# Artificial Intelligence

Program – Search in Pacman

EECS 26' 劉祐廷 111060013

## Q4: A* Search

This algorithm is designed based on BFS algorithm framework. The slight differences between them are the data structure of **process** and **discovered_states**. **process** here is not a pure first-in-first-out queue but a lowest-first priority queue sorted by the **heuristic value** of each element. All the **heuristic values** are counted by the **heuristic function f** as below.

$$f(state) = g(state) + h(state)$$
$$g(state) = g(parent\ state) + the\ cost\ of\ state, \qquad g(start\ state) = 0$$
$$h(state) = ManhattanDistance(state, goal\ state)$$

It is trivial that the values counted by the **heuristic function** are always lower than the actual lowest cost from the start state to the goal state since Manhattan distance counts the least number of steps without any walls. The **heuristic values** are saved in the dictionary **discovered_states** which are indexed by **state**. Below is the structure of **discovered_states**.

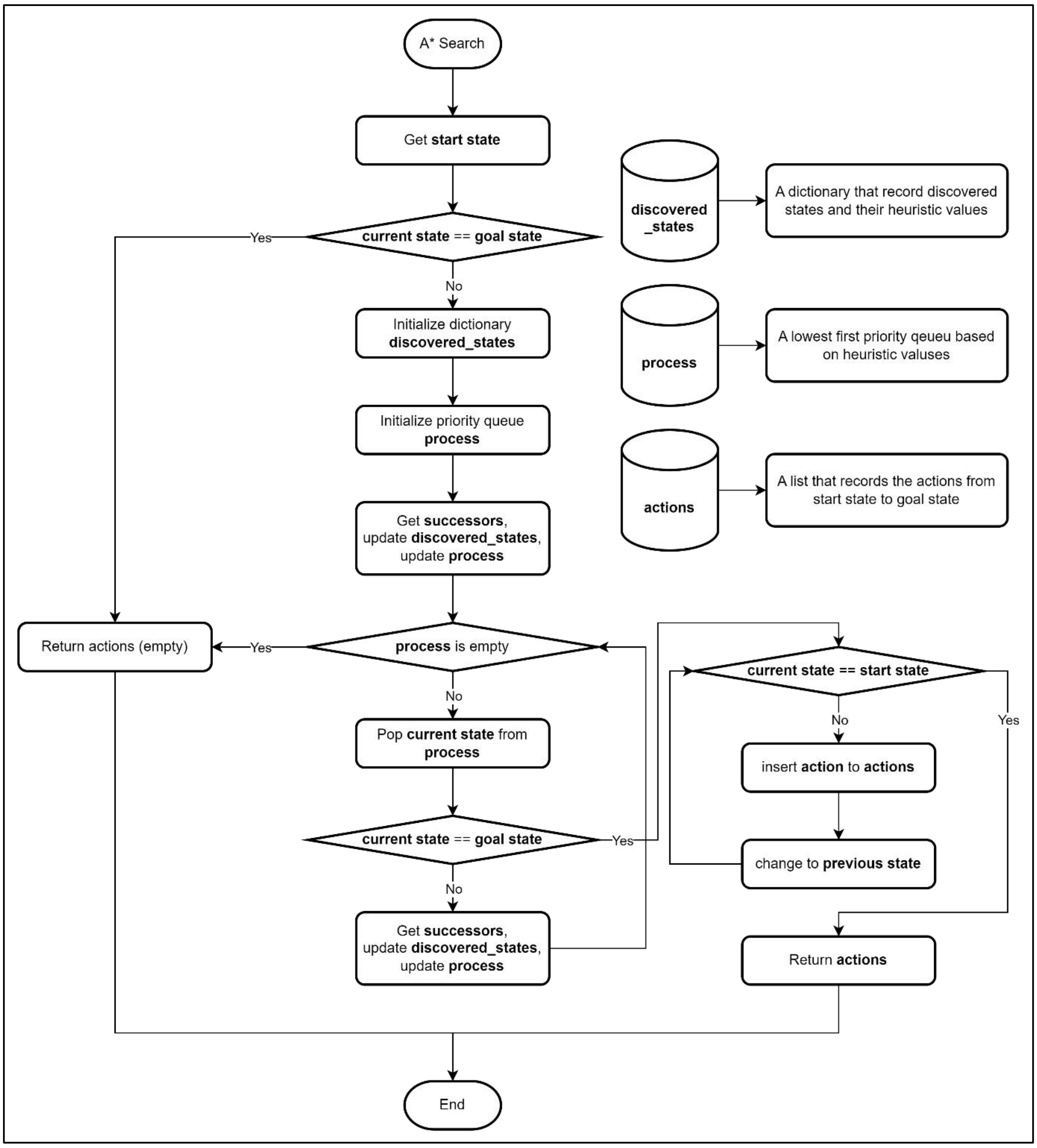
$$discovered\_states = \{state1: (parent\ state, action, heuristice\ value), state2: \ldots\}$$

For every push, **process** sorts the elements by checking the **heuristic values** stored in **discovered_states**. The structure of the nodes in **process** is defined as below.

$$node = (parrent\ state, (current\ state, action, cost))$$

Before every push, **discovered_states** is updated by the new heuristic value. If the state is undiscovered, insert the state with the new heuristic value into **discovered_states** directly. Otherwise, update **discovered_states[state]** if the new heuristic value is less than the value recorded in **discovered_states**.

In summary, this algorithm follows the BFS framework but incorporates a priority queue to explore the lowest-cost path first, guided by the heuristic function. By maintaining **discovered_states** and updating heuristic values dynamically, it ensures an efficient search toward the goal while considering optimality.


A* Search
Get start state
current state == goal state
Yes
No
Initialize dictionary discovered_states
Initialize priority queue process
Get successors, update discovered_states, update process
Return actions (empty)
Yes
process is empty
No
Pop current state from process
current state == goal state
Yes
No
Get successors, update discovered_states, update process
discovered_states
A dictionary that record discovered states and their heuristic values
process
A lowest first priority qeueu based on heuristic valuses
actions
A list that records the actions from start state to goal state
current state == start state
No
Yes
insert action to actions
change to previous state
Return actions
End

Picture 1: The flow chart of the A* search algorithm

## Q6: Corners Problem: Heuristic

Based on Q4, I modified the structure of **state**, **goal state**, and the **heuristic function**. The new structure of **state** is defined as below.

$$state = (current\ position, (positions\ of\ unvisited\ corners))$$

To fit the problem requirement, any two states with the same current position but with different unvisited corners are different, separating different layers with different goals. The **goal state** is defined as below, which means that the goal position is reached and there is only one goal.

$$goal\ state = (len(state[1]) == \ 1\ and\ state[0]\ in\ state[1])$$

In **getSuccessors()** function, the new tuple of **positions of unvisited corners** is remain the same if the current position is not an element of the original tuple of **positions of unvisited corners**. Otherwise, this tuple should be updated as below and combined into the success state.

$$i \ = \ state[1].index(state[0])$$
$$new\ goal\ tuple = \ state[1][:i] + \ state[1][i+1:]$$
$$nextState \ = \ (next\ position, new\ goal\ tuple)$$

The **heuristic function f** is defined here. (Denote Manhattan distance as MD)

$$f(state) = g(state) + h(state)$$
$$g(state) = \ g(parent\ state) + the\ cost\ of\ state, \qquad g(start\ state) = 0$$
$$h(state) = \ \max\,(MD(state[0], state[1][0]), MD(state[0], state[1][0]), \ldots)$$

Here I used max Manhattan distance of different goals, which must be lower than the actual cost from the current state to the goal state. Manhattan distance is admissible. For the case that has more than one goal, the real cost from the current position to the farthest goal with some intermediate goals is greater than the Manhattan distance from the current position to the farthest goal. Therefore, the **heuristic function** I designed is admissible. Since every step can only increase or decrease 1 in Manhattan distance which is not greater than the actual cost of each step, the **heuristic function** is consistent, even when the farthest goal switches. I also experimented with the min Manhattan distance and the mean Manhattan distance, which both are also admissible and consistent. However, their performances are not as good as the performance with the max Manhattan distance. Thus, I chose the max Manhattan distance method to evaluate the heuristic value.

In conclusion, by redefining the **state structure** and refining the **heuristic function**, this approach effectively handles multiple goals while ensuring admissibility and consistency. Through experimentation, the max Manhattan distance heuristic proved to be the most efficient, making it the optimal choice for evaluating heuristic values in this problem.