Group 19: Shasha Alvares, Qisheng Chen, Yash Jagdish Gopani
CS 590 Project 2
Due: Wednesday April 23rd, 2025

Project 2 Report

1. How can you break down a large problem instance into one or more smaller instances? Your answer should include how the solution to the original problem is constructed from the sub-problems and why this breakdown makes sense.

You can break down a large problem instance into one or more smaller instances by first solving the smallest and easiest subproblems, which are the base cases. We can then use those base case solutions to incrementally solve bigger and harder problems. This breakdown makes sense because if you try to solve a big problem first, you have no information to work with. By solving base cases and smaller and easier subproblems first, you are laying the foundation information that can be used and combined to solve the overall larger and more complicated problems.

In terms of the bigger original problem of solving for the greatest score possible in our game, first we can sort our input in ascending numeric space order (the first number on each line) and only deal with the values in our solution, not the indexes.

Starting from the smallest space value, we iteratively check each neighbor and compute the maximum score by combining the current space's points with the best score from any smaller-valued neighbor. We then either store the value of the current space if it is a base case and there are no neighbors of smaller value, or we move to the next neighbor based on which smaller neighbor gives the largest score value when added to the value of the current space out of all the neighbors.

We will continue doing this in ascending space order until we have the greatest score stored for each space (in one dynamic programming array) and the path it took to get to that space (in another dynamic programming array). At the end we can use the results stored in the arrays to find the maximum score and path taken. Overall by calculating and storing the value of the smallest spaces first, we are able to build up to the maximum score efficiently from base cases to larger spaces.

2. What are the base cases of this problem, and what are their solutions?

If a node does not have any lower neighboring values, then it is a base case and the maxScore for the traversal is the value of that current node. For example, in regards to the small-input.txt, the spaces 1, 2, and 3 are all base cases because none of their neighbors are smaller in value, thus from those three spaces as starting points, it is impossible to move.

Group 19: Shasha Alvares, Qisheng Chen, Yash Jagdish Gopani
CS 590 Project 2
Due: Wednesday April 23rd, 2025

3. What data structure would you use to store the partial solutions to this problem? Justify your answer.

Since our coding solution needs to solve the problem iteratively, we could use two dynamic programming arrays to store the partial solutions to this problem. After sorting our input in terms of ascending space number order, one array would store the indexes corresponding to the space number. And the value for that index would be the max possible score for the path traversal. The second array would store the path taken up until that space. The index would again correspond to the space, and the value for that index would be the previous space visited. Then we would just have to continue tracing the previous vertices until we hit the base case. Finally we would reverse the list and print it to get the correct traversal order.

4. Give pseudocode for an algorithm that uses memoization to compute the maximum score.

Algorithm: findMaxScore(v(i))

Sort the algorithm by ascending space value v(i)
Initialize a dp array from 1 to n
Assign each value in the dp array -1 //the maxScore has to be calculated
Initialize maxScore = 0

```
for i from 1 to n:
        //recursively calculate the max score of every space
        maxScore = max(maxScore, memoizedMaxScore(i))
return maxScore

def memoizedMaxScore(i):
        //base case the score has already been calculated
        if dp[i] is not equal to -1:
                return dp[i]

        maxNeighbor = 0 //used to keep track of the maximum neighbor value

        //recursively explore all smaller neighbors
        for each neighbor of v(i):
                If the current neighbor value (v(j)) is smaller than v(i):
                        //Recursively compute the full path score ending at neighbor j
```

```
                    //this iterates through all neighbors and computed the sum each
                    //time it is called to find the greatest score
                    neighborScore = memoizedMaxScore(j)
                    maxNeighbor = max(maxNeighbor, neighborScore)

    //Max Score that will be stored in dp array is the summation  of the current
    // space's points p(i) + the maxNeighbor score that was calculated by summing
    //the p(i)'s of the previous neighbors
    dp[i] = p[i] + maxNeighbor
    return dp[i]
```

5. What is the time complexity of your memoized algorithm?

The time complexity of our memoized algorithm O(n*log(n)+m). The sorting of the input in terms of ascending v(i) takes O(n*log(n)) time and the O(m) is from visiting each vertex once.

6. Give pseudocode for an iterative algorithm for this problem.

```
Sort the input file in terms of ascending space value where space value is v(i)
Instantiate a dp array of size n
Initialize maxScore to 0
Initialize all nodes to undiscovered except for space 1
Start at the first node (for each node in Graph)
//iterating through each space
for i from 1 to n:
    if v(i) does not have any neighbors of smaller value:
        spaceMaxScore = p(i)
        dp[i] = spaceMaxScore
    else if v(i) does have neighbors of smaller values:
        Compare the maximumScore from adding the current space's p(i) to each
        Neighbor's p(neighbor)  individually. Assign maximumScore to the
        greatest addition value.
        for j in N(v(i)):
            spaceMaxScore = dp[j] + p(i)
            if maxScore < spaceMaxScore:
                dp[i] = spaceMaxScore
                maxScore = dp[i]
return maxScore
```

Group 19: Shasha Alvares, Qisheng Chen, Yash Jagdish Gopani
CS 590 Project 2
Due: Wednesday April 23rd, 2025

7. Describe how you could modify your algorithm to identify the maximum-scoring tour, not just the maximum possible score.

In addition to a dynamic programming array that stores the maximum score, we could initialize a second array called prev[] where the index corresponds to the space number, and the value at that index is the space previously visited that had the greatest calculated maximum score. Every time the maxScore is updated, we would also update the prev[] to store the space of the current neighbor the maxScore was calculated for. The path stored corresponds to the path traversed to calculate the maximumScore in the game. This way we would just have to print out the space numbers, and reverse the order to get the path taken.

For example, the prev[] array could look like what I have drawn below. Let's say the highest score of all spaces was found at space 5 in the maximum score array. To get the tour, we would store the path starting from space 5 in the path array. First we would store 5, then v(5) which is 4, then v(4) which is 1, and because v(1) is null we know we have reached the start of the path. Then by reversing the order we would get a path of 1→ 4→5 which is the correct path traveled in this hypothetical example.

Prev[] array:

Prev space: Null	Null	Null	1	4
Space: 1	2	3	4	5

8. Bonus Describe (briefly) how you would modify your algorithm to account for adjacent equal values and wildcards. There is likely no algorithm that is guaranteed to solve this problem in polynomial time, so just focus on solving the problem correctly rather than quickly.

Since the code is required to be an iterative solution, I will describe the modifications using the iterative pseudocode.

To account for adjacent equal values, I would change the comparison condition ("else if v(i) does have neighbors of smaller values") that currently allows traversal only from neighbors with smaller values (v(j) < v(i)) to instead allow neighbors with equal or smaller values (v(j) ≤ v(i)). This would enable traversal through spaces with the same value.

To account for wildcard spaces, I would define: wildcard = "*" and adjust the same condition mentioned in the paragraph above so that traversal is also allowed if

either the current space or the neighbor is a wildcard. Wildcards would be treated as flexible spaces that can connect to any neighbor regardless of value. During iteration, if

a space is a wildcard, it would inherit the maximum score from its neighbors, and also allow continuation to any other space without value constraints.