CS/ECE 374 A ✧ Fall 2023

Homework 8

Due Tuesday, October 24, 2023 at 9pm Central Time

---

1. A *six-sided die* (plural *dice*) is a cube with each side marked with a different number of dots (called *pips*) from 1 to 6. On a *standard* die, numbers on opposite sides always add up to 7.

   A *rolling die maze* is a puzzle involving a standard six-sided die and a grid of squares. You should imagine the grid lying on a table; the die always rests on and exactly covers one square of the grid. In a single step, you can *roll* the die 90 degrees around one of its bottom edges, moving it to an adjacent square one step north, south, east, or west.

   Some squares in the grid may be *blocked*; the die can never rest on a blocked square. Other squares may be *labeled* with a number; whenever the die rests on a labeled square, the number on the *top* face of the die must equal the label. Squares that are neither labeled nor marked are called *free*. You may not roll the die off the edges of the grid. A rolling die maze is *solvable* if it is possible to place a die on the lower left square and roll it to the upper right square under these constraints.

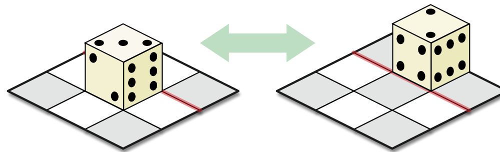


Figure 1. Rolling a (right-handed) die

   Figure 2 shows five rolling die mazes. The first two mazes are solvable using any standard die. Specifically, the first maze can be solved by placing the die on the lower left square with 1 on the top face, and then rolling the die east, north, north, east; the second maze can be solved in 12 moves. The third maze is only solvable using a *right-handed* die, where faces 1, 2, 3 appear in counterclockwise order around a common corner.[1] The last two mazes cannot be solved even with non-standard dice.

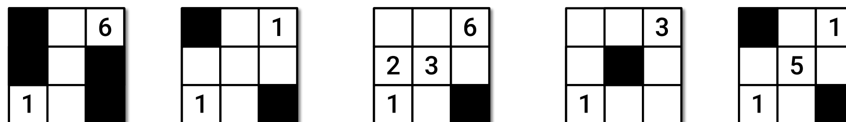


Figure 2. Five rolling die mazes.

   Describe and analyze an algorithm that determines whether a given rolling die maze can be solved with a right-handed standard die. Your input is a two-dimensional array $Label[1..n, 1..n]$, where each entry $Label[i, j]$ stores the label of the square in the ith row and jth column, where 0 means the square is free and -1 means the square is blocked.

   [Hint: You have some freedom in how to place the initial die. There are rolling die mazes that can be solved only if the initial placement is chosen correctly. Describe your solution in high-level language; don't get bogged down in grungy case analysis.]

---

[1] Right-handed dice are more common in the Western hemisphere; left-handed dice are more common in east Asia.

2. The Cheery Hells neighborhood of Sham-Poobanana runs a popular and well-regulated Halloween celebration, attended by thousands of costumed children from all across Poobanana County. To regulate the flood of costumed children, the Cheery Hells Neighborhood Association has designated a walking direction for each stretch of sidewalk.

   After paying the \$25 entrance fee, each child receives a map of the neighborhood, in the form of a directed graph G, whose vertices represent houses. Each edge $v \to w$ indicates that one can walk directly from house v to house w following the designated sidewalk directions. (Anyone caught walking backward along a sidewalk will be ejected from Cheery Hells, without their candy. No refunds.) A special vertex s designates the entrance to Cheery Hells. Children can visit houses as many times as they like, but biometric scanners at every house ensure that each child receives candy only at their *first* visit to each house.

   The children of Cheery Hells have published a secret web site listing the amount of candy that each house in Cheery Hells will give to each visitor.

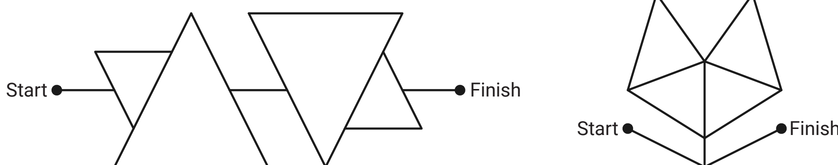
   Describe and analyze an algorithm to compute the maximum amount of candy that a single child can obtain in a walk through Cheery Hells, starting at the entrance node s. The input to your algorithm is the directed graph G, along with a non-negative integer $v.candy$ for each vertex v, describing the amount of candy the correspinding house gives to each first-time visitor.

   [Hint: Think about two special cases first: (1) Cheery Hells is strongly connected, and (2) Cheery Hells is acyclic. Solving only these two special cases is worth half credit.]

3. ***Practice only. Do not submit solutions.***

   One of my daughter's elementary-school math workbooks[2] contains several puzzles of the following type:

   > Complete each angle maze below by tracing a path from Start to Finish that has only acute angles.



   Describe and analyze an algorithm to solve arbitrary acute-angle mazes.

   Your input is a connected undirected graph G, whose vertices are points in the plane and whose edges are straight line segments. Edges do not intersect, except at their common endpoints. For example, a drawing of the letter X would have five vertices and four edges, and the first maze above has 18 vertices and 21 edges. You are also given two vertices Start and Finish.

   Your algorithm should return TRUE if G contains a walk from Start to Finish that has only acute angles, and FALSE otherwise. Formally, a walk through G is valid if, for any two

[2] Jason Batterson and Shannon Rogers, *Beast Academy Math: Practice 3A*, 2012. See https://www.beastacademy.com/resources/printables.php for several more examples.

consecutive edges $u \to v \to w$ in the walk, either $\angle uvw = \pi$ (straight) or $0 < \angle uvw < \pi/2$ (acute). Assume you have a subroutine that can determine in $O(1)$ time whether the angle between two given segments is straight, obtuse, right, or acute.

Solved problem

4. Professor McClane takes you out to a lake and hands you three empty jars. Each jar holds a positive integer number of gallons; the capacities of the three jars may or may not be different. The professor then demands that you put exactly k gallons of water into one of the jars (which one doesn't matter), for some integer k, using only the following operations:

   (a) Fill a jar with water from the lake until the jar is full.

   (b) Empty a jar of water by pouring water into the lake.

   (c) Pour water from one jar to another, until either the first jar is empty or the second jar is full, whichever happens first.

   For example, suppose your jars hold 6, 10, and 15 gallons. Then you can put 13 gallons of water into the third jar in six steps:

   - Fill the third jar from the lake.
   - Fill the first jar from the third jar. (Now the third jar holds 9 gallons.)
   - Empty the first jar into the lake.
   - Fill the second jar from the lake.
   - Fill the first jar from the second jar. (Now the second jar holds 4 gallons.)
   - Empty the second jar into the third jar.

   Describe and analyze an efficient algorithm that either finds the smallest number of operations that leave exactly k gallons in any jar, or reports correctly that obtaining exactly k gallons of water is impossible. Your input consists of the capacities of the three jars and the positive integer k. For example, given the four numbers 6, 10, 15, and 13 as input, your algorithm should return the number 6 (the length of the sequence of operations listed above).

   Solution: Let A, B, C denote the capacities of the three jars. We reduce the problem to breadth-first search in a directed graph $G = (V, E)$ defined as follows:

   - $V = \{(a, b, c) \mid 0 \le a \le A \text{ and } 0 \le b \le B \text{ and } 0 \le c \le C\}$. Each vertex corresponds to a possible ***configuration*** of water in the three jars. There are $(A+1)(B+1)(C+1) = O(ABC)$ vertices altogether.
   - G contains a directed edge $(a, b, c) \to (a', b'c')$ whenever it is possible to change the first configuration into the second in one step. Specifically, G contains an edge from (a, b, c) to each of the following vertices (except those already equal to (a, b, c)):
     - $(0, b, c)$ and $(a, 0, c)$ and $(a, b, 0)$ — dumping a jar into the lake
     - (A, b, c) and (a, B, c) and (a, b, C) — filling a jar from the lake

- $\begin{cases} (0, a+b, c) & \text{if } a+b \le B \\ (a+b-B, B, c) & \text{if } a+b \ge B \end{cases}$ — pouring from jar 1 into jar 2
- $\begin{cases} (0, b, a+c) & \text{if } a+c \le C \\ (a+c-C, b, C) & \text{if } a+c \ge C \end{cases}$ — pouring from jar 1 into jar 3
- $\begin{cases} (a+b, 0, c) & \text{if } a+b \le A \\ (A, a+b-A, c) & \text{if } a+b \ge A \end{cases}$ — pouring from jar 2 into jar 1
- $\begin{cases} (a, 0, b+c) & \text{if } b+c \le C \\ (a, b+c-C, C) & \text{if } b+c \ge C \end{cases}$ — pouring from jar 2 into jar 3
- $\begin{cases} (a+c, b, 0) & \text{if } a+c \le A \\ (A, b, a+c-A) & \text{if } a+c \ge A \end{cases}$ — pouring from jar 3 into jar 1
- $\begin{cases} (a, b+c, 0) & \text{if } b+c \le B \\ (a, B, b+c-B) & \text{if } b+c \ge B \end{cases}$ — pouring from jar 3 into jar 2

Because each vertex has at most 12 outgoing edges, there are at most $12(A+1) \times (B+1)(C+1) = O(ABC)$ edges altogether.

To solve the jars problem, we need to find the ***shortest path*** in G from the start vertex $(0,0,0)$ to any target vertex of the form $(k,\cdot,\cdot)$ or $(\cdot,k,\cdot)$ or $(\cdot,\cdot,k)$.

We can compute this shortest path by calling ***breadth-first search*** starting at $(0,0,0)$, and then examining every target vertex by brute force. If BFS does not visit any target vertex, we report that no legal sequence of moves exists. Otherwise, we find the target vertex closest to $(0,0,0)$ and trace its parent pointers back to $(0,0,0)$ to determine the shortest sequence of moves. The resulting algorithm runs in $O(V+E) = \boldsymbol{O(ABC)}$ ***time***.

We can speed up this algorithm by observing that every move leaves at least one jar either completely empty or completely full. Thus, we only need vertices (a,b,c) where either $a=0$ or $b=0$ or $c=0$ or $a=A$ or $b=B$ or $c=C$; no other vertices are reachable from $(0,0,0)$. The number of non-redundant vertices and edges is $O(AB+BC+AC)$. Thus, if we only construct and search the relevant portion of G, the algorithm runs in $\boldsymbol{O(AB+BC+AC)}$ ***time***. ■

Rubric: 10 points: standard graph reduction rubric

- Brute force construction is fine.

— 1 for calling Dijkstra instead of BFS

- max 8 points for $O(ABC)$ time; scale partial credit.