COMP20007 Design of Algorithms

Dynamic Programming: Part 1

Daniel Beck

Lecture 12

Semester 1, 2023

Fibonacci Numbers

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

$$\begin{gathered} F(n) = F(n-1) + F(n-2), \qquad n > 1, \\ F(0) = 1, \qquad F(1) = 1. \end{gathered}$$

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

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

- 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 smaller than n).
  - Allocates extra memory to store solutions to subproblems.

The Knapsack Problem

Given n items with

- weights: $w_1, w_2, \ldots, w_n$
- values: $v_1, v_2, \ldots, v_n$
- knapsack of capacity W

find the most valuable selection of items that will fit in the knapsack.

We assume that all entities involved are positive integers.

The Knapsack Problem

- weights: $w_1, w_2, \ldots, w_n$
- values: $v_1, v_2, \ldots, v_n$
- knapsack of capacity W

Brute-force solution:

- Try all possible subsets of items, return the most valuable that fits in the knapsack.
- $\Theta(2^n)$

The Knapsack Problem

A Greedy algorithm:

- Assume items have an arbitrary order: $(w_1, v_1), (w_2, v_2), \ldots, (w_n, v_n)$
- Add items one-by-one, in order. When an item does not fit, skip it.
- Not optimal.

The Knapsack Problem

- The Greedy algorithm is reminiscent of Fibonacci: the solution for a set of i items depends on the solutions of a set of $i-1$ items (a subproblem). What is missing?
- The knapsack capacity also leads to subproblems and a corresponding Greedy algorithm: the solution for a knapsack of capacity j depends on the solution for capacity $j-1$.
    - Spoiler: also not optimal.
- The combination of both "classes" of subproblems is what leads to a correct algorithm.

The Knapsack Problem

- Define $F(i, j)$ as the optimal solution for a subset of items $1..i$ and capacity j.
- Case 1: if item i is not in the optimal solution, then $F(i, j) = F(i - 1, j)$
    - This applies either if item i is "not valuable enough" or because it does not fit in the in the knapsack ($j < w_i$).
- Case 2: if item i is in the optimal solution, then we need to take into account its weight.
    - The subproblem is the optimal solution for a knapsack of capacity $j - w_i$.
    - $F(i, j) = F(i - 1, j - w_i)$

The Knapsack Problem

Express the solution recursively:

$$F(i,j) = 0 \text{ if } i = 0 \text{ or } j = 0$$

Otherwise:

$$F(i,j) = \begin{cases} max(F(i-1,j), F(i-1,j-w_i)+v_i) & \text{if } j \geq w_i \\ F(i-1,j) & \text{if } j < w_i \end{cases}$$

- In Fibonacci, we had an array to store solutions.
- Here, we need a matrix of $n+1$ rows and $W+1$ columns.

Example

$$F(i,j) = 0 \text{ if } i = 0 \text{ or } j = 0$$

$$F(i,j) = \begin{cases} max(F(i-1,j), F(i-1,j-w_i)+v_i) & \text{if } j \geq w_i \\ F(i-1,j) & \text{if } j < w_i \end{cases}$$

item	weight	value
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

i/j	0	1	2	3	4	5
0						
1						
2						
3						
4						

The Knapsack Problem

function $\textsc{Knapsack}(v[1..n], w[1..n], W)$

  for $i \leftarrow 0$ to n **do** $F[i, 0] \leftarrow 0$

  for $j \leftarrow 1$ to W **do** $F[0, j] \leftarrow 0$

  for $i \leftarrow 1$ to n **do**

    for $j \leftarrow 1$ to W **do**

      if $j < w_i$ **then**

        $F[i, j] \leftarrow F[i - 1, j]$

      else

        $F[i, j] \leftarrow max(F[i - 1, j], F[i - 1, j - w_i] + v_i)$

  return $\textsc{Backtrace}(F, n, W)$

The algorithm has time (and space) complexity $\Theta(nW)$.

Solving Knapsack with Memory Functions

- To some extent the bottom-up (table-filling) solution is overkill: It finds the solution to every conceivable sub-instance.
- Most entries cannot actually contribute to a solution.
- In this situation, a top-down approach using a memory function is preferable (this technique is also known as memoing).
- The memory function can be stored as a dictionary.

Solving Knapsack with Memory Functions

```
function KNAP(i,j)
    if i = 0 or j = 0 then
        return 0
    if (i, j) is in MEMO then
        return MEMO[(i, j)]
    if j < w_i then
        k ← KNAP(i − 1, j)
    else
        k ← max(KNAP(i − 1, j), KNAP(i − 1, j − w_i) + v_i)
    MEMO[(i, j)] ← k
    return k
function KNAPSACK(v[1..n], w[1..n], W)
    allocate(MEMO)                          ▷ A global dictionary
    KNAP(n, W)                              ▷ v, w are global as well
    return BACKTRACE(MEMO, n, W)
```

Summary

- Dynamic Programming recipe:
    - Split into subproblems.
    - Solutions overlap.
- A DP solution for Knapsack results in a pseudopolynomial time algorithm.
    - Uses two “variables” to split the problem.
    - Can use memory functions to speed it up.

Next lecture: Dynamic Programming applied to graph algorithms.

COMP20007 Design of Algorithms

Dynamic Programming Part 2: Warshall and Floyd algorithms

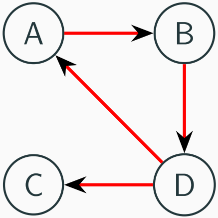
Daniel Beck

Lecture 13

Semester 1, 2023

Transitive Closure

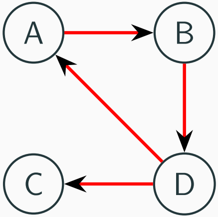
Goal: find all node pairs that have a path between them.



Transitive Closure using DP

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

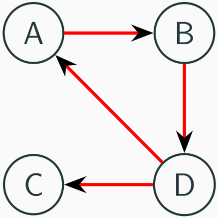
- The solution to a problem can be broken into solutions to subproblems.
  - If there's a path between two nodes i and j which are not directly connected, that path has to go through at least another node k. Therefore, we only need to find if the pairs (i,k) and (k,j) have paths.



Transitive Closure using DP

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

- 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.



Warshall's Algorithm

- In Knapsack, we assume the items have an arbitrary order.
- In Warshall, we assume the nodes have an arbitrary order (from 1 to n).
- For every pair of nodes (i, j), the full problem is: "is there a path between i and j?"
- This can be interpreted as "is there a path between i and j that goes through any subset of nodes from 1 to n?"
- Leading to the subproblem: "is there a path between i and j that goes through any subset of nodes from 1 to k?"

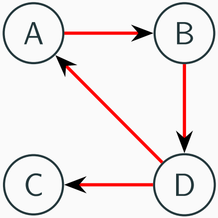
Warshall's Algorithm

- Assume A as the adjacency matrix. Assume R^k as matrix that sets 1 when two nodes are connected through a subset of $1..k$ nodes, and 0 otherwise.
- When $k = 0$, we have the empty subset and $R^0 = A$.
- The goal is to get R^n. We can then get the following recurrence:

$$
\begin{array}{rcl}
R^0_{ij} & = & A_{ij} \\
R^k_{ij} & = & R^{k-1}_{ij} \text{ or } (R^{k-1}_{ik} \text{ and } R^{k-1}_{kj})
\end{array}
$$

Warshall's Algorithm

$$\begin{aligned} R_{ij}^0 &= A_{ij} \\ R_{ij}^k &= R_{ij}^{k-1} \text{ or } (R_{ik}^{k-1} \text{ and } R_{kj}^{k-1}) \end{aligned}$$



Warshall's Algorithm

function $\textsc{Warshall}(A[1..n, 1..n])$
    $R^0 \leftarrow A$
    for $k \leftarrow 1$ to n **do**
        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 the input A to be used for the output, saving memory and simplifying the algorithm.
- 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.

function $\textsc{Warshall2}(A[1..n, 1..n])$
    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 $A[k,j]$ **then**
                        $A[i,j] \leftarrow 1$
    return A

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.

function $\text{Warshall3}(A[1..n, 1..n])$
    for $k \leftarrow 1$ to n **do**
        for $i \leftarrow 1$ to n **do**
            if $A[i,k]$ **then**
                for $j \leftarrow 1$ to n **do**
                    if $A[k,j]$ **then**
                        $A[i,j] \leftarrow 1$
    return A

- Can use bitstring operations.

Analysis of Warshall's Algorithm

- Straightforward analysis: $\Theta(n^3)$ in all cases.
- DFS/BFS from each node is also $\Theta(n^3)$ (if using adjacency matrices)...
- In practice, parallelisation makes Warshall more efficient.

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, and 0 for diagonal cells).

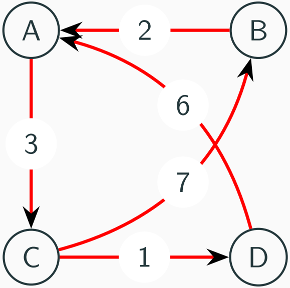
Floyd's Algorithm

- The recurrence follows Warshall's closely:

$$
\begin{aligned}
D^0_{ij} &= W_{ij} \\
D^k_{ij} &= min(D^{k-1}_{ij}, D^{k-1}_{ik} + D^{k-1}_{kj})
\end{aligned}
$$

function $\text{F}\textsc{loyd}(W[1..n, 1..n])$
    $D \leftarrow W$
    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

Floyd's Algorithm



W	A	B	C	D
A	0	∞	3	∞
B	2	0	∞	∞
C	∞	7	0	1
D	6	∞	∞	0

D	A	B	C	D
A	0	10	3	4
B	2	0	5	6
C	7	7	0	1
D	6	16	9	0

Floyd's Algorithm - Why does it work?

- In Warshall, for every pair, each iteration may connect the nodes by including a new intermediate node (if the pair is not already connected).
- In Floyd, the same applies. But if the nodes are already connected, we might need to update the total distance. Two possibilities:
  - The shortest path does not have node k: this is just the same distance as the previous iteration
  - The shortest path has node k: the new distance is the distance of the shortest path between i and k plus the distance of the shortest path between k and j. Both D_{ik}^{k-1} and D_{kj}^{k-1} are already computed in the previous iteration.

Floyd's Algorithm

Obtaining the paths

- The algorithm can be adapted to obtain the full paths (store and update predecessor nodes in a second matrix).

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).

Summary

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

Remember: no lecture next Tuesday (ANZAC day)

Next lecture: Sorting