Team members:
Cong Le Cle155@csu.fullerton.edu
Jack Nguyen nhannguyen7@csu.fullerton.edu
CPSC335-03: Algorithm Engineering
Professor Kevin Wortman
Due date: Friday, November 2, 1 pm.

Project 4- Greedy Gnomes

The Hypothesis

This project will test the following hypothesis:
Polynomial-time dynamic programming algorithms are more efficient than exponential-time exhaustive search algorithms that solve the same problem.

The Greedy Gnomes Problem is
Input: a r×c matrix G where each cell is one of . (earth), g (gold), or X (rock); and G[0][0]= .
Output: a path starting at (0, 0); where each step is either a start, right move, or down move; that does not visit any R cell; and the number of g cells in the path is maximized

We will represent the underground mine-able area as a 2D grid, like the following:

```
.X..X.........X........
......g...............
..................g.....
g...........g....g.....
.Xg........X.X...g.....
........g.......X.g.....
....X......X......gX....
g.g.........X..g.g......
....................X..
...X..g.........g......
g..X......g.......g..X..
gXXX............X.....X.
```

A gnome starts at row 0 and column 0, i.e. coordinate (0, 0), at the top-left corner. Each g cell represents gold, each X represents rock, and each . cell represents earth that may be excavated. A gnome's goal is to plan a route for a mine that maximizes the amount of gold they find, while avoiding rock.
Here is an optimal solution for the previous grid:

```
+X..X.........X........
++++++G++++++++++.....
..................G.....
g...........g....G.....
.Xg........X.X...G.....
........g.......X.G.....
....X......X......GX....
g.g.........X..g.g+.....
...................+..X..
...X..g.........g+.....
g..X......g.......G..X..
gXXX............X.....X.
     steps=29 gold=7
```

In this diagram, the + cells represent earth that the gnome mined through, and the capital G cells represent gold that the gnome harvested. A . still represents earth that was never visited, and lower-case g still represents gold that was not harvested. Observe that the mine starts at (0, 0), and moves

right and down, but not in any other direction.

We will analyze the two algorithms (Exhaustive Optimization and Dynamic Programming) in details by examine their pseudocodes and time complexity.

Part I: Description of the Algorithms:

1. **Exhaustive Optimization Algorithm:**

Since the output definition says that the amount of gold mined must be maximized, so this is an exhaustive optimization algorithm.

Since all paths start at (0, 0) and the only valid moves are right and down, mine paths never backward or upward. So the longest possible path is one that reaches the bottom-right corner of the grid. The grid has r rows and c columns, so this longest path involves (r-1) down moves and (c-1) right moves, for a total of r + c - 2 moves.

There are two kinds of move, right → and down ↓. Coincidentally there are two kinds of bits, 0 and 1. So we can generate move sequences by generating bit strings, using the same method that we used to generate subsets. We loop through all binary numbers from 0 through 2^n-1, and interpret the bit at position *k* as the up/down step at index *k.*

A candidate path is valid when it follows the rules of the greedy gnomes problem. That means that the path stays inside the grid, and never crosses a rock (R) cell.

2. **Dynamic Programming Algorithm:**

This dynamic programming array A stores partial solutions to the problem. In particular,

A[r][c] = the gold-maximizing mine path that starts at (0, 0) and ends at (r, c);
or None if (r, c) is unreachable

Recall that in this problem, some cells are filled with rock and are therefore unreachable by mines. The base case is the solution for A[0][0], which is the trivial path that starts and takes no subsequent steps.

A[0][0] = [start]

We can build a solution for a general case based on pre-existing shorter paths. Gnomes can only dig right and down. So there are two ways a mine path could reach (i, j):

1. The mine path above (i, j) could add a down step.
2. The mine path to the left of (i,j) could add a right step.

The algorithm should pick whichever of these two alternatives is optimal, which in this problem means whichever of the two candidate paths harvests more gold.

However, neither of these paths is guaranteed to exist. The from-above path (1) only exists when we are not on the top row (so when i>0), and when the cell above (i, j) is not rock. Symmetrically, the from-left path (2) only exists when we are not on the leftmost column (so when j>0) and when the cell left of (i, j) is not rock.

Finally, observe that A[i][j] must be None when G[i][j]==X, because a path to (i, j) is only possible when (i, j) is not a rock.

Altogether, the general solution is:

G[i][j] = None if G[i][j]==X

G[i][j] = whichever of *from_above* and *from_left* is non-None and harvests more gold

where *from_above* = None if i=0 or G[i-1][j]==X; or G[i-1][j] + [↓] otherwise

from_left = None if j=0 or G[i][j-1]==X; or G[i][j-1] + [→] otherwise

The optimal solution is not required to end at (r-1, c-1). Indeed, the optimal mine may end anywhere. So after the main dynamic programming loop, there is a post-processing step to find the path with the most gold.

Part II: Pseudocode for the Algorithms and Step Count:

1. Exhaustive Optimization Algorithm:

```
greedy_gnomes_exhaustive(G):
    maxlen = r + c - 2
    best = None
    for len from 0 to maxlen inclusive:                          n
        for bits from 0 to 2^len - 1 inclusive:                  2^n
            candidate = [start]
            for k from 0 to len-1 inclusive:                     n
                bit = (bits >> k) & 1
                if bit == 1:
                    candidate.add(→)
                else:
                    candidate.add(↓)
            if candidate stays inside the grid and never crosses an X cell:
                if best is None or candidate harvests more gold than best:
                    best = candidate
    return best
```

Observation:

Let $n = \max(r, c)$. Then $maxlen \in O(n)$, the outermost for loop repeats $O(n)$ times, the middle for loop over bits repeats $O(2^n)$ times, and the inner for loops repeat $O(n)$ times, and the total run time of this algorithm is $O(n^2 2^n)$. This is a very slow algorithm.

2. Dynamic Programming Algorithm:

```
greedy_gnomes_dyn_prog(G):
    A = new r×c matrix
    # base case
    A[0][0] = [start]
    # general cases
    for i from 0 to r-1 inclusive:                                          n
        for j from 0 to c-1 inclusive:                                      n
            if G[i][j]==X:
                A[i][j]=None
                Continue
            from_above = from_left = None
            if i>0 and A[i-1][j] is not None:
                from_above = A[i-1][j] + [↓]      n
            if j>0 and A[i][j-1] is not None:
                from_left = A[i][j-1] + [→]       n
            A[i][j] = whichever of from_above and from_left is non-None and harvests
            more gold; or None if both from_above and from_left are None
    # post-processing to find maximum-gold path
    best = A[0][0] # this path is always legal, but not necessarily optimal
    for i from 0 to r-1 inclusive:                                          n
        for j from 0 to c-1 inclusive:                                      n
            if A[i][j] is not None and A[i][j] harvests more gold than best:
                best = A[i][j]
    return best
```

Observation:

The time complexity of this algorithm is dominated by the general-case loops. The outer loop repeats n times, the inner loop repeats n times, and creating each of *from_above* and *from_left* takes $O(n)$ time to copy paths, for a total of $O(n^3)$ time. While $O(n^3)$ is not the fastest time complexity out there, it is polynomial so considered tractable, and is drastically faster than the exhaustive algorithm.

Part III: Graphs and Empirical Analysis on the Algorithms:

Empirical Timing Data	End-to-Beginning Algorithm Runtime	Dynamic Programming Algorithm Runtime
Instance Size n	**Elapsed Time (in seconds)**	
15	0.0803707	0.000102286
16	0.164556	0.000131498
17	0.321592	0.00012518
18	0.630174	0.000145242
19	1.4322	0.000162479
20	2.72094	0.00019447
21	5.91575	0.000179775
22	12.0989	0.000208608
23	24.2318	0.000220399
24	52.4215	0.000241428
25	117.548	0.000302922
26	203.798	0.00031321
27	497.947	0.000345179

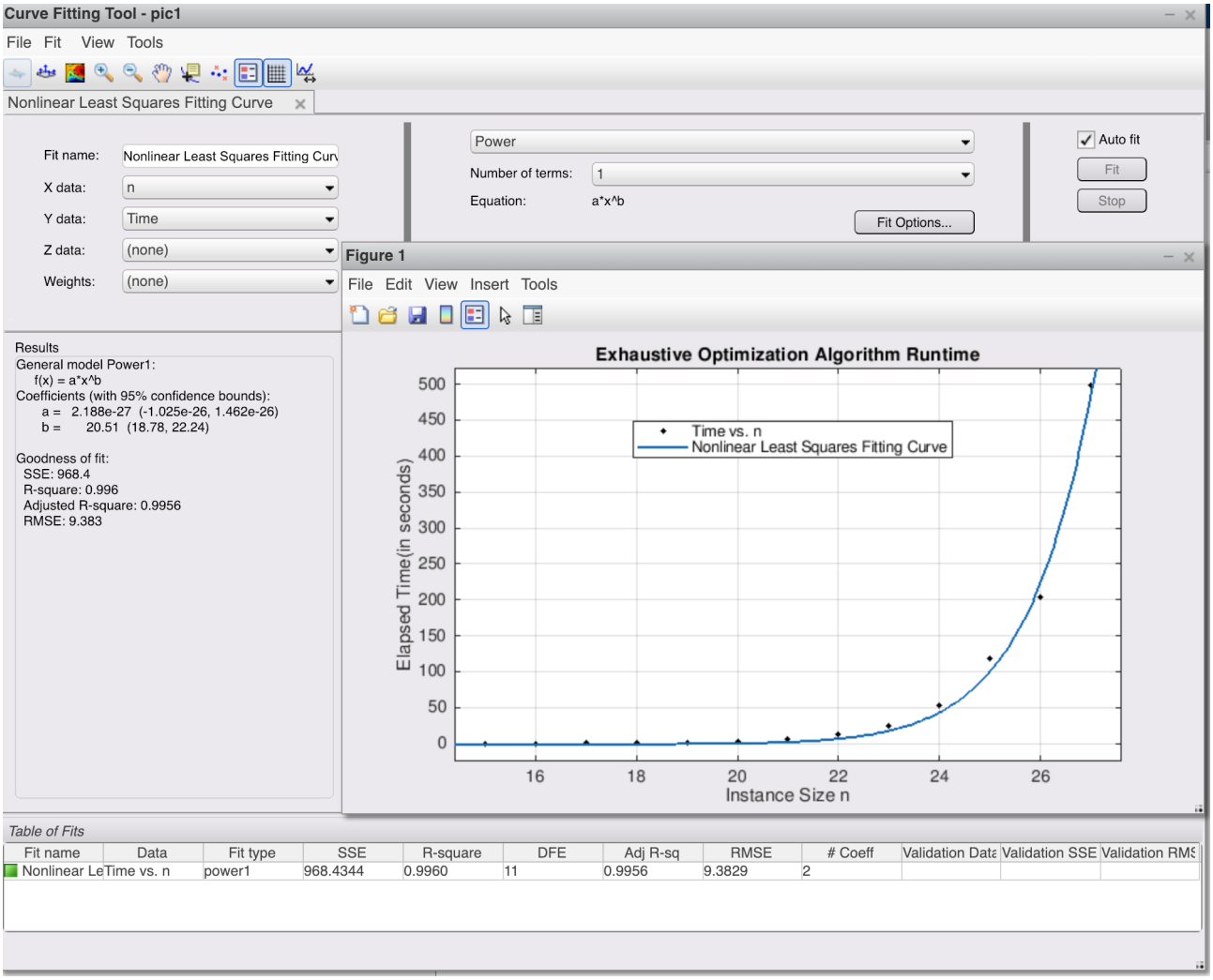

Figure1

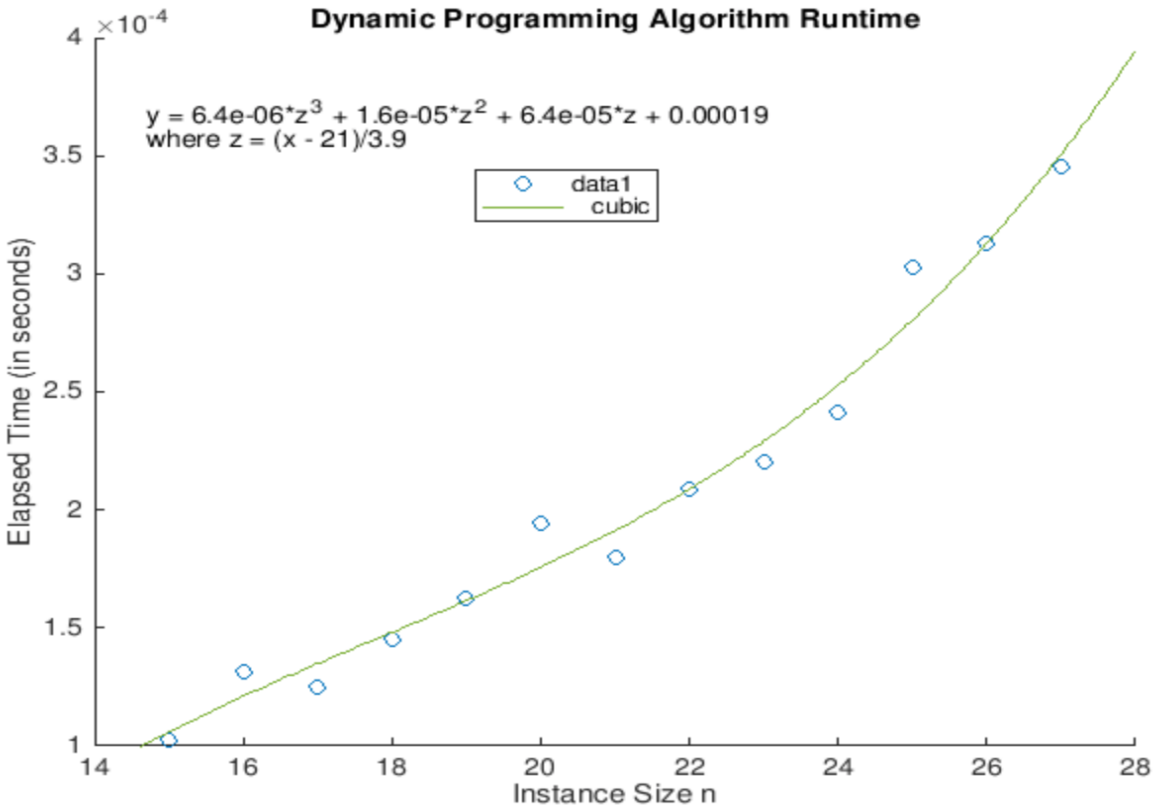

Figure2

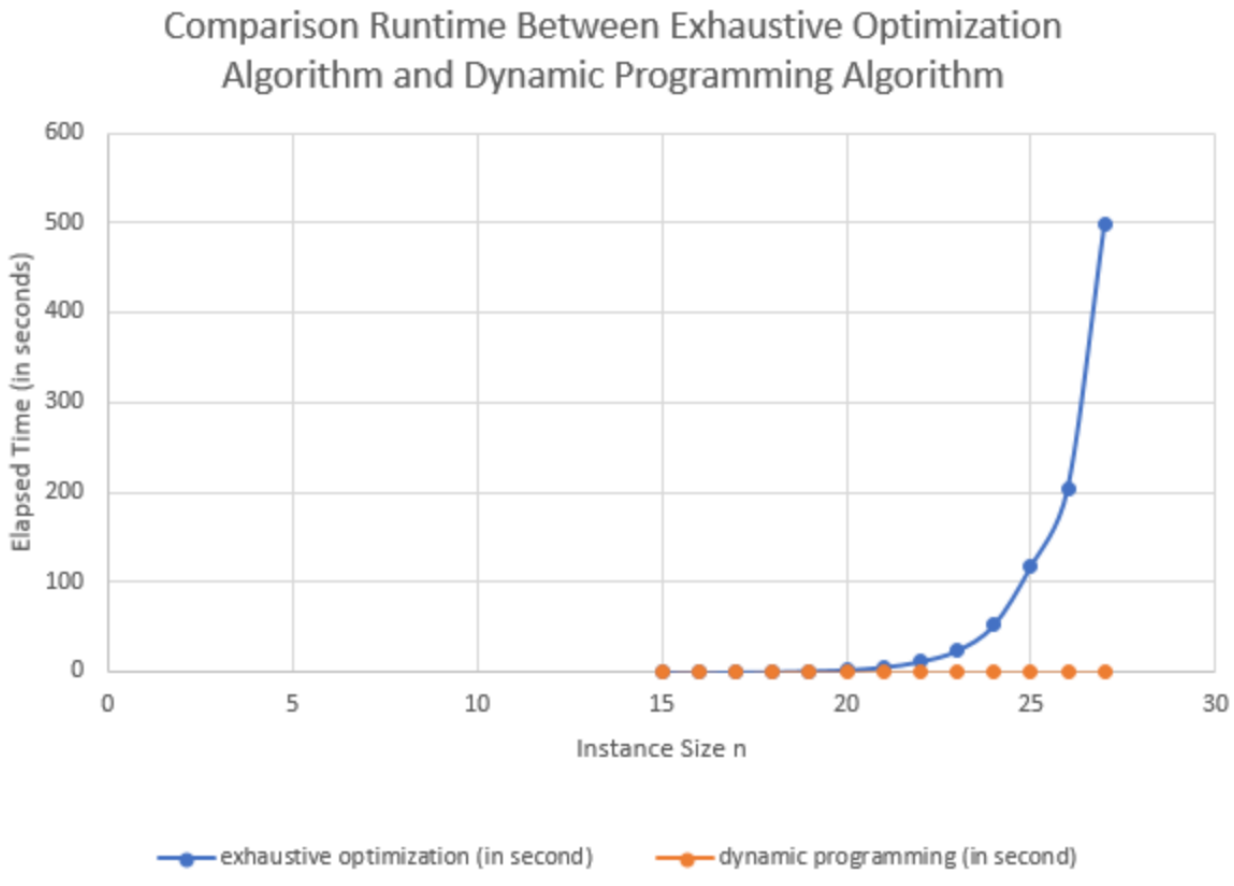

Figure3

Empirical Analysis

- We observe that the running speed for Exhaustive Optimization Algorithm is much slower comparing with the Dynamic Programming Algorithm's. This observation is not surprise us because after mathematically analyzing, we already know that the efficiency class of Exhaustive Optimization Algorithm is $O(n^2 2^n)$ time (exponential time), which is very slow comparing to $O(n^3)$ time (cubic time) of the Dynamic Programming Algorithm.
- The fit line on **Figure1** is a very slow exponential curve, which is ***consistence*** with the efficiency class of Exhaustive Optimization Algorithm $O(n^2 2^n)$ time, exponential time.
- The fit line on **Figure2** is likely close to an cubic curve, which is ***consistence*** with the efficiency class of Dynamic Programming Algorithm as $O(n^3)$ time, cubic time.
- Based on our evidences and observations on **Figure3**, we conclude that the initial hypotheses for this project is consistent with our experimental data: Polynomial-time dynamic programming algorithms are more efficient than exponential-time exhaustive search algorithms that solve the same problem.
- Throughout this experiment, we realize the Exhaustive Optimization Algorithm is much easier to implement than Dynamic Programming Algorithm. Even though time complexity of both algorithms require an exponential time. However, time complexity of Exhaustive Optimization Algorithm as $O(n^2 2^n)$ time (exponential time) is tremendously slower comparing to its alternative as $O(n^3)$ time (cubic time). On the other hand, the benefit of Dynamic Programming Algorithm is that faster time complexity comparing to Exhaustive Optimization Algorithm. However, we encounter a bit difficulty in implementation because Dynamic Programming Algorithm needed the latest C++17 standard library, which is *std::optional*. The latest feature requires us have to learn the new syntax along with implementation of all the tricky logics in Dynamic Programming Algorithm. Overall, we still prefer the second algorithm, Dynamic Programming. The reason for our choice is Dynamic Programming Algorithm are benefit of better time complexity and the innovation of learning new features in programming in general.