CS3383 Unit 3.2: Dynamic Programming Examples

David Bremner

March 10, 2024

Outline

Longest Increasing Subsequence problem

Input Integers $a_1, a_2 \ldots a_n$

Output

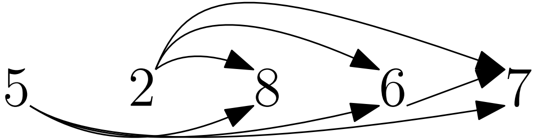
$$a_{i_1}, a_{i_2}, \ldots a_{i_k}$$

Such that

$$i_1 < i_2 \cdots < i_k$$

and

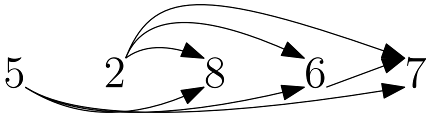
$$a_{i_1} < a_{i_2} < \cdots < a_{i_k}$$



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

Defining subproblems

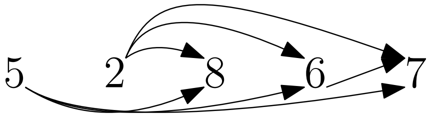
- Define $F(i)$ as the length of longest sequence starting at position i



- Topological sort is trivial

Defining subproblems

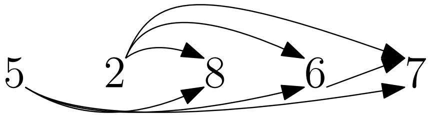
- Define $F(i)$ as the length of longest sequence starting at position i
- We could do n *longest path in DAG* queries.



- Topological sort is trivial

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:

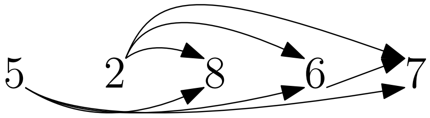


- Topological sort is trivial

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:

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



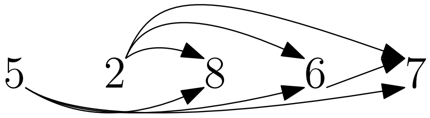
- Topological sort is trivial

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:

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

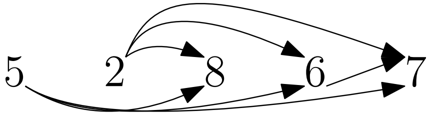
- We could solve this reasonably fast e.g. by memoization.



- 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.

```
def lis(A):
  n = len(A)
  L = [1 for j in range(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)
```

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 for j in range(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

- CLRS 14-5, DPV 6.3, JE3.7
- Minimum number of insertions, deletions, substitutions to transform one string into another.

Example: timberlake $\longrightarrow$ fruitcake

- Using mostly insertions and deletions

```
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.

Edit (Levenshtein) Distance

- CLRS 14-5, DPV 6.3, JE3.7
- Minimum number of insertions, deletions, substitutions to transform one string into another.

Example: timberlake $\longrightarrow$ fruitcake

- Using more substitutions

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

Total cost 7.

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.

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.

Alignments (gap representation)

Theorem (Optimal substructure)

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

proof.

By contradiction □

Subproblems (prefixes)

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

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

justification.

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

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.

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

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?

Edit distance

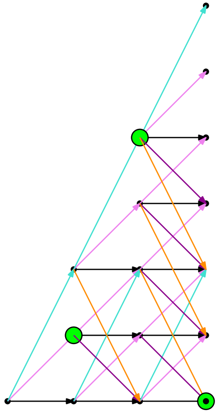
```
def dist(x,y):
  n = len(x); m=len(y)
  E = [ [max(i,j) for j in range(m+1)]
          for i in range(n+1) ]
  for i in range(1,n+1):
    for j in range(1,m+1):
      diff = int(x[i-1] != y[j-1])
      E[i][j] = min(E[i-1][j-1]+diff,
                    E[i-1][j]+1,
                    E[i][j-1]+1)
  return E
```

Tracing back the edits

```
def trace(E,x,y,i,j):
  if (i<1):
    return "i" * j;
  elif (j<1):
    return "d" * i;
  elif x[i-1] == y[j-1]:
    return trace(E,x,y,i-1,j-1)+"."
  elif E[i][j] == E[i-1][j-1] + 1:
    return trace(E,x,y,i-1,j-1)+"s"
  elif E[i][j] ==  E[i-1][j]+1:
    return trace(E,x,y,i-1,j)+ "d"
  else:
    return trace(E,x,y,i,j-1)+ "i"
```

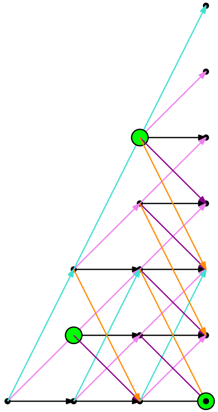
Balloon Flight Planning

- Start at $(0, 0)$



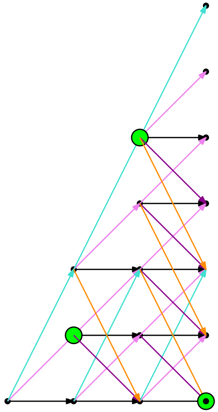
Balloon Flight Planning

- Start at $(0, 0)$
- At every time step, increase or decrease altitude up to k steps, and increase x by 1.

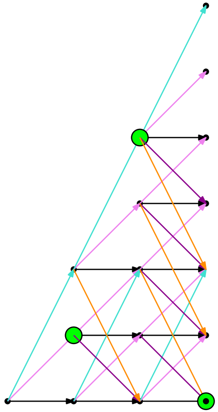


Balloon Flight Planning

- Start at $(0, 0)$
- At every time step, increase or decrease altitude up to k steps, and increase x by 1.
- There is one prize per positive integer x coordinate.



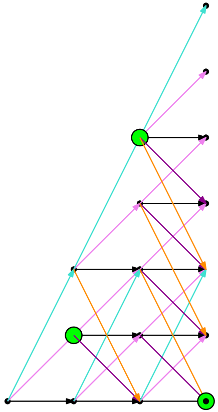
Balloon Flight Planning



- Start at $(0, 0)$
- At every time step, increase or decrease altitude up to k steps, and increase x by 1.
- There is one prize per positive integer x coordinate.
- Maximize value of collected prizes

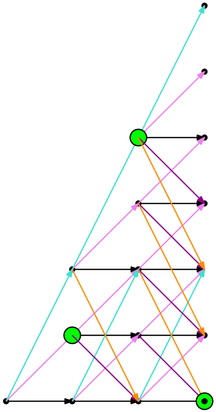
Balloon Flight Planning

- Start at $(0, 0)$
- At every time step, increase or decrease altitude up to k steps, and increase x by 1.
- There is one prize per positive integer x coordinate.
- Maximize value of collected prizes
- We can discretize/simulate the problem as a graph search



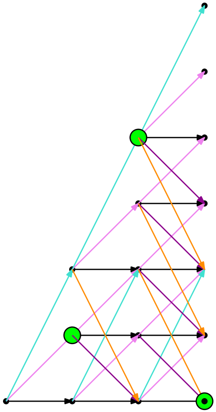
Big Graph is Big

- We can discretize/simulate the problem as a graph search



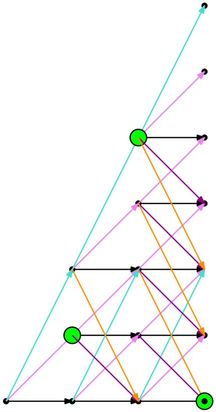
Big Graph is Big

- We can discretize/simulate the problem as a graph search
- After n steps we could reach as high as kn



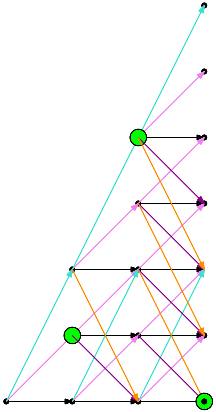
Big Graph is Big

- We can discretize/simulate the problem as a graph search
- After n steps we could reach as high as kn
- Worse, there could be a prize that high



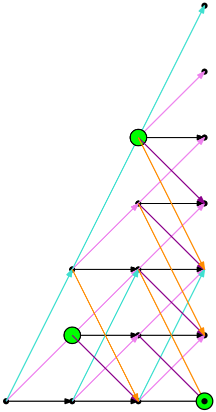
Big Graph is Big

- We can discretize/simulate the problem as a graph search
- After n steps we could reach as high as kn
- Worse, there could be a prize that high
- On the other hand the input (ignoring weights) is only $O(n \log n + n \log k)$.

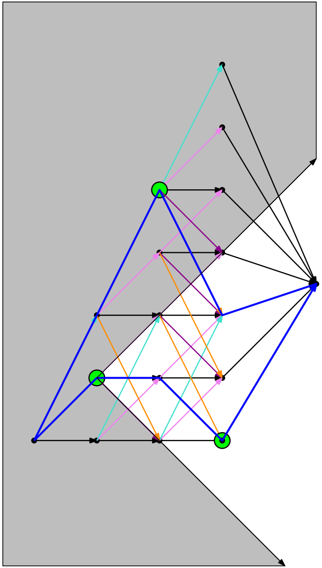


Big Graph is Big

- We can discretize/simulate the problem as a graph search
- After n steps we could reach as high as kn
- Worse, there could be a prize that high
- On the other hand the input (ignoring weights) is only $O(n \log n + n \log k)$.
- This means we have a bad dependence on k; more about this later



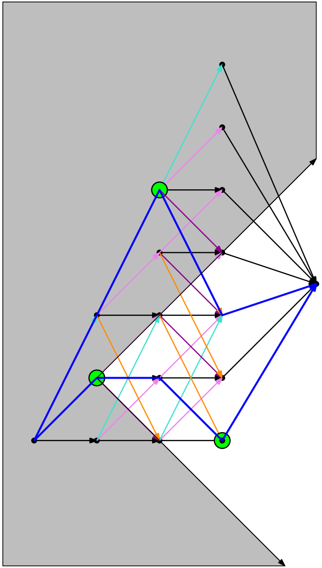
Straightening paths



Lemma (Straightening Paths)

There is a feasible path from p to q iff the segment $[p, q]$ is feasible.

Straightening paths



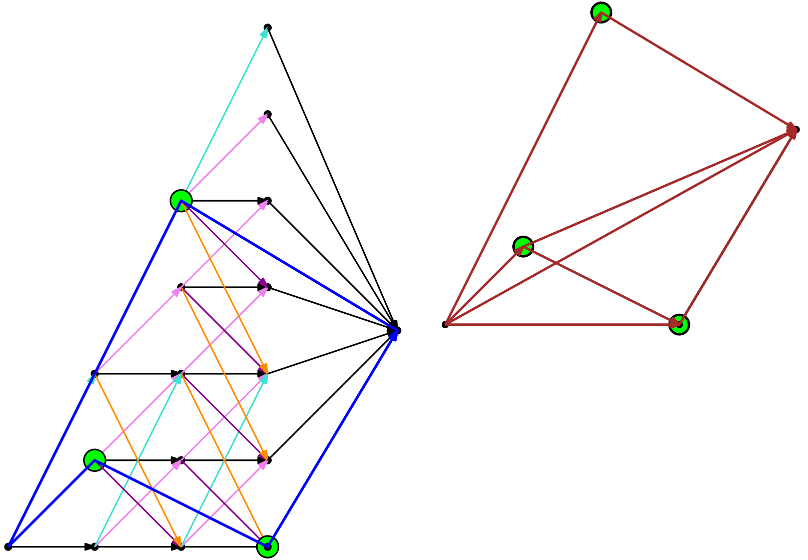
Lemma (Straightening Paths)

There is a feasible path from p to q iff the segment $[p, q]$ is feasible.

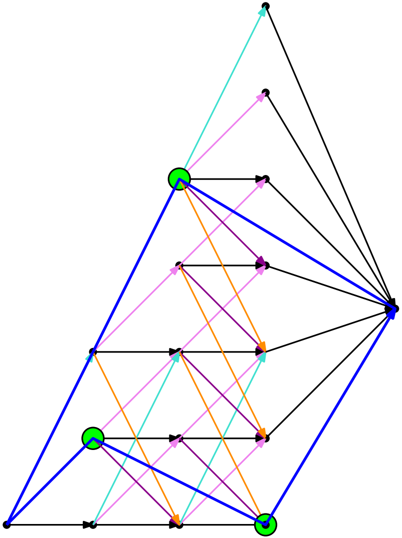
Proof sketch

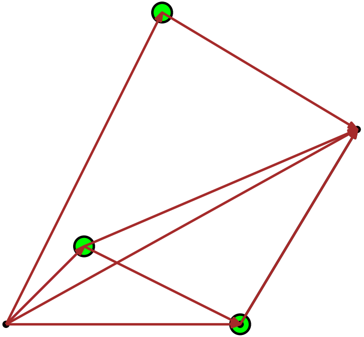
The path cannot escape the cone define by the steepest possible segments.
There is always one step back towards start within cone. Apply induction.

A new graph



A new graph





Improved graph size

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