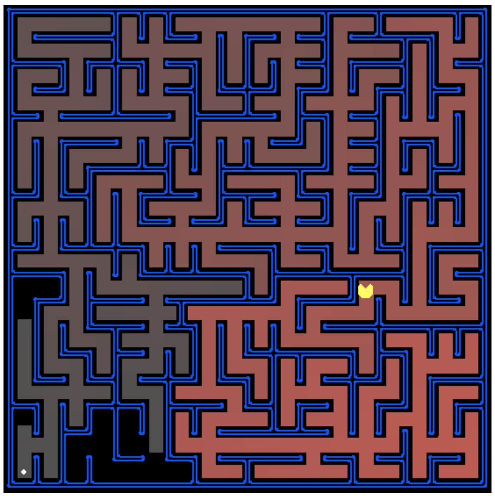
Pacman – Part 1: Searching

Welcome to the first part of the project for building a Pacman AI. In this installment, you will build general search algorithms to navigate through the Pacman world, both to reach a particular location and to collect food efficiently. Detailed instructions on how to do this are provided in Section 1 of the present document. Before getting started, however, it is highly recommended that you read the background material in Section 2, which gives a short introduction to the concepts needed for understanding this project.



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

1 Search algorithms – Implementation

For starters, you can play the game by typing in the command line:

```
python -m pacman
```

The provided project consists of several files, the majority of which you can ignore, except for a few ones you might want to read and understand at some later time. In `agents.py`, you will find a fully implemented `SearchAgent`, which plans out a path through Pacman's world, and then executes that path step-by-step. The search algorithms for formulating a plan are not implemented: **that's your job**! More precisely, all of your search functions need to return a list of actions that will lead the agent from the start to the goal: this is what we call a **plan**.

1.0 Warm-up

In this exercise, you are going to work on the file `ex00.py`. Run the following command

```
python ex00.py
```

to visualize a tiny maze with Pacman standing in the upper-right corner. Now, open the file `ex00.py` and note that the function `my_first_plan` returns an empty list, which is why Pacman does not move! Your job is to fill the list with the directions leading Pacman to the food pellet in the lower-left corner of the maze. Run again the above command when you are finished, and watch Pacman follow the plan you handcrafted. This gives you an idea of what is expected from a search algorithm.

1.1 Breadth-first search

In this exercise, you are going to work on the file `ex01.py`. Your job is to implement the graph search algorithm based on the **breadth-first** strategy; the pseudocode is given below.

```
1. Initialize the frontier as a FIFO queue                     > Use the queue in util.py
2. Push the start state into the frontier
3. Initialize the explored set with the start state            > Use a Python dictionary
4. While the frontier is not empty
    a. Pop a state from the frontier
    b. If it is a goal state, then stop
    c. For each successor of the popped state
        i. If the successor is not in the explored set, then
            1. Push the successor into the frontier
            2. Add the successor to the explored set           > Record the parent state and the action
```

Here are some tips to help you with the implementation.

- Store the "explored set" as a Python dictionary, which you create like this: `explored = {}`
- Use the states as keys of the dictionary: `explored[state]`
- Adding a state to the explored set means storing the tuple "parent state – associated action" as its value in the dictionary, such as: `explored[state] = (parent, action)`

When you are finished, run the following command to test your code on three different mazes.

```
python ex01.py
```

The Pacman board will show an overlay of the states explored, with the order of exploration represented from brighter to darker red. Is the exploration order what you would have expected?

You probably noticed that the above loop does not actually construct the paths; it only explores a maze until the goal state is found. Nonetheless, by storing the explored set as a python dictionary, you linked every explored state to its predecessor state and the corresponding action. With this information, you can easily reconstruct the path from start to goal; the pseudocode is given below.

```
Path reconstruction

1. Initialize the path as an empty list
2. Set the goal state as the current state
3. While the current state is not the start state
    a. Fetch the predecessor and the action of the current state from the explored set
    b. Append the action to the path
    c. Set the predecessor as the current state
4. Reverse the list of actions in the path
```

Append this code to your search algorithm, so that it returns the reconstructed path. Run again the above commands. Does BFS find a least cost solution? If not, check your implementation. Moreover, your search algorithm should also work for the eight-puzzle problem without any changes.

```
python -m puzzle
```

1.2 Uniform-cost search

In this exercise, you are going to work on the file `ex02.py`. Your job is to implement the graph search algorithm based on the **uniform-cost** strategy, which takes into account that the cost of actions may vary from a state to the other. The differences with breadth-first search are minimal. Firstly, the algorithm maintains the "past cost" of explored states (i.e., the sum of costs on the path from the start state), so as to prioritize the expansion of low-cost paths. Secondly, a state is added to the frontier not only when it has never been visited, but also when it has been discovered a new path to that state, and this path is cheaper than the best previous one. The pseudocode is given below, where the modifications with respect to breadth-first search are highlighted in red.

```
1. Initialize the frontier as a priority queue                    > Use the queue in util.py
2. Push the start state into the frontier with priority 0
3. Initialize the explored set with the start state               > Use a Python dictionary
4. Set the past cost of the start state to 0                      > Use a Python dictionary
5. While the frontier is not empty
     a. Pop a state from the frontier
     b. If it is a goal state, then stop
     c. For each successor of the popped state
          i. new cost = past cost of the popped state + cost of the action leading to successor
          ii. If the successor is not in the explored set
              or new cost is less than the past cost of the successor, then
               1. priority = new cost
               2. Push or update the successor into the frontier using the priority above
               3. Add the successor to the explored set
               4. Set the past cost of the successor to new cost
```

Here are some tips to help you with the implementation.

- Store the set of "past costs" as a Python dictionary, which you create like this: `past = {}`
- Use the states as keys of the dictionary: `past[state]`
- Setting the past cost of a state means storing or updating its value in the dictionary, such as: `past[state] = new_cost`. **Read carefully how "new cost" is defined.**

When you are finished, run the following command to test your code on three different scenarios.

```
python ex02.py
```

You should observe a different behavior in the three scenarios, because they use different cost functions (on the same maze). The movement costs are all equal in the first scenario, they are lower on the east side of the maze in the second scenario, and they are lower on the west side of the maze in the third scenario. As a result, the "lowest cost" path is different for the three scenarios.

Uniform-cost search is designed to care about the total cost of explored paths, and not about the number of steps they have. Whenever a state is selected from the frontier, the optimal path to that state has been found. As costs are nonnegative, paths never get shorter as states are added. These two facts together imply that uniform-cost search expands states in order of their optimal path cost. Hence, the first goal state selected for expansion must be the optimal solution.

1.3 A* search

With breadth-first search and uniform-cost search, the frontier expands in all directions. This is a reasonable choice if you are trying to find a path to all locations or to many locations. However, a common case is to find a path to only one location, in which case it is better to expand the frontier towards the goal. A* search uses a heuristic to reorder the explored states so that it is more likely that a goal state will be encountered sooner. If the heuristic does not overestimate the remaining cost to a closest goal, A* search finds an optimal path. With a null heuristic, A* search is identical to uniform-cost search; and if the cost of all actions is equal, it boils down to breadth-first search.

In this exercise, you are going to work on the file `ex03.py`. Your job is to implement the graph search algorithm based on the **A*** strategy, which assumes knowledge of the goal to reorder the priority of explored states so that it is more likely to find a path sooner. A heuristic is used to estimate the "future cost" of unexplored states (i.e., the cost of the cheapest path from the state to the goal). While the algorithm maintains the "past cost" of explored states as in uniform-cost search, the nodes on the frontier are prioritized by the sum of their past and future costs. The pseudocode is given below, where the modifications with respect to uniform-cost search are highlighted in red.

```
1. Initialize the frontier as a priority queue                         > Use the queue in util.py
2. Push the start state into the frontier with priority 0
3. Initialize the explored set with the start state                    > Use a Python dictionary
4. Set the past cost of the start state to 0                           > Use a Python dictionary
5. While the frontier is not empty
    a. Pop a state from the frontier
    b. If it is a goal state, then stop
    c. For each successor of the popped state
        i.  new cost = past cost of the popped state + cost of the action leading to successor
        ii. If the successor is not in the explored set
            or new cost is less than the past cost of the successor, then
            1. priority = new cost + estimated future cost of the successor
            2. Push or update the successor into the frontier using the priority above
            3. Add the successor to the explored set
            4. Set the past cost of the successor to new cost
```

Here are some tips to help you with the implementation.

- A* takes a heuristic function as an argument.
- Heuristic functions take two arguments: a state, and the search problem.

When you are finished, run the following command to test your code on three different scenarios.

```
python ex03.py
```

The three scenarios use different heuristic functions. More specifically, a null heuristic (function that always returns zero) is used in the 1st scenario, the Manhattan distance is used in the 2nd scenario, and the Euclidean distance is used in the 3rd scenario (see the figure in Section 2.7). You should see that A* finds the optimal solution faster in the 2nd scenario than the 1st scenario, with about 221 versus 269 search nodes expanded (but ties in priority may make your numbers differ slightly). What happens on "openMaze" for the various heuristics and movement costs?

1.4 Finding all the corners

The real power of A* search will only be apparent with a more challenging search problem. Now, it's time to formulate a new problem and design a heuristic for it. In corner mazes, there are four dots, one in each corner. Your new search problem is to find the shortest path through the maze that touches all four corners, regardless of whether the maze actually has food there or not.

In this exercise, you are going to work on the file `ex04.py`. Your job is to fill in the missing pieces in the class `CornersProblem` that represents the search problem of "finding all the corners". Firstly, you will need to choose a state representation that encodes all the information necessary to detect whether all four corners have been reached. Then, you will implement the functions `getStartState`, `isGoalState`, and `getSuccessors`. The real work happens in the latter function, where you need to generate all the successors of a given state, according to the legal actions available in that state. Here are some tips to help you with the implementation.

- Read Section 2.1 to help you think of a possible state representation.
- Use a tuple to define a state (tuples can be nested), so it can be used as a dictionary key.
- States must **not** include irrelevant information (like the position of extra food, ghosts, etc.).

When you are finished, run the following command to test your code on two different mazes.

```
python ex04.py
```

The shortest path through "tiny corners" takes 28 steps (and it does not go to the closest food first). Breadth-first search expands 252 nodes on "tiny corners", and just under 2000 search nodes on "medium corners". Next, you will see that A* search can reduce the amount of searching required.

1.5 Heuristic for finding all the corners

In this exercise, you are going to work on the file `ex05.py`. Your job is to implement a heuristic for the search problem of "finding all the corners". Heuristics are functions that take search states, and return numbers that estimate the cost of the remaining path from the state to a nearest goal. To guarantee correctness in graph search, a heuristic must be admissible and consistent.

- **Admissibility**. The heuristic values must be non-negative lower bounds on the actual shortest path cost to the nearest goal (and thus exactly zero at goal states).
- **Consistency**. It must additionally hold that if an action has cost "c", then taking that action can only cause a drop in the heuristic of at most "c" (see the figure in Section 2.8). Note that consistency implies admissibility.
- **Dominance.** More effective heuristics will return values closer to the actual shortest path cost. If a collection of admissible heuristics is available, and none of them dominates any of the others, it is always best to take the maximum: $\mathbf{h(s) = \max\{ h_1(s), \dots, h_N(s) \}}$.
- **Relaxation.** A search problem with fewer restrictions on the actions is called a relaxed problem: in pacman world, this could be a maze without walls. The cost of an optimal solution to a relaxed problem is a consistent heuristic for the original problem.
- **Non-triviality.** The trivial heuristics are the ones that return zero everywhere, or that compute the true completion cost. The former won't save you any time, while the latter will be as difficult as the original problem to evaluate.

Your heuristic must be non-trivial and consistent. It is usually easiest to start out by brainstorming admissible heuristics. Make sure that your heuristic returns 0 at every goal state, and never returns a negative value. Once you have an admissible heuristic that works well, you can check whether it is consistent. The only way to ensure consistency is with a proof. However, inconsistency can often be detected by verifying that for each state you expand, its successors have equal or higher past costs. Moreover, if uniform-cost search and A* search ever return paths of different lengths, your heuristic is inconsistent. When you are finished with your heuristic, run the following command.

```
python ex05.py
```

You will be ranked as follow, depending on how few search nodes your heuristic expands.

Ranking	**1st place**	**2nd place**	**3rd place**	**4th place**	**5th place**
Expanded nodes	< 500	< 1’000	< 1’2000	< 1’600	< 2’0000

1.6 Heuristic for eating all the food

Now you will solve a hard search problem: eating all the Pacman food in as few steps as possible. For this, you need a new search problem definition which formalizes the food-clearing problem, already implemented for you by `FoodSearchProblem` in the file `problems.py`. A solution is defined to be a path that collects all of the food in the Pacman world. If you have written your search methods correctly, A* search with a null heuristic (equivalent to uniform-cost search) should quickly find an optimal solution to different layouts with no code change on your part. You should find that a null heuristic finds a path of length 27 after expanding 5’057 search nodes for “tiny search”, and a path of length 60 after expanding 16’688 nodes for “tricky search”.

```
python ex06.py
```

In this exercise, you are going to work on the file `ex06.py`. Your job is to implement a heuristic for the search problem of “eating all the dots”. When you are finished, run again the above command. Depending on how few states your heuristic expands, you will be ranked as follow.

Ranking	**1st place**	**2nd place**	**3rd place**	**4th place**	**5th place**
Expanded nodes	< 1’000	< 5’000	< 10’000	< 12’000	< 15’000

1.7 Suboptimal search

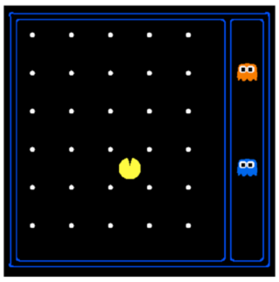
Sometimes, even with A* and a good heuristic, finding the optimal path through all the dots is hard. In these cases, you would still like to find a reasonably good path, quickly. A fast yet suboptimal approach is to write an agent that solves a sequence of search problems, in each of which it always greedily eats the closest dot, until no food is left. This is done by `ClosestDotSearchAgent` in the file `agents.py` (already implemented for you). In this exercise, you are going to work on the file `ex07.py`. Your job is to complete `AnyFoodSearchProblem`, which is missing its goal test (true for any state where some food is being eaten). When you are finished, run the following command.

```
python -m pacman -a ClosestDotSearchAgent -l mediumSearch
```

Your agent solves the maze in under a second with a path cost of 171 on “medium search” maze. This approach won’t always find the shortest possible path through the maze, because “eating all the food” is an NP-hard problem! Look at the folder `layout` for the list of all the mazes you can try.

2 Search algorithms – Background

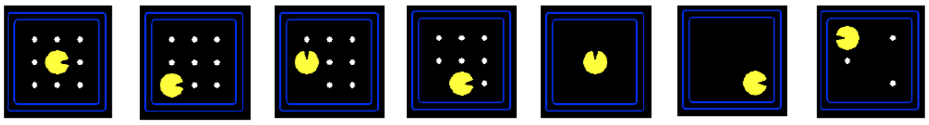
In artificial intelligence, an **agent** is a computational entity that performs a series of **actions** in a complex and dynamic **environment**, in order to reach a narrowly-defined **goal**. To this end, the agent maintains a model of the **world** (a representation of the agent and the environment together), which uses to simulate the execution of various actions and determine their hypothetical consequences, in order to choose the best sequence of actions for achieving its goal. Interesting applications of this framework include game AI, automatic control, and robotics. For example, the Pacman character is an agent that aims to eat all the dots inside an enclosed maze, while avoiding four colored ghosts. The maze, the dots, and the ghosts constitute the environment, with which the agent interacts by moving in one of the four cardinal directions: north, south, east, or west.



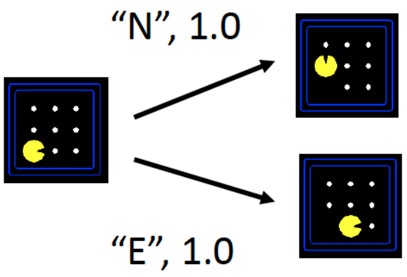
2.1 Search problem

To create a planning agent, we must formally express the world as a **search problem**. Formulating such a problem requires five things.

1. **State space** – Set of all states that are possible in the world. A state represents the picture of the world at a certain point along the progression of the agent's plan to reach its goal.



2. **Action space** – Set of all actions that are possible in the world.
3. **Successor function** – Function that takes in a state-action pair and returns two pieces of information: the future state in which the world would be if the given action is executed in the given state, along with the cost for computing that action.
4. **Initial state** – State in which the agent exists initially.
5. **Goal test** – Function that checks if a state is the goal.



To illustrate these concepts, let's consider two variants of the game of Pacman.

- **Navigation.** The goal is to get from position (x_0, y_0) to position (x*, y*) in the maze optimally.
  - **State**: (x,y) location.
  - **Action**: north, south, east, or west.
  - **Successor**: next (x,y) location.
  - **Goal test**: check if the location is equal to (x*, y*).

  If there are **N** Pacman positions, this search problem has **N** states.

- **Eating all the food.** The goal is to consume all food in the maze optimally.
  - **State**: (x,y) location, and array indicating whether each food pellet has been eaten.
  - **Action**: north, south, east, or west.
  - **Successor**: next (x,y) location and updated array of Booleans.
  - **Goal test**: check if all Booleans are false.

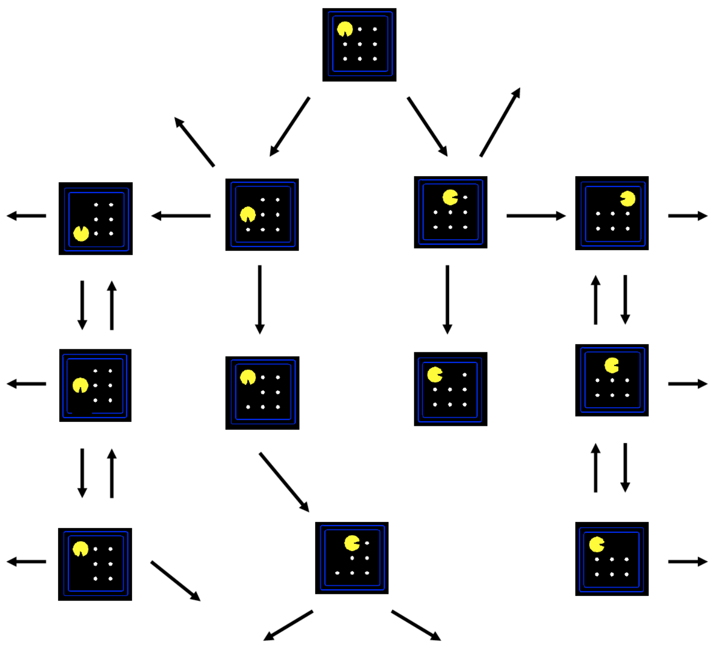
  If there are **N** Pacman positions and **K** food pellets, this search problem has **$N \times 2^K$** states.

2.2 State space graph

A search problem can be mathematically described by a **state space graph**, where

- a **node** represents a search state,
- a **direct edge** represents the action that connects a state to one of its successors,
- an **edge weight** represents the cost of performing the corresponding action.

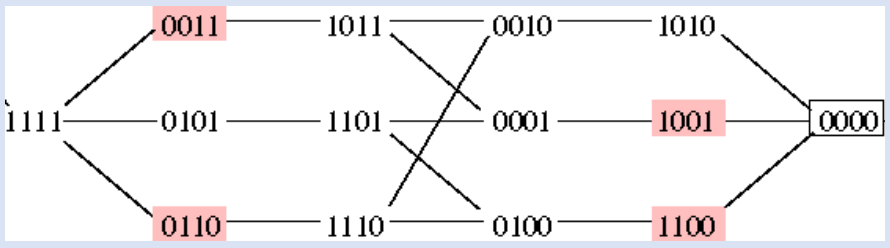
In this framework, a search problem is solved by exploring the state space graph, in order to find a path from the start state to the goal state. This path (typically called **plan**) indicates the sequence of actions that takes the agent from the start to the goal.



The wolf-goat-cabbage problem

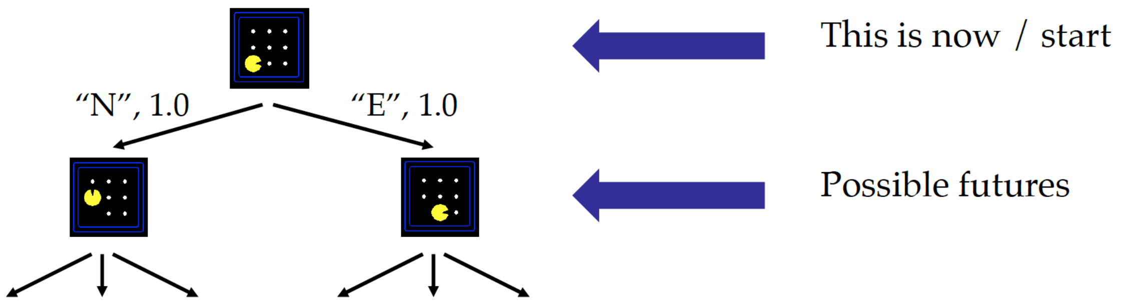
To illustrate the concept of state space graph, consider the following example. You are on the bank of a river with a boat, a cabbage, a goat, and a wolf. Your goal is to get everything to the other side. There are however three restrictions: (1) only you can handle the boat; (2) when you're in the boat, there is only space for one more item; (3) you can't leave the goat alone with the wolf, nor with the cabbage, otherwise something will be eaten.

This task can be expressed as a search problem, where a state indicates the positions of boat, cabbage, goat and wolf. Since there are only two positions (left or right bank of the river), a state is represented by 4 bits (1=left, 0=right, order=BCGW). The state space graph is illustrated below, where all edges are bidirectional, and the colored states are not allowed. The path from the start state (1111) to the goal state (0000) provides the solution to the problem.



2.3 Search tree

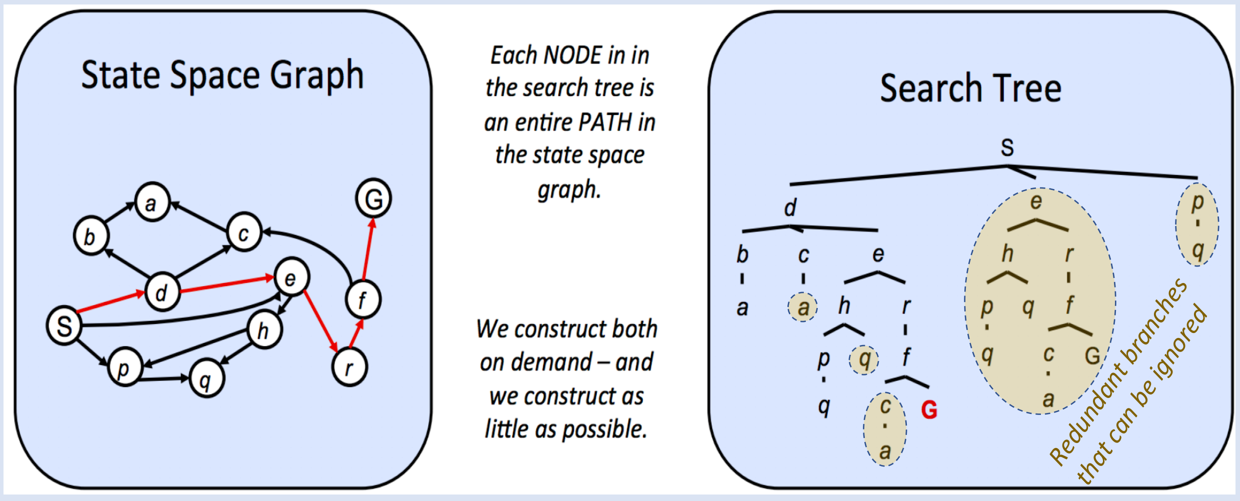
A state space graph is helpful when thinking about the structure of a search problem. However, we cannot build this graph explicitly, because it is exponentially large for any non-trivial problem. What we can actually work with is a structure called **search tree**, which keeps track of all the paths that can be traversed on the state space graph, beginning from the start state. We cannot build the entire search tree either. Rather we try to find a solution, that is a path from the start state to the goal state, while expanding as little of the search tree as possible.



Traversing the state space graph with a search tree

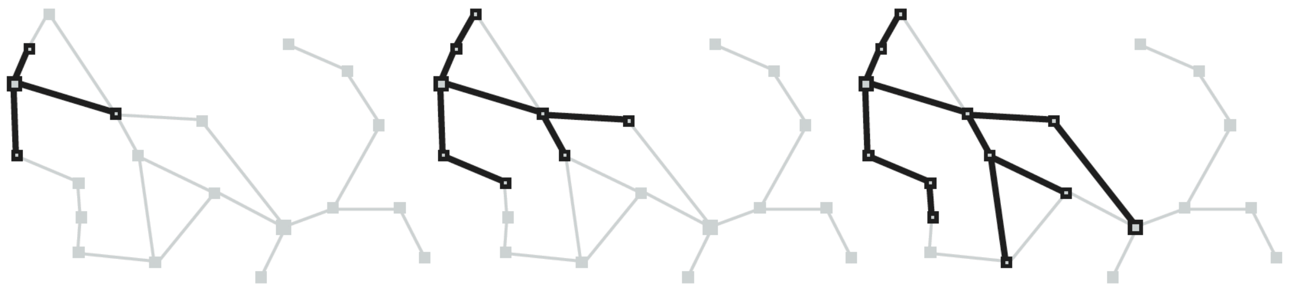
Observe the state space graph and corresponding search tree in the figure below. The highlighted path (S - d - e - r - f - G) in the graph is represented in the search tree by the path from the root (state S) to the leaf highlighted in red (state G). Similarly, each and every path from the start state to any other state is represented in the search tree by a path from the root to some descendant node corresponding to the other state. That is, a node of the search tree encodes an entire path in the state space graph. Since there often exist multiple ways to get from one state to another, states tend to show up multiple times in search trees.

If you are concerned about reaching the goal, there is never any reason to build a search tree with more than one path to any given state. The way to avoid exploring redundant paths is to ignore the tree branches starting with states already explored. By so doing, the search tree contains at most one copy of each state. Failure to detect repeated states may lead to exponentially more work, often causing a tractable problem to become intractable.


State Space Graph
Each NODE in in the search tree is an entire PATH in the state space graph.
We construct both on demand – and we construct as little as possible.
Search Tree
Redundant branches that can be ignored

2.4 Search algorithms

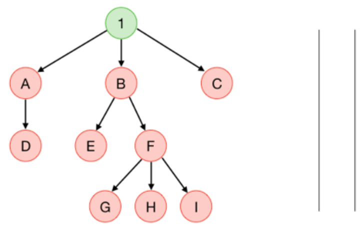
The standard protocol for finding a path from start to goal is to progressively build a search tree, with the start state at the root. The first step is to expand the root by applying each legal action to the start state, thereby generating a set of successor states, which are attached to the root as its children nodes. Then, one of these nodes is chosen to be expanded, and all the *unexplored* successor states are added as its children. The set of all states available for expansion at any given time is called **frontier**. The process of expanding nodes on the frontier continues until either a goal state is found, or there are no more states to explore. The search tree thereby constructed contains at most one copy of each state, so we can think of it as growing a tree directly on the state-space graph. The figure below illustrates the process of generating a search tree with this algorithm (**graph search**).



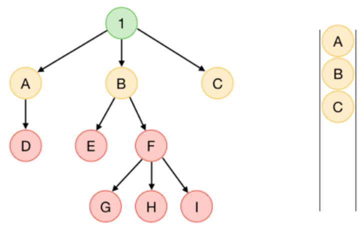
Search algorithms all share the basic structure. The main difference is in the **search strategy**, which determines how they choose the state to explore next. When we have no knowledge of the location of goal states in our search tree, we are forced to use **uninformed search** strategies. But if we have some notion of the direction in which we should focus our search, we can improve performance with **informed search** strategies. We will now cover several such strategies in succession.

2.5 Breadth-first search

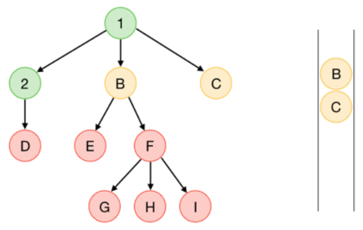
This strategy explores equally in all directions. The root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on. In general, the shallowest unexplored node is chosen for expansion. This is achieved very simply by using a FIFO queue for the frontier. Thus, new nodes go to the back of the queue, and old nodes, which are shallower than the new nodes, get expanded first. The strategy is illustrated in the sequence below.



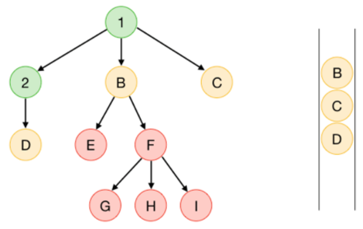
(a) Explore "root"



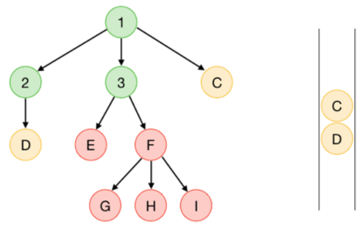
(b) Expand "root" successors



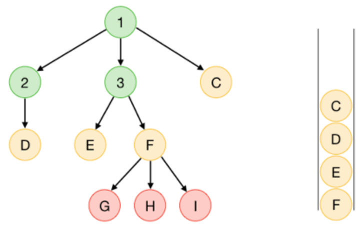
(c) Explore "node A"



(d) Expand "node A" successors



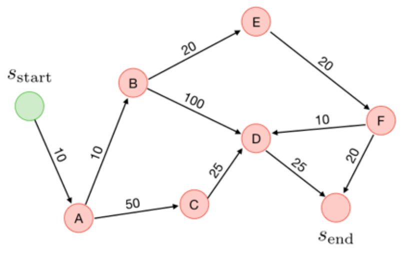
(e) Explore "node B"



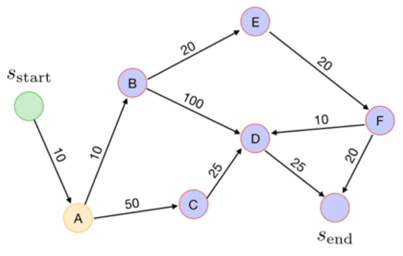
(f) Expand "node B" successors

2.6 Uniform-cost search (Dijkstra algorithm)

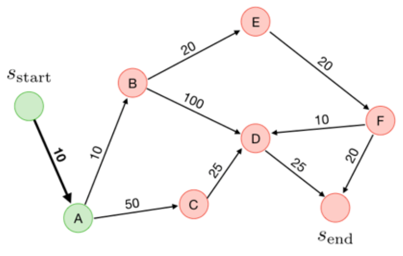
When the cost of all actions is equal, breadth-first search is optimal because it always expands the shallowest unexpanded node. By a simple extension, we can find an algorithm that is optimal with arbitrary costs. Instead of expanding the shallowest node, uniform-cost search expands the node with the lowest path cost from the start state (**past cost**). This is done by storing the frontier as a priority queue ordered by the path cost. In addition, a test is added in case a better path is found to a node currently on the frontier. The strategy is shown in the sequence below.



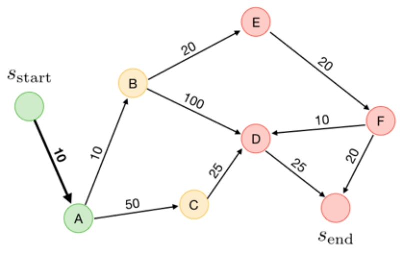
(a) Explore "root"



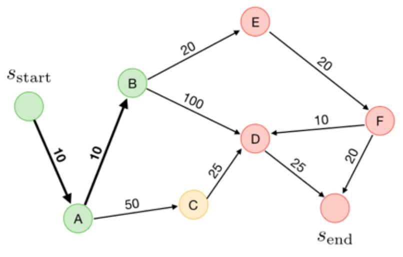
(b) Expand "root" successors



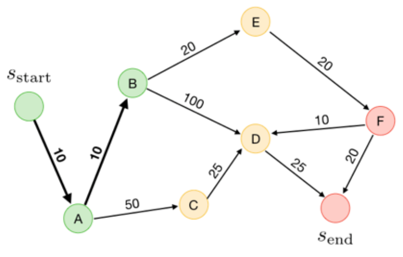
(c) Explore "node A"



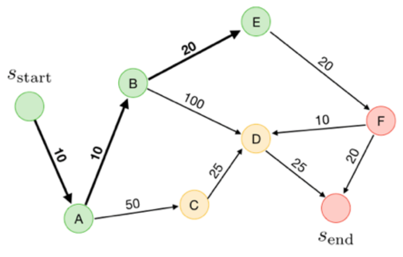
(d) Expand "node A" successors



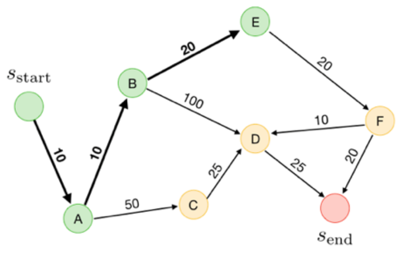
(e) Explore "node B"



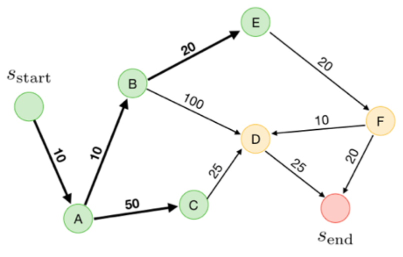
(f) Expand "node B" successors



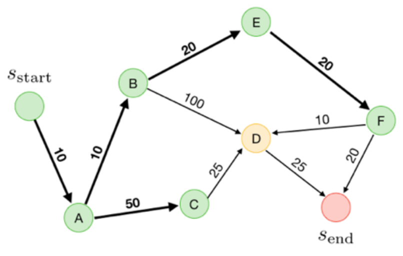
(g) Explore "node E"



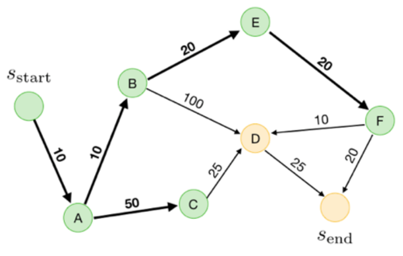
(h) Expand "node E" successors



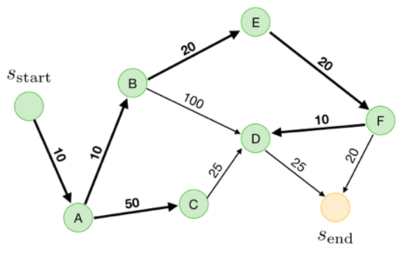
(i) Explore "node C"



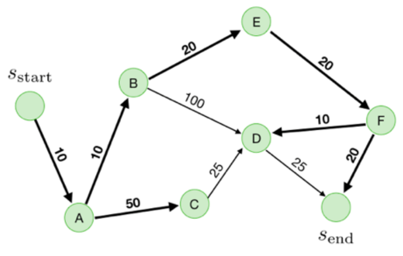
(m) Explore "node F"



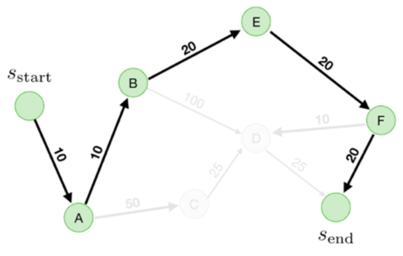
(n) Expand "node F" successors



(o) Explore "node D"



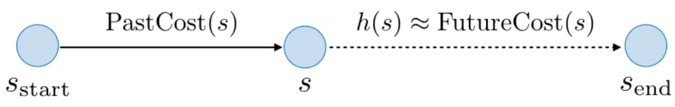
(p) Explore "final node"



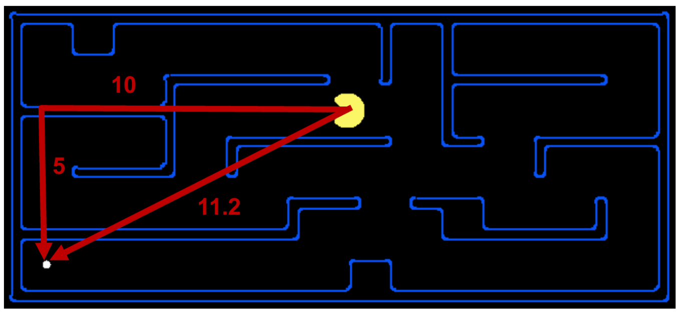
(q) Path with lowest cost

2.7 Greedy best-first search

Instead of expanding the node that is closest to the start state, this strategy expands the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly. To do this, a **heuristic** function "*h(s)*" is used to estimate the cost of the cheapest path from the given state to a goal state (**future cost**). The frontier is stored as a priority queue ordered by the estimated future cost, so that the node with the lowest evaluation of the heuristic function is expanded first.



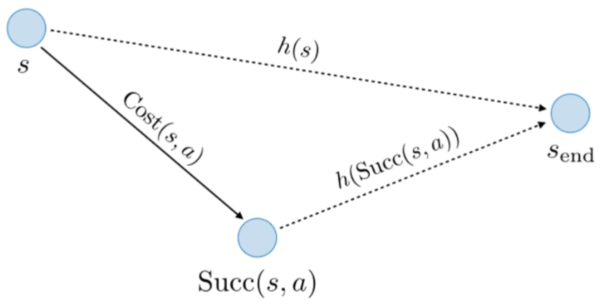
Heuristics are the driving force to prioritize the exploration of paths that seem to be leading closer to a goal. The computation performed by a heuristic function is specific to the search problem being solved, and usually it is a lower bound on the remaining distance to the goal. Turning to our Pacman example, a common heuristic used for pathfinding is the (Euclidean or Manhattan) distance from Pacman's current location to Pacman's desired location, assuming a lack of walls in the maze.



2.8 A* search

Uniform-cost search works well to find the shortest path, but it wastes time exploring in directions that aren't promising, whereas greedy best-first search explores in promising directions, but it may not find the shortest path. To combine their strengths, A* search uses both the actual distance from the start state (past cost) and the estimated distance to the closest goal state (future cost). This is done by storing the frontier as a priority queue in which the nodes are ordered by the sum of their past and future costs, effectively yielding an estimated total cost from start to goal.

A∗ search is more than just a reasonable strategy: it is both complete and optimal, as long as the heuristic function *"h(s)"* that estimates the future cost is **consistent**. This condition is a form of triangular inequality between a state, a successor state, and a goal state, which stipulates that the estimated cost of reaching the goal from a state is no greater than the action cost of getting to a successor, plus the estimated cost of reaching the goal from the successor itself; see figure below.



$$\boxed{h(s) \leqslant \text{Cost}(s,a) + h(\text{Succ}(s,a))}$$