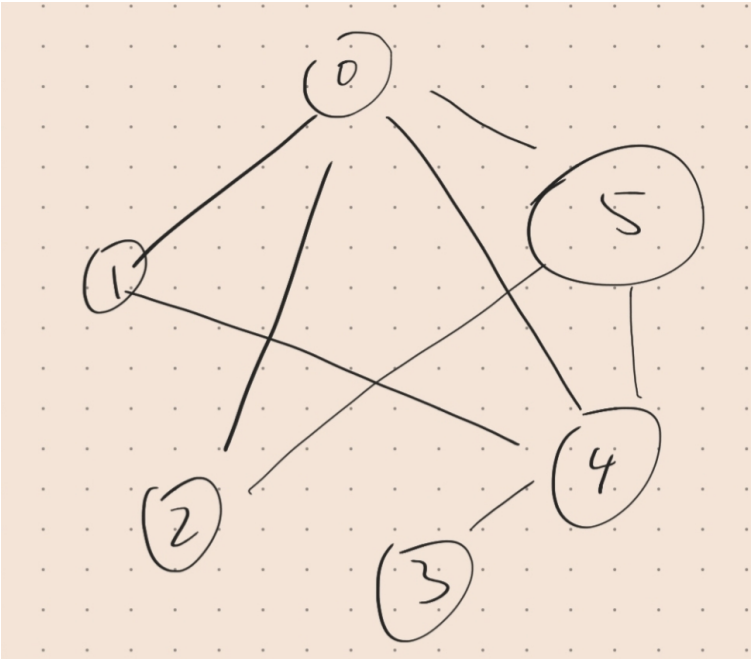
Songhao Li April 1, 2021

6.006 Problem Set

Collaborators: *Yukai Tan*

1 problem 1

(1)



(2)

$$
\begin{aligned}
G = \{&0:[1,2,4,5],\\
&1:[0,4],\\
&2:[0,5],\\
&3:[4],\\
&4:[0,1,3,5],\\
&5:[0,2,4]\}
\end{aligned}
$$

(3){0:2, 1:0, 2:None, 3:4, 4:0, 5:2}

2 Problem 2

(1) Perform a BFS on the graph, starting at the first person among the two people. If the other person among the two people has already been marked, then return True; otherwise keep BFS until finishing marking nodes with 6, return False.

(2) Use Adjacency list. Since nobody has more than 10 friends, the graph should be sparse. Each list in the adjacency list would have a length of at most 10.

3 Problem 3

In addition to a heap (priority queue), we also build a hash table that maps ID to its index in the heap

build(X): build priority queue needs O(n). After building the priority queue, scan the queue to build hash table in O(n) as well.

insert(X): An insertion might update the indices of O(logn) nodes when we heapify, and also update the hash table which requires O(1) for each update; as a result, the runtime is still bounded by O(logn)

delete_max() in priority queue needs O(logn) when we call heapify, and as before updating hash table takes O(1) at each node and the runtime is still bounded by tO(logn)

delete(id): use the hash table to find the index for this id in O(1). Use the index to locate the item with id in the priority queue and delete this item from the priority queue in O(1). Calling heapify and updating hash table takes O(logn) as before. So the runtime is O(logn)

(1) A BFS can do the job.

Scan all edges and store the directly adjacent plants of every planet; technically speaking, for any planet pi, find all pairs of hops that contains pi and store them in H_i.

Initialize a hash table *dist* to store the minimum fluid units required from earth to all planets. When initialize, set all values to MAX.

Algorithm 1 determine whether Kernal is accessible within k fluid units

```
Initialize a hash table dist, set all values to MAX.
Initialize an empty set to_visit
Initialize depth = 0
for i in H_earth do
    dist[j] = h_earth,j
    new_to_visit.add(j)
    depth += 1
end for
while to_visit is nonempty and depth <= k: do
    depth+ = 1
    Initialize set new_to_visit
    for i in to_visit do
        for j in H_i do
            dist[j] = min(dist[i] + hij, dist[j])
            new_to_visit.add(j)
        end for
    end for
    to_visit = new_to_visit
end while
return dist[Kernal] < k
```

Correctness:

It is correct since we tried all possible paths from earth to Kernal with in k trips.

Runtime:

For every depth, the number of edges we tried is smaller than m. Visit every edge takes O(1) time to access the hash table and do the operations. So the total algorithm runtime is bounded by O(km)

(2) We can still make the job done using adjusted version of BFS After consructing $H[i]$ for every planet i as before, we initialize two hash tables: *Distance_end_with_hyper* and *Distance_end_with_normal* with MAX.

Algorithm 2 determine whether Kernal is accessible within k fluid units with hyper-hop

Initialize an empty set *to_visit*
Initialize $depth = 0$
for i in H_{earth} **do**
  $Distance\_end\_with\_normal[i] = h_{earth,i}$
  $Distance\_end\_with\_hyper[i] = 1$
  $new\_to\_visit.add(j)$
  $depth \mathrel{+}= 1$
end for
while *to_visit* is nonempty and $depth <= k$: **do**
  $depth+ = 1$
  Initialize set *new_to_visit*
  for i in *to_visit* **do**
    for j in H_i **do**
      $Distance\_end\_with\_normal[j] = min(Distance\_end\_with\_normal[i] + h_{ij}, Distance\_end\_with\_hyper[i] + h_{ij}, Distance\_end\_with\_normal[j])$
      $Distance\_end\_with\_hyper[j] = min(Distance\_end\_with\_normal[i] + 1, Distance\_end\_with\_hyper[j])$
      $new\_to\_visit.add(j)$
    end for
  end for
  $to\_visit = new\_to\_visit$
end while
return $Distance\_end\_with\_normal[Kernal] < k$ or $Distance\_end\_with\_hyper[Kernal] < k$

Correctness:

As before, the correctness is proven naturally since we have tried all possible paths from earth to Kernal with in k trips.

Runtime:

We have added a constant time operation for every path at every depth, so the total runtime is still bounded by the same complexity as 4(1) which is O(km)

(1) Consider the number of possible configurations as below:

The number of grids in the array except for the last row is (R-1)C; for these cells, there are four possibilities each, with number 0, 1, 2, or 3, so the number of configurations in this part is $4^{(R-1)C}$.

The monkey can stand at one of the column, and may have life 1, 2, or 3. So the total configuration for the game is ...

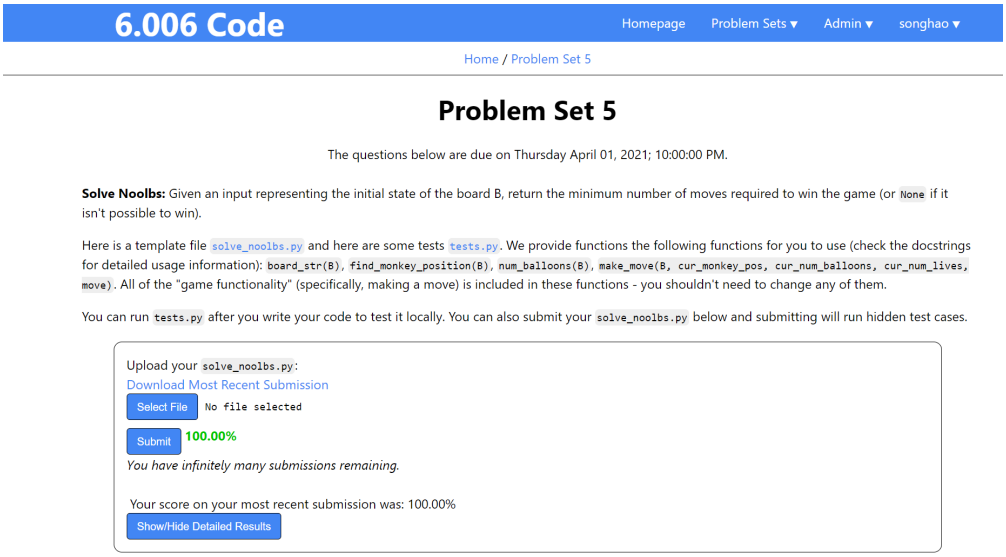
(2) We prove the argument as follows: After 3R moves, either all balloons are already cleared (all cells have value 0), or the game is already dead. If all balloons are already cleared, then the player wins the game and the moves are smaller than 3R. Consider a cell has nonzero value: since the balloon moves one cell (descent or re-spawn) every time the player makes a move, so the remaining balloon in the nonzero cell must have moved at least 3R cells. Given the number of rows = R, the balloon must have hit the ground three times at least, so the monkey has already lost 3 lives, then the player must have already lost the game. So the maximum number of moves in a game is 3R

(3) Design a directed graph. Every node is a configuration and directed edge from node 1 to node 2 exists when configuration 1 can lead to configuration 2 after one turn (while the monkey move left, right, or through a dart straight up.

Initialize the graph as with a node of the current configuration, then compute its neighbors, and neighbors of neighbors. Keep track of the "distance" of a node with the root node. When a node implies lost game, stop finding its neighbors. Find the closest neighbor that wins the game (can be easily done by a BFS), return its distance from the root or return None if no such node exists.

The number of configurations going through is bounded by 3^{3R} in 5(2) since the maximum number of moves is 3R which is the depth of the graph, and every node can have as many as 3 edges to the next level. To build a node takes O(RC) given the side of the board B. As a result the total runtime of this algorithm is $O(RC * 3^{3R})$. Also, to further decrease the average runtime, we can keep track of nodes visited and end finding adjacent nodes of a node that represented a lost game.

The correctness is proven by the nature of this algorithm: we explore all the possible decisions that the monkey can make. Since we can find all ways to win the game, we can surely find the minimum number of moves needed, or return None when no such way exists.



6.006 Code

Homepage Problem Sets ▾ Admin ▾ songhao ▾

Home / Problem Set 5

Problem Set 5

The questions below are due on Thursday April 01, 2021; 10:00:00 PM.

Solve Noolbs: Given an input representing the initial state of the board B, return the minimum number of moves required to win the game (or `None` if it isn't possible to win).

Here is a template file `solve_noolbs.py` and here are some tests `tests.py`. We provide functions the following functions for you to use (check the docstrings for detailed usage information): `board_str(B)`, `find_monkey_position(B)`, `num_balloons(B)`, `make_move(B, cur_monkey_pos, cur_num_balloons, cur_num_lives, move)`. All of the "game functionality" (specifically, making a move) is included in these functions - you shouldn't need to change any of them.

You can run `tests.py` after you write your code to test it locally. You can also submit your `solve_noolbs.py` below and submitting will run hidden test cases.

Upload your `solve_noolbs.py`:

Download Most Recent Submission

Select File No file selected

Submit 100.00%

You have infinitely many submissions remaining.

Your score on your most recent submission was: 100.00%

Show/Hide Detailed Results