6.01 Final Exam Fall 2016

Name: **Section Number:**

Kerberos (Athena) name:

Section	Design Lab Time
1:	Thursday 9:30am
2:	Thursday 2:00pm

Please WAIT until we tell you to begin.

During the exam, you may refer to any written or printed paper material.
You may NOT use any electronic devices (including calculators, phones, etc).

If you have questions, please **come to us at the front** to ask them.

Enter all answers in the boxes provided.

Extra work may be taken into account when assigning partial credit,
but **only work shown on pages with QR codes will be considered.**

Question 1:	18 Points
Question 2:	10 Points
Question 3:	10 Points
Question 4:	16 Points
Question 5:	18 Points
Question 6:	18 Points
Question 7:	18 Points
Question 8:	16 Points
Total:	124 Points

1 Catching the Bus (18 Points)

Ben Bitdiddle needs to catch a bus in an infinite 2-d grid. We know the bus's schedule, and we want to use the graph search algorithms we've built up in 6.01 to plan a path such that Ben and the bus end up in the same grid location at the same time.

Assume that the search starts at time 0 with Ben in location (0,0). On each step, time increases by 1, and Ben can move to any of the **eight** neighboring locations, or he can remain in the same location.

The position of the bus is specified by a procedure `bus_schedule`, which takes a time as input and returns the bus's location at that time as a tuple `(r,c)`.

1.1 Paths

For the following implementations of the bus's schedule, what is one sequence of locations (representing Ben's location at time 0, 1, 2...) that might result from running a breadth-first search in this domain? Enter `None` if no path is returned, either because BFS runs indefinitely, or because it returns without having found a path. Note that there may be more than one correct answer, but you need only enter one.

1.1.1 Schedule 1

```
def bus_schedule(t):
    return (1,t-2)
```

Path found by BFS:

1.1.2 Schedule 2

```
def bus_schedule(t):
    return (2, (3*t) % 8)
```

Path found by BFS:

1.1.3 Schedule 3

```
def bus_schedule(t):
    return (t+1, t+1)
```

Path found by BFS:

1.2 Search Implementation

Recall that, in order to perform a search using the methods we have built up in 6.01, we need the following pieces:

- an appropriate successor function
- a goal test function
- a starting state

Enter your definitions for these pieces below.

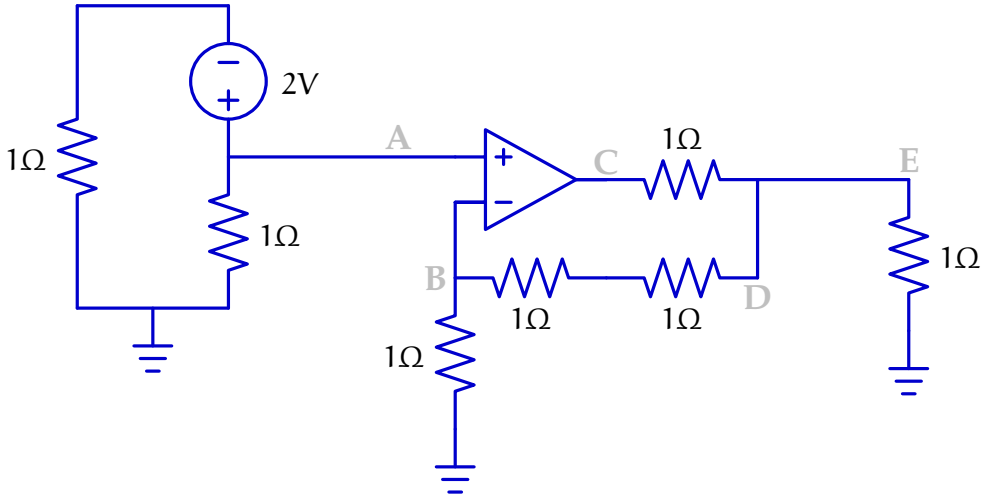
```
DIRECTIONS = [(0,0), (1,0), (0,1), (-1,0), (0,-1), (1,1), (1,-1), (-1,1), (-1,-1)]

def bus_catching_successors(state):
    # your code here
```

```
def goal_test(state):
    #your code here
```

```
#enter your code for start_state below
start_state =
```

2 Op-Amp (10 Points)

In the circuit below, solve for the voltages at the nodes labeled A, B, C, D, and E, all relative to the indicated ground. Make the ideal op-amp assumption, and ignore output limitations of the op-amp. Write your answers (including units) in the boxes below.



V_A =

V_B =

V_C =

V_D =

V_E =

Worksheet (intentionally blank)

3 Admissible (10 Points)

In this question, we will look at automating the process of deciding whether or not given heuristic functions are admissible in a particular search domain.

On the facing page, write a function `admissible_heuristics` that takes five arguments:

- `heuristics`: a list containing the heuristic functions to test
- `all_states`: a list containing all the states in the search space from which the goal is reachable
- `successors`: an appropriate successor function for searching in this domain using `uc_search`
- `goal_test`: a goal test function for searching in this space
- `path_cost`: a function that takes a path (a list of states) and returns the cost of that path

Your function should return a list containing only the heuristic functions from the `heuristics` list that are admissible. If none of the functions are admissible, it should return an empty list.

Note that the source code for the `uc_search` function from `lib601` is included for reference on the last page of this exam, which you may remove.

```
from lib601.search import uc_search

def admissible_heuristics(heuristics, all_states, successors, goal_test, path_cost):
    # your code here
```

4 System (16 Points)

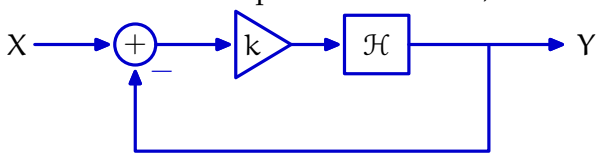
Consider an LTI system described by the following difference equation:

$$y[n+1] = 0.5y[n] + x[n]$$

1. Determine the system functional $\mathcal{H}$ for this system.

$\mathcal{H} =$

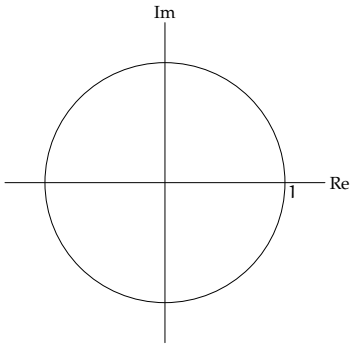
2. $\mathcal{H}$ is now placed in a feedback control loop as shown below, where k is a constant gain:



Determine $\frac{Y}{X}$ for this new system, as a rational polynomial in $\mathcal{R}$ (not including $\mathcal{H}$).

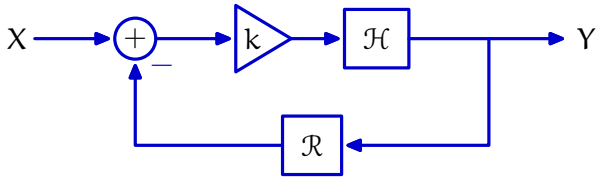
$\frac{Y}{X} =$

3. For K=1, draw the pole(s) of $\frac{Y}{X}$ on the complex plane below.



4. Determine the range of k for which the system is stable, and enter the range in the box below:

5. Now assume a delay is introduced in the feedback loop:



Determine the new system functional $\frac{Y}{X}$ as a rational polynomial in $\mathcal{R}$ (not including $\mathcal{H}$).

$$\frac{Y}{X} =$$

6. For each gain k, which of the following best describes the system's unit sample response?

A. $y[n] = 0$ for all n

B. $y[n]$ constant and nonzero for all $n \geqslant 0$

C. $y[n]$ oscillatory and decaying

D. $y[n]$ oscillatory with constant amplitude

E. $y[n]$ oscillatory and increasing to infinity

F. $y[n]$ monotonic and decaying

G. $y[n]$ monotonic and increasing to infinity

Enter one of these letters in each box below:

$k = -1$:

$k = 1$:

$k = 0$:

5 Return of the Killer Hornets (18 Points)

Consider a world with some population B of nice, pleasant bees and a population H of crazy, evil, killer Japanese giant hornets. Bees don't bother anyone, enjoying life and making honey. Hornets kill bees (when they can catch them). The bee population naturally increases, but when there are hornets around, they kill the bees and decrease the bee population. The hornet population, in the absence of honeybees to hunt, stays the same, or declines a little.

5.1 Initial distribution

Let's assume that the bee population (B) can be low, med or high, and that the hornet population (H) can be low, med, or high. So, there are **9 states**, each corresponding to some value of B and some value of H.

Here is the initial **belief state**, which is written as a joint distribution over B and H, $\Pr(B, H)$.

		Hornets		
		low	med	high
Bees	low	0.04	0.20	0.18
	med	0.08	0.16	0.02
	high	0.28	0.04	0.00

1. What is the marginal distribution over the hornet population, $\Pr(H)$?

2. What is the distribution $\Pr(B|H = low)$?

3. What is the distribution $\Pr(B|H = high)$?

4. Are B and H independent? Explain why or why not.

5.2 Transitions

Let's start by studying how the bee population evolves when there are no hornets, represented by $H_t = low$ (and the hornet population doesn't change).

$$\Pr(B_{t+1} = low \mid H_t = low, B_t = low) = 0.1$$
$$\Pr(B_{t+1} = med \mid H_t = low, B_t = low) = 0.9$$
$$\Pr(B_{t+1} = high \mid H_t = low, B_t = low) = 0.0$$
$$\Pr(B_{t+1} = low \mid H_t = low, B_t = med) = 0.0$$
$$\Pr(B_{t+1} = med \mid H_t = low, B_t = med) = 0.3$$
$$\Pr(B_{t+1} = high \mid H_t = low, B_t = med) = 0.7$$
$$\Pr(B_{t+1} = low \mid H_t = low, B_t = high) = 0.0$$
$$\Pr(B_{t+1} = med \mid H_t = low, B_t = high) = 0.0$$
$$\Pr(B_{t+1} = high \mid H_t = low, B_t = high) = 1.0$$

1. Assume $H_t = low$. For simplicity, also assume that we start out knowing with certainty that the bee population is low. What is the distribution over the possible levels of the bee population (low, med, high) after 1 time step?

2. Assume $H_t = low$. For simplicity, also assume that we start out knowing with certainty that the bee population is low. What is the distribution over the possible levels of the bee population (low, med, high) after 2 time steps?

3. Assume $H_t = low$. For simplicity, also assume that we start out knowing with certainty that the bee population is low. What value does the distribution over the possible levels of the bee population (low, med, high) approach as the number of time steps goes to infinity?

5.3 Observations

Imagine that you are starting with the initial belief state, as given by the joint distribution from problem 5.1. If it helps, you can think of it as the following `DDist` over pairs of values (the first is the value of B, the second is the value of H):

```
DDist({('low', 'low') : 0.04, ('low', 'med') : 0.2, ('low', 'high') : 0.18,
       ('med', 'low') : 0.08, ('med', 'med') : 0.16, ('med', 'high') : 0.02,
       ('high', 'low') : 0.28, ('high', 'med') : 0.04, ('high', 'high') : 0.00})
```

You send an ecologist out into the field to sample the numbers of bees and hornets. The ecologist can't really figure out the absolute numbers of each species, but reports one of three observations:

- **moreH**: means that there are significantly more hornets than bees (that is, that the level of hornets is **high** and the level of bees is **med** or **low**, or that the level of hornets is **med** and the level of bees is **low**).
- **moreB**: means that there are significantly more bees than hornets (that is, that the level of bees is **high** and the level of hornets is **med** or **low**, or that the level of bees is **med** and the level of hornets is **low**).
- **same**: means that there are roughly the same number of hornets as bees (the populations have the same level).

1. If there is no noise in the ecologist's observations (that is, the observation is always true, given the state), and the observation is **moreB**, what is the resulting belief state $\Pr(B_0, H_0 \mid O_0 = \textbf{moreB})$ (the distribution over states given the observation)? Fill in the table below with the joint probabilities in the updated belief:

		Hornets		
		low	med	high
Bees	low			
	med			
	high			

2. Now, we will assume that the ecologist's observations are fallible.

$$\Pr(O_t = \textbf{moreH} \mid B_t < H_t) = 0.9$$
$$\Pr(O_t = \textbf{same} \mid B_t < H_t) = 0.1$$
$$\Pr(O_t = \textbf{moreB} \mid B_t < H_t) = 0.0$$
$$\Pr(O_t = \textbf{moreH} \mid B_t = H_t) = 0.1$$
$$\Pr(O_t = \textbf{same} \mid B_t = H_t) = 0.8$$
$$\Pr(O_t = \textbf{moreB} \mid B_t = H_t) = 0.1$$
$$\Pr(O_t = \textbf{moreH} \mid B_t > H_t) = 0.0$$
$$\Pr(O_t = \textbf{same} \mid B_t > H_t) = 0.1$$
$$\Pr(O_t = \textbf{moreB} \mid B_t > H_t) = 0.9$$

Again starting from the initial belief state, if the observation is **moreB**, what is the belief state $\Pr(B_0, H_0 \mid O_0 = \textbf{moreB})$ (the distribution over states given the observation)? Fill in the table below with the joint probabilities in the updated belief:

		Hornets		
		low	med	high
Bees	low			
	med			
	high			

6 Flood Fill (18 Points)

In computer drawing programs, a "paint bucket" tool is often provided, which re-colors all the cells of a particular color in an enclosed region. When a user clicks on a pixel, that pixel and all surrounding pixels that share the original color are replaced with a different color. Typically, this is implemented with a "flood fill" algorithm that closely resembles the graph search algorithms we have seen in 6.01.

For now, we will assume that an image is represented as a 2-dimensional array (a list of lists) called `image`, where `image[r][c]` returns the color of the pixel in row `r` and column `c`, `image.height` is the number of rows in the image, and `image.width` is the number of columns in the image. Also, assume that the color that was originally clicked is stored as `clicked_color`, that the desired color is stored as `fill_color`, and that a copy of the original image (before the flood fill) is stored as `original_image`.

Consider three possible successor functions for a search in this space:

```
01    def color_successor_1(point):
02        row, col = point
03        neighbors = []
04        for (dr, dc) in [(0,1), (1,0), (0,-1), (-1,0)]:
05            new_r = row+dr
06            new_c = col+dc
07            if 0 <= new_r < image.height and 0 <= new_c < image.width:
08                image[new_r][new_c] = fill_color
09                if original_image[new_r][new_c] == clicked_color:
10                    neighbors.append((new_r, new_c))
11        return neighbors
```

In `color_successor_2`, lines 8-10 instead read:

```
08                if original_image[new_r][new_c] == clicked_color:
09                    image[new_r][new_c] = fill_color
10                neighbors.append((new_r, new_c))
```

In `color_successor_3`, lines 8-10 instead read:

```
08                if original_image[new_r][new_c] == clicked_color:
09                    image[new_r][new_c] = fill_color
10                    neighbors.append((new_r, new_c))
```

Three possible goal tests:

```
def goal_test_a(state):
    return True

def goal_test_b(state):
    return False

def goal_test_c(state):
    return original_image[state[0]][state[1]] != clicked_color
```

And two possible search functions, each implemented as in `lib601` (note that the source code of the `search` function is available on the last page of this exam, which you may remove):

i. BFS with dynamic programming

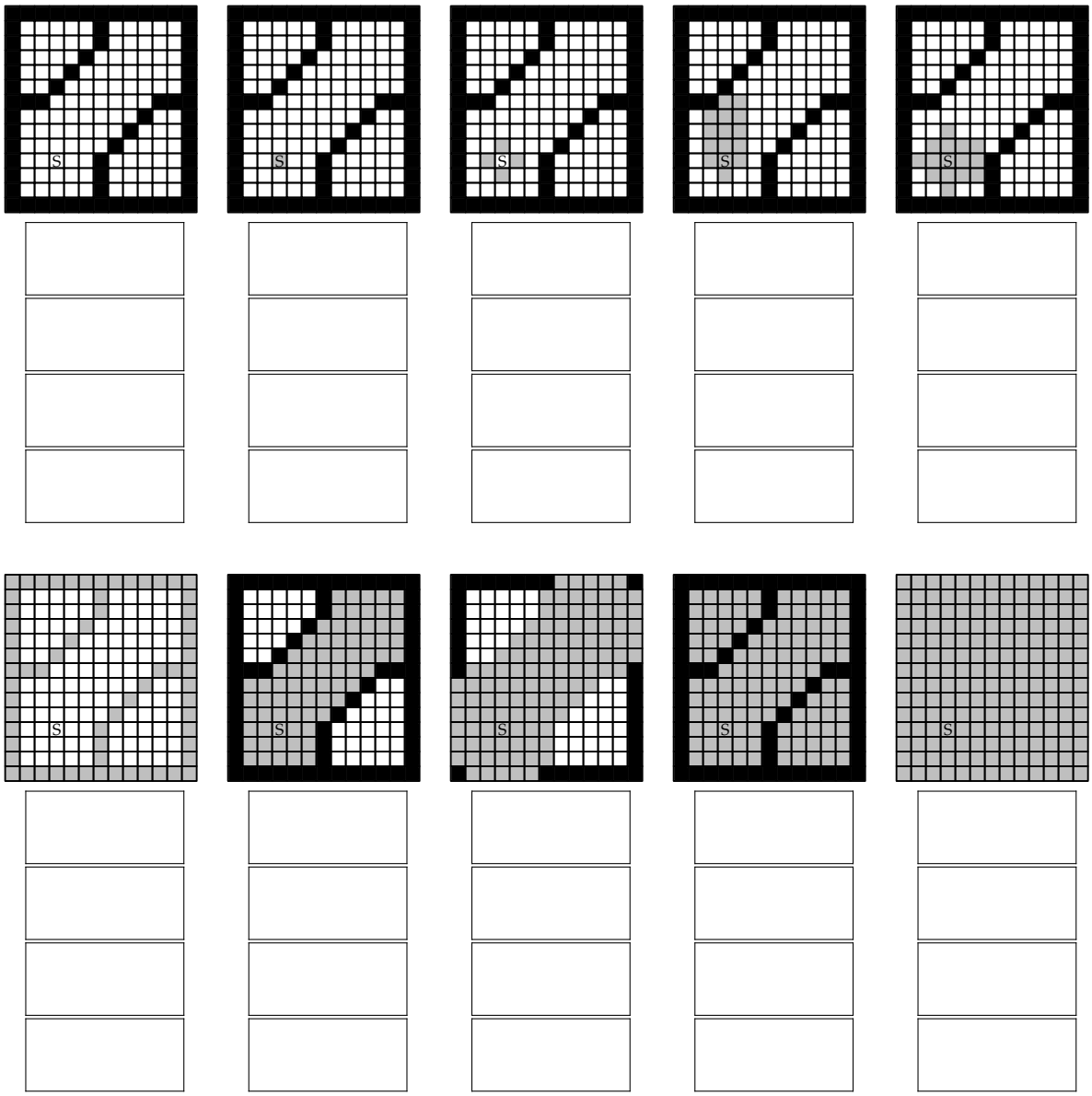
ii. DFS with dynamic programming

Consider all possible combinations of the successor functions, goal test, and search. Each combination is represented by:

- a number (1-3) representing the successor function,
- a letter (a-c) representing the goal test function, and
- a roman numeral (i or ii) representing the search function.

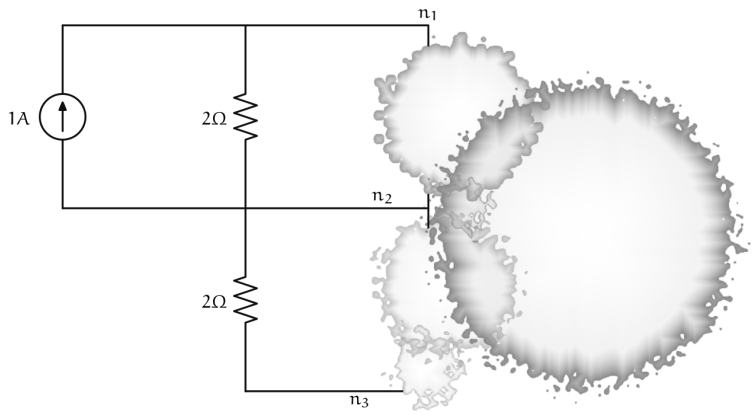
For each combination, enter all three values above in *one of the boxes* below the image that would result from running the search with that combination of inputs to completion, starting from the location labeled "S". For example, if an image would result from running with successor function 1, goal test a, and search i, enter `(1, a, i)` in one of the boxes below that image.

If there are more boxes than you need, leave the remaining boxes blank. If there are too few boxes, enter any subset of the valid answers in the boxes.



7 Coffee Break (18 Points)

Ben Bitdiddle was working on a design for a new circuit when he spilled coffee on his desk, destroying part of his schematic! Here is what he was able to recover:



Before the spill, he noted the following measurements:

- With nothing else connected to the circuit, the potential at n_1 is 12 Volts higher than the potential at n_2.
- With an additional 2Ω resistor connected to n_1 and n_2, the potential measured at n_1 is 7.5 Volts higher than the potential at n_2.
- With nothing else connected to the circuit, the potential at n_2 is 4 Volts higher than the potential at n_3.
- With nothing else connected to the circuit, the resistance measured between n_2 and n_3 is 1Ω.

In the box on the facing page, sketch one possible circuit that might have been obscured by the coffee stain.

Draw the missing part of Ben's circuit in the box below, using only constant-valued resistors, current sources, voltage sources, and/or op-amps. Make sure to label n_1, n_2, and n_3, as well as the values of all resistors and sources in your circuit, including units.

8 Say "Cheese!" (16 Points)

The workers at a cheese shop throw completed wheels of cheese into a barrel that holds 4 cheese wheels, and customers come and buy cheese wheels from the barrel. The shop is still a small-time operation, so they only have two workers.

At the beginning of each hour (at x:01), each worker makes a new cheese wheel and adds it to the barrel with a probability that depends on the number n of cheese wheels already in the barrel:

$$\Pr(\text{make a new cheese wheel}) = \begin{cases} 1/2 & \text{if } n = 0 \\ 1/2 & \text{if } n = 1 \\ 1/4 & \text{if } n = 2 \\ 0 & \text{otherwise} \end{cases}$$

You can assume that each hour, the two workers operate based on the same value of n (measured at x:00). You may also assume that, like all good workers, they never take breaks, they never sleep, they will live forever, and they will never retire.

At the end of each hour (at x:59), each cheese wheel in the barrel is sold (and thus is removed from the barrel) with probability $1/2$, independently of whether the other wheels sell.

For each of the questions below, please enter **a single fraction** in the box, representing the probability in question.

1. At 10:30am on Monday, your belief over the number of cheese wheels in the barrel is given by:

```
DDist({1: 1/3, 3: 2/3})
```

What is the probability that there is exactly one cheese wheel left in the barrel at 11:00am?

2. At 4:30pm on Tuesday, you are told that there are exactly 2 wheels of cheese in the barrel. What is the probability that neither worker will make a wheel of cheese at 5:01pm?

3. At 1:30pm on Wednesday, you have no idea how many cheese wheels are in the barrel (it is equally likely that there are $0, 1, 2, 3, 4$ cheese wheels). You then see that exactly 3 cheese wheels sell at 1:59pm. What is the probability that there is exactly 1 cheese wheel left in the barrel?

4. At 3:30pm on Thursday, you know that there are exactly 2 cheese wheels in the barrel. At 4:30pm, there are exactly 3 cheese wheels in the barrel. What is the probability that both workers made a cheese wheel at 4:01pm?

5. At 12:30pm on Friday, you look and see that there is exactly 1 cheese wheel in the barrel. Coming back at 2:30pm (2 hours later), you are told that no cheese sold in the last two hours, and you notice that there are exactly 3 cheese wheels in the barrel. What is the probability that the same worker made both of the new wheels of cheese?

Worksheet (intentionally blank)

Worksheet (intentionally blank)

Worksheet (intentionally blank)

Worksheet (intentionally blank)

Worksheet (intentionally blank)

Worksheet (intentionally blank)

Worksheet (intentionally blank)

Source Code for `lib601`'s Graph Search Functions

`search` (BFS/DFS)

```
def search(successors, start_state, goal_test, dfs = False):
    if goal_test(start_state):
        return [start_state]
    else:
        agenda = [SearchNode(start_state, None)]
        visited = {start_state}
        while len(agenda) > 0:
            if dfs:
                parent = agenda.pop(-1)
            else:
                parent = agenda.pop(0)
            for child_state in successors(parent.state):
                child = SearchNode(child_state, parent)
                if goal_test(child_state):
                    return child.path()
                if child_state not in visited:
                    agenda.append(child)
                    visited.add(child_state)
        return None
```

`uc_search` (UC/A*)

```
def uc_search(successors, start_state, goal_test, heuristic=lambda s: 0):
    if goal_test(start_state):
        return [start_state]
    agenda = [(heuristic(start_state), SearchNode(start_state, None, cost=0))]
    expanded = set()
    while len(agenda) > 0:
        agenda.sort()
        priority, parent = agenda.pop(0)
        if parent.state not in expanded:
            expanded.add(parent.state)
            if goal_test(parent.state):
                return parent.path()
            for child_state, cost in successors(parent.state):
                child = SearchNode(child_state, parent, parent.cost+cost)
                if child_state not in expanded:
                    agenda.append((child.cost+heuristic(child_state),child))
    return None
```

`SearchNode` class

```
class SearchNode:
    def __init__(self, state, parent, cost = 0.):
        self.state = state
        self.parent = parent
        self.cost = cost

    def path(self):
        p = []
        node = self
        while node:
            p.append(node.state)
            node = node.parent
        return p[::-1]
```

Worksheet (intentionally blank)