Lab 08

Friday, 25 October 2019

DFS vs BFS

- They are both graph traversal algorithms, can be modified to do other graph – related tasks:
- Similarities:
  - Both visit each vertex only once. (so you need to track visited states)
  - Both run in O(V+E).

BFS vs DFS Implementation:

- DFS: Function with recursive call to the unvisited neighbours.
- BFS: Queue, push unvisited neighbours to the back of the queue.

```
void dfs(int vertex) {
    visited[vertex] = true;
    for (auto it: AL[vertex]) {
        if (!visited[it])
            dfs(it);
    }
}
```

```
queue.push(start);
visited[start] = true;

while (!queue.empty()) {
    int v = queue.front(); queue.pop();
    for (auto it: AL[v]) {
        if (!visited[it]) {
            visited[it] = true;
            queue.push(it);
        }
    }
}
```

When can you use either **DFS** or **BFS**

- When you just need to explore all the nodes, and the order in which you explore does not matter.
- Questions like
  - Counting connected components
  - Is it *possible* to go from A to B?
  - *Count* how many points you can visit starting from A.
  - Flood-fill

When to use BFS

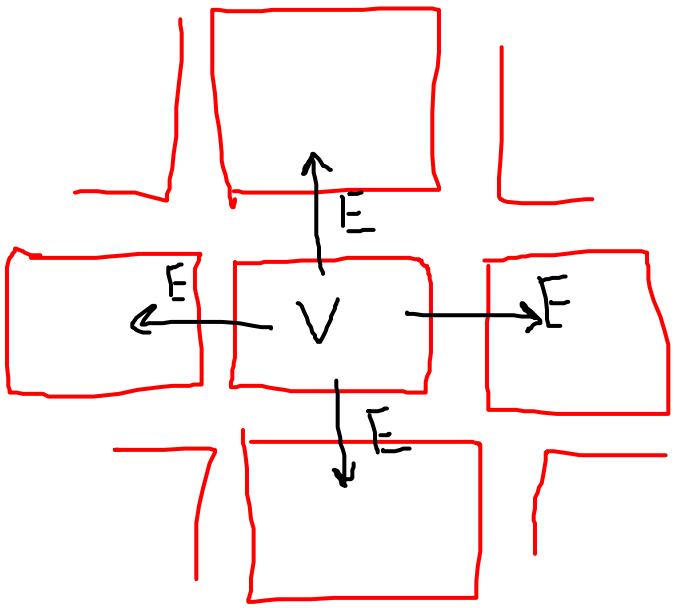
- When the order in which you visit nodes matters. BFS can be used to track number of steps from the origin.
  - Modify the BFS queue to hold a pair: {vertex, distance}
  - If the source vertex has distance *d*, Insert its neighbours with distance *d+1*.
- Examples
  - Find the *least number of jumps* from A to B
  - Flood fill *with limited distance*
  - Distance in a 2D grid.

When to use DFS

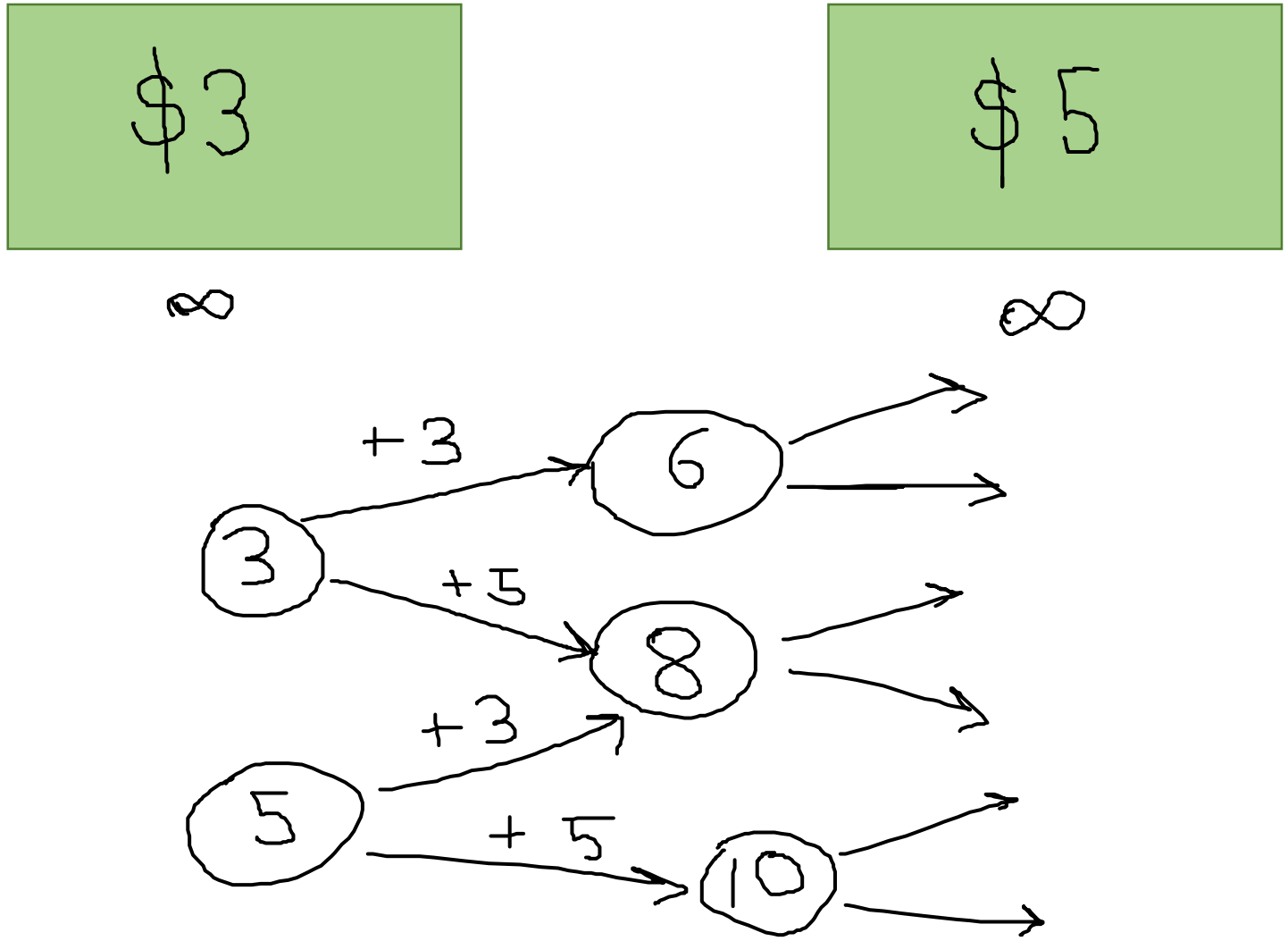
- Cycle/Back edge detection.

- (Not really graph related)
  BST traversal (inorder/preorder/postorder)

- Explicit Graph: When the question phrased in terms of vertices & edges (roads & intersections, airports & flights).
- Implicit Graph: Actually a graph question, but not as obvious:
  - 2D grid: Each spot on a grid (x,y) is a vertex, the 4 directions are edges.



- Less obvious implicit graph:



Hands-on (PE simulation):

- https://nus.kattis.com/problems/daceydice
- You have about 30 minutes to try coding an AC solution
- Lab TA will give gradual hints per 5m interval and live code that hint
- Full AC solution **will not** be given,
  the last hint will be something that is “near AC”
- If you get AC before the last hint is given, Lab TA will recognize you and will count that as a factor to decide the “lab participation points”
- Albeit not graded, you are encouraged to continue working until you get AC after all these hints are poured…

Dacey the Dice (problem summary, after 5m)

Constraints:

$1 \leq N \leq 20$

Multiple TC!

$1 \leq T \leq 100$

Abridged Problem Statement:

Given a grid with size N x N. There are 2 types of cell:

- Empty : Can move into this cell
- Wicked gigantic magnet : Not allowed to move into this cell

Determine whether a dice can move from (S_x, S_y) to (T_x, T_y) with number 5 facing bottom at the finish point.

At a unit of time, you can move in one of the four directions (L, R, U, and D).

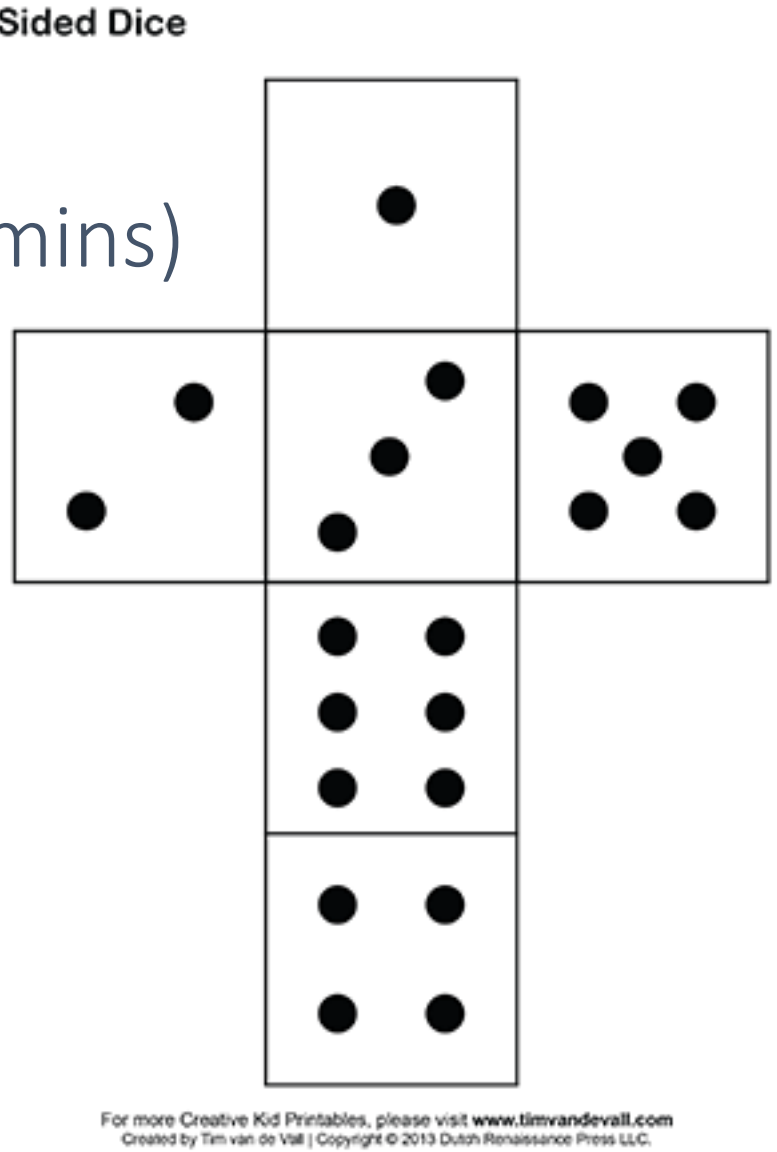
Initial orientation of the dice:

- top : 1 ; bottom : 6
- right : 2 ; left : 5
- up : 4 ; down : 3

Note: every move will rotate the dice 90 degrees.

Output: "yes" or "no".

Dacey the Dice (major hint: after 10 mins)



Key Observation

Sum of values from opposing sides is always 7

At most, we just need to remember 3 values:
top, right, and up

However, actually if we know only 2 of those 3,
we can also infer the other one!

Dacey the Dice (after 15 mins)

Then if we know the value for one of each opposing sides,
we will be able to compute the other :)

the number of possible state is very small
N * N * 6 * 4

Just do simple DFS to mark the states that can be visited from the initial position!

Dacey the Dice (the super big hint, after 20 mins)

Can you get AC in last 20m with this super big hint?

Pseudo-code *(implementation with only 4 parameters is left as a challenge)*

```
void dfs(int row, int col, int top, int right, int up) {
        if (outside_boundary(row, col) || visited[row][col][top][right][up]) return;
        visited[row][col][top][right][up] = true;

        ... // handle if this is the target position
```

roll up

7 - up = back which becomes the top

top becomes up

```
        dfs(row - 1, col       , 7 - up        , right       , top    );
        dfs(row + 1, col       , up            , right       , 7 - top);
        dfs(row    , col + 1, 7 - right, top    , up       );
        dfs(row    , col - 1, right      , 7 - top, up     );
}
```

Dacey the Dice (summary, at the ed)

Time Complexity

N^2	Visit all cells
6	Possible dice orientation
$O(N^2 * 6)$	**Total**

Memory Space

N^2	Grid size
6	Orientation
$O(N^2 * 6^2)$	**Total (both AC)**