Dynamic programming
COMP4500/7500
Advanced Algorithms & Data Structures

November 5, 2019

Overview this week

- Dynamic programming continued:
  - All-pairs shortest paths ($n = |V|$):
    - Straightforward approach ($\Theta(n^4)$)
    - Straightforward improvement ($\Theta(n^3 \lg n)$)
    - Floyd-Warshall algorithm ($\Theta(n^3)$)
    - Johnson's algorithm ($\Theta(n^2 \lg n)$ for sparse graphs) (Note: overview only – not dynamic)
- "Greedy" algorithms:
  - Characteristics
  - Comparison to dynamic programming
  - Examples:
    - Task scheduling
    - Fractional knapsack

Overview today

- Dynamic programming continued:
  - All-pairs shortest paths ($n = |V|$):
    - Straightforward approach ($\Theta(n^4)$)
    - Straightforward improvement ($\Theta(n^3 \lg n)$)
    - Floyd-Warshall algorithm ($\Theta(n^3)$)
    - Johnson's algorithm ($\Theta(n^2 \lg n)$ for sparse graphs)
      (Note: overview only – not dynamic)

Dynamic programming recap

Method for efficiently solving some problems with certain properties:

- optimal substructure and
- overlapping subproblems.

Process:

- Give a (concise & intuitive) recursive definition of the solution;
- Transform into a dynamic programming implementation:
  - Calculate and store solutions to sub-problems in an order that respects sub-problem dependencies.

The key idea:

- No solution to a sub-problem is calculated more than once.
- Massive speed improvements over a naive recursive implementation are possible:
  - from exponential-time to polynomial-time!

Dynamic programming recap

Examples:

- Calculating Fibonacci numbers,
- Calculating longest common subsequences of strings.
- Calculating the fastest order to multiple chains of matrices.

Finding all-pairs shortest paths

Problem: Find the shortest path between all pairs of nodes in a graph.

- Dijkstra's finds all shortest paths from one node.
- Fastest implementation is $O(E + V \lg V)$, which is $O(V^2)$.
- Running Dijkstra's from all nodes is therefore $O(V^3)$. However, this cannot handle negative-weight edges.
- Bellman-Ford is $O(VE)$, which is $O(V^3)$ already, so with an extra loop for each node gives $O(V^4)$.

We can get $O(V^3)$ *and* handle neg-weight edges for all-pairs.

Recursive definition of all-pairs

Simplifying assumptions/notation:

- There are N vertices with ids conveniently $1, 2, ..N$.
- Uses an adjacency-matrix representation, with $weight(i, j)$ as the weight of the edge from vertex i to j
- If no edge exists, the weight is ∞, hence we will always return ∞ as the weight of the shortest path if one does not exist.
- $weight(v, v) = 0$ for all v

Recursive definition of all-pairs

A path from vertex i to j can be either:

- a path with no edges of weight 0, e.g. $\langle i \rangle$ for $i = j$,
- a path with more than one edge consisting of:
  - a path p from vertex i to some vertex k, and
  - the edge (k, j)

  The weight of such a path is $weight(p) + weight(k, j)$

Key insight: *If the shortest path $\langle i, .., k, j \rangle$ from i to j has m edges, then*

- *the path from i to k must have (at most) $m - 1$ edges, and*
- *it must be a shortest path from i to k.*

Recursive definition of all-pairs

Let $shortestPath(i,j)^m$ represent the weight of the shortest path from i to j of at most m edges.

Definition (Shortest path (based on path length))

$$shortestPath(i,j)^0 = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{otherwise} \end{cases}$$

$$shortestPath(i,j)^m = \mathrm{MIN}_{k \in V}(shortestPath(i,k)^{m-1} + weight(k,j))$$

Hence,

$$shortestPath(i,j)^1 = weight(i,j)$$

Each sub-problem is described by 3 parameters, and so a 3D array required to store solutions.

Dynamic implementation of all-pairs

Store **shortestPath**$(\mathbf{i}, \mathbf{j})^{\mathbf{m}}$ at $\mathbf{L}[\mathbf{m}, \mathbf{i}, \mathbf{j}]$ in a $\mathbf{n} \times \mathbf{n} \times \mathbf{n}$ array **L**.

Calculate **L**[**1**], then **L**[**2**] from **L**[**1**], then **L**[**3**] from **L**[**2**] etc.

SLOW-APSP(n)

```
// L[m, i, j] is weight of the shortest path from i to j of at most m edges.
L = new int[n][n][n] // initialise all elements to ∞
L[1] = weights
for m = 2 to n − 1
    d = L[m − 1]
    d′ = L[m]
    for i = 1 to n
        for j = 1 to n
            for k = 1 to n
                d′[i, j] = MIN(d′[i, j], d[i, k] + weight(k, j))
```

$T(n) \in \Theta(n^4)$

Even better

Calculate **L[1]**, then **L[2]** from **L[1]**,
then **L[4]** from **L[2]**), then **L[8]** (from **L[4]**) etc.

FASTER-APSP(n)

```
// L[m, i, j] is weight of the shortest path from i to j of at most m edges.
L = new int[n × n × n] // initialise all elements to ∞
L[1] = weights
m = 1
while m < n − 1
    d = L[m]
    d′ = L[2m]
    for i = 1 to n
        for j = 1 to n
            for k = 1 to n
                d′[i, j] = MIN(d′[i, j], d[i, k] + d[k, j])
    m = 2m
```

$T(n) \in \Theta(n^3 \lg n)$

Floyd-Warshall algorithm

Why did we define our sub-problems in terms of path length?

Instead phrase our sub-problems in terms of which intermediate nodes are in the path.

Floyd-Warshall algorithm

Let

$$shortestPath(i, j, k)$$

be the weight of the shortest path from i to j, using only intermediate vertices $1..k$.

Incrementally add a new node to the intermediate set.

Extending $1..k$ to $1..k+1$, requires checking whether the path formed by:

- going from i to $k+1$ and
- $k+1$ to j

is better than that already found.

Floyd-Warshall algorithm

Let $shortestPath(i, j, k)$ be the weight of the shortest path from i to j, using only intermediate vertices $1..k$.

Definition (Shortest path via a set of intermediate nodes)

$$shortestPath(i, j, 0) = weight(i, j)$$
$$shortestPath(i, j, k+1) =$$
$$\text{MIN}\begin{cases} shortestPath(i, j, k) \\ shortestPath(i, k+1, k) + shortestPath(k+1, j, k)) \end{cases}$$

Note we still have 3 parameters, but have eliminated $\text{MIN}_{k \in V}$.

Floyd-Warshall algorithm

Store **shortestPath(i, j, k)**, at $d_{ij}^{(k)}$. I.e. $d_{ij}^{(k)}$ is the weight of the shortest path from i to j using only intermediate vertices $1..k$,

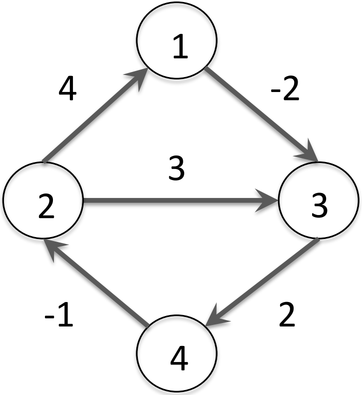
FLOYD-WARSHALL(W)

// W is the weight matrix
$n = W.rows$
$D^{(0)} = W$
for $k = 1$ **to** n
    let $D^{(k)} = (d_{ij}^{(k)})$ be a new $n \times n$ matrix
    for $i = 1$ **to** n
        for $j = 1$ **to** n
            $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
return $D^{(n)}$

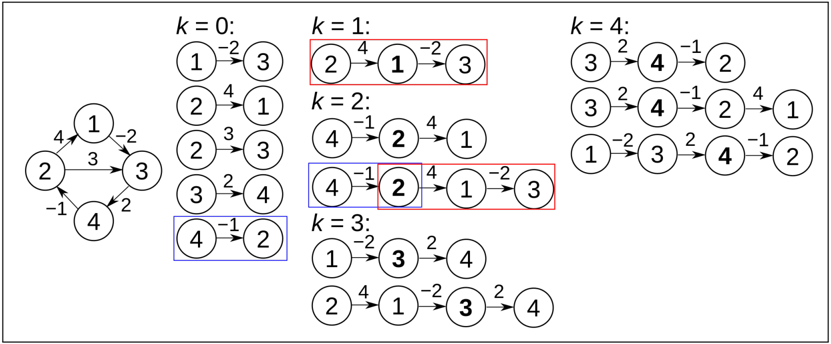
$T(n) \in \Theta(n^3)$

Quick question

Run Floyd-Warshall on the following graph to compute each matrix $D^{(0)}$, $D^{(1)}$ ... $D^{(4)}$.



Example



Aside: Johnson's algorithm

Johnson's algorithm:

- $\Theta(V^3)$ worst case
- but for *sparse graphs* it is $O(V^2 \lg V)$
  (uses an adjacency list representation)
- Strategy:
  - *reweight* to eliminate negative-weight edges
  - add a new "source" vertex
  - Now run Dijkstra's algorithm for each node
- Has relatively high overheads

Dynamic Programming vs Greedy Algorithms

A **dynamic programming** solution might apply to a optimization problem with:

- optimal substructure, e.g.

Definition (Longest common subsequence (LCS))

$$
\begin{aligned}
LCS(\langle\rangle, S_2) &= LCS(S_1, \langle\rangle) = \langle\rangle \\
LCS(S_1.X, S_2.X) &= LCS(S_1, S_2).X \\
LCS(S_1.X, S_2.Y) &= \text{MAX}(LCS(S_1, S_2.Y), LCS(S_1.X, S_2)) \\
&\quad \text{provided } X \neq Y
\end{aligned}
$$

- overlapping subproblems

Dynamic programming solutions solve a problem by:

- solve all of the sub-problems (once each) and then
- use the solutions to choose the sub-problem that will give us the optimal answer.

Greedy choice property

Some optimization problems with *optimal substructure* have the **greedy-choice property**:

- Given a problem, we know which sub-problem will yield an optimal solution *without* having to calculate the solutions to all of the sub-problems it depends on.
- To solve a problem we can make a **greedy choice** (a locally optimal choice) about which sub-problem to solve, and then just solve that one.

If a problem has the greedy choice property then:

Locally optimal choices, lead to a globally optimal solution.

Greedy Algorithms

A **greedy algorithm** may be found for optimization problems with

- optimal substructure, and
- the greedy choice property

Greedy algorithms solve a problem by:

- making a greedy choice (locally optimal choice) and solving (only) the chosen sub-problem.

If applicable, preferable to dynamic programming since we have fewer sub-problems to solve.

Greedy Algorithms we have already seen.

Prim's minimum spanning tree algorithm:

The **minimum spanning tree** of a weighted graph G, that is a superset of tree T (a connected acyclic sub-graph of G) is either:

- Base case: T if the tree is already spanning
- Recursive case: $T \cup \{(u, v)\}$ where (u, v) is the least weight edge leaving T.

I.e. at each stage of the algorithm, we make a **greedy choice** about which edge to include next.

Greedy Algorithms we have already seen.

Kruskals's minimum spanning tree algorithm:

The **minimum spanning tree** of a weighted graph G, that is a superset of a forest of trees T (a spanning acyclic sub-graph of G) is either:

- Base case: T if it is already connected
- Recursive case: $T \cup \{(u, v)\}$ where (u, v) is the least weight edge connecting any two trees in the forest T.

I.e. at each stage of the algorithm, we make a **greedy choice** about which edge to include next.

Greedy Algorithms we have already seen.

Dijkstra's single-source shortest path algorithm:

The **shortest path tree** of a weighted graph G from a source vertex s that is a superset of a shortest-path tree T from s to $G.V - Q$ of the vertices in G is either:

- Base case: T if Q is empty
- Recursive case: $T \cup \{(u, v)\}$ where (u, v) is the edge connecting vertex u in T to the vertex $v \in Q$ that is closest to s (has the highest priority in Q).

If only it were always that simple...

Activity selection problem

Problem: find the combination (subset) of tasks that maximises the number of activities in a finite amount of time

Given:

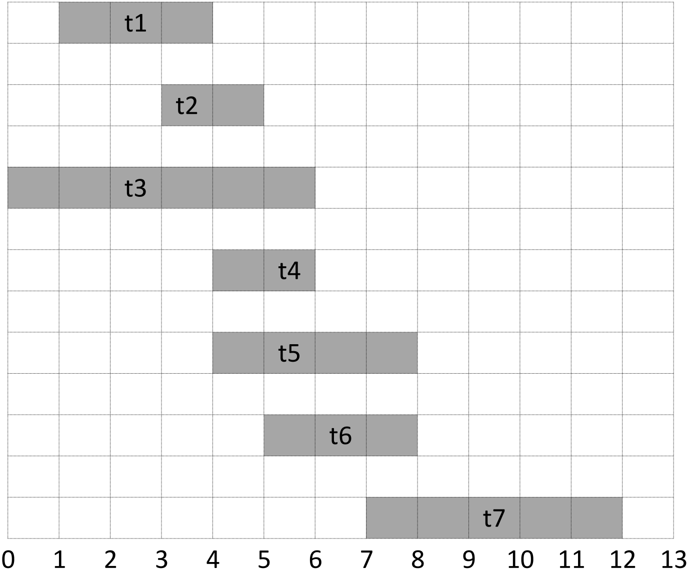
- a list of tasks $t_1, .., t_n$
- their start and finish times:
  $s_1, ..., s_n$
  $f_1, ..., f_n$

Represented as pairs, (start time/finish time):

$$(1,4),(3,5),(0,6)..$$

Which subset of tasks maximises the number of activities?

s	f
1	4
3	5
0	6
4	6
4	8
5	8
7	12



Greedy algorithm: activity selection

Problem: find the combination (subset) of tasks that maximises the number of activities in a finite amount of time

Given:

- a list of tasks $t_1, .., t_n$
- their start and finish times:
  $s_1, ..., s_n$
  $f_1, ..., f_n$

Represented as pairs, (start time/finish time):

$$(1, 4), (3, 5), (0, 6)..$$

Greedy strategy: *always pick the compatible activity (no overlap) that finishes earliest*

Greedy algorithm: activity selection

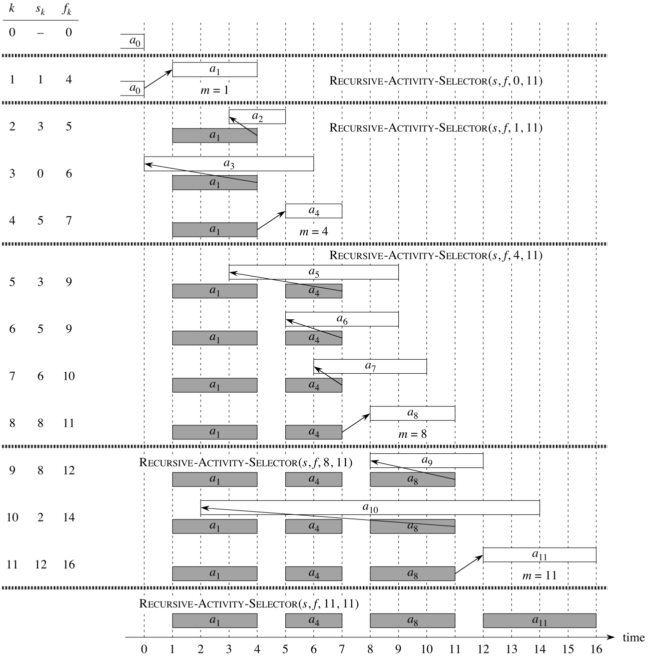
Process:

1. Sort on finish times initially. (For n activities, $\Theta(n \lg n)$)
2. Accumulate compatible activities in set A, initialised to contain the first activity
3. Pick the next activity that starts after the latest finish time so far (k)

GREEDY-ACTIVITY-SELECTOR(s, f)

```
n = s.length
A = {a_1}
k = 1
for m = 2 to n
        if s[m] ≥ f[k]
                A = A ∪ {a_m}
                k = m
return A
```

The loop is $\Theta(n)$, so the dominant factor is the initial sort $\Theta(n \lg n)$


k s_k f_k
0 – 0
1 1 4
2 3 5
3 0 6
4 5 7
5 3 9
6 5 9
7 6 10
8 8 11
9 8 12
10 2 14
11 12 16
RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, 11)
m = 1
RECURSIVE-ACTIVITY-SELECTOR(s, f, 1, 11)
m = 4
RECURSIVE-ACTIVITY-SELECTOR(s, f, 4, 11)
m = 8
RECURSIVE-ACTIVITY-SELECTOR(s, f, 8, 11)
m = 11
RECURSIVE-ACTIVITY-SELECTOR(s, f, 11, 11)
time
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Activity selection

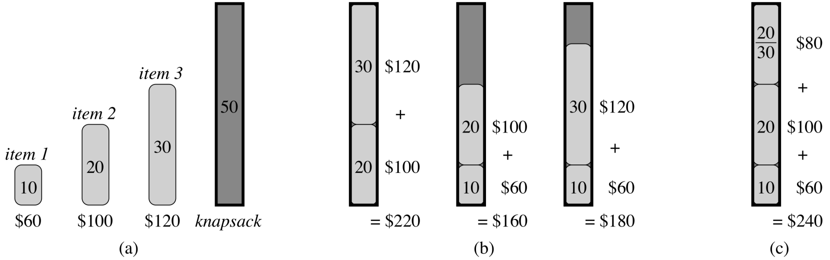
Why does the greedy choice work?

Greedy algorithms: knapsack

Consider a set of items, each with a value v and a weight w. What is the maximum value you can fit into a knapsack, holding a maximum total weight of W?

- Problem 1: **Fractional knapsack**
  You may take part amounts (fractions) of items
  Greedy strategy: take as much as possible of the item that maximises v/w.
  This is optimal
- Problem 2: **Binary (0-1) knapsack**
  You must take all or none of each item
  Greedy strategy: take all of the item that maximises v/w
  This is *not* optimal

Greedy algorithms: knapsack



(a) Three items, knapsack can hold a maximum weight of 50
v/w : Item 1: \$6 Item 2: \$5 Item 3: \$4

(b) For the *binary knapsack*, picking Item 1 is not optimal

(c) For the *fractional knapsack*, picking (all of) Item 1 is optimal

Recap

- Dynamic programming can be used provided
  1. the problem exhibits *optimal substructure*
  2. the problem has *overlapping subproblems*
- Two steps:
  1. Devise an intuitive but inefficient recursive solution
  2. Fill in an array methodically, starting with the base cases
- Covered:
  - All-pairs shortest paths
- Greedy algorithms can be used provided:
  1. the problem exhibits *optimal substructure*
  2. the problem has the *greedy-choice property*
- Covered:
  - Activity scheduling
  - Knapsack problems

(Next week: amortised analysis)