CS3383 Unit 3: Dynamic Programming

David Bremner

February 23, 2018

Outline

Dynamic Programming

- Shortest path in DAG
- Balloon Flight Planning
- Longest Common Subsequence

Contents

March Break Hotels

Scenario

Wanted	Cheap holiday
Costs	Hotel + Taxi, no charge for inconvenience

March Break Hotels

Scenario

Wanted	Cheap holiday
Costs	Hotel + Taxi, no charge for inconvenience

Input

Taxi Cost

	a	b	c	airport
a	0	10	30	50
b	10	0	30	50
c	30	30	0	50
airport	50	50	50	0

Hotel Price

	1	2	3	4
a	100	100	100	100
b	80	40	120	120
c	50	80	80	80

What to do?

The Obvious Algorithm

- Begin at the beginning

What to do?

The Obvious Algorithm

- Begin at the beginning
- Take it day by day

What to do?

The Obvious Algorithm

- Begin at the beginning
- Take it day by day

- How does this do on our example data?

What to do?

The Obvious Algorithm

- Begin at the beginning
- Take it day by day

- How does this do on our example data?
- Can we find a better solution?

What to do?

The Obvious Algorithm

- Begin at the beginning
- Take it day by day

- How does this do on our example data?
- Can we find a better solution?
- What if Taxis charge 1000 to pick up at Hotel C(alifornia)?

It's a trap!

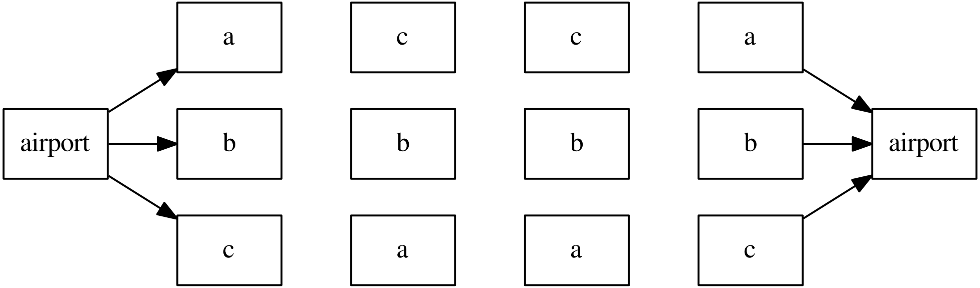
Hotel Price

	1	2	3	4
a	100	100	100	100
b	80	40	120	120
c	50	80	80	80

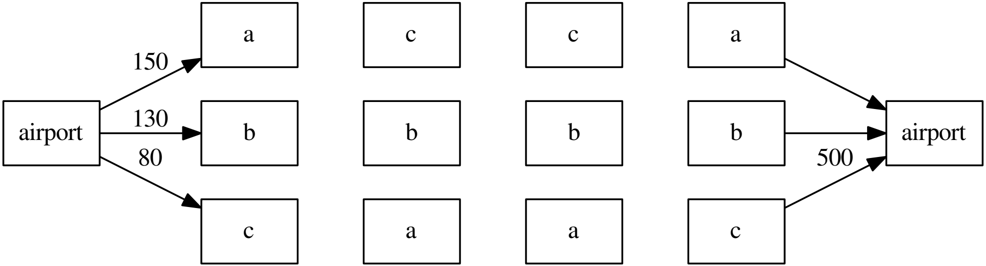
Taxi Cost

	a	b	c	airport
a	0	10	30	50
b	10	0	30	50
c	1000	1000	1000	500
airport	50	50	50	0

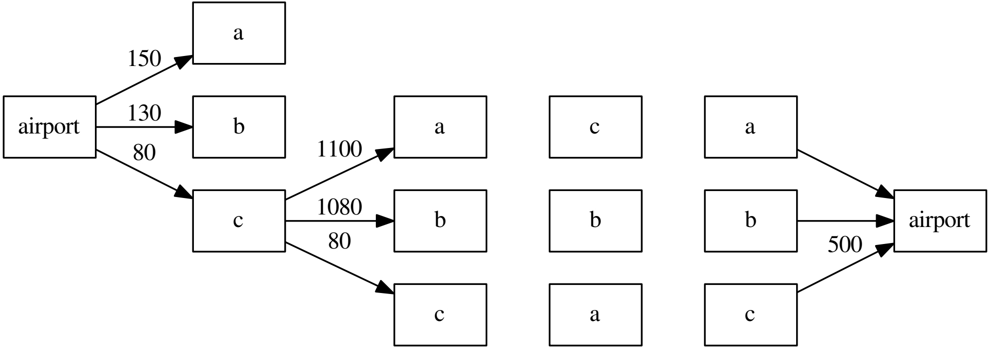
Let's get graphical



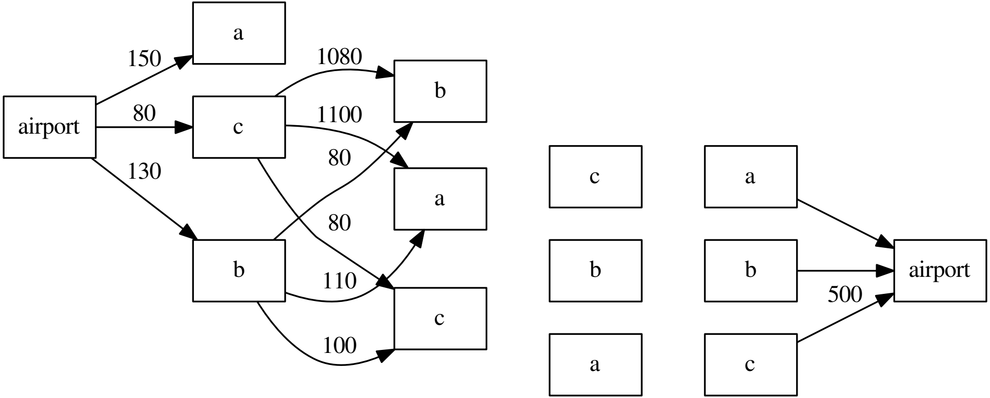
Let's get graphical



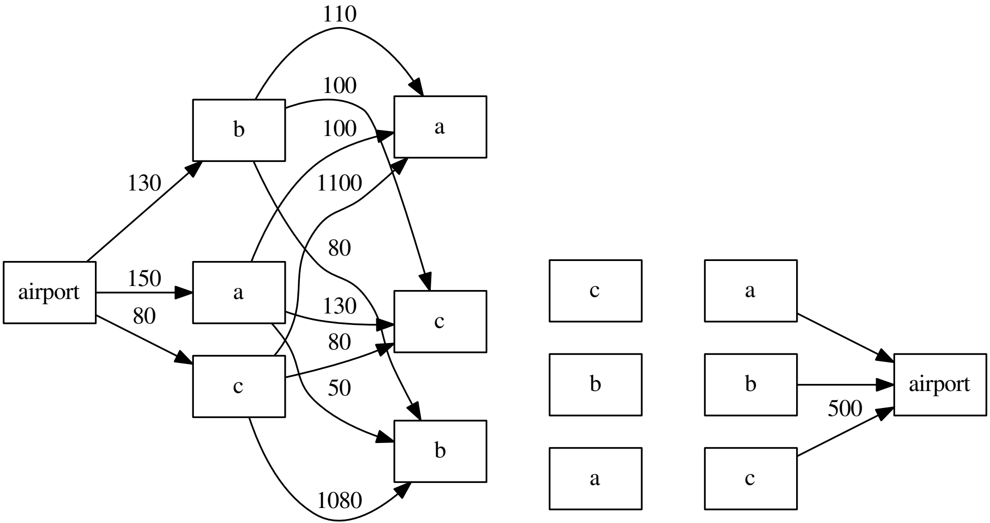
Let's get graphical



Let's get graphical



Let's get graphical



Djikstra considered overkill

- There are no negative edge weights

Djikstra considered overkill

- There are no negative edge weights
- We know how to find the shortest path in such a graph

Djikstra considered overkill

- There are no negative edge weights
- We know how to find the shortest path in such a graph
- Even better, we have an *acyclic* graph (why?)

Djikstra considered overkill

- There are no negative edge weights
- We know how to find the shortest path in such a graph
- Even better, we have an *acyclic* graph (why?)
- So we find a shortest path in linear time after *topological sorting*.

Djikstra considered overkill

- There are no negative edge weights
- We know how to find the shortest path in such a graph
- Even better, we have an *acyclic* graph (why?)
- So we find a shortest path in linear time after *topological sorting*.
- We can do topological sort by DFS or by (essentially) BFS.

"Recursive" topological sort

Recursive version

1. Remove a *source* from the DAG, and put it first.
2. Topologically sort the remaining graph.

- how to quickly find a source?

"Recursive" topological sort

Recursive version

1. Remove a *source* from the DAG, and put it first.
2. Topologically sort the remaining graph.

- how to quickly find a source?
- Use some auxilary data structure to track sources across iterations

"Recursive" topological sort

Recursive version

1. Remove a *source* from the DAG, and put it first.
2. Topologically sort the remaining graph.

- how to quickly find a source?
- Use some auxilary data structure to track sources across iterations

Using a Queue

BFS-like topological sort

```
function TopSort(G)
    $Q \leftarrow$ All Sources
    while !empty(Q) do



    end while
end function
```

Using a Queue

BFS-like topological sort

```
function TopSort(G)
    $Q \leftarrow$ All Sources
    while !empty(Q) do
        $v \leftarrow \mathrm{deq}(Q)$


    end while
end function
```

Using a Queue

BFS-like topological sort

```
function TopSort(G)
    $Q \leftarrow$ All Sources
    while !empty(Q) do
        $v \leftarrow \mathrm{deq}(Q)$
        Output $v$

    end while
end function
```

Using a Queue

BFS-like topological sort

```
function TopSort(G)
    $Q \leftarrow$ All Sources
    while !empty(Q) do
        $v \leftarrow \mathrm{deq}(Q)$
        Output $v$
        Remove $v$, add new sources to $Q$
    end while
end function
```

Using a Queue

BFS-like topological sort

```
1: function TopSort(G)
2:     $Q \leftarrow$ All Sources
3:     while !empty(Q) do
4:         $v \leftarrow \mathrm{deq}(Q)$
5:         Output $v$
6:         Remove $v$, add new sources to $Q$
7:     end while
8: end function
```

- What is the complexity of step 6?

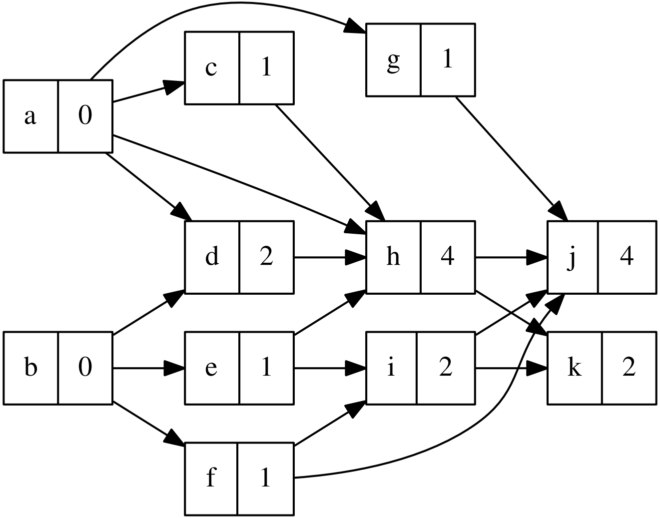
Using a Queue

BFS-like topological sort

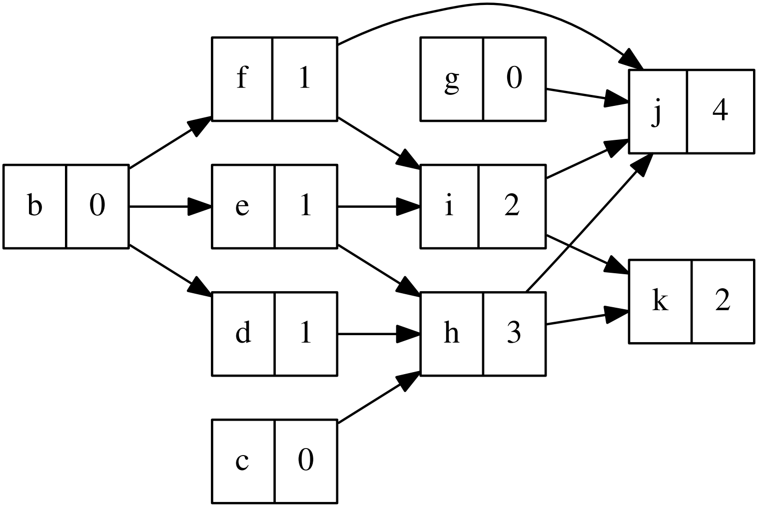
```
1: function TopSort(G)
2:     $Q \leftarrow$ All Sources
3:     while !empty(Q) do
4:         $v \leftarrow \mathrm{deq}(Q)$
5:         Output $v$
6:         Remove $v$, add new sources to $Q$
7:     end while
8: end function
```

- What is the complexity of step 6?
- We can simplify e.g. using counters

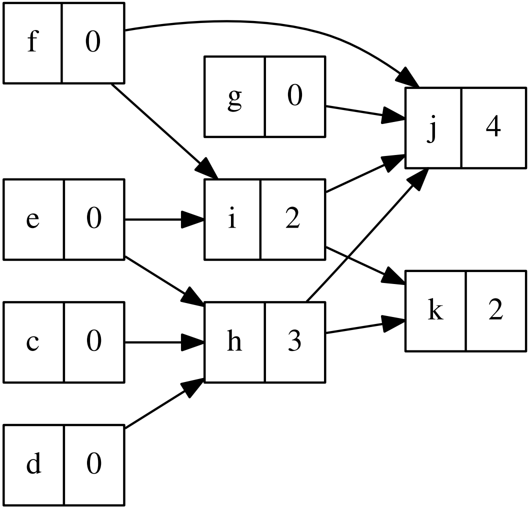
Topological sort with counters



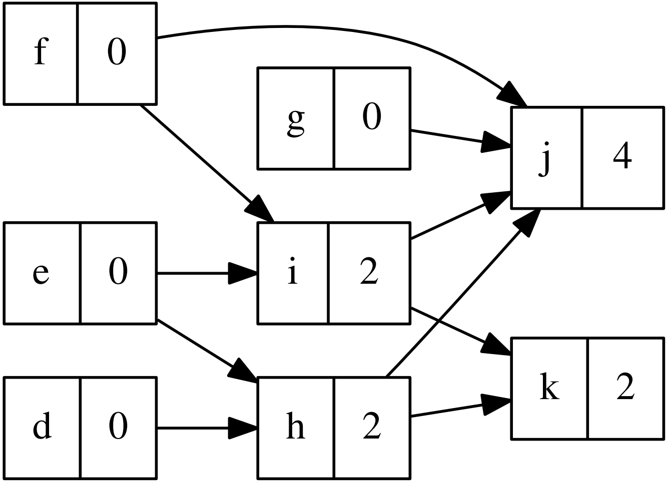
Topological sort with counters



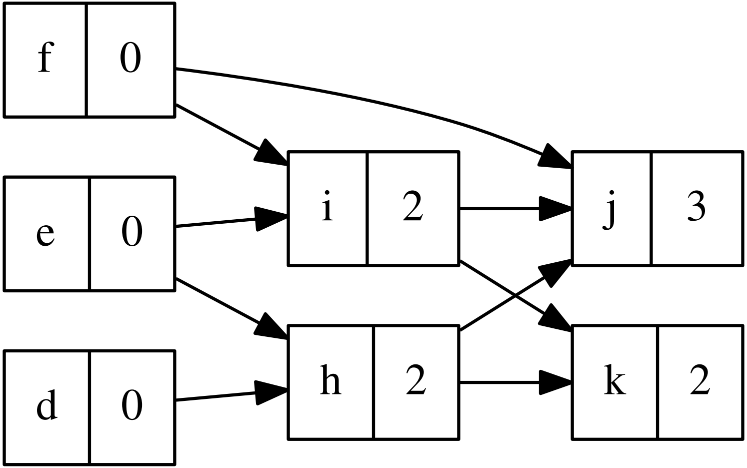
Topological sort with counters



Topological sort with counters



Topological sort with counters



Avoiding a priority queue

```
function $\textsc{RemoveSource}(v, G)$
    for $u$ child of $v$ do
        decrement counter$[u]$
        if counter$[u]$ $==$ 0 then
            enq$(u)$
        end if
    end for
    Remove $v$ from $G$
end function
```

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 Top Sorted Graph

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

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 $L(i)$

So what does this have to do with Dynamic Programming?

Ordered Subproblems

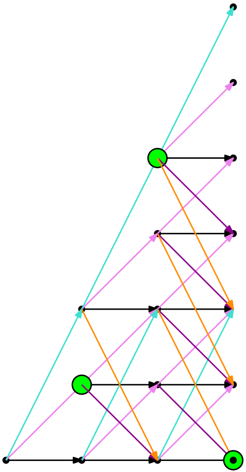
In order to solve our problem in a single pass, we need

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

Contents

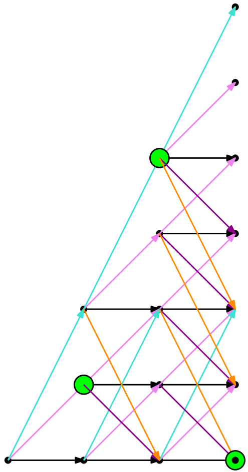
Balloon Flight Planning

- Start at $(0, 0)$



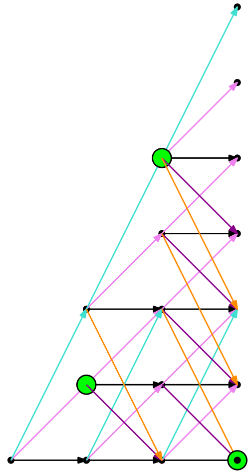
Balloon Flight Planning

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



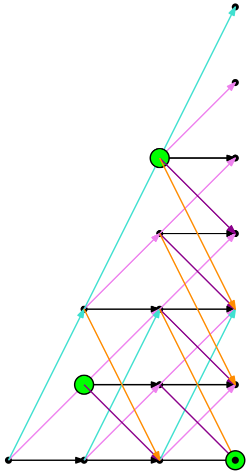
Balloon Flight Planning

- 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.



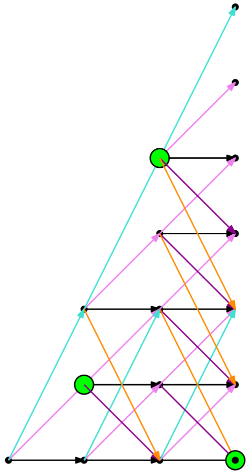
Balloon Flight Planning

- 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



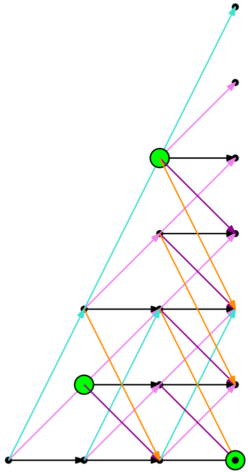
Balloon Flight Planning

- 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



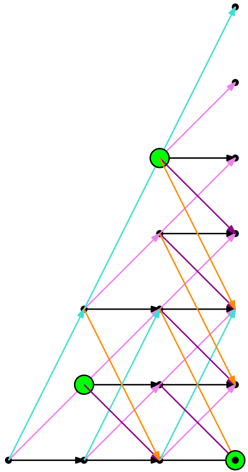
Big Graph is Big

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



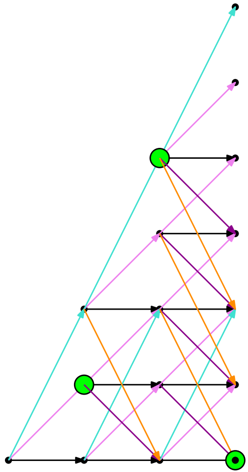
Big Graph is Big

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



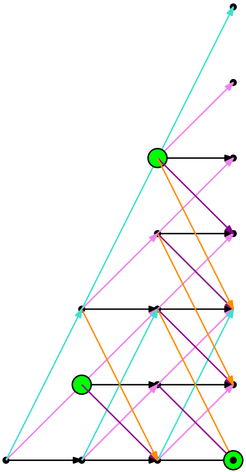
Big Graph is Big

- 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



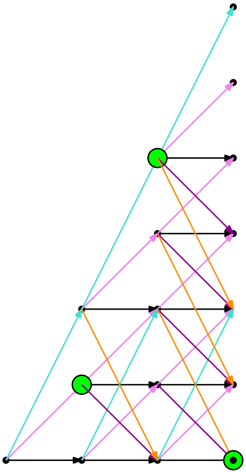
Big Graph is Big

- 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

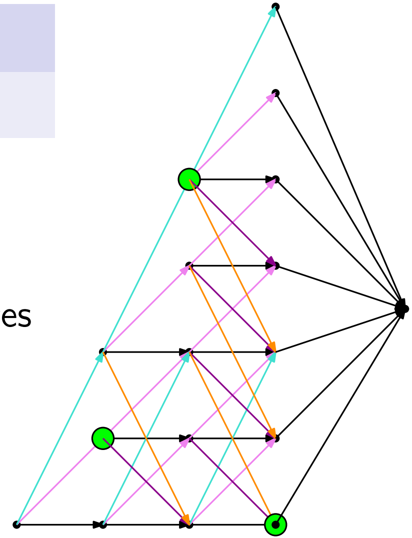


Finding a maximum value path

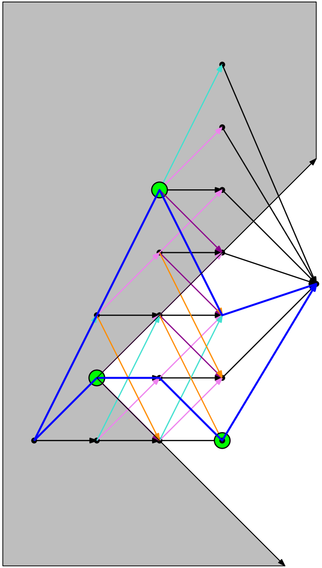
An easy case of a hard problem

In general NP-Hard, but not in DAGs.

```
function 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
```



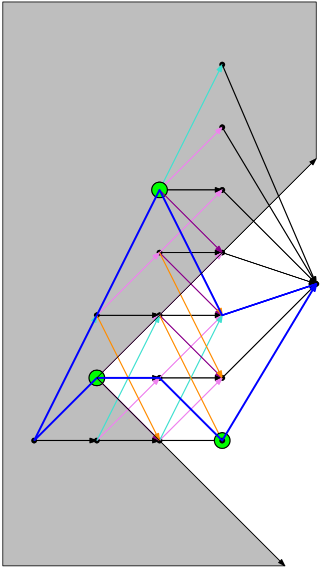
Straightening paths



Lemma (Straightening Paths)

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

Straightening paths



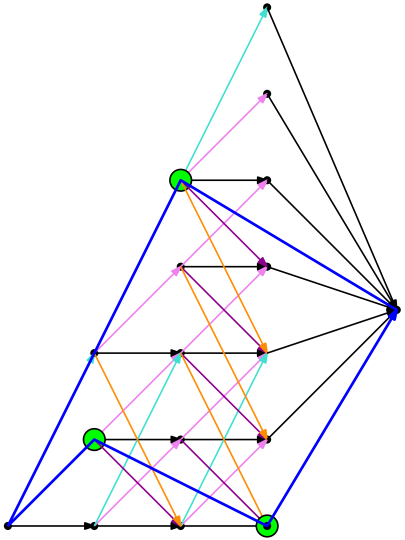
Lemma (Straightening Paths)

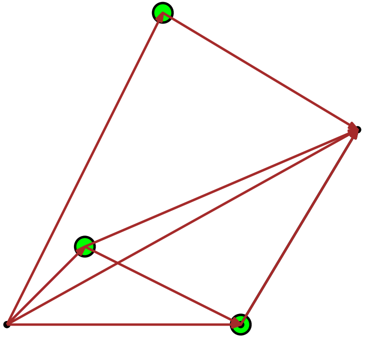
If there is a feasible path from p *to* q *then the segment* $[p, q]$ *is feasible.*

Proof

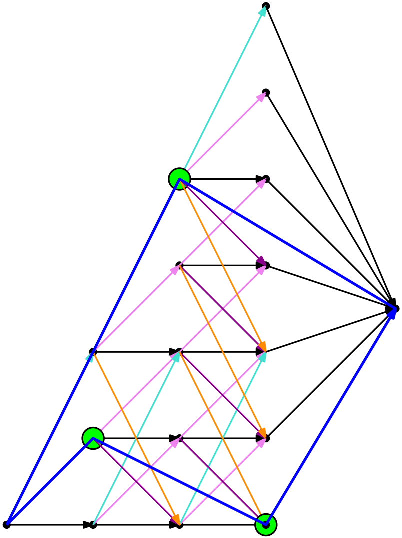
The path cannot escape the cone define by the steepest possible segments.

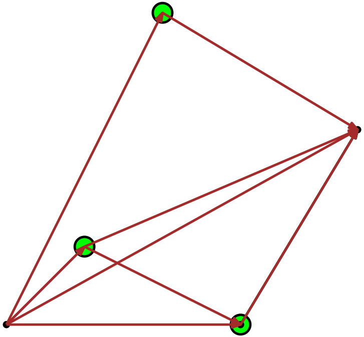
A new graph





A new graph





Improved graph size

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

Contents

Dynamic Programming

- Shortest path in DAG
- Balloon Flight Planning
- Longest Common Subsequence

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

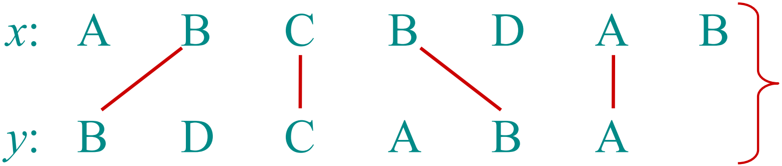
Dynamic programming

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

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 $= O(n2^m)$

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

Recursive formulation

Theorem.

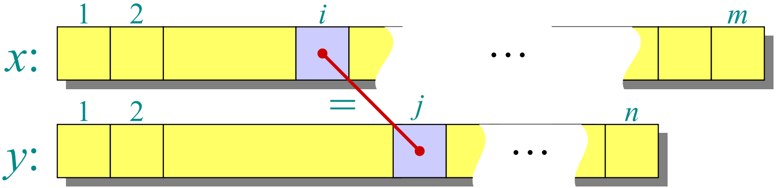
$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max\{c[i-1, j], c[i, j-1]\} & \text{otherwise.} \end{cases}$$

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 $x[i] = y[j]$:

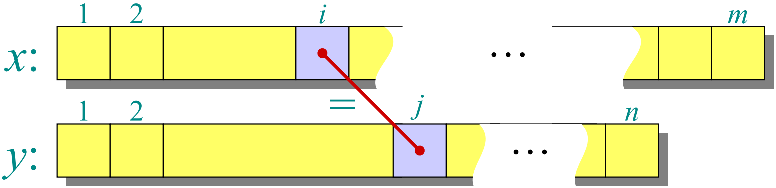


Recursive formulation

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

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.

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 $z = \mathrm{LCS}(x, y)$, 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

```
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$(x, y, i, j-1)\}$
```

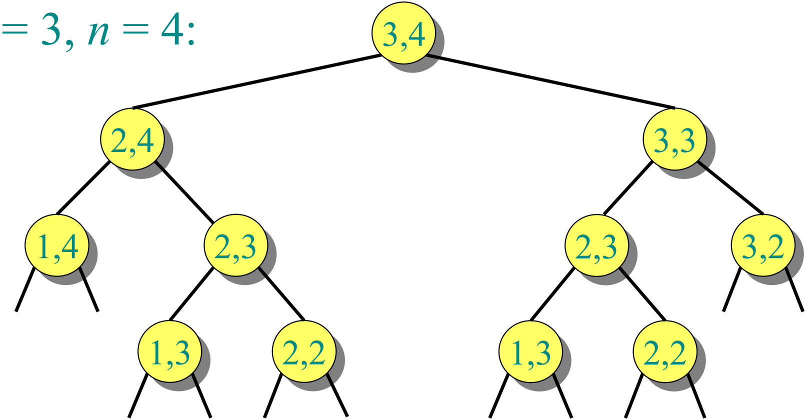
Recursive algorithm for LCS

```
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.

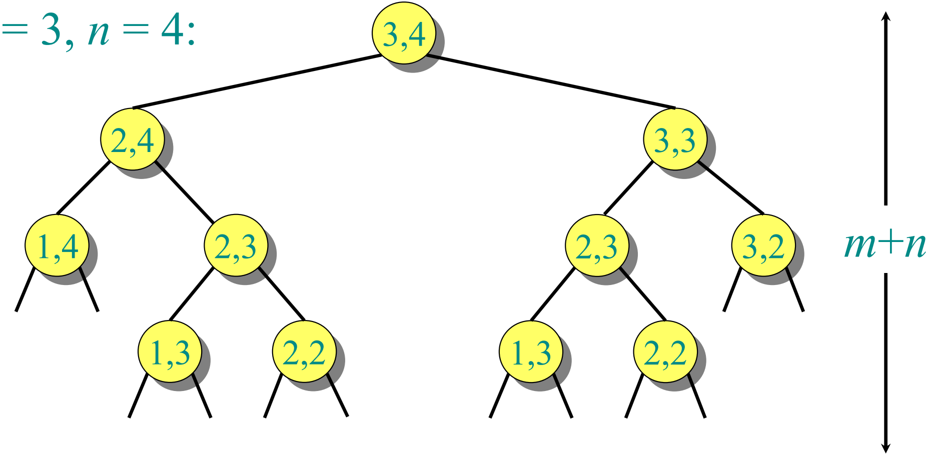
Recursion tree

$m = 3, n = 4$:



Recursion tree

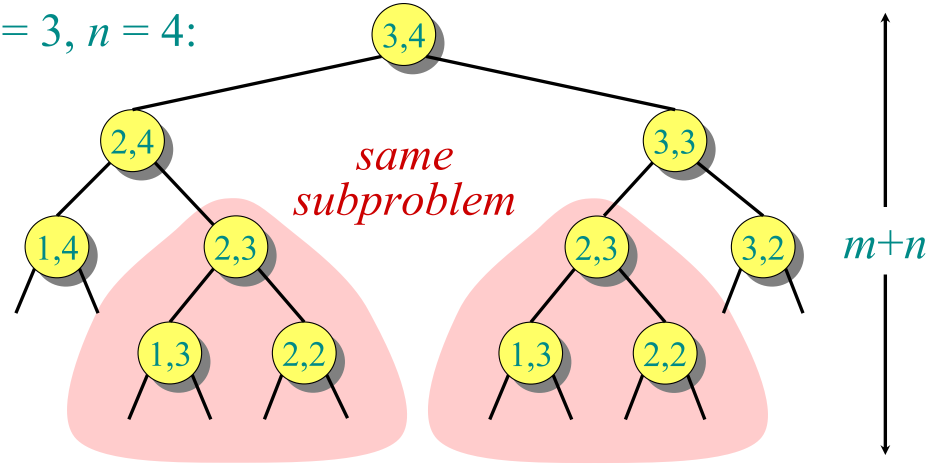
$m = 3$, $n = 4$:



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

Memoization

Recursive Version

```
function $\textsc{Recur}(p_1, \ldots p_k)$
    ⋮
    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
    ⋮
    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 $c[i, j] = \text{NIL}$
        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 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
```