CS3383 Unit 3.2: Dynamic Programming Examples

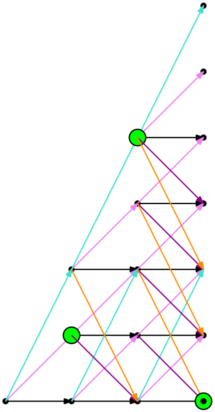
David Bremner

March 10, 2024

Dynamic Programming

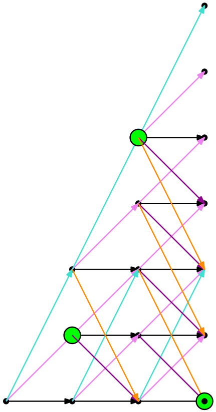
- Balloon Flight Planning
- Longest Increasing Subsequence
- Edit Distance

Balloon Flight Planning



- Start at $(0, 0)$
- every step, rise or fall up to k steps, and increase x by 1.
- one prize per integer $x > 0$.
- discretize the problem as a graph search

Big Graph is Big



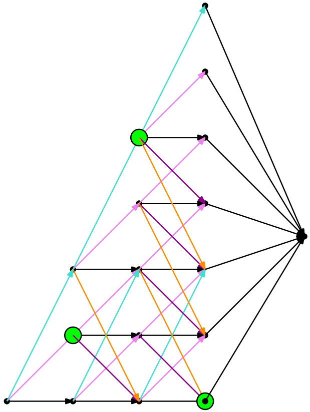
- computed graph is $\Omega(kn)$
- input coordinates $O(n \log n + n \log k)$.
- bad dependence on k; more later

Finding a maximum value path

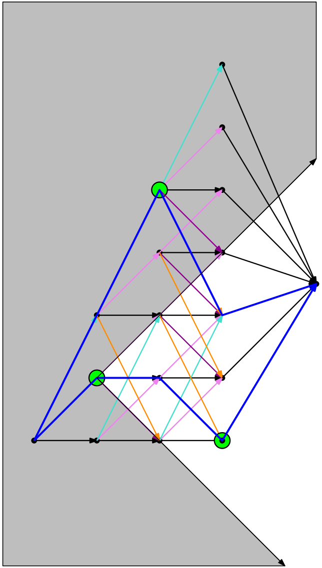
An easy case of a hard problem

In general NP-Hard, but not in DAGs.

function $\textsc{BestPath}(V, E)$
    for $v \in \mathsf{TopSort}(V)$ **do**
        Score[v] $= -\infty$ // unreachable
        for $(u, v) \in E$ **do** // incoming edges
            Score[v] $=$ max(Score[v],
                    Value[v]$+$Score[u])
        end for
    end for
end function



Straightening paths



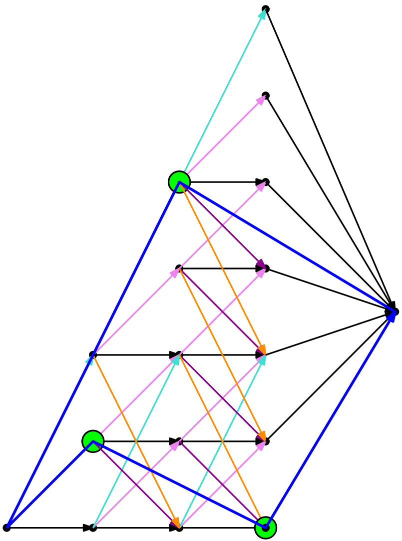
Lemma (Straightening Paths)

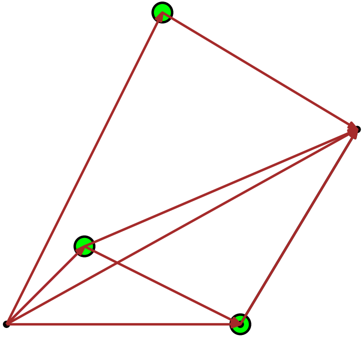
If there is a feasible path from p to q then the segment $[p, q]$ is feasible.

Proof

The path cannot escape the cone define by the steepest possible segments.

A new graph





Improved graph size

The new graph is $O(p^2)$, where $p \leq n$ is the number of prizes.

Longest increasing subsequence problem

Input Integers $a_1, a_2 \dots a_n$

Output

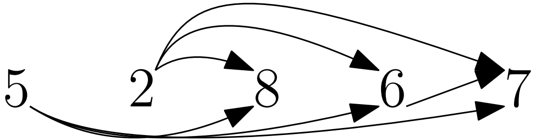
$$a_{i_1}, a_{i_2}, \dots a_{i_k}$$

Such that

$$i_1 < i_2 \cdots < i_k$$

and

$$a_{i_1} < a_{i_2} < \cdots < a_{i_k}$$



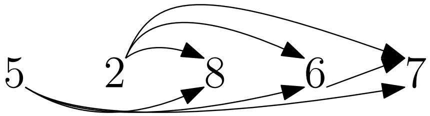
- $(a_i, a_j) \in E$ if $i < j$ and $a_i < a_j$.
- DPV 6.2, JE 3.6

Defining subproblems

- Define $F(i)$ as the length of longest sequence starting at position i
- We could do n longest path in DAG queries.
- Thinking recursively:

$$F(i) = 1 + \max\{F(j) \mid (i,j) \in E\}$$

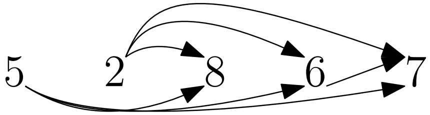
- We could solve this reasonably fast e.g. by memoization.



- Topological sort is trivial

Longest path in DAG, working backwards

- Define $L[i]$ as the longest path *ending* at a_i




```
For i = 1…n:
   L[i] = 1 + max { L(j) | (j,i) in E }
```

- total cost is $O(|E|)$, *after* computing E.

Improving memory use

- We can inline the definition of E.
- $L(i) = 1 + \max\{L(j) \mid j < i \text{ and } a_j < a_i\}$

```
def lis(A):
  n = len(A)
  L = [1] * n
  for i in range(n):
    for j in range(i):
      if A[j] < A[i]:
        L[i] = max(L[i],L[j]+1)
  return max(L)
```

Edit (Levenshtein) Distance

- DPV 6.3, JE3.7
- Minimum number of insertions, deletions, substitutions to transform one string into another.

Example: timberlake $\longrightarrow$ fruitcake

- non optimal solution

```
i i i i   d d d d d s
_ _ _ _ T I M B E R L A K E
F R U I T _ _ _ _ _ C A K E
```

Total cost 10.

Alignments (gap representation)

```
1 1 1 1 0 1 1 1 1 1 1 0 0 0
_ _ _ _ T I M B E R L A K E
F R U I T _ _ _ _ _ C A K E
```

- top line has letters from A, in order, or _
- bottom line has has letters from B or _
- cost per column is 0 or 1.

Theorem (Optimal substructure)

Removing any column from an optimal alignment, yields an opt. alignment for the remaining substrings.

Subproblems (prefixes)

- Define $E[i, j]$ as the minimum edit cost for $A[1 \ldots i]$ and $B[1 \ldots j]$

$$E[i,j] = \begin{cases} E[i, j-1] + 1 & \text{insertion} \\ E[i-1, j] + 1 & \text{deletion} \\ E[i-1, j-1] + 1 & \text{substition} \\ E[i-1, j-1] & \text{equality} \end{cases}$$

justification.

We know deleting a column removes an element from one or both strings; all edit operations cost 1. $\square$

order of subproblems

$$E[i,j] = \begin{cases} E[i-1,j]+1 & \text{deletion} \\ E[i,j-1]+1 & \text{insertion} \\ E[i-1,j-1]+1 & \text{substition} \\ E[i-1,j-1] & \text{equality} \end{cases}$$

- dependency of subproblems is *exactly* the same as LCS, so essentially the same DP algorithm works.
- or just memoize the recursion
- what are the base cases?