Dijkstra's and Dynamic Programming

Lecture 7

Timothy Kim
GWU CSCI 6212

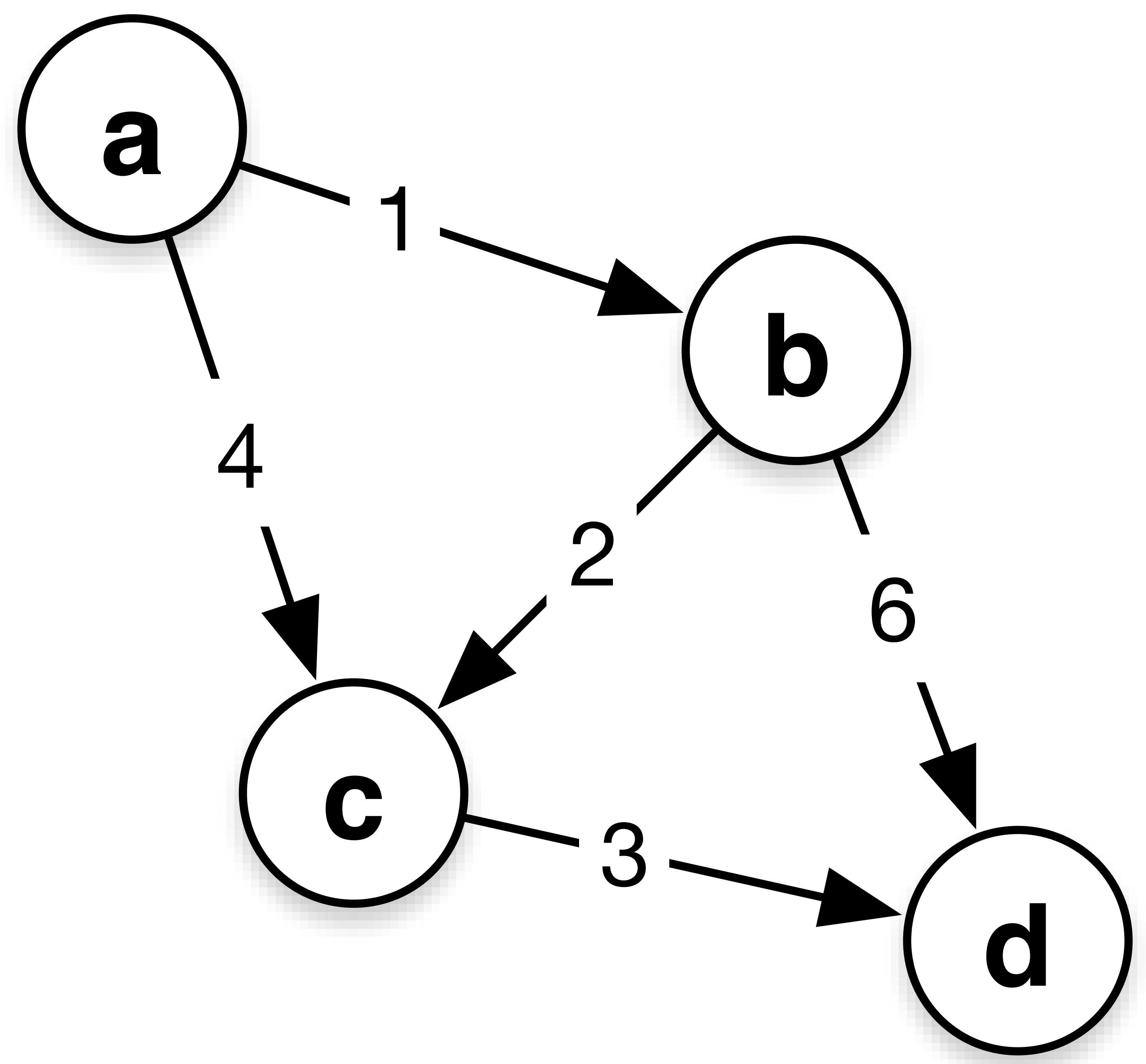
Dijkstra's Shortest Path

- **Problem**: Find the shortest distance between a source vertex to every other vertices.
- **Input**: directed graph G=(V,E), starting vertex, S, and edge lengths, where all edge lengths are non-negative
- **Output**:
    - Lengths of shortest paths from S to all vertices

Algorithm

```
Dijsktra:
  X = {S}
  A[S] = 0
  while X != V:
    for all edges crossing the Cut(X, V-X) -> (u,v):
      (u*, v*) = min(A[u] + length(u,v))
    A[v*] = A[u] + length(u*,v*)
    X += v*
  return A
```

Example



V	a	b	c	d
{a}	0	∞	∞	∞
{a,b}	0	1	∞	∞
{a,b,c}	0	1	3	∞
{a,b,c,d}	0	1	3	6

Proof of Correctness

- **Lemma**:
  - For every directed graph, w/ non negative edge length, Dijsktra's algorithm correctly computes all shortest-path distances. In other words, let A[V] be the output of Dijsktra's, and L[V] be the shortest-path distances, then A[V] = L[V]
- **Proof by Induction** (on the number of iteration)
- **Base Case**: A[s] = L[s] = 0

Proof of correctness

- Inductive Hypothesis:
    - All previous iterations are correct:
        - for all v in X, A[v] = L[v]
- Inductive Step:
    - The algorithm at this point picks an edge (u*, v*) and adds v* to X.
    - A[v*] = A[u*] + LENGTH(u*, v*) ?= L[v*]
    - We know that A[u*] == L[u*] from inductive hypothesis
    - In other words, does all paths s $\rightarrow$ v* has length ≥ L[u*] + LENGTH(u*, v*) ?

Proof of correctness

- Let P $=$ any s $\rightarrow$ v* path.
- P must cross (X, V–X)
- Let (y, z) be the first edge that is on P and crosses the cut(X, V–X), then:
  - Length(s $\rightarrow$ y) ≥ L[y]
  - Length(z $\rightarrow$ v*) ≥ 0 (we don't allow negative edges)
- Length(P) ≥ L[y] + LENGTH(y, z)
- By our algorithm A[u*] + LENGTH(u*, v*) ≤ L[y] + LENGTH(y, z) ≤ Length(P)
- In other words, all paths s $\rightarrow$ v* has length ≥ L[u*] + LENGTH(u*, v*) is true.
- QED

Running Time

- Naive implementation: $O(n^2)$
- Using Heap: $O(m \log n)$

Dynamic Programming

Maximum Weight Independent Set

- Input: Path graph G=(V,E) with non-negative vertices
- Output: An independent set, subset of non-adjacent vertices, with maximum weight
- Brute force solution: exponential running time
- Greedy?
- Dynamic programming?

Principle of Optimality

- "An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision." (See Bellman, 1957, Chap. III.3.)
- In simpler terms, optimal solution must come from "smaller" sub-problems that are optimal.

MWIS Problem

- Assume that we have a solution and work "backwards"
- Let S be the MWIS(G), and v_n = last vertex in G.
- Case 1: $v_n \notin S$,
  - Let G' = G − v_n, then MWIS(G') = S
  - Proof by contradiction (exercise)
- Case 2: $v_n \in S$,
  - Let G'' = G − v_n − v_{n-1}, then MWIS(G'') = S − v_n
  - Proof by contradiction (exercise)

Line of thought

```
MWIS(G):
  if (|G| == 1): return G[1]
  S1 = MWIS(G - V[n])
  S2 = MWIS(G - V[n] - V[n-1])
  return max(S1, S2)
```

- Only if we knew if $v_n \in S$ is true or not, then we can recursively compute G' or G''.
- Since we do not, why not try both and pick the better one?
- But this is same as brute force!

Question

```
MWIS(G):
  if (|G| == 1): return G[1]
  S1 = MWIS(G - V[n])
  S2 = MWIS(G - V[n] - V[n-1])
  return max(S1, S2)
```

- How many distinct calls to MWIS are there?
- O(n), one for each of the distinct prefix of G.
- So let's compute things bottom up!

Algorithm

```
MWIS(G):
  A[0] = 0
  A[1] = V[1]
  for i = 2 to n:
    A[i] = max{A[i-1], A[i-2] + V[i]}
```

Reconstruction

```
S = []
i = n
while i ≥ 1:
  if A[i-1] == A[i]:
    i--
  else:
    S += V[i]
    i -= 2
```

Dynamic Programming

1. Find sub-problems that preserves the Principle of Optimality (requires proof)
2. Derive recurrence from 1
3. Use the recurrence to formulate the algorithm
4. Write a reconstruction algorithm to generate the desired output

Integral Knapsack

- Input:
  - n items with their
    - vi, values (non-negative)
    - wi, sizes (non-negative and integral)
  - W, capacity (non-negative and integral)
- Output:
  - Find S ⊆ {1, 2, ... ,n} such that
    - Maximizes $\sum v_i$
    - Subject to $\sum w_i \leq W$

Principle of Optimality

- Let S be the max-value solution, and impose arbitrary ordering to our items, therefore n is the last item.
- Case 1: $n \notin S$
- Case 2: $n \in S$

Principle of Optimality

- Let S be the max-value solution, and impose arbitrary ordering to our items, therefore n is the last item.
- Case 1: $n \notin S$
    - S must be optimal with the first $n-1$ items with the capacity W.
- Case 2: $n \in S$

Principle of Optimality

- Let S be the max-value solution, and impose arbitrary ordering to our items, therefore n is the last item.
- Case 1: $n \notin S$
    - S must be optimal with the first $n-1$ items with the capacity W.
- Case 2: $n \in S$
    - $S-n$ must be optimal with the first $n-1$ items with capacity $W-w_n$
    - Proof by contradiction (exercise)

Recurrence

- Let $V_{i,x}$ = value of the best solution for $\{1, 2, ..., i\}$ with capacity X.
- $V_{i,x} = \begin{cases} v_{i-1,x} & \text{if } w_i > X \\ \max\{v_{i-1,x}\ ,\ v_i + v_{i-1,\ x-w[i]}\} & \text{otherwise} \end{cases}$

Subproblem Identification

- We need to apply the objective function (sum of v_i) to
    - All possible prefix of item {1, 2, ... n}
    - All possible residual capacity of {0, 1, ..., W}

Algorithm

```
Let A = 2d-Array
Init A[0,x] = 0 for all x = 0, 1, ... ,W
for i = 1 to n:
  for x = 0 to W:
    if (w[i] > x) A[i,x] = A[i-1,x]
    else A[i,x] = max{A[i-1,x], A[i-1,x-w[i]] + v[i]}
return A[n,W]
```

- $n = 4$
- $W = 6$
- $v = \{3, 2, 4, 4\}$
- $w = \{4, 3, 2, 3\}$

6	0	3	3	7	**8**
5	0	3	3	6	8
4	0	3	3	4	4
3	0	0	2	4	4
2	0	0	0	4	4
1	0	0	0	0	0
0	0	0	0	0	0
	0	**1**	**2**	**3**	**4**

```
Let A = 2d-Array
Init A[0,x] = 0 for all x = 0, 1, ... ,W
for i = 1 to n:
  for x = 0 to W:
    if (w[i] > x) A[i,x] = A[i-1,x]
    else A[i,x] = max{A[i-1,x], A[i-1,x-w[i]] + v[i]}
return A[n,W]
```

Example

- $n = 4$
- $W = 6$
- $v = \{3, 2, 4, 4\}$
- $w = \{4, 3, 2, 3\}$

6	0	3	3	7	**8**
5	0	3	3	6	8
4	0	3	3	4	4
3	0	0	2	4	4
2	0	0	0	4	4
1	0	0	0	0	0
0	0	0	0	0	0
	0	**1**	**2**	**3**	**4**

Reconstruction

- Homework

- "I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes. An interesting question is, Where did the name, dynamic programming, come from? The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word research. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term research in his presence. You can imagine how he felt, then, about the term mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word "programming". I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying. I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that it's impossible to use the word dynamic in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities."