Algorithm Design & Analysis (CS3383)[1]

Unit 4 : Dynamic Programming

Rasoul Shahsavarifar

March 11, 2019

[1]Thanks to Dr. Ptricia Evans and Dr. David Bremner at UNB, Dr. Erik Demaine at MIT for sharing the teaching stuffs

Outline

Dynamic Programming and Examples

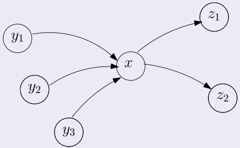
- Shortest path in Directed Acyclic Graph (DAG)
- Longest increasing subsequence
- Longest Common Subsequence
- Edit Distance
- Balloon Flight Planning

Contents

Directed Acyclic Graph (DAG)

A DAG is a directed graph ($G = (V, E)$), where $E = \{< u, v > \ \mid\ u, v \in V\}$, that has no cycle. Similar to tree, but not necessary connected,...

The vertices in DAG can be ordered topologically, so that all edges go in the same direction. Essentially, we can list the vertices so that all vertices that "precode" a vertex x come before x in the order.



More discussion on DAG

A DAG is often used to represent a set of tasks and their relationships. An edge $< u, v >$ indicates that task v uses the result of task u. We can find the topological order by: repeatedly finding a vertex u with no incoming edge, making u the next vertex in the order removing u from the (copy of) graph.

Application & Example

- (DAG is the same as Recursion tree) for some algorithms. e.g. Mergesort. See the tree(board)
- How about DAG vs. Recursion tree for Fibonacci recursion relation (board)

Shortest Paths in DAGs

- Every path in a DAG goes through nodes in linearized (topological sort) order.

Shortest Paths in DAGs

- Every path in a DAG goes through nodes in linearized (topological sort) order.
- *every node is reached via its predecessors*

Shortest Paths in DAGs

- Every path in a DAG goes through nodes in linearized (topological sort) order.
- *every node is reached via its predecessors*
- So we need a single loop after sorting.

Shortest Paths in DAGs

- Every path in a DAG goes through nodes in linearized (topological sort) order.
- *every node is reached via its predecessors*
- So we need a single loop after sorting.

Dist in Topological Sorted Graph

- Initialize all$\mathrm{dist}(*) = \infty$
- $\mathrm{dist}(s) = 0$
- foreach $v \in V\ \{s\}$ in top. sort order
- $\quad \mathrm{dist}(v) = \min_{(u,v) \in E} \mathrm{dist}(u) + l(u, v)$

So what does this have to do with Dynamic Programming?

Ordered Subproblems

In order to solve our problem in a single pass, we need

- An ordered set of subproblems $L(i)$
- See Figure 6.1 in DPU Textbook

So what does this have to do with Dynamic Programming?

Ordered Subproblems

In order to solve our problem in a single pass, we need

- An ordered set of subproblems $L(i)$
- Each subproblem $L(i)$ can be solved using only the answers for $L(j)$, for $j < i$.
- See Figure 6.1 in DPU Textbook

Contents

Longest increasing subsequence problem

Input Integers $a_1, a_2 \dots a_n$

Output

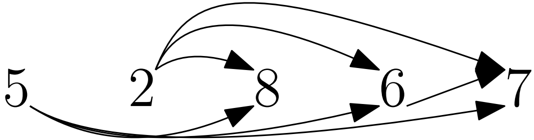
$$a_{i_1}, a_{i_2}, \dots a_{i_k}$$

Such that

$$i_1 < i_2 \dots < i_k$$

and

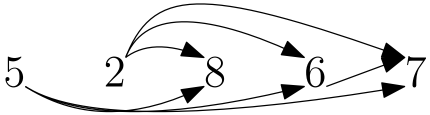
$$a_{i_1} < a_{i_2} < \dots < a_{i_k}$$



- $(a_i, a_j) \in E$ if $i < j$ and $a_i < a_j$.
- DPV 6.2, JE 5.2

Defining subproblems

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



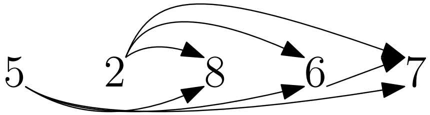
$$F(i) = 1 + \max\{F(j) \mid (i,j) \in E\}$$

- 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.

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

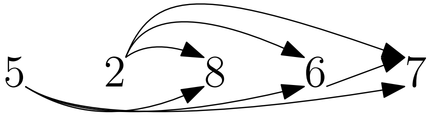


- Topological sort is trivial

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:

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



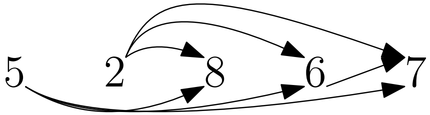
- Topological sort is trivial

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:

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

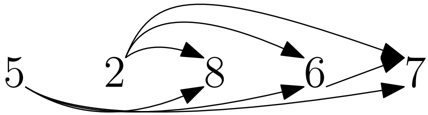
- We could solve this reasonably fast e.g. by memoization.



- 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.

Improving memory use

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

Improving memory use

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

function $\mathrm{LIS}(a_1 \dots a_n)$
    $\forall i \; L[i] = -\infty$
    for $i \in 1 \dots n$ **do**
        for $j \in 1 \dots i-1$ **do**
            if $a_j < a_i$ **then**
                $L[i] \leftarrow \max(L[i], L[j] + 1)$
            end if
        end for
    end for
    return $\max(L[1] \dots L[n])$
end function

LIC Cont.

Another key to dynamic programming is *subproblem reuse* – a subproblem result forms part of more than one later calculation, so we save the results for reuse.

Question:

How can we also determine the subsequence itself?

LIC Cont.

Another key to dynamic programming is *subproblem reuse* – a subproblem result forms part of more than one later calculation, so we save the results for reuse.

Question:

How can we also determine the subsequence itself?

A possible Answer:

We can save the best previous value (what subproblem actually formed part of the result), and trace back from the optimum answer. In short, design a "trace" procedure.

Another pseudocode with "trace" for LIC

$opt \leftarrow 1$
$last \leftarrow 1$
for j from 1 to n
$L[j] \leftarrow 1$
$p[j] \leftarrow 0$
for i from 1 to $j-1$
    if $(a_i < a_j)$ and $(L[j] < 1 + L[i])$
        $L[j] \leftarrow 1 + L[i]$
        $p[j] \leftarrow i$
if $opt < L[j]$
    $opt \leftarrow L[j]$
    $last \leftarrow j$
return $trace(last)$

$trace(j)$: returns sequence
  $S \leftarrow$ empty sequence
  if $j > 1$
    $S \leftarrow trace(p[j])$
  append a_j to S
  return S

Contents

Ordering Subproblems

- In the two problems we saw so far, the DAG of subproblem dependence was defined by time.
- In general this need not be the case; a very natural way of deriving this DAG is from a recursive algorithm.
- We'll explore this strategy with the Longest Common Subsequence problem.

Dynamic programming

Design technique, like divide-and-conquer.

Example: ***Longest Common Subsequence (LCS)***

- Given two sequences $x[1 \,.\,.\, m]$ and $y[1 \,.\,.\, n]$, find a longest subsequence common to them both.

Dynamic programming

Design technique, like divide-and-conquer.

Example: ***Longest Common Subsequence (LCS)***

- Given two sequences $x[1\,.\,.\,m]$ and $y[1\,.\,.\,n]$, find a longest subsequence common to them both.

"a" *not* "the"

Dynamic programming

Design technique, like divide-and-conquer.

Example: *Longest Common Subsequence (LCS)*

- Given two sequences $x[1 \, . \, . \, m]$ and $y[1 \, . \, . \, n]$, find a longest subsequence common to them both.

"a" *not* "the"

x: A B C B D A B

y: B D C A B A

Dynamic programming

Design technique, like divide-and-conquer.

Example: *Longest Common Subsequence (LCS)*

- Given two sequences $x[1 \,.\,.\, m]$ and $y[1 \,.\,.\, n]$, find a longest subsequence common to them both.

"a" *not* "the"

x: A B C B D A B

y: B D C A B A

BCBA = LCS(x, y)

functional notation, but not a function

Brute-force LCS algorithm

Check every subsequence of $x[1 \, . \, . \, m]$ to see if it is also a subsequence of $y[1 \, . \, . \, n]$.

Brute-force LCS algorithm

Check every subsequence of $x[1 \,.\,.\, m]$ to see if it is also a subsequence of $y[1 \,.\,.\, n]$.

Analysis

- Checking $= O(n)$ time per subsequence.
- 2^m subsequences of x (each bit-vector of length m determines a distinct subsequence of x).

Worst-case running time $= O(n2^m)$

$= \text{exponential time.}$

Pruning subproblems

- Part (but only part) of the problem is that the brute force algorithm considers many sequences that can't possibly be the maximal one.
- In order to recursively compute an optimal answer, an obvious strategy is to compute answers that are optimal for some subset of the input
- Unlike in strong induction proofs, considering all smaller subsets is clearly a losing strategy.

Towards a better algorithm

Simplification:

1. Look at the *length* of a longest-common subsequence.
2. Extend the algorithm to find the LCS itself.

Towards a better algorithm

Simplification:

1. Look at the *length* of a longest-common subsequence.
2. Extend the algorithm to find the LCS itself.

Notation: Denote the length of a sequence s by $|s|$.

Towards a better algorithm

Simplification:

1. Look at the *length* of a longest-common subsequence.
2. Extend the algorithm to find the LCS itself.

Notation: Denote the length of a sequence s by $|s|$.

Strategy: Consider ***prefixes*** of x and y.

- Define $c[i, j] = |\text{LCS}(x[1 \,.\,.\, i], y[1 \,.\,.\, j])|$.
- Then, $c[m, n] = |\text{LCS}(x, y)|$.

Recursive formulation

Theorem.

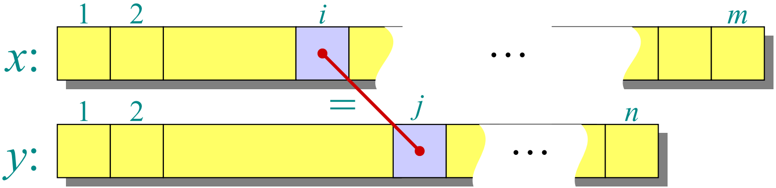
$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max\{c[i-1, j], c[i, j-1]\} & \text{otherwise.} \end{cases}$$

Recursive formulation

Theorem.

$$c[i,j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max\{c[i-1, j],\ c[i, j-1]\} & \text{otherwise.} \end{cases}$$

Proof. Case $x[i] = y[j]$:

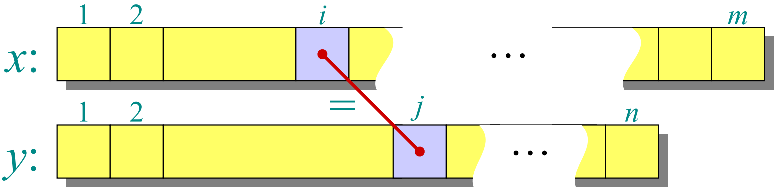


Recursive formulation

Theorem.

$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max\{c[i-1, j], c[i, j-1]\} & \text{otherwise.} \end{cases}$$

Proof. Case $x[i] = y[j]$:



Let $z[1 \,.\,.\, k] = \mathrm{LCS}(x[1 \,.\,.\, i], y[1 \,.\,.\, j])$, where $c[i, j] = k$. Then, $z[k] = x[i]$, or else z could be extended. Thus, $z[1 \,.\,.\, k-1]$ is CS of $x[1 \,.\,.\, i-1]$ and $y[1 \,.\,.\, j-1]$.

Proof (continued)

Claim: $z[1\,.\,.\,k{-}1] = \mathrm{LCS}(x[1\,.\,.\,i{-}1], y[1\,.\,.\,j{-}1])$.

Suppose w is a longer CS of $x[1\,.\,.\,i{-}1]$ and $y[1\,.\,.\,j{-}1]$, that is, $|w| > k{-}1$. Then, ***cut and paste***: $w \,||\, z[k]$ (w concatenated with $z[k]$) is a common subsequence of $x[1\,.\,.\,i]$ and $y[1\,.\,.\,j]$ with $|w \,||\, z[k]| > k$. Contradiction, proving the claim.

Proof (continued)

Claim: $z[1\,.\,.\,k-1] = \text{LCS}(x[1\,.\,.\,i-1], y[1\,.\,.\,j-1])$.

Suppose w is a longer CS of $x[1\,.\,.\,i-1]$ and $y[1\,.\,.\,j-1]$, that is, $|w| > k-1$. Then, ***cut and paste***: $w \,||\, z[k]$ (w concatenated with $z[k]$) is a common subsequence of $x[1\,.\,.\,i]$ and $y[1\,.\,.\,j]$ with $|w \,||\, z[k]| > k$. Contradiction, proving the claim.

Thus, $c[i-1, j-1] = k-1$, which implies that $c[i, j] = c[i-1, j-1] + 1$.

Other cases are similar. □

Dynamic-programming hallmark #1

Optimal substructure

An optimal solution to a problem (instance) contains optimal solutions to subproblems.

Dynamic-programming hallmark #1

Optimal substructure

An optimal solution to a problem (instance) contains optimal solutions to subproblems.

If $z = \text{LCS}(x, y)$, then any prefix of z is an LCS of a prefix of x and a prefix of y.

The trouble with recursion

- Although recursion is a useful step to a dynamic programming algorithm, naive recursion is often expensive because of repeated subproblems

Recursive algorithm for LCS

$$
\begin{array}{l}
\mathrm{LCS}(x, y, i, j) \\
\quad \textbf{if } x[i] = y[j] \\
\qquad \textbf{then } c[i, j] \leftarrow \mathrm{LCS}(x, y, i\!-\!1, j\!-\!1) + 1 \\
\qquad \textbf{else } c[i, j] \leftarrow \max\left\{ \begin{array}{l} \mathrm{LCS}(x, y, i\!-\!1, j), \\ \mathrm{LCS}(x, y, i, j\!-\!1) \end{array} \right\}
\end{array}
$$

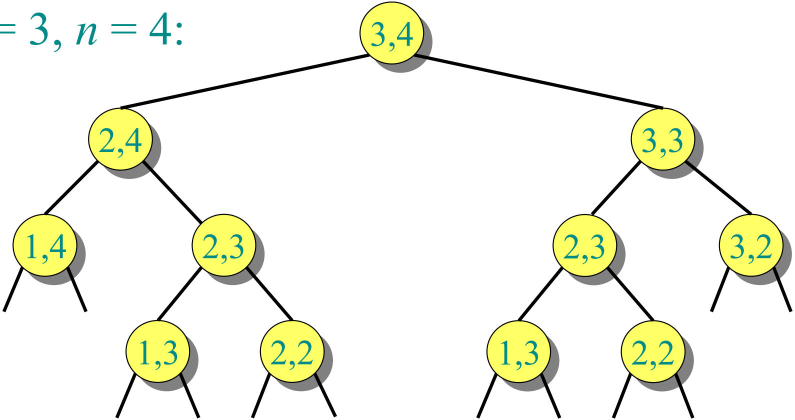
Recursive algorithm for LCS

LCS(x, y, i, j)
  if $x[i] = y[j]$
    then $c[i, j] \leftarrow$ LCS$(x, y, i-1, j-1) + 1$
    else $c[i, j] \leftarrow \max\{LCS(x, y, i-1, j)$,
            LCS$(x, y, i, j-1)\}$

Worst-case: $x[i] \neq y[j]$, in which case the algorithm evaluates two subproblems, each with only one parameter decremented.

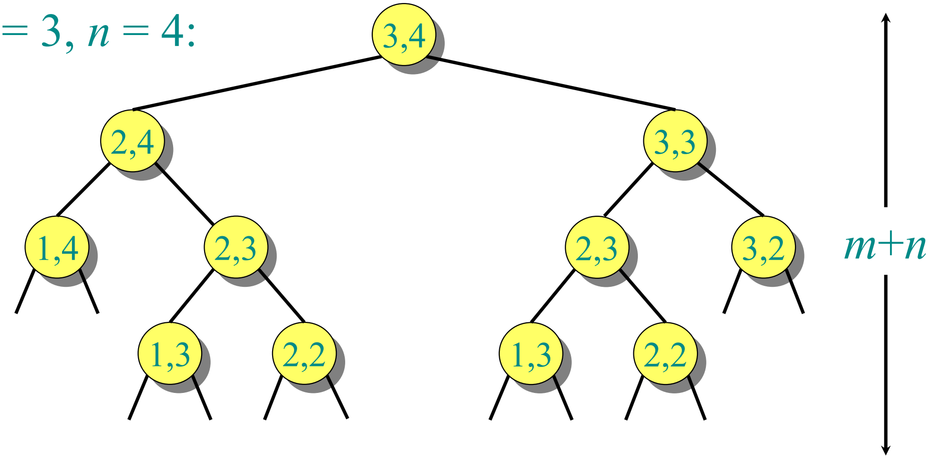
Recursion tree

$m = 3, n = 4$:



Recursion tree

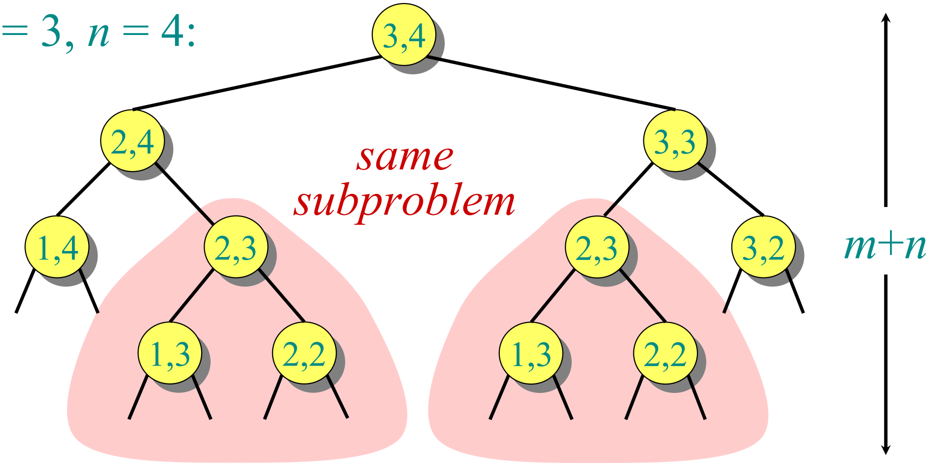
$m = 3,\ n = 4$:



Height $= m + n \Rightarrow$ work potentially exponential.

Recursion tree

$m = 3$, $n = 4$:



Height $= m + n \Rightarrow$ work potentially exponential, but we're solving subproblems already solved!

Dynamic-programming hallmark #2

Overlapping subproblems

A recursive solution contains a "small" number of distinct subproblems repeated many times.

Dynamic-programming hallmark #2

Overlapping subproblems

A recursive solution contains a "small" number of distinct subproblems repeated many times.

The number of distinct LCS subproblems for two strings of lengths m and n is only mn.

Memoization

Recursive Version

function $\textsc{Recur}(p_1, \ldots p_k)$

$\vdots$

return val

end function

Memoization

Recursive Version

function $\textsc{Recur}(p_1, \ldots p_k)$
$\vdots$
return val
end function

Memoized version

function $\textsc{Memo}(p_1, \ldots p_k)$
if cache$[p_1, \ldots p_k] \neq$ NIL **then**
return cache$[p_1, \ldots p_k]$
end if
$\vdots$
cache$[p_1, \ldots p_k] =$ val
return val
end function

Memoization algorithm

Memoization: After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

Memoization algorithm

Memoization: After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

LCS(x, y, i, j)
    if $c[i, j] = \mathrm{NIL}$
        then if $x[i] = y[j]$
            then $c[i, j] \leftarrow$ LCS$(x, y, i-1, j-1) + 1$
            else $c[i, j] \leftarrow \max\{LCS(x, y, i-1, j)$, LCS$(x, y, i, j-1)\}$

same as before

Memoized LCS (with base case)

function $\mathrm{LCS}(x, y, i, j)$
    if $(i < 1)$ **or** $(j < 1)$ **then**
        return 0
    end if
    if $c[i, j] = \mathrm{NIL}$ **then**
        if $x[i] = y[j]$ **then**
            $c[i, j] \leftarrow LCS(x, y, i-1, j-1) + 1$
        else
            $c[i, j] \leftarrow \max(LCS(x, y, i-1, j),$
                $LCS(x, y, i, j-1))$
        end if
    end if
    return $c[i, j]$

- $c[i, j]$ written at most once.

Memoized LCS (with base case)

function $\mathrm{LCS}(x, y, i, j)$
    if $(i < 1)$ **or** $(j < 1)$ **then**
        return 0
    end if
    if $c[i, j] = \mathrm{NIL}$ **then**
        if $x[i] = y[j]$ **then**
            $c[i, j] \leftarrow LCS(x, y, i - 1, j - 1) + 1$
        else
            $c[i, j] \leftarrow \max(LCS(x, y, i - 1, j),$
                $LCS(x, y, i, j - 1))$
        end if
    end if
    return $c[i, j]$

- $c[i, j]$ written at most once.
- returned value written immediately

Memoized LCS (with base case)

function $\mathrm{LCS}(x, y, i, j)$
  if $(i < 1)$ **or** $(j < 1)$ **then**
    return 0
  end if
  if $c[i, j] = \mathrm{NIL}$ **then**
    if $x[i] = y[j]$ **then**
      $c[i, j] \leftarrow LCS(x, y, i-1, j-1) + 1$
    else
      $c[i, j] \leftarrow \max(LCS(x, y, i-1, j),$
        $LCS(x, y, i, j-1))$
    end if
  end if
  return $c[i, j]$

- $c[i, j]$ written at most once.
- returned value written immediately
- charge all work to writes

Eliminating Recursion completely

function $\mathrm{LCS}(x, y)$
  $\forall i : c[i, 0] = 0$
  $\forall j : c[0, j] = 0$
  for $i \in 1 \ldots |x|$ **do**
    for $j \in 1 \ldots |y|$ **do**
      if $x[i] = y[j]$ **then**
        $c[i, j] \leftarrow c[i-1, j-1] + 1$
      else
        $c[i, j] \leftarrow \max(c[i-1, j], c[i, j-1])$
      end if
    end for
  end for
end function

Comparing Memoized to Iterative LCS

- Asymptotic time is the same

Comparing Memoized to Iterative LCS

- Asymptotic time is the same
- Iterative version is typically faster/more robust in practice

Comparing Memoized to Iterative LCS

- Asymptotic time is the same
- Iterative version is typically faster/more robust in practice
- memoized version is easier to derive (even automatically) from the recursive version.

Comparing Memoized to Iterative LCS

- Asymptotic time is the same
- Iterative version is typically faster/more robust in practice
- memoized version is easier to derive (even automatically) from the recursive version.
- Iterative version is easier to analyze

Comparing Memoized to Iterative LCS

- Asymptotic time is the same
- Iterative version is typically faster/more robust in practice
- memoized version is easier to derive (even automatically) from the recursive version.
- Iterative version is easier to analyze
- Both versions add extra memory use to pure recursion.

Comparing Memoized to Iterative LCS

- Asymptotic time is the same
- Iterative version is typically faster/more robust in practice
- memoized version is easier to derive (even automatically) from the recursive version.
- Iterative version is easier to analyze
- Both versions add extra memory use to pure recursion.
- Memoization never solves unneeded subproblems.

Reading back the sequence

function $\textsc{Backtrack}(i, j)$
    if $(i < 1)$ **or** $(j < 1)$ **then**
        return ""
    end if
    if x[i] = y[j] **then**
        return backtrack$(i-1, j-1) + x[i]$
    end if
    if $c[i, j-1] > c[i-1, j]$ **then**
        return backtrack$(i, j-1)$
    else
        return backtrack$(i-1, j)$
    end if
end function

Contents

Edit (Levenshtein) Distance

- DPV 6.3, JE5.5
- 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

- DPV 6.3, JE5.5
- 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)

If we remove any column from an optimal alignment, we have an optimal alignment for the remaining substrings.

Alignments (gap representation)

Theorem (Optimal substructure)

If we remove any column from an optimal alignment, we have an optimal alignment for the remaining substrings.

proof.

By contradiction $\square$

Subproblems (prefixes)

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

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

justification.

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

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

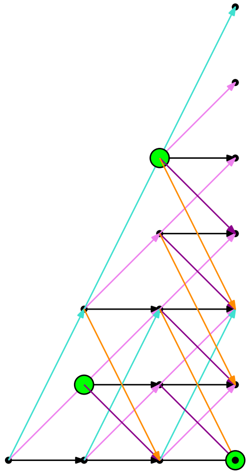
$$E[i,j] = \begin{cases} E[i-1,j]+1 & \text{deletion} \\ E[i,j-1]+1 & \text{insertion} \\ E[i-1,j-1]+1 & \text{substition} \\ E[i-1,j-1] & \text{equality} \end{cases}$$

- 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?

Contents

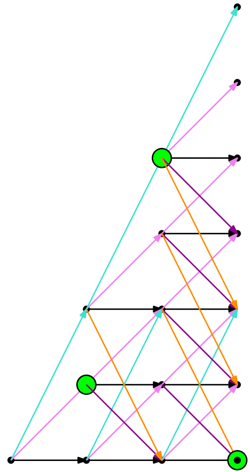
Balloon Flight Planning

- Start at $(0, 0)$



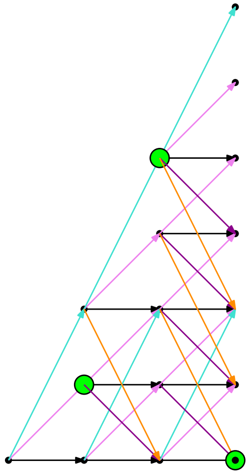
Balloon Flight Planning

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



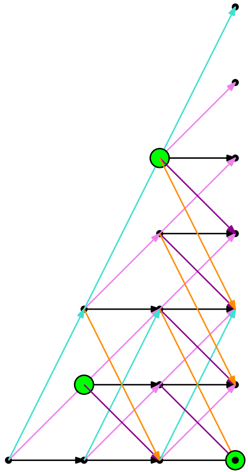
Balloon Flight Planning

- 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.



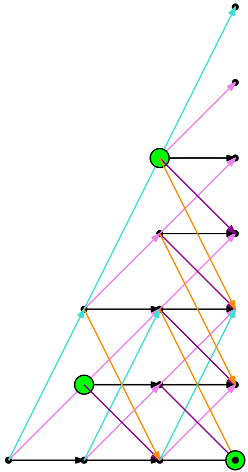
Balloon Flight Planning

- 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



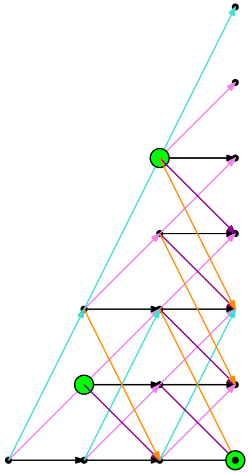
Balloon Flight Planning

- 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



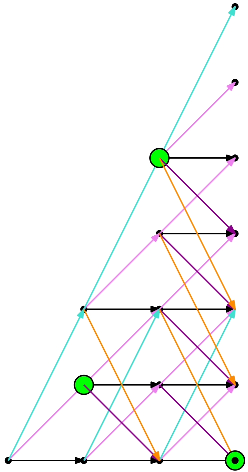
Big Graph is Big

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



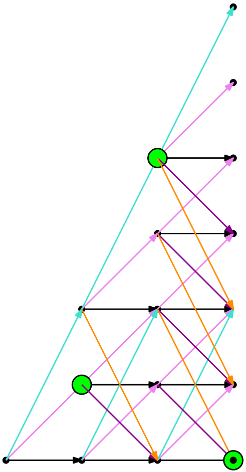
Big Graph is Big

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



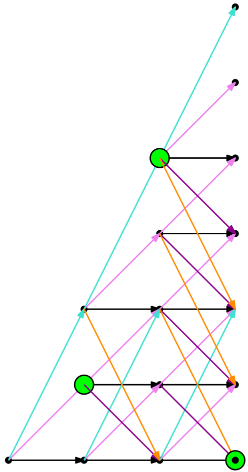
Big Graph is Big

- 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



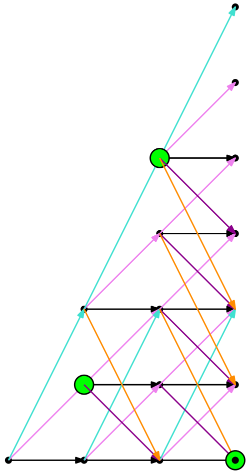
Big Graph is Big

- 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

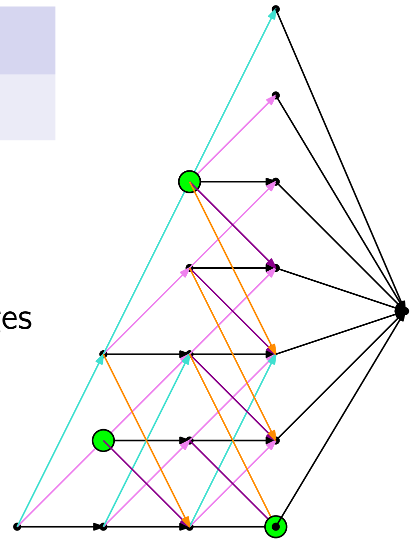


Finding a maximum value path

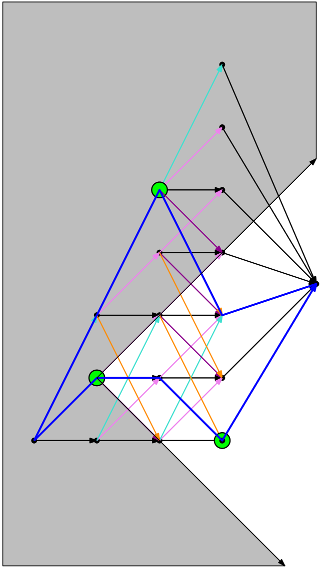
An easy case of a hard problem

In general NP-Hard, but not in DAGs.

function $\textsc{BestPath}(V, E)$
  for $v \in$ 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



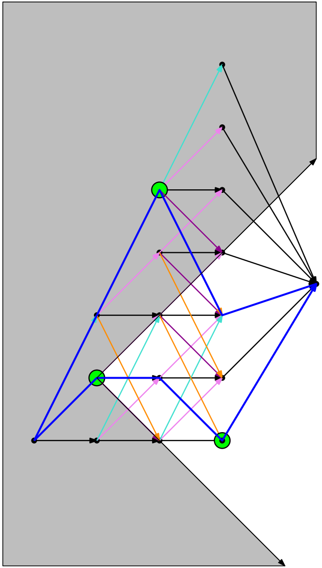
Straightening paths



Lemma (Straightening Paths)

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

Straightening paths



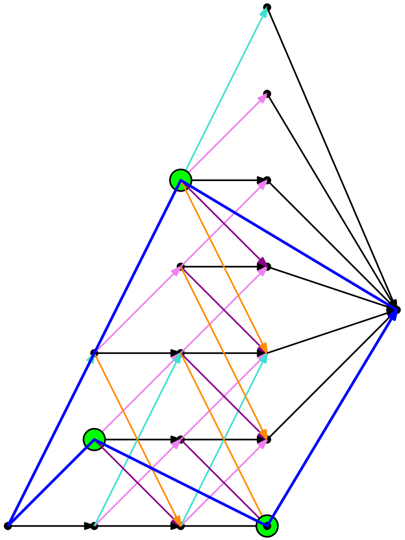
Lemma (Straightening Paths)

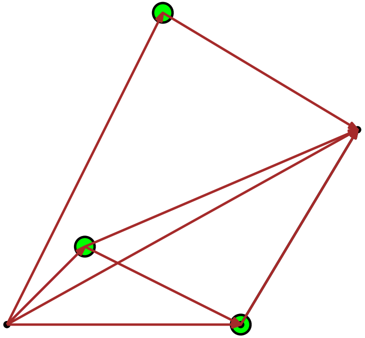
If there is a feasible path from p to q then the segment $[p, q]$ is feasible.

Proof

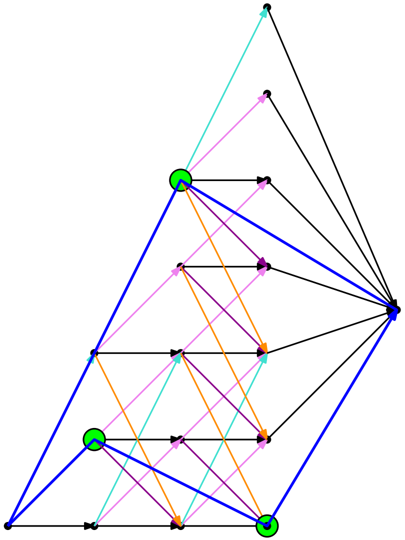
The path cannot escape the cone define by the steepest possible segments.

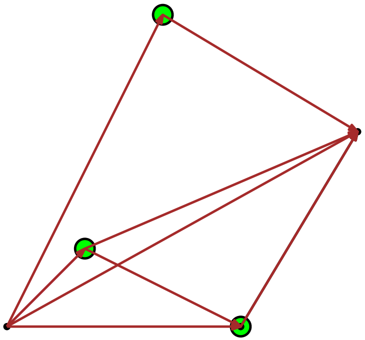
A new graph





A new graph





Improved graph size

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