COMP20007 Design of Algorithms

Dynamic Programming Part 1: Warshall and Floyd algorithms

Daniel Beck

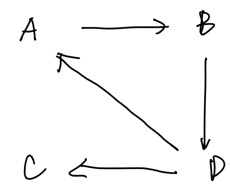
Lecture 19

Semester 1, 2020

Transitive closure problem

for each node v, what other nodes in the graph I can reach if I start from node v.

Graph:



Adjacency matrix

	A	B	C	D
A	0	1	0	0
B	0	0	1	0
C	0	0	0	0
D	1	0	1	0

transitive closure problem

look at the node it can reach

	A	B	C	D
A	1	1	1	1
B	1	1	1	1
C	0	0	1	0
D	1	1	1	1

① one way is
for each node, do DFS

lots of reduntant work

② dynamic programming

Fibonacci Numbers

$$0, 1, 1, 2, 3, 5, 8, 13, \ldots$$

$$F(n) = F(n-1) + F(n-2), \qquad n > 1,$$
$$F(0) = 1, \qquad F(1) = 1.$$

Fibonacci Numbers

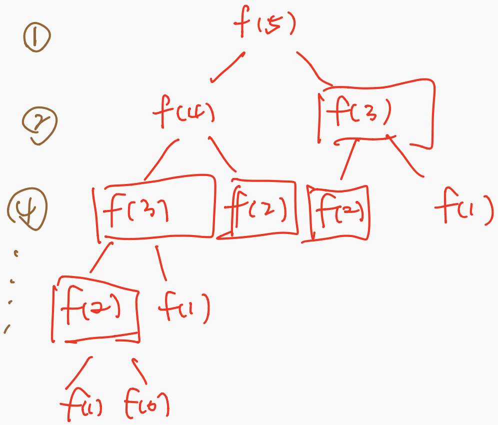
$$0, 1, 1, 2, 3, 5, 8, 13, \ldots$$

$$F(n) = F(n-1) + F(n-2), \qquad n > 1,$$
$$F(0) = 1, \qquad F(1) = 1.$$

function $\textsc{Fibonacci}(n)$
    if $n == 0$ or $n == 1$ **then return** 1
    return $\textsc{Fibonacci}(n-1) + \textsc{Fibonacci}(n-2)$

Fibonacci Numbers

function FIBONACCI(n)
  if $n == 0$ or $n == 1$ **then return** 1
  return FIBONACCI($n - 1$) + FIBONACCI($n - 2$)



→ trading memory for speed

Storing Intermediate Solutions

- Allocate an array of size n to store previous solutions.

Storing Intermediate Solutions

- Allocate an array of size n to store previous solutions.

function FIBONACCIDP(n)

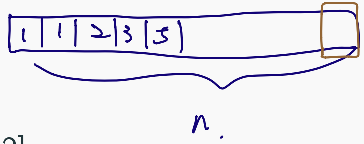
    $F[0] \leftarrow 1$

    $F[1] \leftarrow 1$

    for $i = 2$ to n **do**

        $F[i] = F[i-1] + F[i-2]$

    return $F[n]$



Storing Intermediate Solutions

- Allocate an array of size n to store previous solutions.

function $\textsc{FibonacciDP}(n)$
  $F[0] \leftarrow 1$
  $F[1] \leftarrow 1$
  for $i = 2$ to n **do**
    $F[i] = F[i-1] + F[i-2]$
  return $F[n]$

- From exponential to linear complexity.

Dynamic Programming

Dynamic Programming

- The solution to a problem can be broken into solutions to subproblems (recurrence relations).

Dynamic Programming

- The solution to a problem can be broken into solutions to subproblems (recurrence relations).
- Solutions to subproblems can **overlap** (calls to F for all values lesser than n).
  - Allocates extra memory to store solutions to subproblems.

Dynamic Programming

- The solution to a problem can be broken into solutions to subproblems (recurrence relations).
- Solutions to subproblems can **overlap** (calls to F for all values lesser than n).
  - Allocates extra memory to store solutions to subproblems.
- DP is mostly related to optimisation problems (but not always, see Fibonacci).
  - Optimal solution should be obtained through optimal solutions to subproblems (not always the case).

Transitive Closure using DP

Goal: find all node pairs that have a path between them.

Transitive Closure using DP

Goal: find all node pairs that have a path between them.

- The solution to a problem can be broken into solutions to subproblems.

Transitive Closure using DP

Goal: find all node pairs that have a path between them.

- The solution to a problem can be broken into solutions to subproblems.
  - If there's a path between two nodes i and j which are not directly connected, that path has to go through at least another node k. Therefore, we only need to find if the pairs (i,k) and (k,j) have paths.

$$i \to \cdots \quad k \to \cdots \quad j$$

Transitive Closure using DP

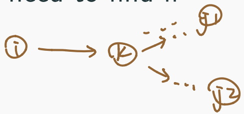
Goal: find all node pairs that have a path between them.

- The solution to a problem can be broken into solutions to subproblems.
  - If there's a path between two nodes i and j which are not directly connected, that path has to go through at least another node k. Therefore, we only need to find if the pairs (i,k) and (k,j) have paths.
- Solutions to subproblems can overlap.

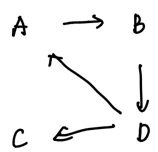
Transitive Closure using DP

Goal: find all node pairs that have a path between them.

- The solution to a problem can be broken into solutions to subproblems.
  - If there's a path between two nodes i and j which are not directly connected, that path has to go through at least another node k. Therefore, we only need to find if the pairs (i,k) and (k,j) have paths.



- Solutions to subproblems can overlap.
  - If the pairs (i,j_1) and (i,j_2) have paths that go through k, then finding if the pair (i,k) has a path is part of the solutions for both problems.



Adjacency matrix
(allow no point in between).

	A	B	C	D
A	0	1	0	0
B	0	0	0	1
C	0	0	0	0
D	1	0	1	0

AND operation
look at the row and column of ~~the~~ node allowed ~~to~~ add in

Allow A in between
don't have edge to itself

	A	B	C	D
A	0	1	0	0
B	0	0	0	1
C	0	0	0	0
D	1	1	1	0

= 1 change

(not require that
⇒ if B can reach D directly without going to A, we can still set that to 1.

allow A . B in between

	A	B	C	D.
A	0	1	0	1
B	0	0	0	1
C	0	0	0	0
D	1	1	1	1

* If 1, just copy
if 0, look at matrix

C is no help, just copy the matrix

allow A, B, C

	A	B	C	D
A	0	1	0	1
B	0	0	0	1
C	0	0	0	0
D	1	1	1	1

allow A B, C, D

	A	B	C	D
A	1	1	1	1
B	1	1	1	1
C	0	0	0	0
D	1	1	1	1

build matrix $O(n^2)$
how many matrix $O(n^3)$

Warshall's Algorithm

- Assume nodes can be numbered from 1 to n, with A being the adjacency matrix.

Warshall's Algorithm

- Assume nodes can be numbered from 1 to n, with A being the adjacency matrix.

all pairs connected
with 0 node
between

$$R^0_{ij} = A[i,j]$$

→ adjacency matrix

$$R^k_{ij} = R^{k-1}_{ij} \text{ or } (R^{k-1}_{ik} \text{ and } R^{k-1}_{kj})$$

already
connected

if 1 → 1
0 → check condition

path from i to k

if both 1 → become 1

path from k to j

Warshall's Algorithm

- Assume nodes can be numbered from 1 to n, with A being the adjacency matrix.

$$
\begin{aligned}
R^0_{ij} &= A[i,j] \\
R^k_{ij} &= R^{k-1}_{ij} \text{ or } (R^{k-1}_{ik} \text{ and } R^{k-1}_{kj})
\end{aligned}
$$

function $\textsc{Warshall}(A[1..n, 1..n])$
    $R^0 \leftarrow A$
    for $k \leftarrow 1$ to n **do** R^k
        for $i \leftarrow 1$ to n **do**
            for $j \leftarrow 1$ to n **do**
                $R^k[i,j] \leftarrow R^{k-1}[i,j]$ **or** $(R^{k-1}[i,k]$ **and** $R^{k-1}[k,j])$
    return R^n

Warshall's Algorithm

- We can allow input A to be used for the output, simplifying things.

  adjacency matrix

Warshall's Algorithm

- We can allow input A to be used for the output, simplifying things.
- Namely, if $R^{k-1}[i, k]$ (that is, $A[i, k]$) is 0 then the assignment is doing nothing. And if it is 1, and if $A[k, j]$ is also 1, then $A[i, j]$ gets set to 1.

$0 \to 1$ ✓

$1 \to 0$ X

Warshall's Algorithm

- We can allow input A to be used for the output, simplifying things.
- Namely, if $R^{k-1}[i,k]$ (that is, $A[i,k]$) is 0 then the assignment is doing nothing. And if it is 1, and if $A[k,j]$ is also 1, then $A[i,j]$ gets set to 1.

for $k \leftarrow 1$ to n **do**
    for $i \leftarrow 1$ to n **do**
        for $j \leftarrow 1$ to n **do**
            if $A[i,k]$ **then**   if there is a path between i, k
                if $A[k,j]$ **then**   if there is a path between k, j
                    $A[i,j] \leftarrow 1$

Warshall's Algorithm

- Now we notice that $A[i, k]$ does not depend on j, so testing it can be moved outside the innermost loop.

Warshall's Algorithm

- Now we notice that $A[i, k]$ does not depend on j, so testing it can be moved outside the innermost loop.

for $k \leftarrow 1$ to n **do**
    for $i \leftarrow 1$ to n **do**
        if $A[i, k]$ **then** one check instead of n checks
            for $j \leftarrow 1$ to n **do** $\rightarrow$ faster
                if $A[k, j]$ **then**
                    $A[i, j] \leftarrow 1$

Warshall's Algorithm

- Now we notice that $A[i,k]$ does not depend on j, so testing it can be moved outside the innermost loop.

```
for k ← 1 to n do
    for i ← 1 to n do
        if A[i, k] then
            for j ← 1 to n do
                if A[k, j] then
                    A[i, j] ← 1
```

- Can use bitstring operations.

Analysis of Warshall's Algorithm

- Straightforward analysis: $\Theta(n^3)$ in all cases.

Analysis of Warshall's Algorithm

- Straightforward analysis: $\Theta(n^3)$ in all cases.
- In practice:
  - Ideal for dense graphs.

Analysis of Warshall's Algorithm

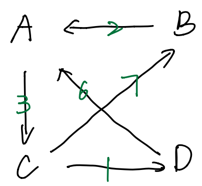
- Straightforward analysis: $\Theta(n^3)$ in all cases.
- In practice:
  - Ideal for dense graphs.
  - Not the best for sparse graphs (#edges $\in O(n)$): DFS from each node tends to perform better.

Floyd's Algorithm: All-Pairs Shortest-Paths

- Floyd's algorithm solves the all-pairs shortest-path problem for weighted graphs with positive weights.

directed + no weight → transitive closure problem.

FLOYD



	A	B	C	D
A	∞	∞	3	∞
B	2	∞	∞	∞
C	∞	7	∞	1
D	6	∞	∞	∞

① allow A

	A	B	C	D.
A	∞	∞	3	∞
B	2	∞	5	∞
C	∞	7	∞	1
D	6	∞	9	∞

② allow A, B

	A	B	C	D.
A	∞	∞	3	∞
B	2	∞	5	∞
C	9	7	12	1
D	6	∞	9	∞

③ allow A B C

	A	B	C	D
A	12	10	3	4
B	2	12	5	6
C	9	7	12	1
D	6	16	9	10

④ allow A B C D

	A	B	C	D
A	10	10	3	4
B	2	12	5	6
C	7	7	10	1
D	6	16	9	10

Floyd's Algorithm: All-Pairs Shortest-Paths

- Floyd's algorithm solves the all-pairs shortest-path problem for weighted graphs with positive weights.
- Similar to Warshall's, but uses a weight matrix W instead of adjacency matrix A (with ∞ values for missing edges)

Floyd's Algorithm: All-Pairs Shortest-Paths

- Floyd's algorithm solves the all-pairs shortest-path problem for weighted graphs with positive weights.
- Similar to Warshall's, but uses a weight matrix W instead of adjacency matrix A (with ∞ values for missing edges)
- It works for directed as well as undirected graphs.

Floyd's Algorithm

- The recurrence follows Warshall's closely:

Floyd's Algorithm

- The recurrence follows Warshall's closely:

Distance matrix

update distance matrix

$$
\begin{aligned}
D_{ij}^{0} &= W[i,j] \\
D_{ij}^{k} &= min(D_{ij}^{k-1}, \underbrace{D_{ik}^{k-1} + D_{kj}^{k-1}}_{\text{potential new part including } k})
\end{aligned}
$$

path already there

Floyd's Algorithm

- The recurrence follows Warshall's closely:

$$
\begin{aligned}
D_{ij}^{0} &= W[i,j] \\
D_{ij}^{k} &= min(D_{ij}^{k-1}, D_{ik}^{k-1} + D_{kj}^{k-1})
\end{aligned}
$$

function $\textsc{Floyd}(W[1..n, 1..n])$
 $D \leftarrow W$
 for $k \leftarrow 1$ to n **do**
  for $i \leftarrow 1$ to n **do**
   for $j \leftarrow 1$ to n **do**
    $D[i,j] \leftarrow min(D[i,j], D[i,k] + D[k,j])$
 return D

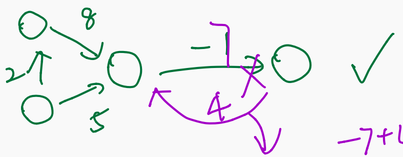
$\Theta(n^3)$

Negative weights

- Negative weights are not necessarily a problem, but negative cycles are.

Negative weights

- Negative weights are not necessarily a problem, but negative cycles are.
- These trigger arbitrarily low values for the paths involved.



$-7+4-7+4+\cdots \to -\infty$

not work for negative cycle

Negative weights

- Negative weights are not necessarily a problem, but negative cycles are.
- These trigger arbitrarily low values for the paths involved.
- Floyd's algorithm can be adapted to detect negative cycles (by looking if diagonal values become negative).

go back to same node with negative weight

Summary

- Dynamic programming is another technique that trades memory for speed.

Summary

- Dynamic programming is another technique that trades memory for speed.
  - Breaks into subproblems.
  - Store overlapping solutions in memory.

Summary

- Dynamic programming is another technique that trades memory for speed.
  - Breaks into subproblems.
  - Store overlapping solutions in memory.
- Warshall's algorithm: find the transitive closure of a graph.

Summary

- Dynamic programming is another technique that trades memory for speed.
  - Breaks into subproblems.
  - Store overlapping solutions in memory.
- Warshall's algorithm: find the transitive closure of a graph.
- Floyd's algorithm: all-pairs shortest paths.

Summary

- Dynamic programming is another technique that trades memory for speed.
  - Breaks into subproblems.
  - Store overlapping solutions in memory.
- Warshall's algorithm: find the transitive closure of a graph.
- Floyd's algorithm: all-pairs shortest paths.

Next lecture: Dynamic Programming part 2.