Dynamic Programming (Ch. 15)

Good solution for problems that take exponential time to solve by brute-force methods.

Typically applied to optimization problems, where there are many possible solutions, each solution has a particular value, and we wish to find the solution with an optimal (minimal or maximal) value.

For many of these problems, we must consider all subsets of a possibly very large set, so there are 2^n possible solutions -- too many to consider sequentially for large n.

Dynamic Programming

Divide-and-conquer algorithms find an optimal solution by partitioning a problem into independent subproblems, solving the subproblems recursively, and then combining the solutions to solve the original problem.

Dynamic programming is applicable when the sub-problems are not independent, i.e. when they share subsubproblems.

Dynamic Programming

Developed by Richard Bellman in the 1950s. Not a specific algorithm, but a technique (like divide-and-conquer).

This process takes advantage of the fact that subproblems have optimal solutions that lead to an overall optimal solution.

DP is often useful for problems with overlapping subproblems. These algorithms typically solve each subproblem once, record the result in a table, and use the information from the table to solve larger problems.

Computing the nth Fibonacci number is an example of a non-optimization problem to which dynamic programming can be applied.

F(n) = F(n-1) + F(n-2) for n >= 2

F(0) = 0 and F(1) = 1.

Fibonacci Numbers

A straightforward, but inefficient algorithm to compute the nth Fibonacci number uses a top-down approach:

RFibonacci (n)
1. if n = 0 then return 0
2. else if n = 1 then return 1
3. else return RFibonacci (n-1) + RFibonacci (n-2)

This approach uses calls on the same numbers many times, leading to an exponential running time.

Fibonacci Numbers

A more efficient, bottom-up approach starts with 0 and works up to n, requiring only n values to be computed:

Fibonacci(n)
1. f[0] = 0
2. f[1] = 1
3. **for** i = 2 … n
4. f[i] = f[i-1] + f[i-2]
5. return f[n]

The technique of storing answers to smaller subproblems is one type of bottom-up programming.

All-Pairs Shortest Paths (Ch. 25)

The all-pairs shortest path problem (APSP) is example where DP can help.
input: **a directed graph G = (V, E) with edge weights**
goal: **find a minimum weight (shortest) path between every pair of vertices in V**
Can we do this with algorithms we've already seen, say SSSP algorithms?

Solution 1: run Dijkstra's algorithm V times, once with each $v \in V$ as the source node (requires no negative-weight edges in E)
If G is dense with an array implementation of Q
$O(V \cdot V^2) = O(V^3)$ time
If G is sparse with a binary heap implementation of Q
$O(V \cdot ((V + E) \log V)) = O(V^2 \log V + VE \log V)$ time

All-Pairs Shortest Paths

Solution 2: run the Bellman-Ford algorithm V times (negative edge weights allowed), once from each vertex.
$O(V^2E)$, which on a dense graph is $O(V^4)$

Solution 3: Use an algorithm designed for the APSP problem.
E.g., Floyd's Algorithm (also allows negative edge weights)
introduces a *dynamic programming* technique that
uses an adjacency matrix representation of G = (V, E)

Floyd's algorithm uses the fact that all shortest paths are composed of shortest sub-paths.

No shortest-path algorithm can work on a graph with negative edge weights.

Warshall's Transitive Closure Algorithm

Input: Adjacency matrix A of G as matrix of 1s and 0's
Output: Transitive Closure (reachability matrix) $R^{(n)}$ of G

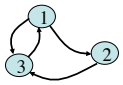
Assumes vertices are numbered 1 to |V|, |V| = n and there are no edge weights. Finds a series of boolean matrices $R^{(0)}$, ..., $R^{(n)}$

Solution for $R^{(n)}$:
Define $r_{ij}^{(k)}$ as the element in the ith row and jth column to be 1 iff there is a path between vertices i and j using only vertices numbered ≤ k.
$R^{(0)}$ = A, original adjacency matrix (only 1's in matrix are direct edges)
$R^{(n)}$ = the matrix we want to compute

$R^{(k)}$'s elements are: $R^{(k)}[i, j] = r_{ij}^{(k)} = r_{ij}^{(k-1)} \vee (r_{ik}^{(k-1)} \wedge r_{kj}^{(k-1)})$

Lowercase r is element in matrix and capital R is entire matrix.

Warshall's Algorithm



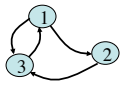
$R^{(0)} = A =$

0	1	1
0	0	1
1	0	0

Matrix $R^{(0)}$ contains the nodes reachable in one hop

For $R^{(1)}$, there is a 1 in row 3, col 1 and there are 1s in row 1, columns 2 and 3, so put 1s in positions 3,2 and 3,3.

Warshall's Algorithm



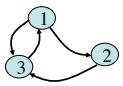
$R^{(1)} =$

0	1	1
0	0	1
1	1	1

Matrix $R^{(1)}$ contains the nodes reachable in one hop or on 2 hop paths that go through vertex 1.

For $R^{(2)}$, there is no change because 1 can get to 3 through 2 but there is already a direct path between 1 and 3.

Warshall's Algorithm



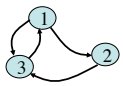
$R^{(2)} =$

0	1	1
0	0	1
1	1	1

Matrix $R^{(2)}$ contains the nodes reachable in one hop or on paths that go through vertices 1 or 2.

For $R^{(3)}$, there is a 1 in row 1, col 3 and col 1, row 3, so put a 1 in position 1,1. Also, there is a 1 in row 2, col 3 and col 2, row 3, so put a 1 in position 2,1. Also, there is a 1 in row 2, col 3 and col 2, row 3, so put a 1 in position 2,2.

Warshall's Algorithm



$R^{(3)} =$

1	1	1
1	1	1
1	1	1

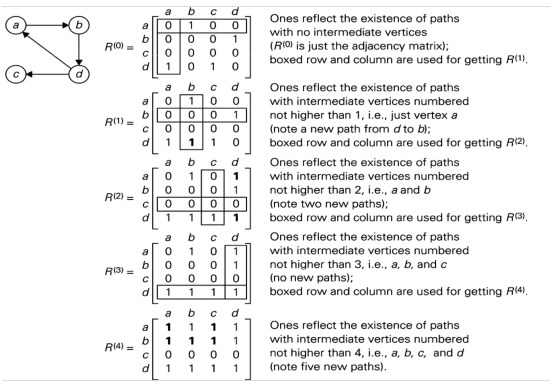
Matrix $R^{(3)}$ contains the vertices reachable in one hop or on paths that go through vertices 1, 2, and 3.

Warshall's Algorithm

- Main idea: a path exists between two vertices i, j, iff
  - there is an edge from i to j; or
  - there is a path from i to j going through vertex 1; or
  - there is a path from i to j going through vertex 1 and/or 2; or
  - there is a path from i to j going through vertex 1, 2, and/or 3; or
  - ...
  - there is a path from i to j going through any of the other vertices

R_0

0	0	1	0
1	0	0	1
0	0	0	0
0	1	0	0



Warshall's Algorithm

Warshall (A[1…n,1…n])

1. n = rows[A]
2. $R^{(0)} = A$
3. for k = 1 to n do
4. for i = 1 to n do
5. for j = 1 to n do
6. $R_{ij}^{(k)} = R_{ij}^{(k-1)} \vee R_{ik}^{(k-1)} \wedge R_{kj}^{(k-1)}$
7. return $R^{(n)}$

Time efficiency?

Space efficiency?

Floyd's APSP Algorithm

Input: Adjacency matrix A
Output: Shortest path matrix $D^{(n)}$ and predecessor matrix $\Pi^{(n)}$

Observation: When G contains no negative-weight cycles, all shortest paths consist of at most |V| – 1 edges

Assumes vertices are numbered 1 to |V|

Relies on the *Optimal Substructure Property*:
All sub-paths of a shortest path are shortest paths.

Solution for D:

Define $D^{(k)}[i, j] = d_{ij}^{(k)}$ as the minimum weight of any path from vertex i to vertex j, such that all intermediate vertices are in {1, 2, 3, ..., k}

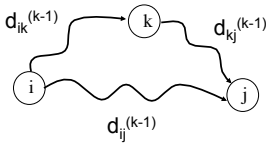
$D^{(0)}$ = A, original adjacency matrix (only paths are single edges)

$D^{(n)}$ the matrix we want to compute

$D^{(k)}$'s elements are: $D^{(k)}[i, j] = d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$

Recursive Solution for $D^{(k)}$

$$D^{(k)}[i, j] = d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$$



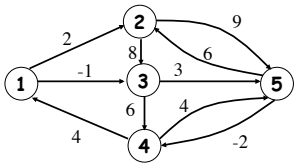
The only intermediate nodes on the paths from i to j, i to k or k to j are in the set of vertices {1, 2, 3, ..., k-1}.
If k is included in shortest i to j path, then a shortest path has been found that includes k.
If k is not included in shortest i to j path, then the shortest path still only includes vertices in the set 1…k-1.

Floyd's APSP Algorithm

Use adjacency matrix A for G = (V, E):

$$A[i, j] = a_{ij} = \begin{cases} w(i, j) & \text{if } (i,j) \in E \\ 0 & \text{if } i = j \\ \infty & \text{if } i \neq j \text{ and } (i, j) \notin E \end{cases}$$

	1	2	3	4	5
1	0	2	-1	∞	∞
2	∞	0	8	∞	9
3	∞	∞	0	6	3
4	4	∞	∞	0	4
5	∞	6	∞	-2	0

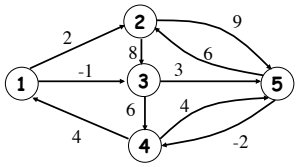


Floyd's APSP Algorithm

Use adjacency matrix Π to keep track of predecessors:

$$\pi^{(0)}_{ij} = \begin{cases} i & \text{if } i \neq j \text{ and } w(i,j) < \infty \\ \varnothing & \text{if } i = j \text{ or } w(i,j) = \infty \end{cases}$$

	1	2	3	4	5
1	Ø	1	1	Ø	Ø
2	Ø	Ø	2	Ø	2
3	Ø	Ø	Ø	3	3
4	4	Ø	Ø	Ø	4
5	Ø	5	Ø	5	Ø



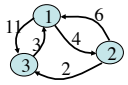
π_{ij} is predecessor of j on some shortest path from i

Floyd's Simplified APSP Algorithm

Floyd-Warshall-APSP(D,Π)

```
1. n = D.rows
2. for k = 1 to n
3.     for i = 1 to n
4.         for j = 1 to n
5.             if  d[ i ][ j ] > d[ i ][ k ] + d[ k ][ j ]
6.                 d[ i ][ j ] = d[ i ][ k ] + d[ k ][ j ]
7.                 π[ i ][ j ]= π[ k ][ j ]
8. return D and Π
```

Operation of F-APSP Algorithm



$D^{(0)} = A =$

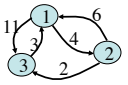
0	4	11
6	0	2
3	∞	0

3->1, 1->2 = 7

$\Pi^{(0)} =$

∅	1	1
2	∅	2
3	∅	∅

Operation of F-APSP Algorithm



$D^{(1)} =$

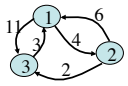
0	4	11
6	0	2
3	7	0

1->2, 2->3 = 6

$\Pi^{(1)} =$

∅	1	1
2	∅	2
3	1	∅

Operation of F-APSP Algorithm



$D^{(2)} =$

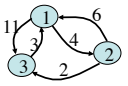
0	4	6
6	0	2
3	7	0

2->3, 3->1 = 5

$\Pi^{(2)} =$

∅	1	2
2	∅	2
3	1	∅

Operation of F-APSP Algorithm



$D^{(0)} = A =$

0	4	11
6	0	2
3	∞	0

$D^{(1)}$

0	4	11
6	0	2
3	7	0

3→ 1 → 2

$D^{(2)}$

0	4	6
6	0	2
3	7	0

1→ 2 → 3

$D^{(3)}$

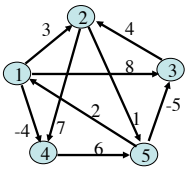
0	4	6
5	0	2
3	7	0

2→ 3 → 1

F-APSP Algorithm

$D^{(0)} = A =$

0	3	8	-4	∞
∞	0	∞	7	1
∞	4	0	∞	∞
∞	∞	∞	0	6
2	∞	-5	∞	0



$D^{(1)} =$

0	3	8	-4	∞
∞	0	∞	7	1
∞	4	0	∞	∞
∞	∞	∞	0	6
2	5	-5	-2	0

The darkened squares represent shorter paths through vertex 1.
5→1→2, 5→1→4

Then look at all paths that go through vertex 2.

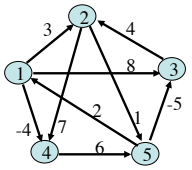
F-APSP Algorithm

$D^{(2)} =$

0	3	8	-4	4
∞	0	∞	7	1
∞	4	0	11	5
∞	∞	∞	0	6
2	5	-5	-2	0

The darkened squares represent shorter paths through vertex 2.
1→2→5; 3→2→4;
3→2→5

Then look at all paths that go through vertex 3.



F-APSP Algorithm

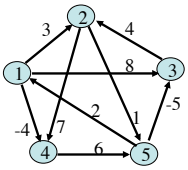
$D^{(3)} =$

0	3	8	-4	4
∞	0	∞	7	1
∞	4	0	11	5
∞	∞	∞	0	6
2	-1	-5	-2	0

The darkened square represents a shorter path through vertex 3.
5→3→2

D(3) is all direct routes or routes through nodes 1, 2, and 3.

Then look at all paths that go through vertex 4.



F-APSP Algorithm

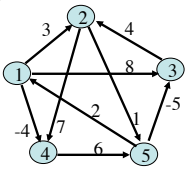
$D^{(4)} =$

0	3	8	-4	2
∞	0	∞	7	1
∞	4	0	11	5
∞	∞	∞	0	6
2	-1	-5	-2	0

The darkened square represents a shorter path through vertex 4.
1→4→5

D(4) is all direct routes or routes through nodes 1, 2, 3, and 4.

Then look at all paths that go through vertex 5.



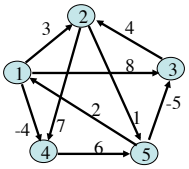
F-APSP Algorithm

$D^{(5)} =$

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

1→4→5→3→2, 1→4→5→3
2→5→1, 2→5→3, 2→5→1→4
3→2→5→1, 3→2→5→1→4
4→5→1, 4→5→3→2, 4→5→3

The darkened squares represent shorter paths through vertices 1...5.

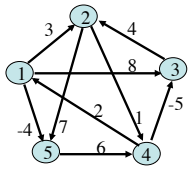


FW-APSP Algorithm

4 → 1 → 2 = 5
4 → 1 → 5 = -2

$D^{(0)} = D =$

0	3	8	∞	-4
∞	0	∞	1	7
∞	4	0	∞	∞
2	∞	-5	0	∞
∞	∞	∞	6	0



$D^{(1)} =$

0	3	8	∞	-4
∞	0	∞	1	7
∞	4	0	∞	∞
2	5	-5	0	-2
∞	∞	∞	6	0

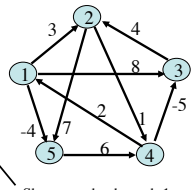
1 → 2 → 4 = 4

3 → 2 → 4 = 5
3 → 2 → 5 = 11

FW-APSP Algorithm

$\pi^{(0)} = A =$

∅	1	1	∅	1
∅	∅	∅	2	2
∅	3	∅	∅	∅
4	**∅**	4	∅	**∅**
∅	∅	∅	5	∅



$\pi^{(1)} =$

∅	1	1	**∅**	1
∅	∅	∅	2	2
∅	3	∅	**∅**	**∅**
4	1	4	∅	1
∅	∅	∅	5	∅

Shorter paths through 1

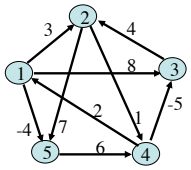
Shorter paths through 2

F-APSP Algorithm

$D^{(2)} =$

0	3	**8**	4	-4
∞	0	**∞**	1	7
∞	**4**	**0**	**5**	**11**
2	**5**	-5	0	-2
∞	∞	**∞**	6	0

4 → 3 → 2 = -1

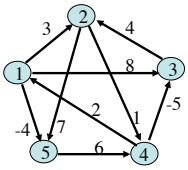


F-APSP Algorithm

$\pi^{(2)} =$

∅	1	1	2	1
∅	∅	∅	2	2
∅	3	∅	2	2
4	**1**	4	∅	1
∅	∅	∅	5	∅

Shorter path through 3



F-APSP Algorithm

$D^{(3)} =$

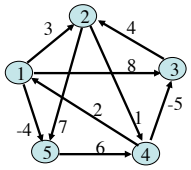
0	3	**8**	**4**	-4
∞	0	**∞**	1	**7**
∞	4	0	5	**11**
2	**-1**	**-5**	**0**	**-2**
∞	**∞**	**∞**	6	0

2 → 4 → 3 = -4
1 → 2 → 4 → 3 = -1

2 → 4 → 1 → 5 = -1
3 → 2 → 4 → 1 → 5 = 3

2 → 4 → 1 = 3
3 → 2 → 4 → 1 = 7

5 → 4 → 1 = 8
5 → 4 → 3 → 2 = 5
5 → 4 → 3 = 1

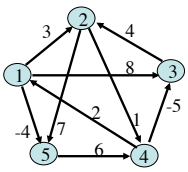


F-APSP Algorithm

$D^{(3)} =$

∅	1	**1**	2	1
∅	∅	**∅**	2	**2**
∅	3	∅	2	**2**
4	3	4	∅	1
∅	**∅**	**∅**	5	∅

Shorter paths through 4

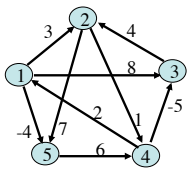


F-APSP Algorithm

$D^{(4)} =$

0	**3**	**-1**	**4**	-4
3	0	-4	1	7
7	4	0	5	11
2	-1	-5	0	-2
8	**5**	**1**	**6**	**0**

1 → 5 → 4 = 2
1 → 5 → 4 → 3 = -3
1 → 5 → 4 → 3 → 2 = 1

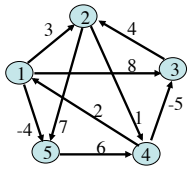


F-APSP Algorithm

$\pi^{(4)} =$

∅	1	4	2	1
4	∅	4	2	1
4	3	∅	2	1
4	3	4	∅	1
4	3	4	5	∅

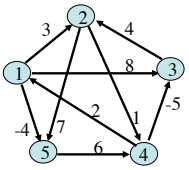
Shorter paths through 5



F-APSP Algorithm

$D^{(5)} =$

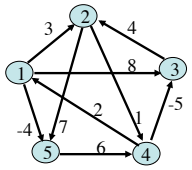
0	1	-3	2	-4
3	0	-4	1	7
7	4	0	5	11
2	-1	-5	0	-2
8	5	1	6	0



F-APSP Algorithm

$\pi^{(5)} =$

∅	3	4	5	1
4	∅	4	2	1
4	3	∅	2	1
4	3	4	∅	1
4	3	4	5	∅



Running Time of Floyd's-APSP

Lines 3 – 6: $|V^3|$ time for triply-nested **for** loops

Overall running time = $\theta(V^3)$

The code is tight, with no elaborate data structures and so the constant hidden in the θ-notation is small.

Much better than exponential time! So big win!