Safest Path Home

Daniel Kuelbs & Sasha Ronaghi

I. Problem description.

Suppose you are downtown at a restaurant and you want to find the safest walking path home. You live in a well-planned city with perfectly grid-like streets.

With a map of the city and data about crime rates, lighting, and how populated each street is, you decide to calculate a safety rating for each possible street. The data used to calculate the safety rating of a street is accessible through the street class, for which you should write the following functions. You may initialize a street either by providing all the variables or none of them, in which case all ints will be set to zero and sidewalk is true.

- int getDensity()
- int getCrime()
- int getLight()
- bool isSidewalk()
- int getSafetyRating()

The formula for the safety rating is calculated by taking 2 * (light level) + (population density) - 3 * (crime rate). You don't walk on streets without sidewalks.

Since your city is nicely planned, you are able to represent the street layout with a grid of streets like so: Grid<street> city. Given a certain grid of streets, the goal is to find the path from the top left of the grid to the bottom right of the grid with the highest total safety rating. That is, we want the path with the highest value if we sum over all the individual streets it contains. A path is represented by a vector of streets: Vector<street>.

For testing purposes, it will be necessary to write a function bool areEqual(Vector<street> path1, Vector<street> path2), which compares two paths and spits out true if they are equal and false otherwise. Then, write a function int getPathSafetyVector(Vector<street> path) which takes in a vector of streets and returns its total safety rating.

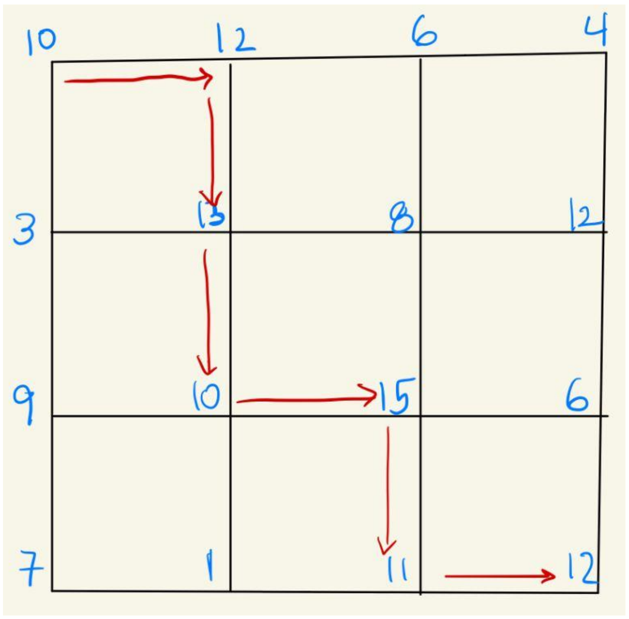
Finally, write the function Vector<street> safestPath(Grid<street> city), which takes in a Grid<street> and outputs the Vector<street> with the highest safety rating. Your function will take in a grid of streets and return a vector of streets representing the safest path that avoids streets without sidewalks.

For simplicity, you may assume that there is always a valid path (a lack of sidewalks won't prevent you from getting to your destination in any case), and that all grids will be at least 2x2. Furthermore, all paths will only ever move to the right and down--nobody wants to walk backwards on their way home.

If you take an iterative approach as opposed to a recursive one, a wide range of ADT's will be useful: stacks, queues, priority queues, sets, and vectors may all be used. If you utilize recursion, defining a helper function can be useful depending on how you go about finding the safest path.

```
Vector<street> safestPath(Grid<street> city){
  /* TODO: Fill me in! */
  return {};
}
```

Example:



Each of the numbers is the respective safety rating of the street in this 4x4 city. Note that in the graphical representation, streets are located at the corners/intersections of the black lines. The red path would be returned by your function.

II. Solutions and Test Cases.

Required Functions:

```
bool areEqual(Vector<street> path1, Vector<street> path2){
  if (path1.size() != path2.size()){
     return false;
  }
  for (int i = 0; i < path1.size(); i++){
     if (path1[i].getCrime() != path2[i].getCrime()
           || path1[i].getLight() != path2[i].getLight()
           || path1[i].getDensity() != path2[i].getDensity()
           || (!path1[i].isSidewalk() && path2[i].isSidewalk())
           || (path1[i].isSidewalk() && !path2[i].isSidewalk())){
        return false;
        }
    }
    return true;
}

int getPathSafetyVector(Vector<street> path){
    int output = 0;
    for (street s: path){
        output += s.getSafetyRating();
    }
    return output;
}
```

Solution 1:

```
void safestPath1Helper(Grid<street> city, int row, int col, Vector<street> path,
PriorityQueue<Vector<street>>& solutions){
    if (row == city.numRows() - 1 && col == city.numCols() - 1){
        solutions.enqueue(path, -(getPathSafetyVector(path)));
        //negative because safer paths will have a lower priority value
    }
    else {
        Vector<street> rightPath = path;
        Vector<street> downPath = path;
        if (row < city.numRows() - 1 && city[row + 1][col].isSidewalk()){
            downPath.add(city[row + 1][col]);
            safestPath1Helper(city, row + 1, col, downPath, solutions);
        }
        if (col < city.numCols() - 1 && city[row][col + 1].isSidewalk()){
            rightPath.add(city[row][col + 1]);
            safestPath1Helper(city, row, col + 1, rightPath, solutions);
        }
    }
}

Vector<street> safestPath1(Grid<street> city){
    PriorityQueue<Vector<street>> solutions;
    Vector<street> path;
    path.add(city[0][0]);
    safestPath1Helper(city, 0, 0, path, solutions);
    return solutions.dequeue();
}
```

The first solution is a typical recursive backtracking approach with a helper function. The program proceeds left to right and top to bottom, finding all possible paths and enquing them in a priority queue with priority equal to the additive inverse of their safety rating. This ensures that the highest safety rating has the lowest priority value, and thus will be dequeued first when we call solutions.dequeue(). Let **N** be the number of rows in the grid and let **M** be the number of columns. Since this function will call itself at most twice, it will have a runtime of **O((NM)^2)**. This is the most elegant solution to the problem, but requires finding all of the solutions.

Solution 2:

```
Vector<street> getSaferPath(Vector<street> path1, Vector<street> path2){
  if (path1.size() > path2.size()){
     return path1;
  }
  else if (path2.size() > path1.size()){
     return path2;
  }
  else {
     if (getPathSafetyVector(path1) >= getPathSafetyVector(path2)){
        return path1;
     }
     return path2;
  }

}


Vector<street> safestPath2Helper(Grid<street> city, int row, int col, Vector<street>& path){
  if (row == city.numRows() - 1 && col == city.numCols() - 1){ // end of path
     return path;
  }
  else if (row == city.numRows() - 1){ //hit the side wall
     if (!city[row][col + 1].isSidewalk()){
        return {};
     }
     path.add(city[row][col + 1]);
     return safestPath2Helper(city, row, col + 1, path);
  }
  else if (col == city.numCols() - 1){
     if (!city[row + 1][col].isSidewalk()){
        return {};
     }
     path.add(city[row + 1][col]);
     return safestPath2Helper(city, row + 1, col, path);
  }
  else {
     Vector<street> rightPath = path;
```

```
        Vector<street> downPath = path;
        if (city[row + 1][col].isSidewalk()){
            downPath.add(city[row + 1][col]);
        }
        if (city[row][col + 1].isSidewalk()){
            rightPath.add(city[row][col + 1]);
        }
        rightPath = safestPath2Helper(city, row, col + 1, rightPath);
        downPath = safestPath2Helper(city, row + 1, col, downPath);
        return getSaferPath(rightPath, downPath);
    }
    return {};
}
```

The second solution is another recursive approach, which compares each path as they are found and then returns the one with the highest safety rating. Here we do not need to store all of the possible solutions, instead we simply compare as we go and return the optimal path. Let **N** be the number of rows in the grid and let **M** be the number of columns. Then the runtime of this function is also **O((NM)^2)**, since it calls itself twice each time. This solution is less elegant than Solution 1, but returns the safest path through recursion right away.

Solution 3:

```
Set<GridLocation> generateValidMoves(Grid<GridLocation>& city, GridLocation cur,
Grid<street>& cityStreet) {
    int cur_x = cur.row;
    int cur_y = cur.col;
    Vector<GridLocation> possible_locations = {GridLocation(cur_x, cur_y + 1),
                                              GridLocation(cur_x +  1, cur_y)};
    Set<GridLocation> neighbors;
    for (GridLocation location : possible_locations) {
        if (city.inBounds(location) &&
                cityStreet[location.row][location.col].isSidewalk()) {
            neighbors.add(location);
        }
    } /
    return neighbors;
}

int getGridLocPathSafety(Stack<GridLocation> path, Grid<street>& city){
    int output = 0;
    while (!path.isEmpty()){
        GridLocation loc = path.pop();
        output += city[loc.row][loc.col].getSafetyRating();
```

```
  }
  return output;
}

Stack<GridLocation> safestPath3Helper(Grid<GridLocation>& city, Grid<street>& cityStreet) {
  Stack<GridLocation> path;
  PriorityQueue<Stack<GridLocation>> solutions;
  Queue<Stack<GridLocation>> paths;
  GridLocation entry = GridLocation(0,0);
  GridLocation exit = {city.numRows()-1,  city.numCols()-1};
  path.push(entry);
  paths.enqueue(path);
  Set<GridLocation> checked_moves;
  while (paths.size() > 0){
     Stack<GridLocation> curr_path = paths.dequeue();
     checked_moves.add(curr_path.peek());
     if (curr_path.peek() == exit) {
        solutions.enqueue(curr_path, -(getGridLocPathSafety(curr_path, cityStreet)));
        //negative because safer paths will have a lower priority value
     }
     else{
        Set<GridLocation> valid_moves = generateValidMoves(city, curr_path.peek(), cityStreet);
        for (GridLocation move : valid_moves) {
           if (!checked_moves.contains(move)) {
              Stack<GridLocation> new_path = curr_path;
              new_path.push(move);
              paths.enqueue(new_path);
           }
        }
     }

  }
  return solutions.dequeue();
}

Vector<street> safestPath3(Grid<street>& cityStreet){
  Grid<GridLocation> city(cityStreet.numRows(), cityStreet.numCols());
  Stack<GridLocation> stackOutput = safestPath3Helper(city, cityStreet);
  Vector<street> output;
  while (!stackOutput.isEmpty()){
     GridLocation loc = stackOutput.pop();
     output.insert(0, cityStreet[loc.row][loc.col]);
  }
  return output;
```

```
}
```

The third solution takes an iterative approach to the problem using a few helper functions. It takes the city as both a grid of GridLocation and streets, in which each GridLocation corresponds to a street. Using Stacks and Queues, the helper function safestPath3Helper finds each potential path in terms of a GridLocation and uses the helper function getGridLocPathSafety to store the path in a Priority Queue with their safety rating as the priority value. The solution then converts the path of GridLocations to a path of streets and returns the vector of streets of the safest path. Let **N** be the number of rows in the grid and let **M** be the number of columns. Then the runtime of this function is **O((NM)(N + M - 1)^2)**, since it goes through each row and column during the iteration. Then, the function does vector insertion, which is 0(k^2) where k is the size of the vector because the vector moves all of the elements for each insert. The vector size is (N + M - 1). This solution is the least efficient, but allows for the use of ADTs and iteration to prevent stack overflow for especially large grids.

Test Cases:

Test cases were implemented to cover city grids of varying complexity and size. We also ran our functions on a grid with multiple safest paths.

```
STUDENT_TEST("Very simple example"){
  street sdwlk =  street(2, 3,  4, false);
  street street1 =  street(10, 1, 1, true);

  Grid<street> city = {{street1, sdwlk},
                      {street1, street1}};
  Vector<street> expectedPath = {street1, street1, street1};

  Vector<street> actual1 = safestPath1(city);
  Vector<street> actual2 = safestPath2(city);
  Vector<street> actual3 = safestPath2(city);


  EXPECT(areEqual(expectedPath, actual1));
  EXPECT(areEqual(expectedPath, actual2));
  EXPECT(areEqual(expectedPath, actual3));

  //n is total number of elements in the grid
  TIME_OPERATION(4, safestPath1(city));
  TIME_OPERATION(4, safestPath2(city));
  TIME_OPERATION(4, safestPath3(city));
}

STUDENT_TEST("Simple example"){
  street sdwlk =  street(2, 3,  4, false);
```

```
    street street1 =  street(10, 1, 1, true);
    street street2 =  street(0, 0, 0, true);

    Grid<street> city = {{street2, street1, street2},
                         {street2, sdwlk, street2},
                         {street2, street2, street2}};
    Vector<street> expectedPath = {street2, street1, street2, street2, street2};

    Vector<street> actual1 = safestPath1(city);
    Vector<street> actual2 = safestPath2(city);
    Vector<street> actual3 = safestPath2(city);

    EXPECT(areEqual(expectedPath, actual1));
    EXPECT(areEqual(expectedPath, actual2));
    EXPECT(areEqual(expectedPath, actual3));

    TIME_OPERATION(9, safestPath1(city));
    TIME_OPERATION(9, safestPath2(city));
    TIME_OPERATION(9, safestPath3(city));
}

STUDENT_TEST("Complicated example"){
    street sdwlk =  street(2, 3,  4, false);
    street street1 =  street(10, 1, 1, true);
    street street2 =  street(0, 0, 0, true);

    Grid<street> city = {{street2, street1, street2, street1, street2},
                         {street2, sdwlk, street2, street1, street2},
                         {street2, street2, street2, street1, street1},
                         {street2, street2, sdwlk, street1, street1},
                         {sdwlk, street2, sdwlk, street2, street2}};
    Vector<street> expectedPath = {street2, street1, street2, street1, street1, street1, street1,
street1, street2};
    Vector<street> actual1 = safestPath1(city);
    Vector<street> actual2 = safestPath2(city);
    Vector<street> actual3 = safestPath2(city);

    EXPECT(areEqual(expectedPath, actual1));
    EXPECT(areEqual(expectedPath, actual2));
    EXPECT(areEqual(expectedPath, actual3));

    TIME_OPERATION(25, safestPath1(city));
    TIME_OPERATION(25, safestPath2(city));
    TIME_OPERATION(25, safestPath3(city));
```

```
}

STUDENT_TEST("More complicated example"){
  street sdwlk =  street(2, 3,  4, false);
  street street1 =  street(10, 1, 1, true);
  street street2 =  street(0, 0, 0, true);
  street street3 = street(50, 1, 60, true);
  street street4 = street (0, 15, 3, true);

  Grid<street> city = {{street2, street3, street2, street1, street2, sdwlk},
                  {street2, sdwlk, street3, street4, street2, street1},
                  {street2, street2, street2, street1, street3, street4},
                  {street2, street2, sdwlk, street1, street1, street3},
                  {sdwlk, street2, sdwlk, street2, street2, street3}};
  Vector<street> expectedPath = {street2, street3, street2, street3, street2, street1, street3,
street1, street3, street3};
  Vector<street> actual1 = safestPath1(city);
  Vector<street> actual2 = safestPath2(city);
  Vector<street> actual3 = safestPath2(city);

  EXPECT(areEqual(expectedPath, actual1));
  EXPECT(areEqual(expectedPath, actual2));
  EXPECT(areEqual(expectedPath, actual3));

  TIME_OPERATION(30, safestPath1(city));
  TIME_OPERATION(30, safestPath2(city));
  TIME_OPERATION(30, safestPath3(city));
}

STUDENT_TEST("Two paths of same safety rating"){
  street sdwlk =  street(2, 3,  4, false);
  street street1 =  street(3, 2, 1, true);
  street street2 =  street(5, 1, 4, true);

  Grid<street> city = {{street2, street2, street1},
                  {street1, sdwlk, street2},
                  {street1, street1, street2}};
  Vector<street> expectedPath1 = {street2, street2, street1, street2, street2};
  Vector<street> expectedPath2 = {street2, street1, street1, street1, street2};

  Vector<street> actual1 = safestPath1(city);
  Vector<street> actual2 = safestPath2(city);
  Vector<street> actual3 = safestPath2(city);
  EXPECT(areEqual(expectedPath1, actual1) || areEqual(expectedPath2, actual1));
  EXPECT(areEqual(expectedPath1, actual2) || areEqual(expectedPath2, actual2));
  EXPECT(areEqual(expectedPath1, actual3) || areEqual(expectedPath2, actual3));
}
```

III. Problem Motivation.

Conceptual Motivation

This project challenges students to think hard about recursive backtracking and abstract data structures. While the different approaches use recursion and ADT's to varying degrees, they all require the student to be able to systematically traverse the grid, access data from a custom class, and pick an optimal solution out of the many generated ones.

Additionally, this project requires the student to:

- Create testing infrastructure to handle custom classes
- Pass ADT's by reference
- Work with nested ADT's

Personal Motivation

Out of the many concepts in the CS 106B curriculum, we found recursive backtracking and abstract data structures to be especially interesting and powerful tools. After the mid-quarter diagnostic, we also decided it would be worthwhile to strengthen our understanding of recursive backtracking and recursion in general.

One of the main reasons we chose this project was because we thought coming up with alternate solutions to a typical recursive backtracking problem would be a good challenge. It was difficult but absolutely worthwhile to write the three different solutions--a recursive backtracking algorithm that stores all possible solutions in a priority queue, a recursive program that compares paths and only returns the one with the highest safety rating, and an iterative algorithm which uses a plethora of abstract data structures to find all possible paths and returns the safest one.

The premise of this problem is motivated by the potential dangers of walking alone in a city or college campus, especially at night. Oftentimes, map applications prioritize speed when providing a route for consumers. For some, especially those who identify as female, they would rather take a route that is safe (has more light, greater population density, and lower crime rate) over a fast route, especially when alone or at night. This project tries to solve this gap in most map applications and provide a safest route given light, population, and crime data.

IV. Concept Mastery/Common Misconceptions.

This project requires students to have a solid understanding of recursion/recursive backtracking, classes, and abstract data structures.

Recursion/Recursive Backtracking:

Recursive backtracking is generally tricky, but some common errors that might occur when trying to implement the recursive solutions (1 and 2) might be forgetting to make a helper function with more variables or not properly exploring the grid (passing in a row and column separately and iterating to the right and down).

Classes:
In handling the street class, cryptic errors arise when using the = operator on vectors of streets. This can be addressed by writing a helper function areEqual, which takes in two paths (street vectors) and returns a bool if they contain all the same streets. This is done by comparing the getCrime, getLight, and getDensity values of each individual street in each vector.

Abstract Data Structures:
Each of our solutions utilizes abstract data structures in some form or another, and students will likely run into issues like trying to loop through a queue/priority queue, off-by-one while handling with the grid of streets/path vectors, or peeking empty stacks/queues. Students may have trouble figuring out which data structure to use when storing paths for later comparison like in solution 1. For example, if a vector or regular queue is used as opposed to a priority queue, the solution can still work, but one has to write extra helper functions to do the comparison, as opposed to taking advantage of the properties of a priority queue.