order, not expansion order. On success, the second grid changes to **Returned path**, with  marking its cells, including start and goal; the ordered coordinates appear below. If no path exists, the second grid retains the expansions. Unfinished or invalid searches show a plain maze and an error. These controls also apply to Q3 and Q4.

The replay shows your returned expansion order, not a live frontier. A visible route does not prove correctness: check it with `--test`. Rerun after editing.

With multiple food cells, BFS returns a route with the fewest moves to **any** food. Try `python3 q2.py multiple_food --replay`.

5. Explain What You Observed

For `branching`, report the number of expanded states and the number of moves in the returned path. In at most 100 words, explain why BFS finds a shortest route here. Name one expanded state outside your returned route and explain why BFS examined it. Complete the supplied `q2_answers.md` template.

For Q2, submit `q2.py` and `q2_answers.md`, together with your completed Q1 file `q1.py`.

Q3: Find a Route Using DFS-GSA

Implement `dfs_gsa` in `q3.py` using **DFS-GSA** (depth-first graph search). Keep its supplied optional parameters and default setup, as in Q2. Q3 uses Q1's `successors`; Q2 is needed only for comparison.

DFS uses a **LIFO stack** (last in, first out), following a branch before returning to another pending branch.

1. Implement the Search

Use Q2's input, return dictionary and graph-search loop. Keep the same `(state, path)` pairs, explored set and expansion list. **The only algorithm change is `pop()` instead of `popleft()`**, so the newest frontier entry is removed first.

Keep the goal test immediately after removal and append successors in **North, East, South, West** order, including duplicates. Do not reverse the order: if East and South are legal, append East then South; DFS examines South next.

2. Run and Check

```
python3 q3.py corridor
python3 q3.py corridor --test
python3 q3.py multiple_food --test
python3 q3.py --test
```

The corridor should match Q2's two-move route. Next, `multiple_food` checks your direction order: append North, East, South, West without reversing. Aim for 1/1 on each small case, then 7/7 overall.

3. Compare the Two Replays

```
python3 q2.py branching --replay
python3 q3.py branching --replay
```

Compare their paths, then repeat with `playground`. DFS reaches the first food in its exploration order, which need not be the nearest. Q5/Q6 later collect all six foods.

4. Explain the Difference

Run BFS-GSA and DFS-GSA on `branching`. Complete the supplied `q3_answers.md` template. Report each algorithm's number of expanded states and returned path length in moves.

In at most 100 words, explain how FIFO versus LIFO causes the different exploration orders. Does DFS find a shortest route in this maze? Explain why examining fewer states would not, by itself, imply a shorter returned path.

For Q3, submit `q3.py` and `q3_answers.md`, together with your completed `q1.py`.

Q4: Find the Cheapest Route Using UCS-GSA

Implement `ucs_gsa(problem)` in `q4.py` using the lecture's **UCS-GSA** (uniform-cost graph search). Find the lowest **total path cost**, which may require more moves than BFS.

Keep your completed `q1.py` unchanged. Q4 supplies weighted `successors(problem, state)`, returning `(action, next_state, step_cost)` tuples in North, East, South, West order. It reuses Q1 but does not depend on Q2 or Q3.

1. Inspect Move Costs

Pay the cost of the **cell you enter**: **1 (normal)**, **3 (moderate)**, or **5 (expensive)**. Every unnumbered cell, including the food, costs 1. The initial path costs 0.

Inspect the input maze with this command (initial replay excerpt after implementation; a plain maze is shown while Q4 is unfinished):

```
$ python3 q4.py corridor --replay
Map: maps/weighted_corridor.json
Maze (unchanged)   Search progress
  012               012
0 ●3·               0
Entry costs: (0, 1)=3
```

Entering the middle cell costs 3; leaving it for the food costs 1. The unchanged maze preserves every cost; the search-progress grid shows expansions separately.

The case `test_cases/q4/corridor.json` references `maps/weighted_corridor.json`, which contains `"grid": ["●3·"]` and a title. The grid defines the start, food, walls and costs. Your Q1 functions still receive ordinary open cells.

2. Implement the Priority-Queue Loop

Use Q2's search interface with Q4's supplied weighted `successors`.

1. Start with a path containing only the initial state, at total cost 0.
2. Remove the path with the lowest accumulated cost. Test for the goal immediately. Do not stop when a goal is generated.
3. Skip an already-explored state. Otherwise mark it explored and record its expansion.
4. Add every successor path with the accumulated cost plus that move's `step_cost`. Keep duplicate frontier entries, as in the lecture.
5. If the frontier empties without reaching a goal, report failure.

The starter imports `heappush` (add an entry) and `heappop` (remove the smallest entry). Use `(total_cost, path)` entries. After removing an entry, the current state is `path[-1]`. Python handles ties automatically: compare whole paths at the first differing state, by row then column. At equal cost, `[(1, 1), (1, 2)]` comes before `[(1, 1), (2, 1)]`. Do not add a custom tie-break rule.

Return the Route and Its Cost

Return Q2's `'expanded'` and `'path'` entries, plus `'cost'`: the sum of move costs along the path, or `None` if no food is reachable. Return exactly these three keys.

3. Run the Small Example

```
$ python3 q4.py corridor
Running Q4: corridor
Solution found.
Moves: 2
Expanded states: 2
Total cost: 4
Path: [(0, 0), (0, 1), (0, 2)]
```

The two moves cost 3 and 1, giving a total of 4.

4. Check Your Work

```
python3 q4.py corridor --test
python3 q4.py --test
```

Aim for 1/1 on the corridor, then 8/8 overall. They check cheaper detours, ties, cycles, multiple food cells and unreachable food. UCS chooses the cheapest reachable food, which need not be the nearest in moves. On `playground`, every move costs 1. Try `python3 q4.py multiple_food --replay`.

5. Follow the Cheapest Route

```
python3 q4.py branching --replay
python3 q4.py playground --replay
```

Step through with Enter, then inspect the returned path and total cost. Q4's `branching` uses Q2's layout with different costs, so you can compare the routes.

6. Compare Moves with Cost

Complete the supplied `q4_answers.md` template. For Q4's `branching`, report UCS's number of moves and total cost. Compare its route with your Q2 BFS route for `branching`: evaluate that BFS route using **Q4's move costs**, rather than assuming every move still costs 1.

Calculate the BFS route's cost from Q4's grid: skip the starting cell, then add the cost of each cell the route enters.

In at most 100 words, explain which route is cheaper and why BFS's fewest-moves guarantee does not guarantee the cheapest route here. Explain why UCS waits until a goal path is removed from the priority queue before returning it.

For Q4, submit `q4.py`, `q4_answers.md` and your completed `q1.py`.

Q5: Eat All the Food

Previously, search stopped at any food cell. Now Pacman must eat **every food dot**. Every move costs 1. Reuse BFS-GSA to find a route with the fewest moves.

1. Decide What a State Must Remember

Imagine Pacman returns to a junction after eating a food dot. Its position is unchanged, but the task remaining has changed. In `q5_answers.md`, explain why position alone is no longer a sufficient state.

Choose your own Python representation. It must record Pacman's position and which food remains, and satisfy these rules:

- Each state must be **hashable**, so you can put it in the explored `set`, and must not change after creation.
- The same position and remaining food must compare equal and have the same hash, regardless of the order in which food was eaten.
- Different positions or remaining food must produce different states.
- Generating successors must not modify the parent state or the supplied problem.

A Python list, dictionary or mutable set is not hashable. Putting one inside a tuple does not solve this: every element of the tuple must also be hashable.

Briefly explain your representation in `q5_answers.md`. You may use standard-library types; no classes are required.

2. Implement the Problem Definition

Implement these functions in `q5.py`. All parameters are inputs.

Function	Required behavior
<code>initial_state(problem)</code>	Return a state containing the starting position and all food.
<code>pacman_position(state)</code>	Return the position as a <code>(row, column)</code> tuple.
<code>remaining_food(state)</code>	Return the remaining food positions as a tuple of coordinate tuples, sorted by row then column, without duplicates.
<code>is_goal(problem, state)</code>	Return <code>True</code> exactly when no food remains; otherwise <code>False</code> .
<code>successors(problem, state)</code>	Return a list of <code>(action, child_state, 1)</code> tuples in North, East, South, West order. Remove food at the destination from each child state.

The supplied `problem` contains `grid` as in Q1, `pacman_start` as a coordinate tuple, and `food_positions` as a tuple of all food coordinates. There is one Pacman and at least one food dot, in separate cells. Food disappears when entered and stays eaten when revisited.

Use `position_successors(problem, pacman_position(state))` to obtain legal moves from your completed Q1. It is already imported. The tests do not request successors after all food is eaten. The two display functions let the runner inspect any representation you choose; their return format does not dictate how you store information internally.

Run `python3 q5.py corridor --test`. Once the five problem functions are correct, `corridor problem definition: PASSED` confirms this checkpoint. If Q2 is unfinished, the next error names its BFS function.

3. Reuse Your Search

Q5 already imports your completed Q2 BFS and supplies this adapter:

```
from q2 import bfs_gsa as search_bfs
def bfs_gsa(problem):
    return search_bfs(problem, is_goal=is_goal, successors=successors)
```

Keep this supplied adapter and Q2's search unchanged. It passes Q5's goal and successor functions into Q2. The runner sets `problem['start_state']` using `initial_state(problem)`.

The reused BFS returns a dictionary whose `expanded` and `path` lists contain complete Q5 states, not just positions. Its `path` is `None` if any food is unreachable.

4. Build Confidence with Small Cases

Start with one food dot, then two:

```
python3 q5.py
python3 q5.py corridor --test
python3 q5.py two_food --test
```

The corridor takes **2 moves**; `two_food` takes **3**. Tests check the problem definition, then BFS. Tests compare both expansion order and the returned path using position and remaining food, independently of your internal state format. Mismatches show the first difference. A valid path can still be replayed with `--replay` even if its ordering fails a test.

Replay a route that returns through a junction:

```
python3 q5.py return_visit --replay
```

Q5 replays **moves along the returned path**, with eaten food disappearing. Press Enter for the next move or `q` to quit. The counter shows the food remaining; Q2–Q4 replays show expansions instead.

5. Solve the Larger Maze

Inspect the winding route (initial maze excerpt, available even before implementation):

```
$ python3 q5.py long_route --replay
Map: maps/long_route.json
      1
    01234567890
0  ██████████
1  █●  .  .  █
2  ██████████
3  █.  .  █
4  █ ██████████
5  █ .  .  █
6  ██████████
7  █.  █
8  ██████████
```

This maze has one shortest solution. Before testing, trace the route and predict the move count. Then run:

```
python3 q5.py long_route --test
python3 q5.py --test
```

Aim for 6/6. `playground` is the branching maze used by the manual game. The final case, `unreachable`, must report that no route collects all food.

In `q5_answers.md`, also report `long_route`'s move count and explain why BFS itself needed no change. Keep your full answer to Q5 within 200 words.

For Q5, submit `q5.py`, `q5_answers.md`, and your completed `q1.py` and `q2.py`.

Q6: Guide the Search with A*-GSA

Keep Q5's state representation and problem functions. Implement `heuristic(problem, state)` and `a_star_gsa(problem)` in `q6.py` to collect all food in the fewest moves while exploring more selectively. Q6 imports your Q5 functions. Its six test cases reference the same shared maps as Q5, with separate expected results.

1. Estimate the Remaining Distance

For position `(r, c)` and food `(fr, fc)`, the **Manhattan distance** is:

```
abs(r - fr) + abs(c - fc)
```

Use `pacman_position(state)` and `remaining_food(state)` to inspect your state. Return the **largest** Manhattan distance to any remaining food as an integer; return `0` when no food remains. Ignore walls when calculating this estimate. Do not modify the state or problem.

For example, from `(2, 2)` to food at `(2, 4)` and `(5, 2)`, the distances are 2 and 3, so the heuristic is 3.

Check this function before implementing A*:

```
python3 q6.py corridor --test
```

With Q5 complete and the heuristic correct, you should see `corridor heuristic: PASSED`, followed by `NotImplementedError: Implement a_star_gsa`. The case passes fully only after both functions are implemented.

2. Choose Which Path to Explore Next

A* orders paths by $f = g + h$: moves made so far plus estimated moves remaining. Follow the lecture's A* priority update, using the familiar graph-search explored set:

1. Start with the one-state path. Its priority is `heuristic(problem, start)` because its cost so far is zero.
2. Remove the lowest-priority entry and test for the goal immediately.
3. Skip an already-explored state. Otherwise mark it explored and record its expansion.
4. Add every successor path, including duplicates, in the supplied North, East, South, West order. Update its priority with `new_f = f + step_cost - heuristic(problem, state) + heuristic(problem, child)`.
5. If the frontier empties, return `path=None`.

The update removes the parent's estimate, adds the move cost, and adds the child's estimate.

In the corridor, the start has `f=2` and `h=2`. Moving one cell gives `step_cost=1` and `h(child)=1`. Therefore, `new_f = 2 + 1 - 2 + 1 = 2`, matching `g(child) + h(child) = 1 + 1`.

The lecture's runnable example uses tree search. Here, the explored set prevents repeated expansion. This heuristic is **consistent**: $h(\text{state}) \leq 1 + h(\text{child})$ for every legal move. Therefore, this graph search does not need to reopen explored states.

3. Handle Equal Priorities

Use heap entries `(f, path_key(path), path)`. The supplied `path_key` compares whole paths using each state's position, then sorted remaining food. This preserves the lecture's whole-path tie-breaking idea without requiring your state containers to support ordering. Keep the helper unchanged.

Return Q5's dictionary with exactly `'expanded'` and `'path'`, containing your complete states. Exclude goals and skipped duplicates from expansions, as before.

4. Run and Check

```
python3 q6.py
python3 q6.py corridor --test
python3 q6.py --test
```

The corridor takes **2 moves** and expands **2 states**. Aim for **1/1**, then **6/6**. Tests also check the heuristic with multiple food dots and with no food remaining.

Replay your solution using the same controls as Q5:

```
python3 q6.py playground --replay
```

5. Compare A* with BFS

Run both searches on the branching `playground` maze:

```
python3 q5.py playground
python3 q6.py playground
```

Both commands report the solution length and number of expanded states. In `q6_answers.md`, record those two measurements for each algorithm. Both should find a shortest solution; their routes may differ because their frontier ordering differs. Also compare `corridor`: does A* always expand fewer states?

In at most 200 words, explain:

- Would summing the Manhattan distances to all remaining food always be admissible? Justify with a small maze in your written answer. Keep the maximum-distance heuristic in `q6.py`.
- Why the original maximum-distance heuristic satisfies $h(\text{state}) \leq 1 + h(\text{child})$, including when a move eats a food dot.
- How `g + h` changes the search effort compared with BFS on `playground`.

For Q6, submit `q6.py`, `q6_answers.md`, and your completed `q1.py`, `q2.py` and `q5.py`.