Dynamic Programming

CSE 101: Design and Analysis of Algorithms

Lecture 19

CSE 101: Design and analysis of algorithms

- Dynamic programming
  - Reading: Chapter 6
- Homework 7 is due Dec 6, 11:59 PM
- This Friday's discussion section is the final exam review
- No instructor, TA, or tutor office hours during finals week. Ask questions on Piazza.
- Final exam is Friday, Dec 14, 7:00 PM-9:59 PM
  - Practice problems released this Friday

Dynamic Programming

- Dynamic programming is an algorithmic paradigm in which a problem is solved by identifying a collection of subproblems and tackling them one by one, smallest first, using the answers to small problems to help figure out larger ones, until they are all solved

The knapsack problem

- Suppose you are a burglar who breaks into a store and you want to leave with the maximum value of items
- Example: your knapsack can only hold 13 lbs and the items in the store are

	Silver	4 lbs jewery	Diamond jacket	Golden suitcases	Platinum Nintendo	Solid ruby bowling ball
Value	4	9	12	15	19	21
Weight	2	4	5	7	8	9

The knapsack problem

- What is the maximum value you can have from a list of items a[1],…,a[n] where each item has a value v[i] and a weight w[i] given that you cannot have more weight than W

The knapsack problem, algorithm

```
Knapsack(w[1…n],v[1…n],W)
KS(j,0):=0 for all j
KS(0,w):=0 for all w
for w from 1 to W
    for j from 1 to n
        KS(j,w):= max(KS(j,w-w(j)),KS(j-1,w))
Return KS(n,W)
```

Base cases

Ordering

The knapsack problem, runtime

```
Knapsack(w[1…n],v[1…n],W)
KS(j,0):=0 for all j
KS(0,w):=0 for all w
for w from 1 to W
    for j from 1 to n
        KS(j,w):= max(KS(j,w-w(j)),KS(j-1,w))
Return KS(n,W)
```

W times

n times

Total runtime O(nW)

The knapsack problem, runtime

- We just showed that the dynamic programming runtime is O(nW)
  - Reasonable when W is small, but is exponential in the input size
    - For example, $W = 75 = 1001011_2 \approx 2^7$, so an input size of 7 must do 75 operations
- Brute force is 2^n (also exponential)
- It is unknown whether or not there is a polynomial algorithm to this problem
- Interesting variant: if integers are encoded in *unary* (e.g., $12 = 11111111111_1$), then there is a polynomial algorithm

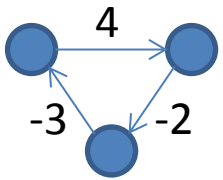
Dynamic programming

SHORTEST PATHS

Shortest path

- We have a breadth-first search style algorithm for finding shortest paths in graphs with positive edge weights—Dijkstra's algorithm
- But, what if there are negative edge weights? For example, what if edges represent bank transactions, where deposits are positive and withdrawals are negative (or gains and losses in altitude on a flight)?
- If there are negative cycles, then shortest path is not defined, because we can cycle around arbitrarily, getting more and more negative

For example

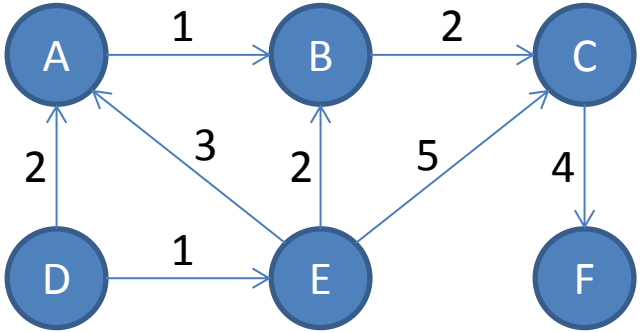


What about graphs with no cycles?

- First, let's try to solve shortest paths for directed acyclic graphs with some negative edges
- We are still trying to minimize the sum of the edge weights along a path

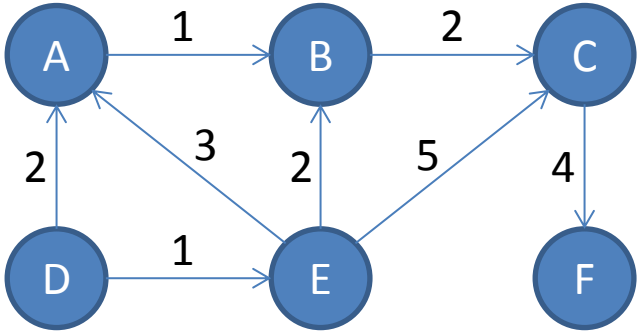
Shortest path in directed acyclic graphs

Example

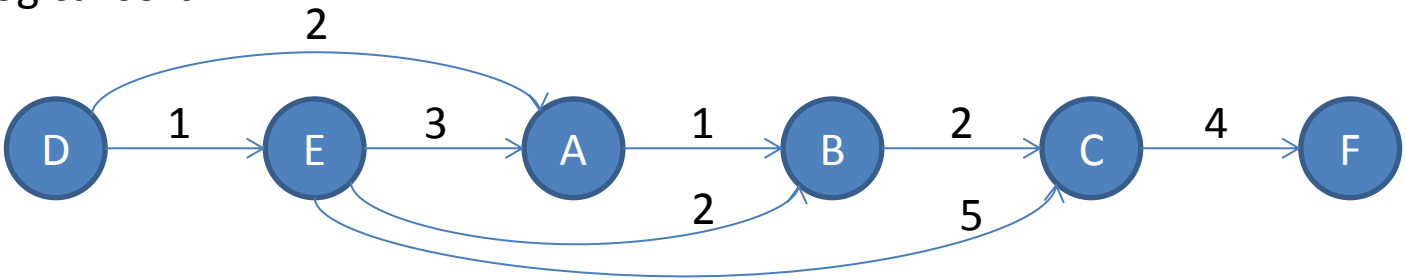


Shortest path in directed acyclic graphs

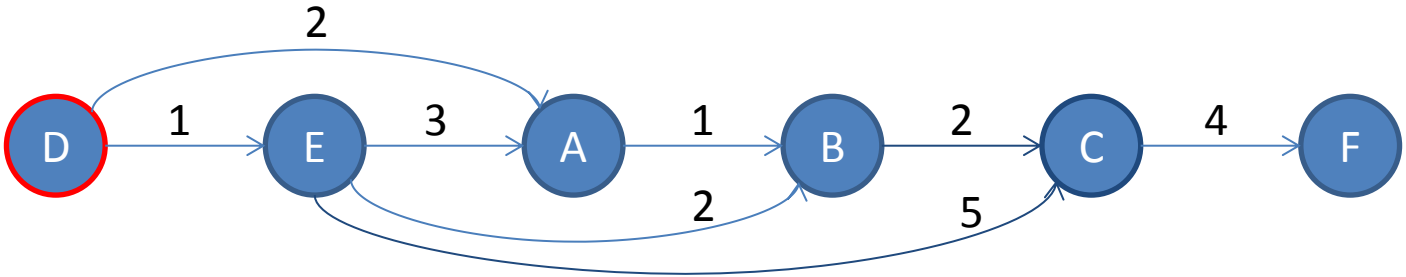
Example



Topological sort

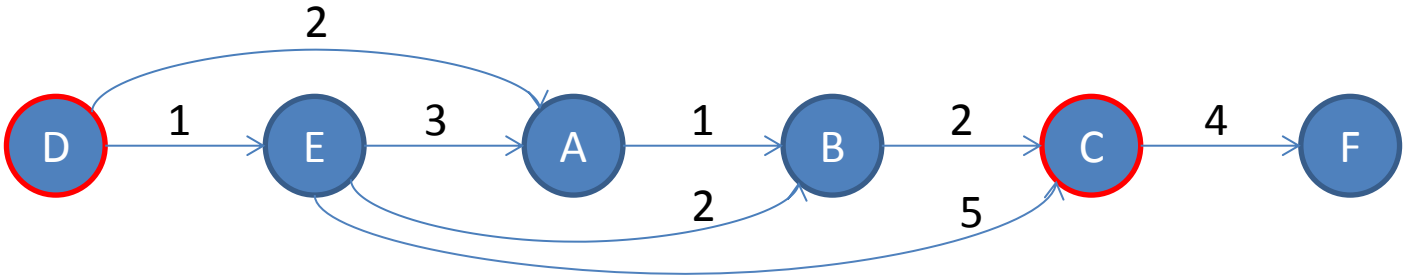


Shortest path in directed acyclic graphs



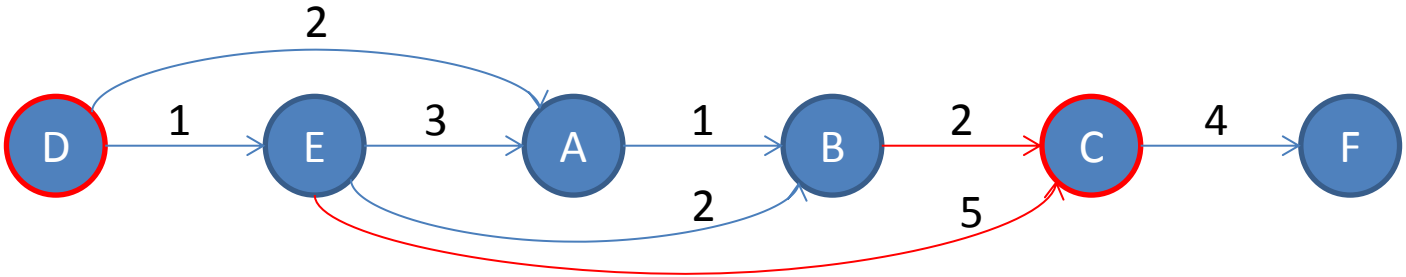
- Shortest distance from node D to another node x will be denoted dist(x)

Shortest path in directed acyclic graphs



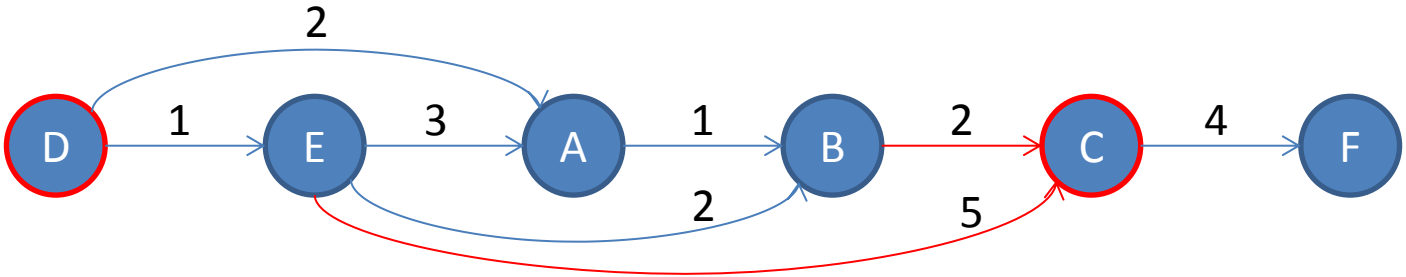
- Shortest distance from node D to another node x will be denoted dist(x)
- How do we determine the shortest distance from D to C?

Shortest path in directed acyclic graphs



- Shortest distance from node D to another node x will be denoted dist(x)
- How do we determine the shortest distance from D to C?
  - The shortest distance from D to C is
    dist(C) = min(dist(E) + 5, dist(B) + 2)

Shortest path in directed acyclic graphs



- Shortest distance from node D to another node x will be denoted dist(x)
- How do we determine the shortest distance from D to C?
  - The shortest distance from D to C is
    dist(C) = min(dist(E) + 5, dist(B) + 2)
- This kind of relation can be written for every node
- Since it is a directed acyclic graph, the arrows only go to the right, so by the time we get to node x, we have all the information needed

Dynamic programming steps

- Step 1: Define subproblems and corresponding array
- Step 2: What are the base cases?
- Step 3: Give recursion for subproblems
- Step 4: Find bottom-up order
- Step 5: What is the final output?
- Step 6: Put it all together into an iterative algorithm that fills in the array step by step

- For analysis:
- Step 7: Correctness proof
- Step 8: Runtime analysis

Dynamic programming steps

- Step 0: Define the problem
- Step 1: Define subproblems and corresponding array
- Step 2: What are the base cases?
- Step 3: Give recursion for subproblems
- Step 4: Find bottom-up order
- Step 5: What is the final output?
- Step 6: Put it all together into an iterative algorithm that fills in the array step by step

Shortest path in directed acyclic graphs

- Step 0: Define the problem

Shortest path in directed acyclic graphs

- Step 0: Define the problem
  - Given a directed acyclic graph with vertices $v_1, \ldots, v_n$ in topological order and starting vertex s, where $1 \leq s \leq n$, output the shortest distance from s to $v_1, \ldots, v_n$

Shortest path in directed acyclic graphs

- Step 0: Define the problem
  - Given a directed acyclic graph with vertices $v_1, \ldots, v_n$ in topological order and starting vertex s, where $1 \leq s \leq n$, output the shortest distance from s to $v_1, \ldots, v_n$
- Step 1: Define subproblems and corresponding array

Shortest path in directed acyclic graphs

- Step 0: Define the problem
  - Given a directed acyclic graph with vertices $v_1, \ldots, v_n$ in topological order and starting vertex s, where $1 \leq s \leq n$, output the shortest distance from s to $v_1, \ldots, v_n$
- Step 1: Define subproblems and corresponding array
  - Given a directed acyclic graph with vertices $v_1, \ldots, v_n$ in topological order and starting vertex s, output the shortest distance from s to $v_1, \ldots, v_i$
  - Let $d[i] = \text{dist}(v_i)$ the shortest distance from s to v_i

Shortest path in directed acyclic graphs

- Step 2: What are the base cases?

Shortest path in directed acyclic graphs

- Step 2: What are the base cases?

  $\text{dist}(s) = 0, \text{dist}(v) = \infty \text{ if } v \neq s$

Shortest path in directed acyclic graphs

- Step 2: What are the base cases?
  $\text{dist}(s) = 0, \text{dist}(v) = \infty \text{ if } v \neq s$
- Step 3: Give recursion for subproblems

Shortest path in directed acyclic graphs

- Step 2: What are the base cases?
  $\text{dist}(s) = 0, \text{dist}(v) = \infty \text{ if } v \neq s$
- Step 3: Give recursion for subproblems
  - What is the second to last vertex in path to v_i?

Shortest path in directed acyclic graphs

- Step 2: What are the base cases?
  $\text{dist}(s) = 0, \text{dist}(v) = \infty \text{ if } v \neq s$
- Step 3: Give recursion for subproblems
  - What is the second to last vertex in path to v_i? Let's call it vertex u

Shortest path in directed acyclic graphs

- Step 2: What are the base cases?

  $\text{dist}(s) = 0, \text{dist}(v) = \infty \text{ if } v \neq s$

- Step 3: Give recursion for subproblems
  - What is the second to last vertex in path to v_i?

    Let's call it vertex u

  $$\text{dist}(v_i) = \min_{(u,v_i) \in E} \{\text{dist}(u) + w(u, v_i)\}$$

Shortest path in directed acyclic graphs

- Step 4: Find bottom-up order

Shortest path in directed acyclic graphs

- Step 4: Find bottom-up order
  - Topological order

Shortest path in directed acyclic graphs

- Step 4: Find bottom-up order
  - Topological order
- Step 5: What is the final output?

Shortest path in directed acyclic graphs

- Step 4: Find bottom-up order
  - Topological order
- Step 5: What is the final output?
  $[\text{dist}(v_1), \ldots, \text{dist}(v_n)]$

Shortest path in directed acyclic graphs

- Step 6: Put it all together into an iterative algorithm that fills in the array step by step

Shortest path in directed acyclic graphs

- Step 6: Put it all together into an iterative algorithm that fills in the array step by step

initialize all dist(.) values to infinity

dist(s):=0

for each $v \in V \backslash \{s\}$ in topological order

$$\text{dist}(v) = \min_{(u,v) \in E} \{\text{dist}(u) + w(u,v)\}$$

Example: Longest increasing subsequence

- Given a sequence of distinct positive integers a[1],…,a[n], an increasing subsequence is a sequence a[i_1],…,a[i_k] such that $i_1 < \cdots < i_k$ and a[i_1]<…<a[i_k]

- Example: 15, 18, 8, 11, 5, 12, 16, 2, 20, 9, 10, 4
  - 5, 16, 20 is an increasing subsequence

Longest increasing subsequence

- Given a sequence of distinct positive integers a[1],…,a[n], an increasing subsequence is a sequence a[i_1],…,a[i_k] such that $i_1 < \cdots < i_k$ and a[i_1]<…<a[i_k]

- Example: 15, 18, 8, 11, 5, 12, 16, 2, 20, 9, 10, 4
  - 5, 16, 20 is an increasing subsequence
  - How long is the longest increasing subsequence?

Longest increasing subsequence

- Given a sequence of distinct positive integers a[1],…,a[n], an increasing subsequence is a sequence a[i_1],…,a[i_k] such that $i_1 < \cdots < i_k$ and a[i_1]<…<a[i_k]

- Example: 15, 18, 8, 11, 5, 12, 16, 2, 20, 9, 10, 4
  - 5, 16, 20 is an increasing subsequence
  - How long is the longest increasing subsequence?
    - 8, 11, 12, 16, 20

Viewing longest increasing subsequence as shortest path in directed acyclic graph

- What could the vertices be?

Viewing longest increasing subsequence as shortest path in directed acyclic graph

- What could the vertices be?
  - Positions in the array, and 0

Viewing longest increasing subsequence as shortest path in directed acyclic graph

- What could the vertices be?
  - Positions in the array, and 0
- When is there an edge?

Viewing longest increasing subsequence as shortest path in directed acyclic graph

- What could the vertices be?
  - Positions in the array, and 0
- When is there an edge?
  - There is an edge from I to J if A[I] < A[J] or I=0

Viewing longest increasing subsequence as shortest path in directed acyclic graph

- What could the vertices be?
  - Positions in the array, and 0
- When is there an edge?
  - There is an edge from I to J if A[I] < A[J] or I=0
- What are the weights of edges?

Viewing longest increasing subsequence as shortest path in directed acyclic graph

- What could the vertices be?
  - Positions in the array, and 0
- When is there an edge?
  - There is an edge from I to J if A[I] < A[J] or I=0
- What are the weights of edges? -1

Directed acyclic graphs are canonical for dynamic programming

- Consider a graph whose vertices are the distinct recursive calls an algorithm makes and where calls are edges from the subproblem to the main problem
- This graph had better be a directed acyclic graph or we are in deep trouble
- This graph should be small or dynamic programming will not help much
- Bottom-up order = topological sort

What about graphs with cycles?

- What if the graph has both negative weights and cycles?

What about graphs with cycles?

- What if the graph has both negative weights and cycles?
  - If the graph has negative cycles, some distances are negative infinity. Otherwise,

$$\text{dist}(C) = \min_{v|(v,C)\text{is an edge}} \{\text{dist}(v) + w(v, C)\}$$

What about graphs with cycles?

- What if the graph has both negative weights and cycles?
  - If the graph has negative cycles, some distances are negative infinity. Otherwise,

$$\text{dist}(C) = \min_{v|(v,C)\text{is an edge}} \{\text{dist}(v) + w(v, C)\}$$

- But, this does not give a consistent recursion, because the recursion graph is not a directed acyclic graph, so recursion loops around cycles

What about graphs with cycles?

- What if the graph has both negative weights and cycles?
  - If the graph has negative cycles, some distances are negative infinity. Otherwise,

$$\text{dist}(C) = \min_{v|(v,C)\text{is an edge}} \{\text{dist}(v) + w(v, C)\}$$

- But, this does not give a consistent recursion, because the recursion graph is not a directed acyclic graph, so recursion loops around cycles
- How can we make recursion acyclic when the graph is cyclic?
- When would we even want to do this?

Example: exchange rates

- We usually deal with exchange rates as being determined so that you always lose a bit when exchanging one currency for another

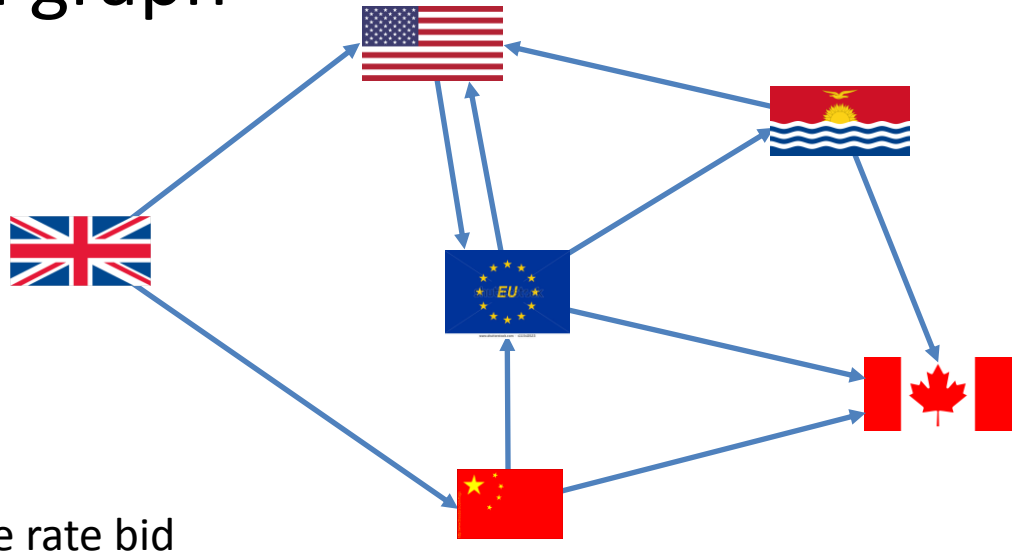
Exchange rates

- But, there is actually no central authority that sets exchange rates. They are actually set by a market where individual buyers and sellers make offers to trade with different prices, sometimes varying from the current rate. This allows currencies to float, changing relative values as market conditions change
- This means the best way to change one currency for another might involve a sequence of trades, rather than just pick the one best rate

Pair	Rate	Bid	Ask	Time
AUD/CAD	1.0119	1.0119	1.0120	8/11/14 12:46 PM
AUD/CHF	0.8397	0.8394	0.8399	8/11/14 12:46 PM
AUD/EUR	0.6921	0.6921	0.6922	8/11/14 12:46 PM
AUD/GBP	0.5519	0.5518	0.5520	8/11/14 12:46 PM
AUD/JPY	94.6585	94.6318	94.6852	8/11/14 12:46 PM
AUD/NZD	1.0947	1.0945	1.0949	8/11/14 12:46 PM
AUD/USD	0.9264	0.9264	0.9265	8/11/14 12:45 PM
CAD/AUD	0.9882	0.9881	0.9883	8/11/14 12:46 PM
CAD/CHF	0.8298	0.8295	0.8300	8/11/14 12:46 PM
CAD/EUR	0.6839	0.6839	0.6840	8/11/14 12:46 PM
CAD/GBP	0.5454	0.5453	0.5455	8/11/14 12:46 PM
CAD/JPY	93.5433	93.5174	93.5692	8/11/14 12:46 PM
CAD/NZD	1.0818	1.0816	1.0820	8/11/14 12:46 PM
CAD/USD	0.9155	0.9155	0.9155	8/11/14 12:46 PM
CHF/AUD	1.1909	1.1906	1.1913	8/11/14 12:46 PM
CHF/CAD	1.2051	1.2048	1.2055	8/11/14 12:46 PM
CHF/EUR	0.8243	0.8240	0.8245	8/11/14 12:46 PM

Exchange rates

- Model as a graph



Vertices = currencies
Edges = bids for trades
Edge weight = exchange rate bid

Objective function

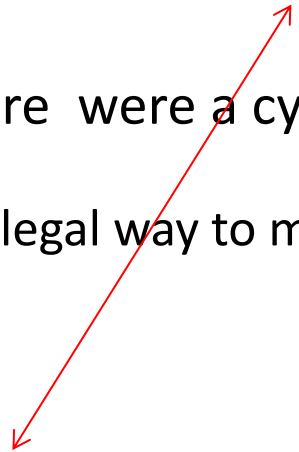
- How good is a sequence of trades?
- What function are we trying to maximize or minimize?
- What is the best thing that could happen for us?

Comparing objective functions

- We are trying to find the path that maximizes the product of the edge weights along a path
- What would be ***really*** good is if there were a cycle whose product was bigger than 1
  - This is called an arbitrage, and is a legal way to make money for free. But, they only last a short time.

Comparing objective functions

- We are trying to find the path that **maximizes the product** of the edge weights along a path
- What would be ***really*** good is if there were a cycle whose product was bigger than 1
  - This is called an arbitrage, and is a legal way to make money for free. But, they only last a short time.



- What we know how to do
  - Shortest path problem: **minimize the sum** of the edge weights
  - How can we map one to the other?
  - What function would map a product to a sum?

Modifying the graph

- Define $w'(e) = -\log(w(e))$
- Then

  $\Sigma_{\{e \in path\}}\, w'(e) = -\log\left(\Pi_{\{e \in path\}} w(e)\right)$
- Thus, the smallest sum of $w'(e)$ is the largest product of $w(e)$ = best trades

Revising the goal

- The revised goal
  - If there is any negative cycle in w', then find one such negative cycle (arbitrage!)
  - If there is not a negative cycle, then find the best exchange rate from US dollars to every other currency = shortest path

Preventing infinite recursion

- Let's put a budget T on how deep we will recurse
  - This corresponds to looking at paths of length at most T

Bound the number of edges

- Given a graph with vertices $v_0, v_1, v_2, \ldots, v_n$, a starting vertex v_0, and weighted edges (possibly negative), give the shortest path from v_0 to each other vertex $v_1, \ldots, v_n$ or determine if there is a negative cycle
- Step 0: Define the problem
  - Bound the number of edges

Bound the number of edges

- Given a graph with vertices $v_0, v_1, v_2, \ldots, v_n$, a starting vertex v_0, and weighted edges (possibly negative), give the shortest path from v_0 to each other vertex $v_1, \ldots, v_n$ or determine if there is a negative cycle
- Step 0: Define the problem
  - Bound the number of edges
    - Given a graph with vertices $v_0, v_1, v_2, \ldots, v_n$, a starting vertex v_0, and weighted edges (possibly negative), give the shortest path from v_0 to each other vertex $v_1, \ldots, v_n$ that has T or fewer edges

Bound the number of edges

- Step 0: Define the problem
  - Given a graph with vertices $v_0, v_1, v_2, \ldots, v_n$, a starting vertex v_0, and weighted edges (possibly negative), give the shortest path from v_0 to each other vertex $v_1, \ldots, v_n$ that has T or fewer edges
- Step 1: Define subproblems and corresponding array

Bound the number of edges

- Step 0: Define the problem
  - Given a graph with vertices $v_0, v_1, v_2, \ldots, v_n$, a starting vertex v_0, and weighted edges (possibly negative), give the shortest path from v_0 to each other vertex $v_1, \ldots, v_n$ that has T or fewer edges
- Step 1: Define subproblems and corresponding array
  - Let B[i,t] by the shortest path from v_0 to v_i with t or fewer edges

Bound the number of edges

- Step 2: What are the base cases?

Bound the number of edges

- Step 2: What are the base cases?

  $B[0,0] = 0$

  $B[i,0] = \infty$ for $i \neq 0$

Bound the number of edges

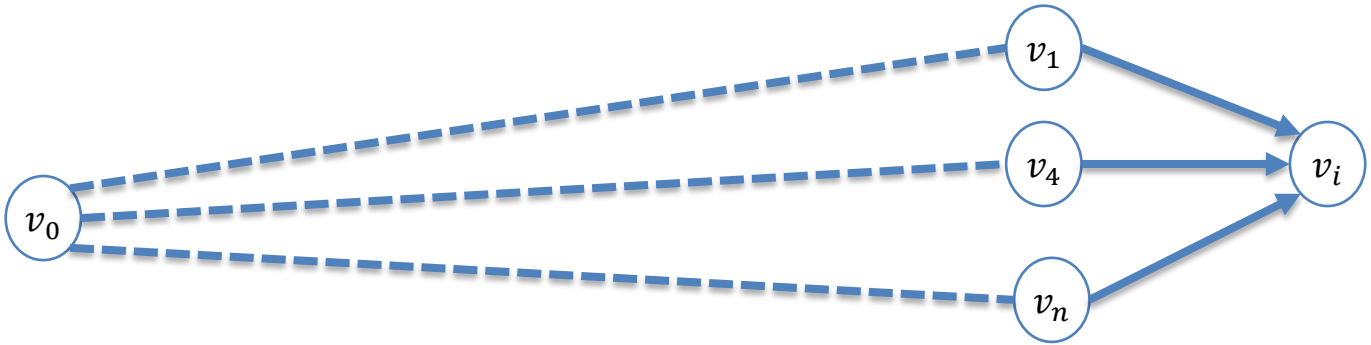
- Step 3: Give recursion for subproblems

Bound the number of edges

- Step 3: Give recursion for subproblems
  - What simple question can we ask to split up the possible paths from v_0 to v_i?

Bound the number of edges

- Step 3: Give recursion for subproblems
  - What simple question can we ask to split up the possible paths from v_0 to v_i?
    - What is the second to last vertex in the shortest path from v_0 to v_i (that uses at most t edges)?

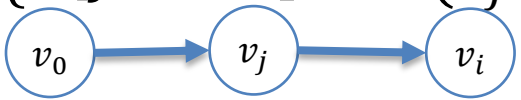


Bound the number of edges

- Step 3: Give recursion for subproblems
  - What simple question can we ask to split up the possible paths from v_0 to v_i?
    - What is the second to last vertex in the shortest path from v_0 to v_i (that uses at most t edges)?

$$B[i,t] = \min \begin{cases} B[0,t-1] + w(v_0, v_i) \\ \vdots \\ B[n,t-1] + w(v_n, v_i) \end{cases}$$

$$= \min_{(v_j,v_i)\in E} \{B[j,t-1] + w(v_j, v_i)\}, B[i,t-1]$$



Bound the number of edges

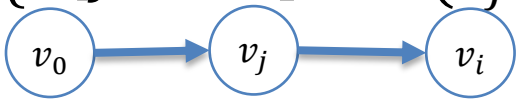
- Step 3: Give recursion for subproblems
  - What simple question can we ask to split up the possible paths from v_0 to v_i?
    - What is the second to last vertex in the shortest path from v_0 to v_i (that uses at most t edges)?

$$B[i,t] = \min \begin{cases} B[0,t-1] + w(v_0, v_i) \\ \vdots \\ B[n,t-1] + w(v_n, v_i) \end{cases}$$

$$= \min_{(v_j,v_i)\in E} \{B[j,t-1] + w(v_j, v_i)\}, B[i,t-1]$$

The shortest path from v_0 to v_j

The shortest path from v_0 to v_i



Bound the number of edges

$$B[i,t] = \min_{(v_j,v_i)\in E}\{B[j,t-1] + w(v_j,v_i)\}, B[i,t-1]$$

- Step 4: Find bottom-up order

Bound the number of edges

$$B[i,t] = \min_{(v_j,v_i)\in E}\{B[j,t-1] + w(v_j,v_i)\}, B[i,t-1]$$

- Step 4: Find bottom-up order
  - You must do the t values in increasing order from t=0,1,2,…,n
  - The i values can be done in any order

Bound the number of edges

- Step 5: What is the final output?

Bound the number of edges

- Step 5: What is the final output?
  - Since we are doing the bounded problem, then we output the array
    $[B(0,T), B(1,T), B(2,T), B(3,T), \dots, B(n,T)]$

Bound the number of edges

- Step 5: What is the final output?
  - Since we are doing the bounded problem, then we output the array
    $[B(0,T), B(1,T), B(2,T), B(3,T), \ldots, B(n,T)]$
  - Can we use this to detect negative cycles?

Bound the number of edges

- Preventing infinite recursion
  - If there are no negative cycles:
    - Then all shortest paths are simple (i.e., do not repeat vertices)

Bound the number of edges

- Preventing infinite recursion
  - If there are no negative cycles:
    - Then all shortest paths are simple (i.e., do not repeat vertices)
    - So, the shortest path has at most |V|-1 = n-1 edges

Bound the number of edges

- Preventing infinite recursion
  - If there are no negative cycles:
    - Then all shortest paths are simple (i.e., do not repeat vertices)
    - So, the shortest path has at most |V|-1 = n-1 edges
    - So, the array values will never improve after t gets bigger than n-1 (i.e., B(i,n-1)=B(i,n)=B(i,n+1)=…)

Bound the number of edges

- Preventing infinite recursion
  - This means that if there exists an i such that $B(i, n-1) \neq B(i, n)$, then there is a negative cycle!
  - So, to solve the main problem, compute all values of $B(i, n-1)$ and then, as you are computing $B(i, n)$, if anything changes, conclude that there is a negative cycle

Bellman-Ford algorithm

- BellmanFord(G=(V,E), V=$v_0, v_1, \ldots, v_n$, starting vertex v_0, edgeweights $w(e)$)
  - If there is a negative cycle, then output "negative cycle!!!!"; otherwise, output the shortest lengths of paths to each vertex from v_0

Bellman-Ford algorithm

BellmanFord(G=(V,E), V=$v_0, v_1, \dots, v_n$, starting vertex v_0, edgeweights $w(e)$)

Initialize BF(i,t)=∞ for all $0 \le i, t \le n$

BF(0,0)=0

for t $= 1, \dots, n$:

for $i = 0, \dots, n$:

$$BF(i,t) = \min_{(v_j,v_i)\in E}\left[BF(j,t-1) + w(v_j, v_i)\right]$$,[BF(i, t-1)]

for $i = 0, \dots, n$:

if $BF(i, n-1) \neq BF(i, n)$:

return “negative cycle!!!!”

return [BF(0,n),BF(1,n),BF(2,n),…,BF(n,n)]

Bellman-Ford algorithm, runtime

BellmanFord(G=(V,E), V=$v_0, v_1, \dots, v_n$, starting vertex v_0, edgeweights $w(e)$)

Initialize BF(i,t)=∞ for all $0 \le i, t \le n$

BF(0,0)=0

for t $= 1, \dots, n$: n times

for $i = 0, \dots, n$: n + m times

$$BF(i,t) = \min_{(v_j,v_i)\in E}\left[BF(j,t-1) + w(v_j,v_i)\right]$$,[BF(i, t-1)]

for $i = 0, \dots, n$:

if $BF(i,n-1) \neq BF(i,n)$:

return "negative cycle!!!!"

return [BF(0,n),BF(1,n),BF(2,n),…,BF(n,n)]

Total runtime O(n(n + m))
(if connected graph, then O(nm))

Floyd-Warshall algorithm

- Suppose you wanted the shortest path between all pairs of vertices
- You could run Bellman Ford n times for a total runtime of $O(n^2 m)$
- Floyd-Warshall accomplishes this in $O(n^3)$ time!

Floyd-Warshall algorithm

- Step 1: Define subproblems and corresponding array

Floyd-Warshall algorithm

- Step 1: Define subproblems and corresponding array
  - Let FW(i, j, t) be the shortest path from v_i to v_j using only the vertices $\{v_1, \ldots, v_t\}$ as intermediate vertices

Floyd-Warshall algorithm

- Step 1: Define subproblems and corresponding array
  - Let FW(i, j, t) be the shortest path from v_i to v_j using only the vertices $\{v_1, \ldots, v_t\}$ as intermediate vertices
- Step 2: What are the base cases?

Floyd-Warshall algorithm

- Step 1: Define subproblems and corresponding array
  - Let FW(i, j, t) be the shortest path from v_i to v_j using only the vertices $\{v_1, \ldots, v_t\}$ as intermediate vertices
- Step 2: What are the base cases?

  FW$(i, j, 0) = w(i, j)$

Floyd-Warshall algorithm

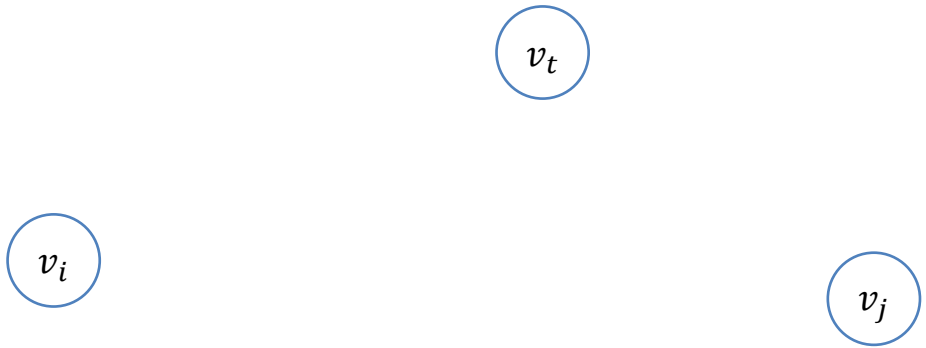
- Let FW(i, j, t) be the shortest path from v_i to v_j using only the vertices $\{v_1, \dots, v_t\}$ as intermediate vertices
- Step 3: Give recursion for subproblems

Floyd-Warshall algorithm

- Let FW(i, j, t) be the shortest path from v_i to v_j using only the vertices $\{v_1, \ldots, v_t\}$ as intermediate vertices
- Step 3: Give recursion for subproblems
  - What can we ask about FW(i, j, t)?

Floyd-Warshall algorithm

- Let FW(i, j, t) be the shortest path from v_i to v_j using only the vertices $\{v_1, \ldots, v_t\}$ as intermediate vertices
- Step 3: Give recursion for subproblems
  - What can we ask about FW(i, j, t)?
    - Does going through v_t improve the length?



Floyd-Warshall algorithm

- Let FW(i, j, t) be the shortest path from v_i to v_j using only the vertices $\{v_1, \ldots, v_t\}$ as intermediate vertices
- Step 3: Give recursion for subproblems
  - What can we ask about FW(i, j, t)?
    - Does going through v_t improve the length?

$$FW(i, j, t) = \min[FW(i, j, t-1), FW(i, t, t-1) + FW(t, j, t-1)]$$

See traveling salesperson problem in textbook for more details

Next lecture

- NP-complete problems
  - Reading: Chapter 8