Dynamic Programming Project

November 4, 2016

Authors:

1. Hanna Barringer
2. Saad Elbeleidy
3. Austin Leo

1 Problem

The Astronomy club is faced with the following algorithmic problem. There are `n` consecutive astronomical events they could observe on a particular night that occur exactly one minute apart. Thus event `j` occurs at minute j. Also, event `j` occurs at integer coordinate `d_j` in the sky (we're assuming the sky is one-dimensional). The telescope's initial position at minute 0 is assumed to be coordinate 0 and the club is required to observe the last event `n` (occurring at minute `n`). The catch here is that the telescope can only be moved one coordinate per minute. So, at minute 1, the telescope can be moved to coordinate location 1 or -1 (or it could remain at location 0).

The optimization problem you have to solve is: given the coordinates of each of the `n` events, find a viewable subset of maximum size, subject to the requirement that it should contain event `n`.

Example:

In the example below, the optimal solution is to observe events `{1,3,6,9}`. Note that the telescope has time to move from one event in this set to the next event moving at one coordinate location per minute:

Event	1	2	3	4	5	6	7	8	9
Coordinate	1	-4	-1	4	5	-4	6	7	-2

2 Solution Approach

The algorithm begins by first by creating two lists, *allPossibleFromStart* and *allPossibleFromEnd*, which will be used to store the possible solutions generated. Generating the lists begins by taking a subset of the list of events C starting with the first element and adding the next element each iteration.

2.1 Initial List Generation

Given a sample set C of [0, 1, 2, 3]:

```
Subset 1: [0]
Subset 2: [0, 1]
Subset 3: [0, 1, 2]
Subset 4: [0, 1, 2, 3]
```

allPossibleFromStart will begin at the first element of the subset and begin at the initial coordinate position of the telescope, 0. The telescope may only move at maximum 1 coordinate per minute and the iteration of each subset will check that the proceding elements can be reached by the telescope in time. If the telescope can possibly see the event it is appeneded to the current list as `True`. Once the subset has been processed the current list is appeneded to *allPossibleFromStart*.

For example, the telescope begins at position 0 and may only reach either coordinate -1 or 1 by the next minute. By minute 2, the telescope can see events from coordinate -2 to 2 and if the event at this time is within these bounds it will be `True`. However, if there is an event *j* at coordinate 5 by minute 3 the telescope cannot possibly reach *j* from its starting position and therefore will have a `False` added.

allPossibleFromEnd will repeat this process, but rather than starting from the first element and the telescope's initial position of 0, it will begin at the position and coordinate of the last event in the subset. This will guarantee that the last event in the subset is observed.

Once both lists have been generated they are *AND*ed together and assigned to a new list *allPossibleFromStartAndEnd*. This list will contain all solutions of events that can be reached by the telescope in time and additionally observe the last event.

2.2 Determining The Solution From List

From *allPossibleFromStartAndEnd*, a list of the *bestCases* of each solution will be generated. This is where the recursive relation occurs. Using *allPossibleFromStartAndEnd* we take each list *s* and create a new list that is the length *l* of *s* of *l - 1* `False`s and a single `True` at the end. This signifies that the last astronomical event has been seen. We then iterate through *s* starting from the end moving towards the front. As we move towards the front we will add the elements to our best case list for that subset. If a `True` is seen at position *i* we will follow a new procedure.

Keep in mind that the *bestCases* lists all the best cases for the subset of that length. We will compare our current subset to the corresponding *bestCases* subset of length *i* and if the element in *s* is `True` at *i* the remaining elements in *s* will be replaced by the *bestCases* subset. By storing the best case of each subset we can avoid recalculating the remaining elements of *s* if we reach a `True` by substituting in the best case subset of that length. This subset is added as the next list in our *bestCases* until we have fully processed *allPossibleFromStartAndEnd*. The final best case is the solution.

3 Psudocode

Assume *C* is the list of coordinates for the astronomical events with the indices *j* being the minute they each occur at.

```
global allPossibleFromStart
global allPossibleFromEnd

def getAllPossibilites(C):
    n = length(C)
    possibleStart = []
    for i = 1 to n:
        check to see if event C[i] is within abs(i) of the starting position
        if it is:
            possibleStart.append(True)
        else:
            possibleStart.append(False)

    allPossibleFromStart = possibleStart

    for i = 1 to n:
        check to see if event C[i] is within abs(n-i) of the end event position
        if it is:
            possibleEnd.append(True)
        else:
            possibleEnd.append(False)

        allPossibleFromEnd.append(possibleEnd)

        for p = 1 to len(possibleEnd)
            if possibleEnd[p] == true
                getAllPossibilites(possibleEnd(C[:p]))
                #Recursively call with fewer elements of C


getAllPossibilities(C)

compare allPossibleFromStart and allPossibleFromEnd
allPossibleFromStartAndEnd = elements in common of the two
bestCases = list of n elements for each subset

for index s, subset subgroup in allPossibleFromStartAndEnd:
    if bestCases has not been initialized:
        bestCases[s] = list of s Falses + True
        # No events have been seen except the last

        for index e, subset event in subgroup:
            if we have not reached the end and an event has been witnessed:
                substitute elements from the previous bestCases subset
                of length e into current subset

                if there are more events witnessed in current subset:
                    bestCases[s] = current subset

return the last element in bestCases
```

4 Traceback Algorithm

Based on our implementation, the traceback algorithm is simply the reverse of the solution that is returned. Since solutions are developed by copying the previous item's best path, every "True" value is on the path of how to reach the solution. It is therefore the traceback path to the solution.

5 Complexity Analysis

The algorithm is split into the following components:

1. List Generation O(n^2)
    1. Reachable From Start (O(n))
    2. Reachable From End (O(n^2))
    3. Combining Reachable States (O(n^2))
2. Solving (O(n^2))

These components result in a combined complexity of the algorithm at O(n^2).

Details about the breakdown of the complexity for each component are available in the comments of the implementation section below.

6 Implementation

```
In [64]: allPossibleFromEnd = []
         allPossibleFromStart = []

         def reset():
             global allPossibleFromEnd, allPossibleFromStart
             allPossibleFromEnd = []
             allPossibleFromStart = []


         def getPossibilities(C,debug=0): # Total of O(n^2)
         #     Get the possible values that are reachable from start and end
             n = len(C)
             minsEnd = [C[-1]-(n-i-1) for i in range(n-1)] + [C[-1]] # O(n)
             maxsEnd = [C[-1]+(n-i-1) for i in range(n-1)] + [C[-1]] # O(n)
             minsStart = [-i for i in range(1,n+1)]
             maxsStart = list(range(1,n+1))

             if debug:
                 print(minsEnd)
                 print(C)
                 print(maxsEnd)

                 print(minsStart)
                 print(C)
```

```
        print(maxsStart)

    possibleEnd = [(C[i]>=minsEnd[i]
                    and C[i]<=maxsEnd[i])
                   for i in range(n)] # O(n)

    if debug:
        print(possibleEnd)

    global allPossibleFromEnd
    if allPossibleFromEnd == []:
        allPossibleFromEnd = [0]* n

    global allPossibleFromStart
    if allPossibleFromStart == []:
        allPossibleFromStart = [(C[i]>=minsStart[i]
                                 and C[i]<=maxsStart[i])
                                for i in range(n)]

    allPossibleFromEnd[n-1] = possibleEnd

    for p in range(1,len(possibleEnd)):
        if possibleEnd[p]:
            getPossibilities(C[:p])
            # This part leads to the O(n^2) portion


def solve(C, debug=0): # Total of O(n^2)
    n = len(C)
    getPossibilities(C, debug)
    global allPossibleFromStart, allPossibleFromEnd

    if debug:
        print("All possible form start")
        print(allPossibleFromStart)
        print()
        print("All possible from end")
        for l in allPossibleFromEnd:
            print(l)
        print()

#     Create the table of all posibilities of
#     being reached from both start and end
    possibleFromStartAndEnd = [[(allPossibleFromStart[i]
                                 and allPossibleFromEnd[j][i])
                                for i in range(j+1)]
                               for j in range(n)] # O(n^2)
    if debug:
        print("Possible from start and end")
        for l in possibleFromStartAndEnd:
            print(l)
        print()

        scores = [sum(i) for i in possibleFromStartAndEnd]
        print(scores)
        print()

#       Find all possible solutions

    bestCases = [0]*n

    # This portion takes O(n^2)
    for s, subgroup in enumerate(possibleFromStartAndEnd):
        if bestCases[s] == 0:
            bestCases[s] = [False]*(s) + [True]
        for e, event in enumerate(subgroup):
            if e < len(subgroup)-1 and event:
                if bestCases[e] == 0:
                    bestCases[e] = [False]*(e) + [True]

                potentialSolution = bestCases[e] + [False]*(s-e-1) \
                + [True]

                if sum(potentialSolution) > sum(bestCases[s]):
                    bestCases[s] = potentialSolution


    solution = [i+1 for i in range(n-1) if bestCases[-1][i]] + [n]

    traceback = list(solution)
    traceback.reverse()

    if debug:
        print("Solution:")
        print(solution)
        print()
        print("Traceback:")
        print(traceback)
        print()

    return solution
```

6.1 Implementation Examples

```
In [65]: print(solve([1,-4,-1,4,5,-4,6,7,-2]))
         reset()

[1, 3, 6, 9]


In [66]: print(solve([1,2,3,4,5,6,7,8,9]))
         reset()

[1, 2, 3, 4, 5, 6, 7, 8, 9]


In [67]: print(solve([1,2,-3,4,-5,6,-7,8,9]))
         reset()

[1, 2, 4, 6, 8, 9]


In [68]: print(solve([0,2,3,4,5,6,7,8,9]))
         reset()

[2, 3, 4, 5, 6, 7, 8, 9]


In [69]: print(solve([1,1,2,3,4,5,6,7,1]))
         reset()

[1, 2, 3, 4, 5, 9]


In [70]: solve([1,-1,2,-2,3,-3,2,-2,1,-1,0],1)
         reset()

[-10, -9, -8, -7, -6, -5, -4, -3, -2, -1, 0]
[1, -1, 2, -2, 3, -3, 2, -2, 1, -1, 0]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
[-1, -2, -3, -4, -5, -6, -7, -8, -9, -10, -11]
[1, -1, 2, -2, 3, -3, 2, -2, 1, -1, 0]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
[True, True, True, True, True, True, True, True, True, True, True]
All possible form start
[True, True, True, True, True, True, True, True, True, True, True]

All possible from end
[True]
[False, True]
[True, False, True]
[True, True, False, True]
```

```
[True, False, True, False, True]
[True, True, False, True, False, True]
[True, True, True, False, True, False, True]
[True, True, True, True, False, True, False, True]
[True, True, True, True, True, False, True, False, True]
[True, True, True, True, True, True, True, True, False, True]
[True, True, True, True, True, True, True, True, True, True, True]

Possible from start and end
[True]
[False, True]
[True, False, True]
[True, True, False, True]
[True, False, True, False, True]
[True, True, False, True, False, True]
[True, True, True, False, True, False, True]
[True, True, True, True, False, True, False, True]
[True, True, True, True, True, False, True, False, True]
[True, True, True, True, True, True, True, True, False, True]
[True, True, True, True, True, True, True, True, True, True, True]

[1, 1, 2, 3, 3, 4, 5, 6, 7, 9, 11]

Solution:
[1, 3, 5, 7, 9, 11]

Traceback:
[11, 9, 7, 5, 3, 1]
```

```
In [71]: solve([1,-4,-1,4,5,-4,6,7,-2],1)
         reset()
```

```
[-10, -9, -8, -7, -6, -5, -4, -3, -2]
[1, -4, -1, 4, 5, -4, 6, 7, -2]
[6, 5, 4, 3, 2, 1, 0, -1, -2]
[-1, -2, -3, -4, -5, -6, -7, -8, -9]
[1, -4, -1, 4, 5, -4, 6, 7, -2]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
[True, True, True, False, False, True, False, False, True]
All possible form start
[True, False, True, True, True, True, True, True, True]

All possible from end
[True]
[False, True]
[True, False, True]
```

```
[True, False, False, True]
[True, False, False, True, True]
[True, True, True, False, False, True]
[True, False, False, True, True, False, True]
[True, False, False, True, True, False, True, True]
[True, True, True, False, False, True, False, False, True]

Possible from start and end
[True]
[False, False]
[True, False, True]
[True, False, False, True]
[True, False, False, True, True]
[True, False, True, False, False, True]
[True, False, False, True, True, False, True]
[True, False, False, True, True, False, True, True]
[True, False, True, False, False, True, False, False, True]

[1, 0, 2, 2, 3, 3, 4, 5, 4]

Solution:
[1, 3, 6, 9]

Traceback:
[9, 6, 3, 1]
```