# Project 3

Spring 2023 CPSC 335 - Algorithm Engineering

Instructor: Prof. Sampson Akwafuo

# Soccer Opponent Avoidance

*Polynomial versus Exponential Time*

## Hypothesis

This project will test the following hypotheses:

1. Exhaustive search algorithms can be implemented, and produce correct outputs.
2. Algorithms with exponential or factorial running times are extremely slow, probably too slow to be of practical use.

To test this hypothesis, you will implement two algorithms:

1. a $O(n.2^n)$ - time exhaustive search algorithm for the *soccer opponent avoidance problem;* and
2. a $\mathrm{O}(n^2) -$ time dynamic programming algorithm for the same problem.

### The Soccer Opponent Avoidance Problem

The ***soccer opponent avoidance problem*** is a puzzle that comes from a modified real-life scenario, where players of a team maneuver through an opponent-filled field in order to find a safe passage to reach the goal post. For example, we could consider the Red Team trying to overcome the Blue Team. The field is represented by a square $\boldsymbol{r \times c}$ grid with r rows and c columns. In this modified game, unlike in real-life, players of the opposing team are stationary, and are located in some cells. The Red Team starts from the top-left corner of the field, at (row, column) location (0,0), dribbles past the opponents, and aims to reach the goal post in the bottom-right corner at location (r$-\boldsymbol{1, c-1}$). The occupied cells are impenetrable and movements towards the goal post can only be done by moving right, from location $(\boldsymbol{i, j})$ to $(\boldsymbol{i, j+1})$; or down, from $(\boldsymbol{i, j})$ to $(\boldsymbol{i+1, j})$. A path may not go through an occupied cell, so the Red Team can only move to a new location, if there is no opponent at that location.

The field can be displayed as a 2D grid, like the following:

```
……X.X
X……..
…X…X.
..X….X.
.X….X..
….X….
..X…..X
……..
```

The Red Team starts at row 0 and column 0, i.e. coordinate (0,0) at the top-left corner. Each period represents a passable spot and each X represents an opponent-occupied spot (i.e. an impenetrable spot). The Red Team's goal is to plan a passable route to cross the field, and reach the opponents goal post, while avoiding the occupied cells. **The problem objective is to compute the number of different paths to cross the field. T**wo paths are different if they differ by at least one location.

For the previous grid, the optimal solution is **102**.

This puzzle can be defined as an algorithmic problem.

***soccer opponent avoidance problem***
**input:** a $\boldsymbol{r} \times \boldsymbol{c}$ matrix F, where each cell is either (passable) or X (impassable); and F[0][0]=.
**output:** the number of different paths starting at (0,0) and ending at (r−1,c−1); where each step is either a start, right move, or down move; and does not visit any X cell

Note that if the start or end location is blocked, there are no valid paths, so the output should be 0.

## Exhaustive Search

Our first algorithm for solving this problem is exhaustive. The output definition says that the number of different paths, so this is an exhaustive search algorithm that keeps a counter and does not return after such a path is found but increment the counter instead.

The following is a first draft of the exhaustive search algorithm.

```
soccer_exhaustive(G):
  maxno = total number of different paths originating at (0,0) and ending at (r − 1, c − 1)
  counter = 0 (number of valid paths in G)
  for len from 0 to maxno inclusive:
     for each possible sequence S of {→, ↓} encoded as len:
        candidate = [start] + S
        if candidate is valid:
           counter++
  return counter
```

This is not quite clear, because the precise value of *maxno*, method of generating the sequences S, and verifying candidates, are all vague.

Since all paths start at $(0,0)$ and the only valid moves are right and down, valid paths are never backward or upward. Hence, any valid path must reach the bottom-right corner of the grid. The grid has $r$ rows and $c$ columns, so this path involves $(r-1)$ down moves and $(c-1)$ right moves, for a total of $n = r + c - 2$ moves.
There are two kinds of move, down ↓ and right →. The two kinds of bits, 0 and 1 may represent these. So, we can generate move sequences by generating bit strings. 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 soccer avoidance problem. That means that the path stays inside the grid, and never crosses an opponent (X) cell.

Combining these ideas, a better and clearer algorithm is obtained.

```
soccer_exhaustive(G):
    len = r + c − 2
    counter = 0
    for bits from 0 to 2^len − 1 inclusive:
       candidate = empty list of moves
       for k from 0 to len − 1 inclusive:
          bit = (bits >> k) & 1
          if bit == 1:
             candidate.add(→)
          else:
             candidate.add(↓)
       if candidate stays inside the grid, never crosses an X cell, and ends at (r − 1, c − 1):
          counter++
     return counter
```

Let $n = r + c - 2$. Then the initial for loop repeats $0(2^n)$ times, and the inner for loops repeats $O(n)$ times, and the total run time of this algorithm is $O(n \cdot 2^n)$. This is a very slow algorithm. The $2^n - 1$ expression could cause an integer overflow bug when $2^n$ is too large.

## Dynamic Programming

This problem can also be solved using a dynamic programming approach. The dynamic programming $matrix\ A$ stores partial solutions to the problem. In particular,
$A[i][j]$ = the number of different valid paths that start at (0,0) and end at $(i, j)$; or 0 if $(i, j)$ is unreachable.

Recall that in this problem, some cells are filled with Blue Team players and are therefore unreachable by a valid path. The base case is the value for $A[0][0]$, which is the trivial path that starts at (0,0) and takes no subsequent steps: $A[0][0] = 1$. A solution for a general case can be built based on pre-existing shorter paths. The Red Team player can only move right and down. So, there are two ways a player can reach location $(i, j)$.

1. A path reaches the location above $(i, j)$, then moves down.
2. A path reaches the location left of $(i, j)$, then moves right.

The algorithm should count both alternatives. 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 passable. Symmetrically, the from-left path (2) only exists when we are not on the leftmost column (so when $j > 0$) and when the cell to the left of $(i, j)$ is passable.

Finally, observe that $A[i][j]$ must be 0 when $F[i][j] = X$, because a path to $(i, j)$ is only possible when $(i, j)$ is passable.

Therefore, the general solution is:

$A[i][j] = 0,$ if $F[i][j] = X$

$A[i][j] = above + left$ where

above $= 0$ $if\ i = 0$ or $F[i-1][j] = X$, or $A[i-1][j]$ otherwise

$left = 0$ $if\ i = 0$ $or\ F[i][j-1] = X,$ $or\ A[i][j-1]$ $otherwise$ otherwise ($F[i][j] = .$)

Putting these parts together yields a complete dynamic programming algorithm:

```
soccer_dyn_prog(F):
    //corner case: initial cell is impassible
```

```
if F[0][0] == X:
    return 0
A = new r×c matrix initialized to zeroes
//base case
A[0][0] = 1
//general cases
for i from 0 to r − 1 inclusive:
  for j from 0 to c − 1 inclusive:
    if F[i][j] == X:
      A[i][j] = 0
      continue
    above = from_left = 0
    if i > 0 and F[i − 1][j] == . :
      above = A[i − 1][j]
    if j > 0 and F[i][j − 1] == . :
      left = A[i][j − 1]
    A[i][j] += above + left
return A[r − 1][c − 1]
```

The time complexity of this algorithm is dominated by the general-case loops. For the sake of this analysis, let $n = max(r, c)$. The outer loop repeats $O(n)$ times, the inner loop repeats $O(n)$ times, and each iteration takes $O(1)$ time, so the total time is $O(n^2)$. While $O(n^2)$ is not the fastest time complexity, it is polynomial, and therefore considered tractable. It is drastically faster than the exhaustive algorithm.

# Implementation

1. Add your name or group members name (and email addresses) to a Readme file. The file should also contain instructions on how to run your submitted codes. The Readme file can be a simple .txt or .docx file.
2. Next, implement both algorithms in either C++ or Python.
3. Your codes should be well commented, including names of member(s).

## Empirical Analysis

After you have finalized your code, conduct an empirical analysis of each of the two algorithms. For each algorithm:

1. Gather empirical timing data by running your implementations for various values of $n$. Use a C++ or Python timing program. You will need enough data points to establish the shape of the best-fit curve (at least 6 data points or more), and you should use $n$ sizes that are large enough to produce large time values (multiple seconds or even minutes) to minimize instrumental error.
2. Draw a scatter plot and fit line for your timing data. You could use a spreadsheet program (e.g. Excel, Google Sheets, Numbers, OpenOffice). The instance size $n$ should be on the horizontal axis and elapsed time should be on the vertical axis. Your plot should have a title; and each axis should have a label and units of measure.
3. Conclude whether or not your empirically-observed time efficiency data is consistent, or inconsistent, with the big-O efficiency classes predicted above.

## Report Document

Produce a brief written project report in PDF format. Your report should include the following:

a. A title, indicating that the report is about project 3; with the name and CSUF-issued email address of every team member.
b. Scatter plots for each of the two algorithms.
c. Answers to the following questions. (Each answer should be at least one complete sentence.)
    I. Is there a noticeable difference in the performance of the two algorithms?
    II. According to your experimental observation, which of the implementation is faster, and by how much?
    III. Are your empirical analyses consistent with the predicted big-O efficiency class for each algorithm? Justify your answer.
    IV. Is this evidence consistent or inconsistent with hypothesis 1? Justify your answer.
    V. Is this evidence consistent or inconsistent with hypothesis 2? Justify your answer.

## Grading Rubric

The suggested grading rubric is given below:

Report (40)

a. Answers to questions I. – V above = 15 (3 points each)
b. Readme file with detailed instructions, names and email addresses = 5 points
c. Scatter plots for the algorithms = 20 ( 10 points each)

Algorithm 1 (30 points)

a. Clear and complete codes with comments and exponential complexity =5 points
b. Successful compilation of codes= 15 points
c. Produces accurate results = 10 points

Algorithm 2 (30 points)

a. Clear and complete codes with comments and quadratic complexity =5 points
b. Successful compilation of codes= 15 points
c. Produces accurate results = 10 points

## Submitting your code

Submit your files to the Project 3 Assignment on Canvas. It allows for multiple submissions. Your codes should be submitted in their executable extensions (.py or .cpp), and report in PDF. All files should be submitted **separately**. Do not zip or use .rar.

Ensure your submissions are your own works. Be advised that your submissions may be checked for plagiarism using automated tools. Do not plagiarize. As stated in the syllabus, a submission that involves academic dishonesty will receive a 0% score on that assignment. A repeat offense will result in an "F" in the class and the incident will be reported to the Office of Student Conduct.

## Deadline

The project deadline is **Friday, April 21, by 11:59 pm** on Canvas.
Penalty for late submission (within 48 hours) is as stated in the syllabus. Projects submitted more than 48 hours after the deadline will not be accepted.