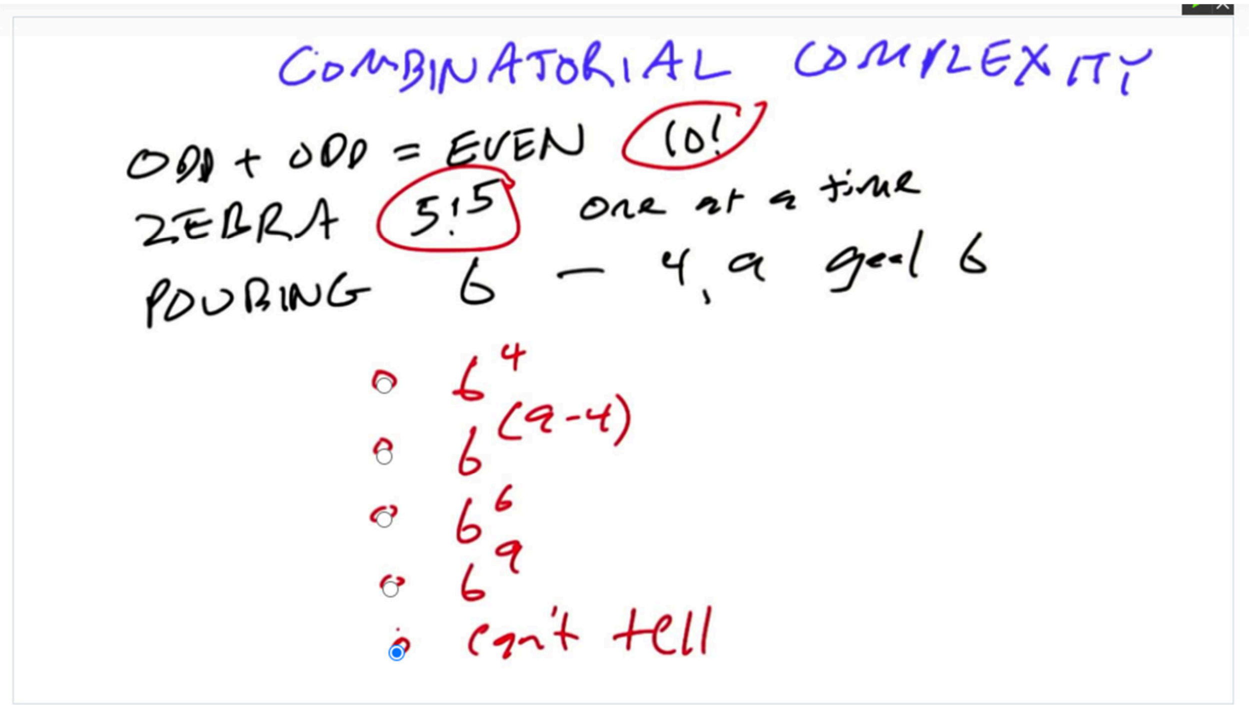
Combinatorial Complexity



COMBINATORIAL COMPLEXITY

ODD + ODD = EVEN 10!

ZEBRA 5!5 one at a time

POURING 6 — 4, 9 goal 6

○ 6^4

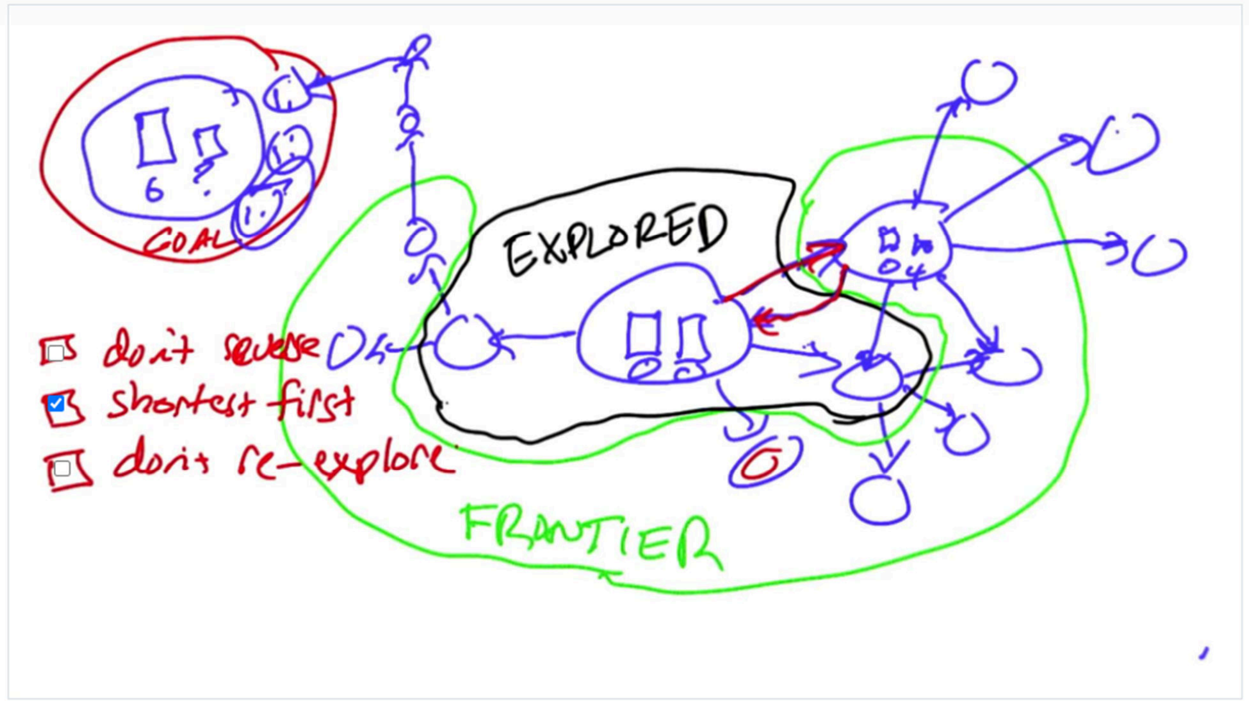
○ $6^{(9-4)}$

○ 6^6

○ 6^9

● can't tell

Exploring The Space

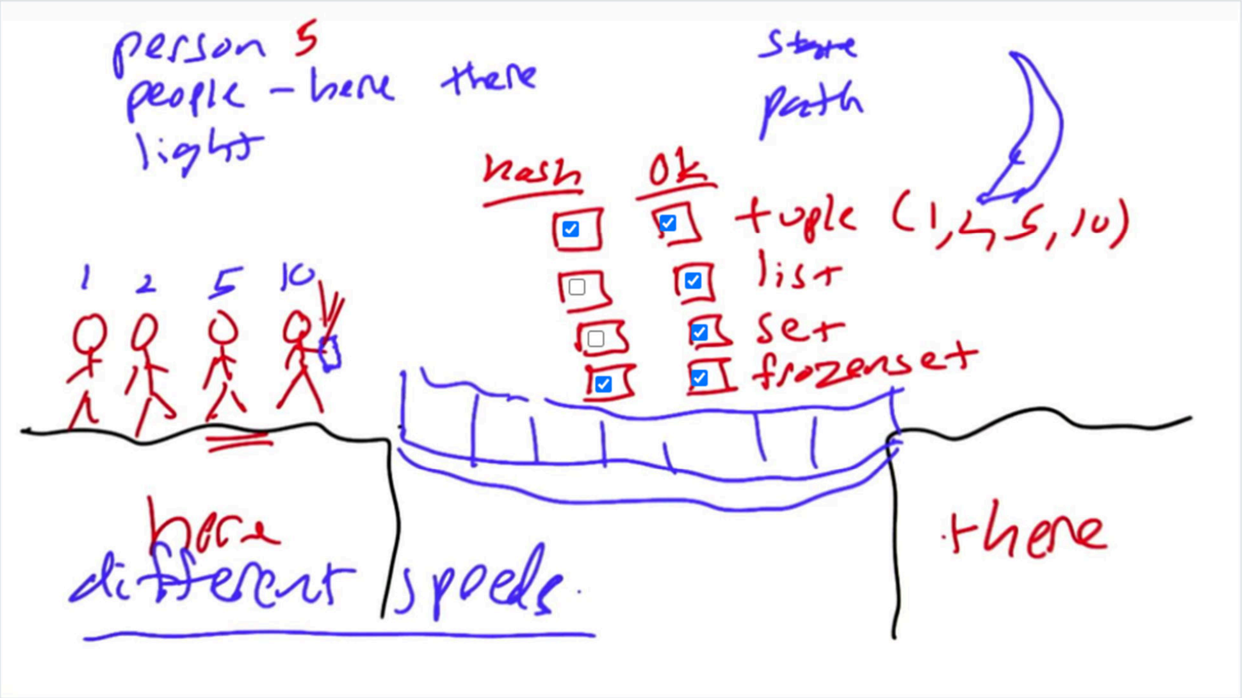


□ don't reverse

☑ shortest first

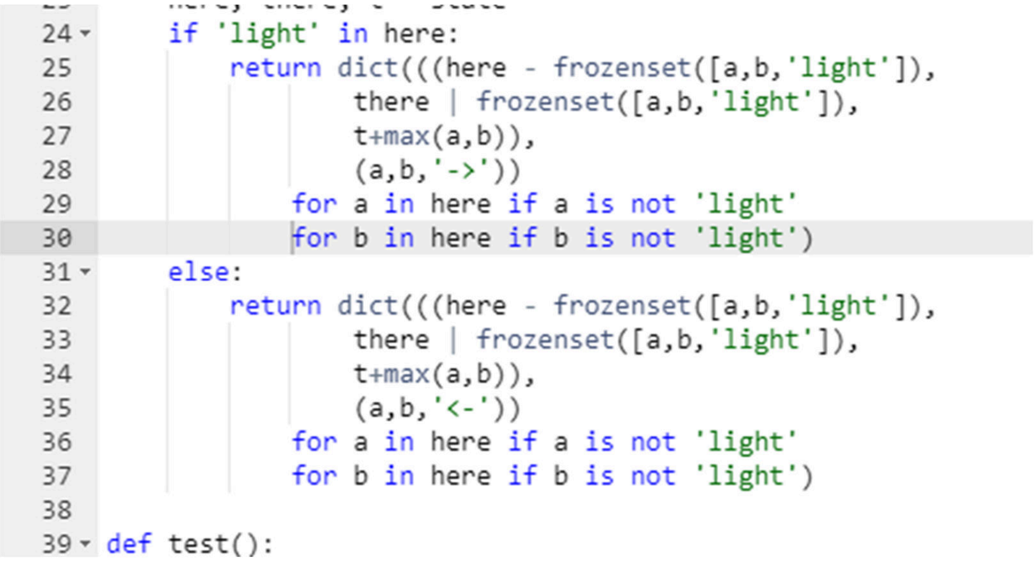
□ don't re-explore

Representing State



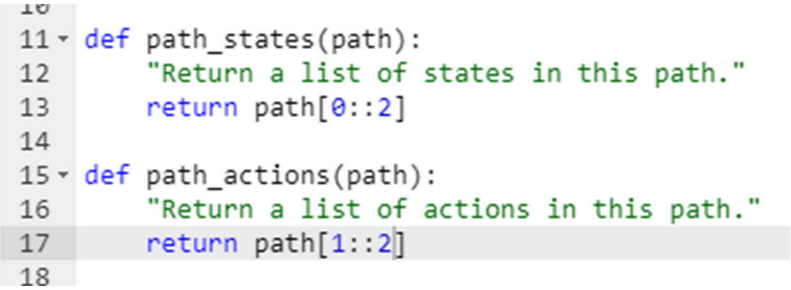
	hash	ok
tuple (1,2,5,10)	☑	☑
list	☐	☑
set	☐	☑
frozenset	☑	☑

Bridge Successors



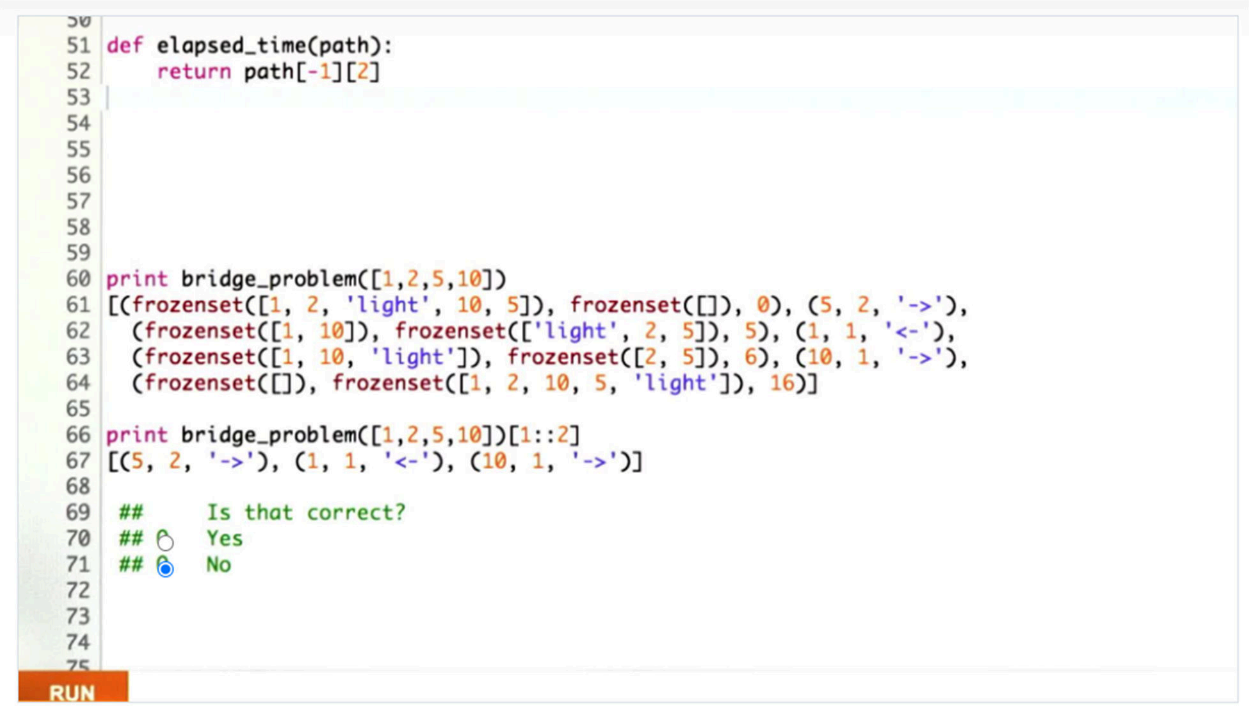
```
    here, there, t = state
    if 'light' in here:
        return dict(((here - frozenset([a,b,'light']),
                whenthere | frozenset([a,b,'light']),
                t+max(a,b)),
                (a,b,'->'))
            for a in here if a is not 'light'
            for b in here if b is not 'light')
    else:
        return dict(((here - frozenset([a,b,'light']),
                there | frozenset([a,b,'light']),
                t+max(a,b)),
                (a,b,'<-'))
            for a in here if a is not 'light'
            for b in here if b is not 'light')

def test():
```

Paths Actions States



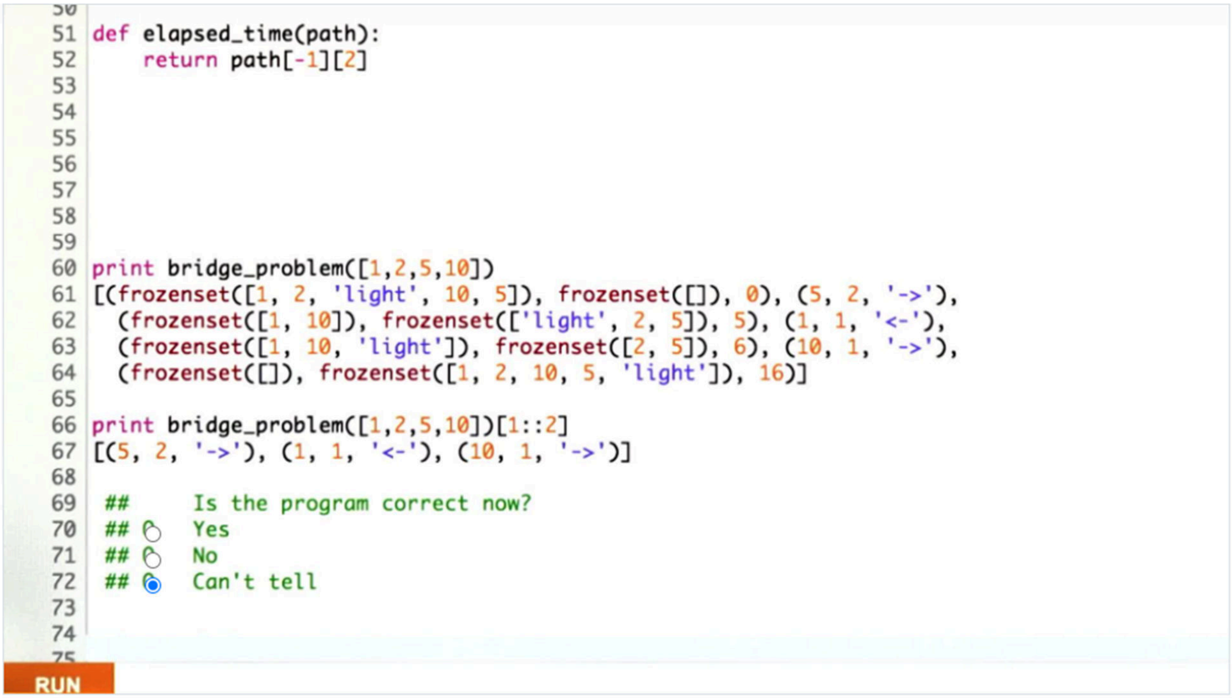
```
def path_states(path):
    "Return a list of states in this path."
    return path[0::2]

def path_actions(path):
    "Return a list of actions in this path."
    return path[1::2]

```

Bridge solution



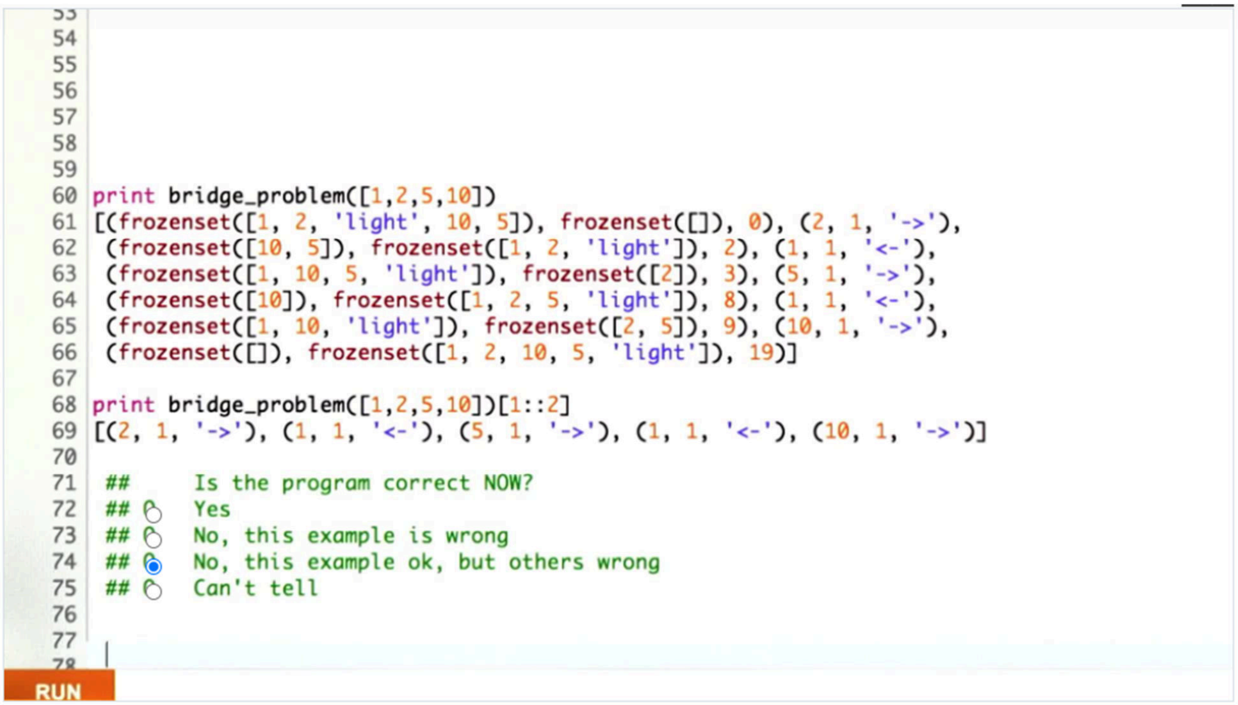
```
def elapsed_time(path):
    return path[-1][2]







print bridge_problem([1,2,5,10])
[(frozenset([1, 2, 'light', 10, 5]), frozenset([]), 0), (5, 2, '->'),
  (frozenset([1, 10]), frozenset(['light', 2, 5]), 5), (1, 1, '<-'),
  (frozenset([1, 10, 'light']), frozenset([2, 5]), 6), (10, 1, '->'),
  (frozenset([]), frozenset([1, 2, 10, 5, 'light']), 16)]

print bridge_problem([1,2,5,10])[1::2]
[(5, 2, '->'), (1, 1, '<-'), (10, 1, '->')]

 ##      Is that correct?
 ## ○    Yes
 ## ●    No
```

Debugging



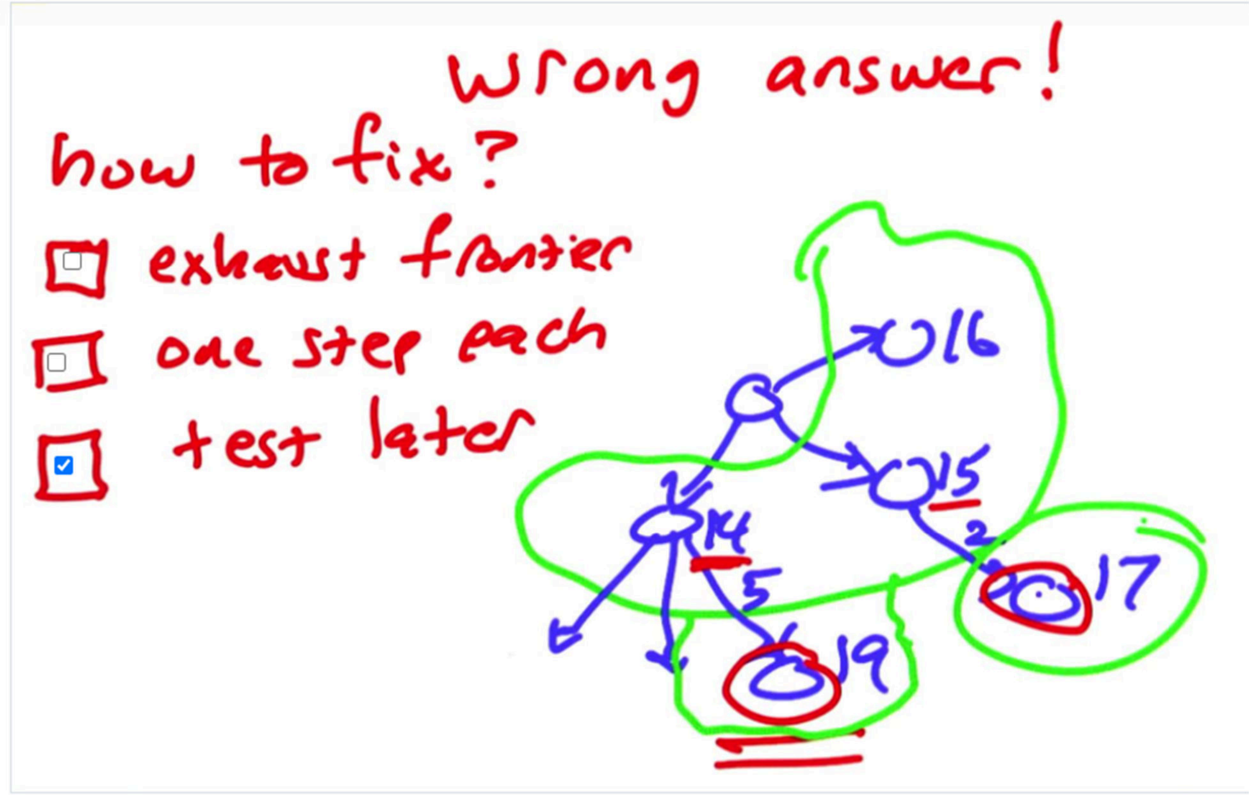
```
def elapsed_time(path):
    return path[-1][2]







print bridge_problem([1,2,5,10])
[(frozenset([1, 2, 'light', 10, 5]), frozenset([]), 0), (5, 2, '->'),
  (frozenset([1, 10]), frozenset(['light', 2, 5]), 5), (1, 1, '<-'),
  (frozenset([1, 10, 'light']), frozenset([2, 5]), 6), (10, 1, '->'),
  (frozenset([]), frozenset([1, 2, 10, 5, 'light']), 16)]

print bridge_problem([1,2,5,10])[1::2]
[(5, 2, '->'), (1, 1, '<-'), (10, 1, '->')]

 ##     Is the program correct now?
 ## ○  Yes
 ## ○  No
 ## ●  Can't tell
```

Did It Work

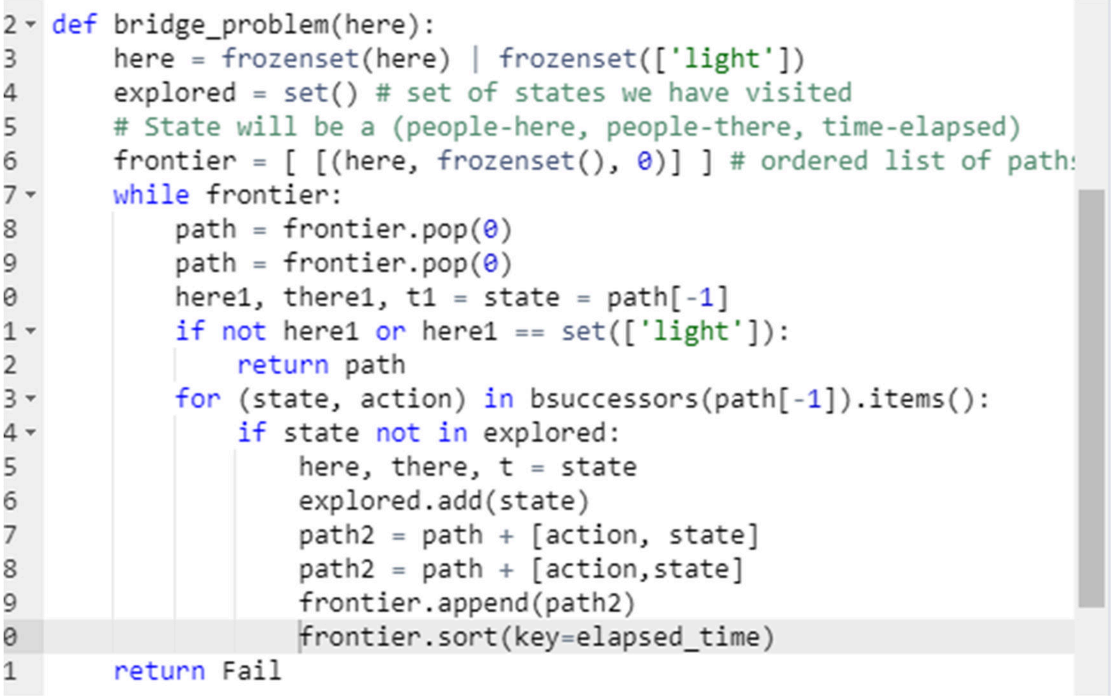


```






print bridge_problem([1,2,5,10])
[(frozenset([1, 2, 'light', 10, 5]), frozenset([]), 0), (2, 1, '->'),
 (frozenset([10, 5]), frozenset([1, 2, 'light']), 2), (1, 1, '<-'),
 (frozenset([1, 10, 5, 'light']), frozenset([2]), 3), (5, 1, '->'),
 (frozenset([10]), frozenset([1, 2, 5, 'light']), 8), (1, 1, '<-'),
 (frozenset([1, 10, 'light']), frozenset([2, 5]), 9), (10, 1, '->'),
 (frozenset([]), frozenset([1, 2, 10, 5, 'light']), 19)]

print bridge_problem([1,2,5,10])[1::2]
[(2, 1, '->'), (1, 1, '<-'), (5, 1, '->'), (1, 1, '<-'), (10, 1, '->')]

 ##     Is the program correct NOW?
 ## ○  Yes
 ## ○  No, this example is wrong
 ## ●  No, this example ok, but others wrong
 ## ○  Can't tell
```

Improving The Solution

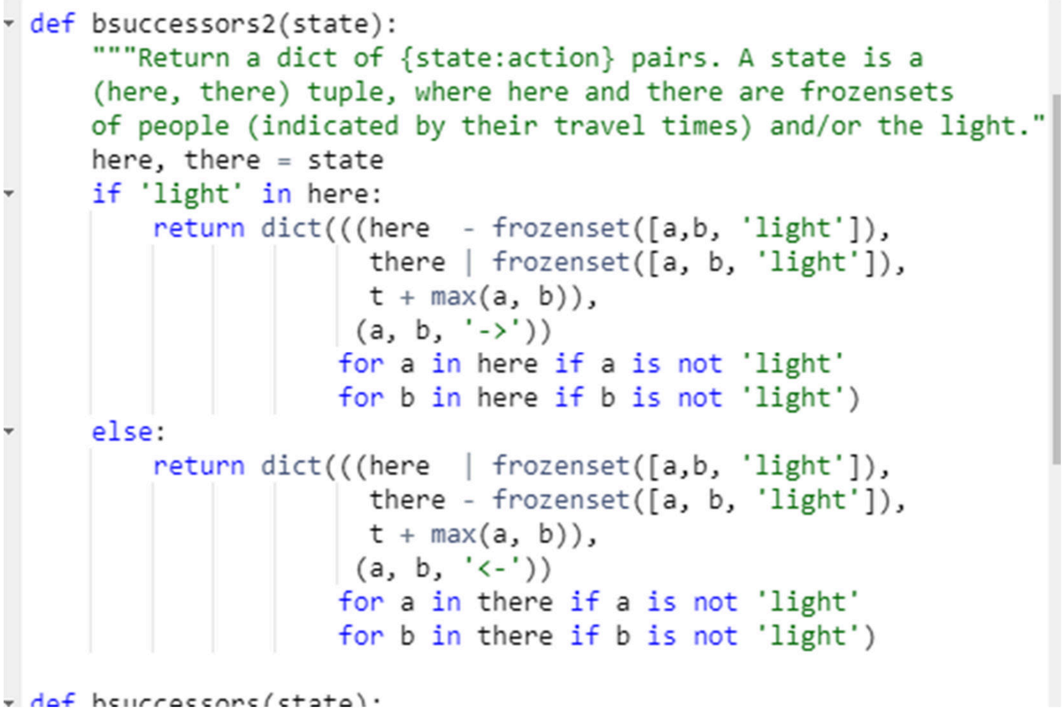


Modify Code



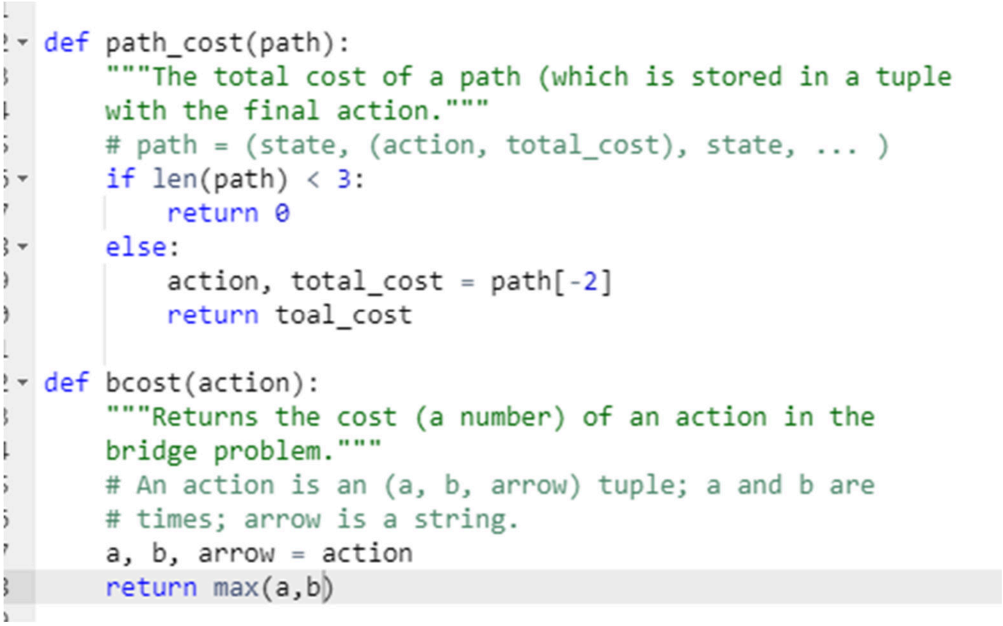
```
def bridge_problem(here):
    here = frozenset(here) | frozenset(['light'])
    explored = set() # set of states we have visited
    # State will be a (people-here, people-there, time-elapsed)
    frontier = [ [(here, frozenset(), 0)] ] # ordered list of paths
    while frontier:
        path = frontier.pop(0)
        path = frontier.pop(0)
        here1, there1, t1 = state = path[-1]
        if not here1 or here1 == set(['light']):
            return path
        for (state, action) in bsuccessors(path[-1]).items():
            if state not in explored:
                here, there, t = state
                explored.add(state)
                path2 = path + [action, state]
                path2 = path + [action,state]
                frontier.append(path2)
                frontier.sort(key=elapsed_time)
    return Fail
```

Refactoring Paths



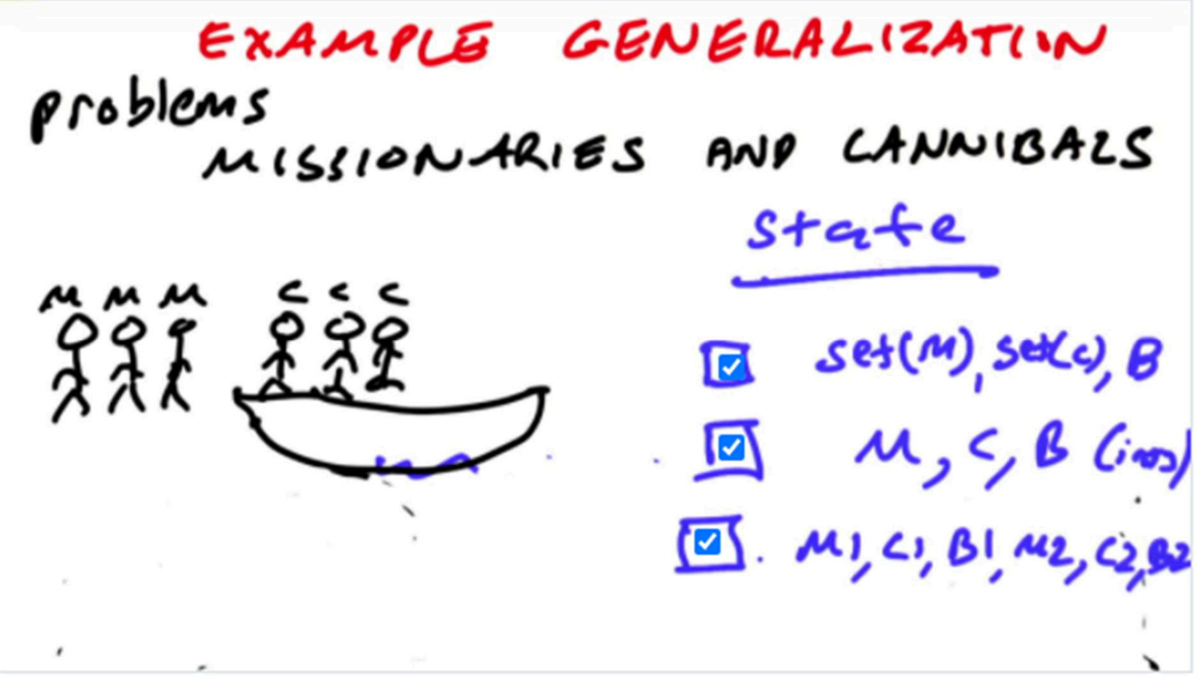
```
def bsuccessors2(state):
    """Return a dict of {state:action} pairs. A state is a
    (here, there) tuple, where here and there are frozensets
    of people (indicated by their travel times) and/or the light."
    here, there = state
    if 'light' in here:
        return dict(((here  - frozenset([a,b, 'light']),
                      there | frozenset([a, b, 'light']),
                      t + max(a, b)),
                     (a, b, '->'))
                    for a in here if a is not 'light'
                    for b in here if b is not 'light')
    else:
        return dict(((here  | frozenset([a,b, 'light']),
                      there - frozenset([a, b, 'light']),
                      t + max(a, b)),
                     (a, b, '<-'))
                    for a in there if a is not 'light'
                    for b in there if b is not 'light')

def bsuccessors(state):
```

Calculating Costs



```
def path_cost(path):
    """The total cost of a path (which is stored in a tuple
    with the final action."""
    # path = (state, (action, total_cost), state, ... )
    if len(path) < 3:
        return 0
    else:
        action, total_cost = path[-2]
        return toal_cost

def bcost(action):
    """Returns the cost (a number) of an action in the
    bridge problem."""
    # An action is an (a, b, arrow) tuple; a and b are
    # times; arrow is a string.
    a, b, arrow = action
    return max(a,b)
```

Missionaries And Cannibals



EXAMPLE GENERALIZATION

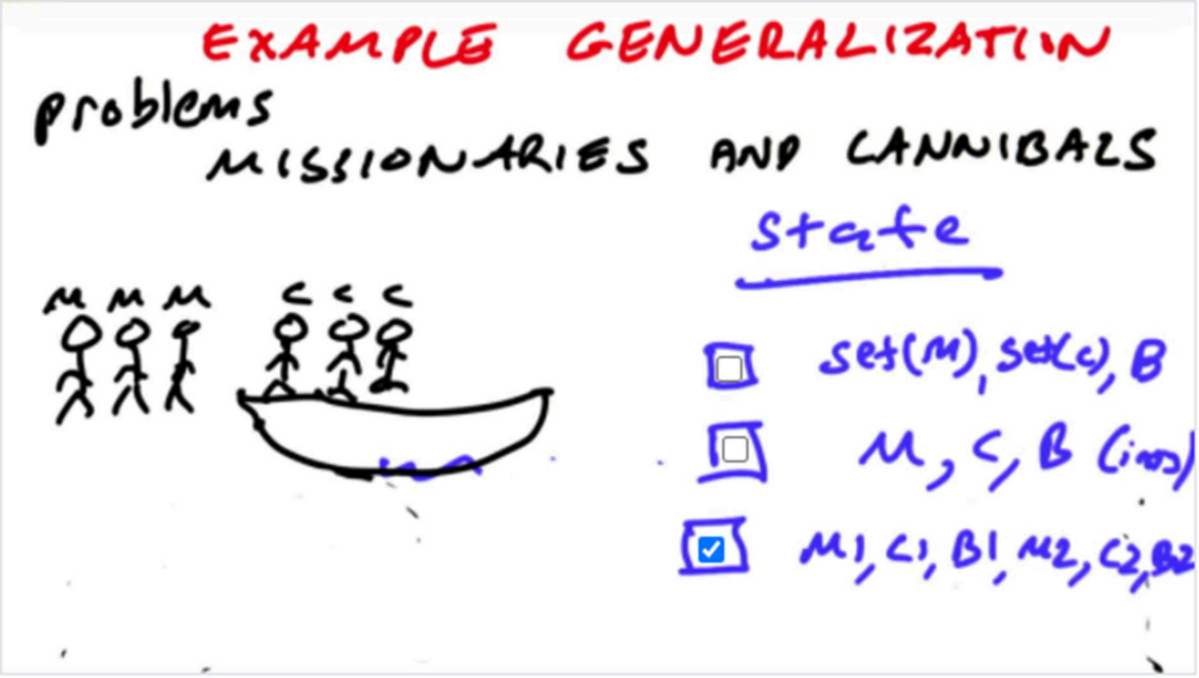
problems

MISSIONARIES AND CANNIBALS

State

- ☑ set(M), set(C), B
- ☑ M, C, B (ints)
- ☑ M1, C1, B1, M2, C2, B2

Generalized State



EXAMPLE GENERALIZATION

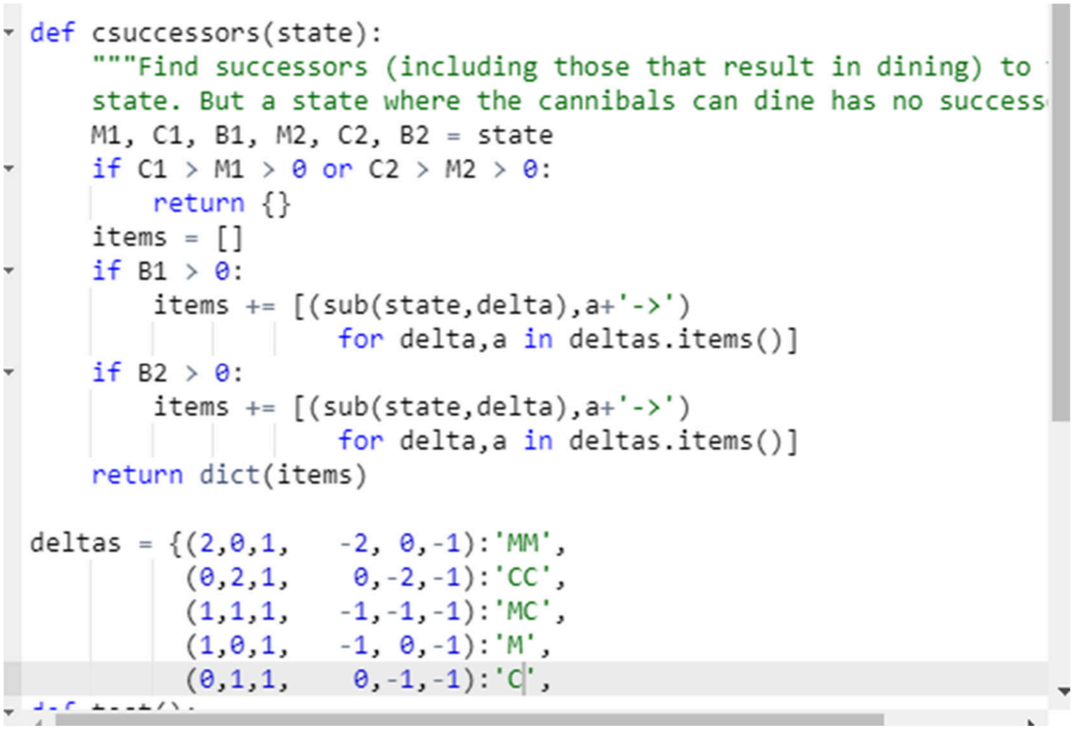
problems

MISSIONARIES AND CANNIBALS

State

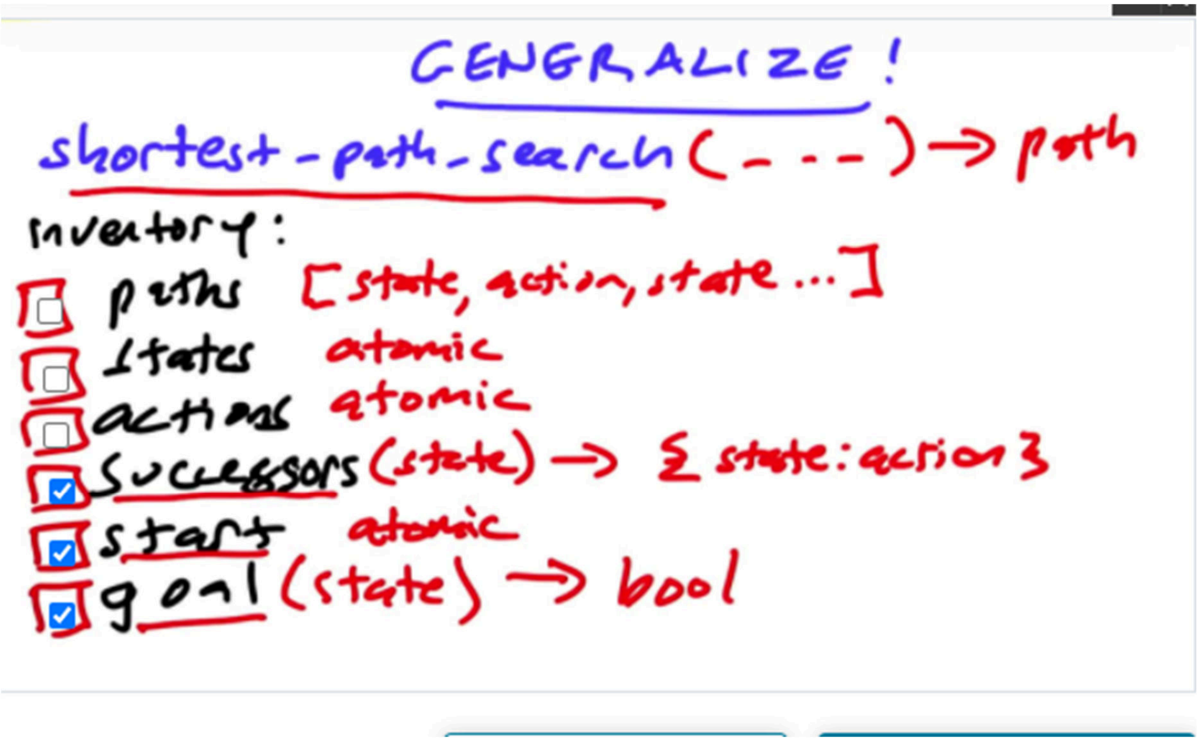
- □ set(M), set(C), B
- □ M, C, B (ints)
- ☑ M1, C1, B1, M2, C2, B2

Csuccessors



```
def csuccessors(state):
    """Find successors (including those that result in dining) to
    state. But a state where the cannibals can dine has no success
    M1, C1, B1, M2, C2, B2 = state
    if C1 > M1 > 0 or C2 > M2 > 0:
        return {}
    items = []
    if B1 > 0:
        items += [(sub(state,delta),a+'->')
                  for delta,a in deltas.items()]
    if B2 > 0:
        items += [(sub(state,delta),a+'->')
                  for delta,a in deltas.items()]
    return dict(items)

deltas = {(2,0,1,   -2, 0,-1):'MM',
          (0,2,1,    0,-2,-1):'CC',
          (1,1,1,   -1,-1,-1):'MC',
          (1,0,1,   -1, 0,-1):'M',
          (0,1,1,    0,-1,-1):'C',
```

Shortest Path Search



GENERALIZE !

shortest-path-search(- - -) -> path

inventory:

☐ paths [state, action, state ...]

☐ states atomic

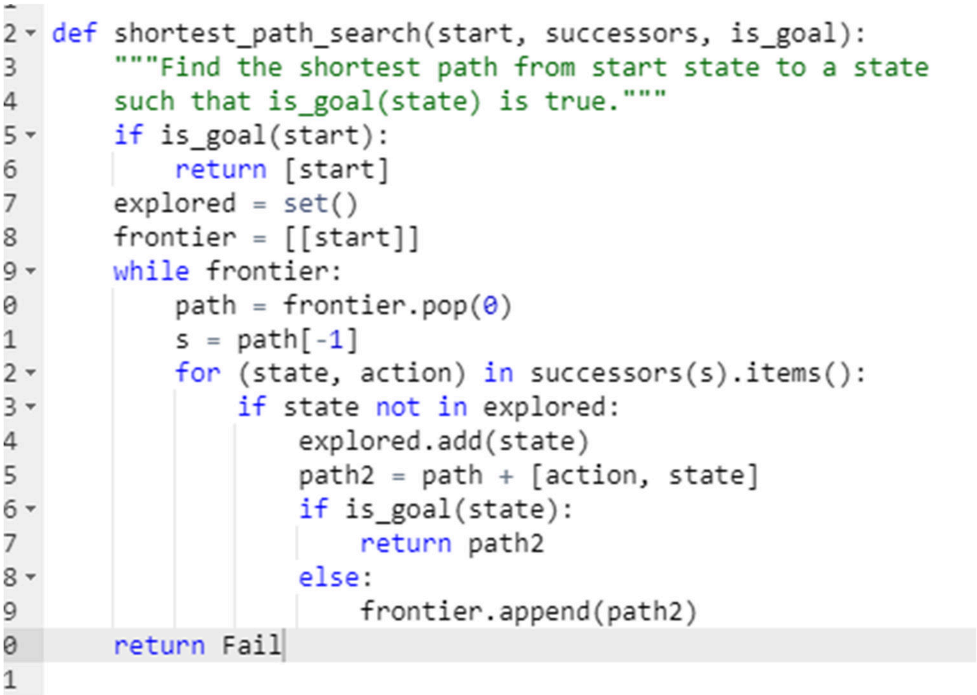
☐ actions atomic

☑ successors(state) -> {state:action}

☑ start atomic

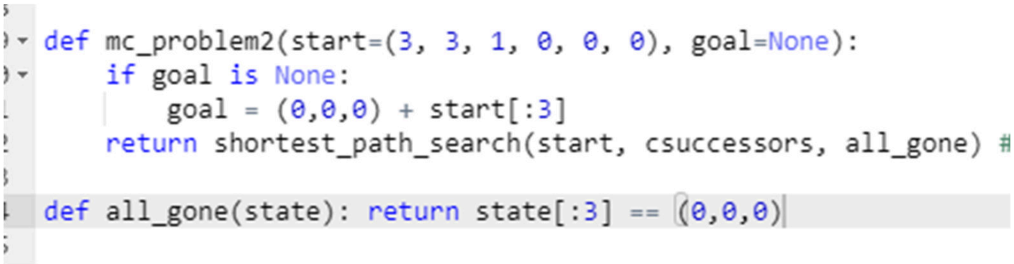
☑ goal(state) -> bool

Sps Function



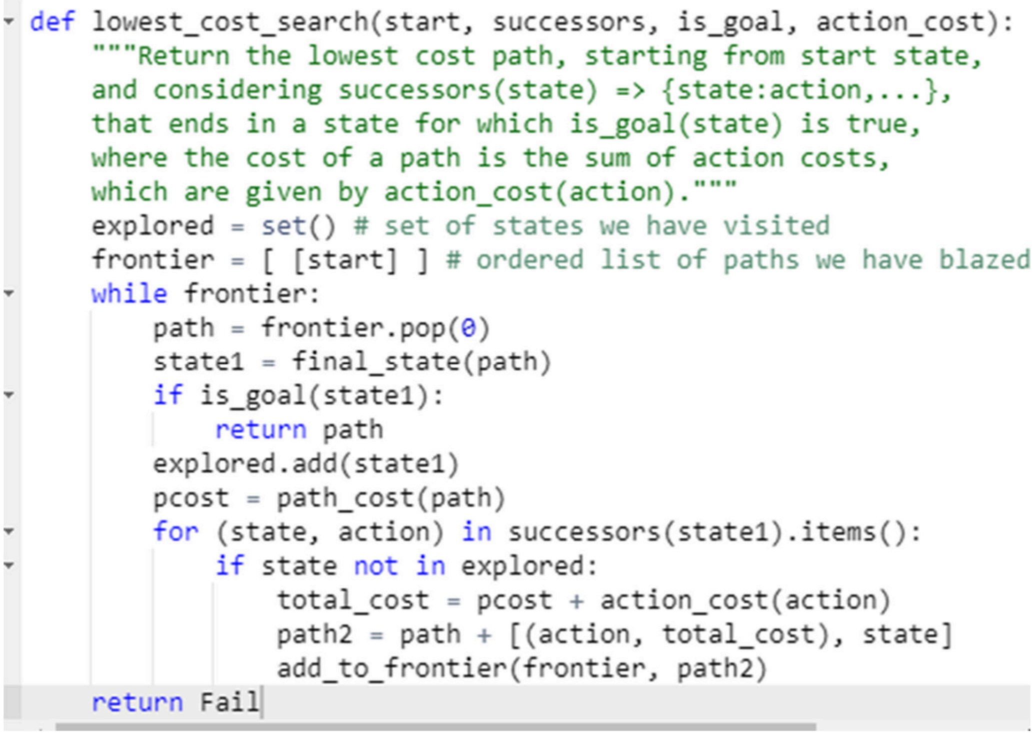
```python
def shortest_path_search(start, successors, is_goal):
    """Find the shortest path from start state to a state
    such that is_goal(state) is true."""
    if is_goal(start):
        return [start]
    explored = set()
    frontier = [[start]]
    while frontier:
        path = frontier.pop(0)
        s = path[-1]
        for (state, action) in successors(s).items():
            if state not in explored:
                explored.add(state)
                path2 = path + [action, state]
                if is_goal(state):
                    return path2
                else:
                    frontier.append(path2)
    return Fail
```

Cleaning Up Mc Problem



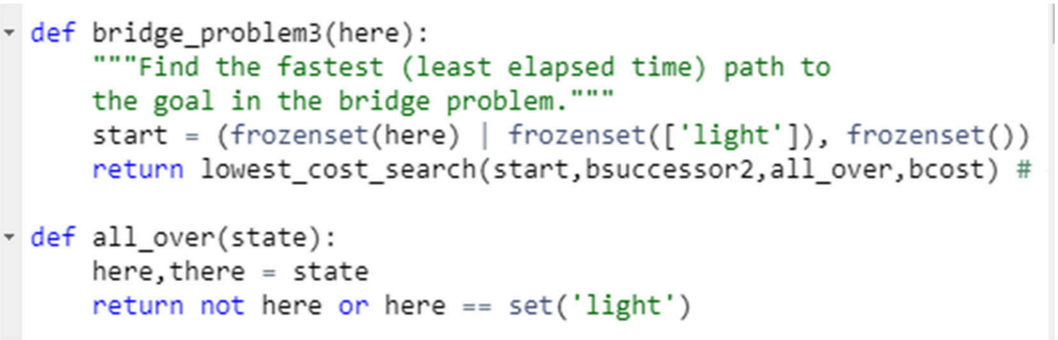
```python
def mc_problem2(start=(3, 3, 1, 0, 0, 0), goal=None):
    if goal is None:
        goal = (0,0,0) + start[:3]
    return shortest_path_search(start, csuccessors, all_gone) #

def all_gone(state): return state[:3] == (0,0,0)
```

Lowest Cost Search



```
def lowest_cost_search(start, successors, is_goal, action_cost):
    """Return the lowest cost path, starting from start state,
    and considering successors(state) => {state:action,...},
    that ends in a state for which is_goal(state) is true,
    where the cost of a path is the sum of action costs,
    which are given by action_cost(action)."""
    explored = set() # set of states we have visited
    frontier = [ [start] ] # ordered list of paths we have blazed
    while frontier:
        path = frontier.pop(0)
        state1 = final_state(path)
        if is_goal(state1):
            return path
        explored.add(state1)
        pcost = path_cost(path)
        for (state, action) in successors(state1).items():
            if state not in explored:
                total_cost = pcost + action_cost(action)
                path2 = path + [(action, total_cost), state]
                add_to_frontier(frontier, path2)
    return Fail
```

Back To Bridge Problem



```
def bridge_problem3(here):
    """Find the fastest (least elapsed time) path to
    the goal in the bridge problem."""
    start = (frozenset(here) | frozenset(['light']), frozenset())
    return lowest_cost_search(start,bsuccessor2,all_over,bcost) #

def all_over(state):
    here,there = state
    return not here or here == set('light')
```