Part 6: Dynamic Programming

Contents

[document finalized]

Dynamic Programming

Historic term:

Programming refers to "planning" or "optimization".

Other examples: "Linear Programming", "Quadratic Programming", which deal with optimization subject to linear or quadratic constraints.

Dynamic programming is a technique for solving problems with overlapping subproblems.

Rather than solving overlapping subproblems again and again, the idea is to store obtained solutions and reuse them later as needed.

Therefore, dynamic programming trades space for speed.

Sometimes, the gain is considerable.

Often, dynamic programming is used for solving optimization problems quickly. We will discuss some of these problems after introducing the concept using two simple non-optimization problems.

Fibonacci Numbers

Consider computing Fibonacci sequence elements:

$$F(0) = 0, F(1) = 1$$
$$F(n) = F(n-1) + F(n-2),\ n \geq 2$$

n	0	1	2	3	4	5	6	7	8	9	10
$F(n)$	0	1	1	2	3	5	8	13	21	34	55

Question: how do we compute $F(n)$ quickly?

Direct implementation based on recursive definiton:

```
function fib1(n)
if n < 2 then
  return n
else
  return fib1(n-1) + fib1(n-2)
end
```

Repeated work leads to exponential runtime!

To see this let $T(n)$ denote the number of additions executed in the computation of `fib1(n)`. Then for $n \geq 4$:

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + 1 \\ &= [T(n-2) + T(n-3) + 1] + T(n-2) + 1 \\ &\geq 2T(n-2) + 2 \quad [T(n-3) \geq 0] \\ &\geq 2T(n-2) \\ &\geq 2^i T(n-2i) \end{aligned}$$

and $T(0) = T(1) = 0$, $T(2) = 1$, and $T(3) = 2$.

In the second expansion step we see that $F(n-2)$ is computed twice.

For even n we reach base case 2 for $i = (n-2)/2$.

$$T(n) \geq 2^{n/2-1} T(2) = (\sqrt{2})^n/2 \doteq 1.414^n/2$$

For odd n a similar exponential lower bound holds.

This means that our function has at least exponential runtime!

To speed things up we need to avoid repeated computations.

Iterative (or non-recursive) implementation using an array to store previously computed Fibonacci numbers:

```
function fib2(n)
A[0] <- 0
A[1] <- 1
for i <- 2 to n do
  A[i] <- A[i-1] + A[i-2]
end
return A[n]
```

At the time we want to compute $F(i)$, the previous values $F(i-1)$ and $F(i-2)$ have been computed already and stored in the array. Therefore, without any work repetition, `fib2` computes $F(n)$ in time $\Theta(n)$, at the cost of $\Theta(n)$ space.

The space requirement can be reduced to $\Theta(1)$ by noting that at any given time we only need to memorize the two immediate predecessors.


```
function fib3(n)
x <- 0  // previous-previous value
y <- 1  // previous value
for i <- 2 to n do
  nexty <- x + y  // next value
  x <- y          // prev.-prev = prev
  y <- nexty      // previous = next
end
return y
```


Computing Binomial Coefficients

From elementary combinatorics you may recall that the **binomial coefficient** $\binom{n}{k}$ is the number of k-element subsets of an n-element set ($0 \le k \le n$).

The name binomial coefficients comes from the binomial formula which holds for all $a, b \in \mathbb{R}$ and $n \in \mathbb{N}$:

$$(a+b)^n = \sum_{i=0}^{n} \binom{n}{i} a^{n-i} b^i$$

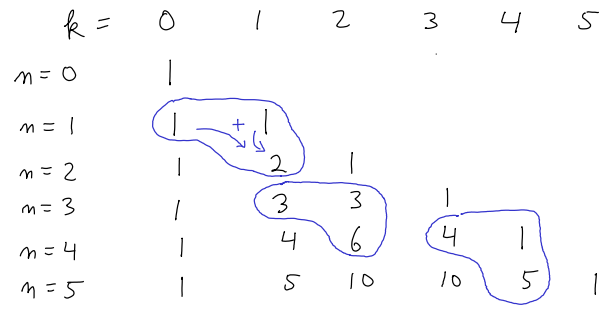
Among the many properties of $\binom{n}{k}$ are the following:

1. $\binom{n}{0} = \binom{n}{n} = 1$
2. $\binom{n}{k} = \binom{n}{n-k}$
3. $\binom{n}{k} = \frac{n!}{k!(n-k)!}$
4. $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \quad (0 < k < n)$

Property 4 suggest a recursive way of computing $\binom{n}{k}$. Rather than using the straightforward recursive function that would solve many subproblems repeatedly, we apply the dynamic programming idea of saving intermediate results and organizing the computation in such a

way that we can reuse prior results as much as possible.

We fill a table of $\binom{n}{k}$ values row by row like so:



$\binom{n}{0} = \binom{n}{n} = 1$, $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$, $k, n \ge 1$

1. For each new row n we set the $(n,0)$ and (n,n) entries to 1.

2. For computing the (n,k) entries we make use of property 4 by adding entries $(n-1,k-1)$ and $(n-1,k)$, i.e. going up one step and going left from there.

So, $\binom{n}{k}$ can be computed just based on additions ("Pas-

cal's triangle").

```
function choose(n,k)
for i <- 0 to n do
  for j <- 0 to min(i,k) do
    if j = 0 or j = i then
      C[i,j] <- 1
    else
      C[i,j] <- C[i-1,j-1] + C[i-1, j]
    end
  end
end
return C[n,k]
```

The runtime of this function is $O(nk)$ (two nested loops: one iterates $\Theta(n)$ times, the other $O(k)$ times.

The space requirement is also $O(nk)$.

The space requirement can be reduced considerably.

Dynamic Programming and Optimization

Dynamic programming can also be used for combinatorial optimization.

Two conditions have to be met:

1. Optimal substructure. Check whether the problem exhibits the optimal substructure property, i.e. an optimal solution to the problem contains within it optimal solutions to subproblems. Examples: subpaths of optimal paths are optimal, subtrees of optimal search trees are optimal.
2. Overlapping subproblems. When a recursive algorithm revisits subproblems repeatedly, we say that the optimization problem has overlapping subproblems. In this case we can apply the dynammic programming idea to save time by storing solutions to subproblems and looking them up when we need them later.

At an additional cost of memory we can speed up the reconstruction of an optimal solution by storing which choice we made in each subproblem in a table.

The Knapsack Problem

Recall Knapsack Problem:

Given n items of known weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ and a knapsack of capacity W, find the most valuable subset of items that fit into the knapsack.

In the Brute Force section we saw an exponential time algorithm that enumerated all item subsets.

Here we assume that all weights and the knapsack capacity are integers.

To design a dynamic programming algorithm we need to derive a recurrence relation that expresses a solution to an instance in terms of solutions to smaller subinstances.

Define an instance by the first i items ($i \leq n$)

$$(w_1, v_1), \ldots, (w_i, v_i)$$

and knapsack capacity j ($1 \leq j \leq W$) and $V[i, j]$ as the value of an optimal solution to this instance.

Lecture 23

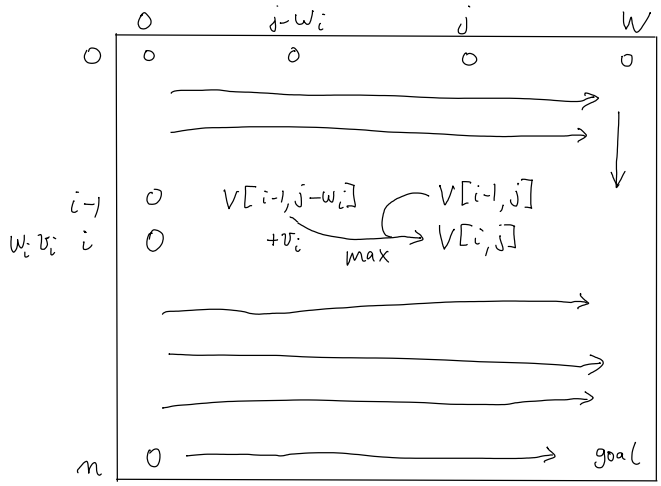
Observations:

- For optimal subsets that don't include item i the optimal value is
$$V[i-1, j]$$
- For optimal subsets that include item i the optimal value is
$$v_i + V[i-1, j-w_i]$$

Therefore, the value of the optimal solution is the maximum of these values, i.e.

$$V[i,j] = \begin{cases} \max\left\{V[i-1,j],\ v_i + V[i-1, j-w_i]\right\}, & w_i \leq j \\ V[i-1,j], & w_i > j \end{cases}$$

with $V[0, j] = 0$ for $j \geq 0$ and $V[i, 0] = 0$ for $i \geq 0$.



Strategy for filling the matrix?

Similar to the binomial coefficient computation: top-down, left to right.

Example: $W = 5$

Weight	Value	$i\backslash j$	0	1	2	3	4	5
		0	0	0	0	0	0	0
$w_1 = 2$	$v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1$	$v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3$	$v_2 = 20$	3	0	10	12	22	30	32
$w_4 = 2$	$v_2 = 15$	4	0	10	15	25	30	37

$$V[i,j] = \max\Big\{V[i-1,j],\ v_i + V[i-1,j-w_i]\Big\}$$

Reconstructing an optimal solution:

$V[4,5] \neq V[3,5] \Rightarrow$ item 4 selected, $5 - w_4 = 3$ left

$V[3,3] = V[2,3] \Rightarrow$ item 3 not selected

$V[2,3] \neq V[1,3] \Rightarrow$ item 2 selected, $3 - w_2 = 2$ left

$V[1,2] \neq V[0,2] \Rightarrow$ item 1 selected, $1 - w_1 = 0$ left

Thus, an optimal solution consists of items 1, 2, and 4.

The time and space requirements of this algorithm are both in $\Theta(nW)$ and the time needed to find the composition of an optimal solution is in $O(n)$ (exercise).

Memory Functions

The direct top-down approach to finding a solution leads to an algorithm that solves common subproblems more than once and hence can be very inefficient. The classic dynamic programming approach, on the other hand, works bottom-up: it fills a table with solutions to all smaller subproblems.

Sometimes, not all subproblems have to be solved to obtain optimal solutions. The top-down approach does not suffer from this inefficiency. Therefore, it is natural to try to combine the strengths of both methods.

So called **memory functions** accomplish this goal by proceeding top-down but storing and using previously computed function values.

Example: Memory function version of Knapsack

```
function KS(w[1..n], v[1..n], W)
clear M
return MemKS(n, W, M, w, v)


function MemKS(i, j, M, w[1..n], v[1..n])
if i <= 0 or j <= 0 then return 0 end
val <- lookup(M, i, j)
if val < 0 then  // new
  if j < w[i] then
    val <- MemKS(i-1, j, M, w, v)
  else
    val <- max(MemKS(i-1, j, M, w, v),
               v[i] + MemKS(i-1, j-w[i], M, w, v))
  end
  store(M, i, j, val)    // memoize value
end
return val
```

To be effective, both locating entry (i, j) and clearing data structure `M` must be fast.

As an example consider using an array for storing `V[i,j]`

The memory function based approach may only visit a small fraction of this array. So, initializing every element with -1 in the beginning could take more time than the actual top-down dynamic programming run.

How can we avoid initializing the array everytime we call `KS` for solving a new Knapsack problem?

Rather than using the value itself to distinguish whether we have visited the entry before (≥ 0?), we can use integers `M[i,j]` and a global generation count `C`, with the convention that `M[i,j] = C` iff entry `(i,j)` has already been computed in the current generation.

So, to "clear M" in the `KS` function we simply incremenent the generation count `C` which invalidates all previously used entries in constant time. Only in the beginning and when the generation count wraps around to 0, the entire array needs to be filled with 0 and `C` set to 1.

This only happens every $(2^K - 1)$-th time, if we store K-bit integers $\rightsquigarrow$ exponential time savings!

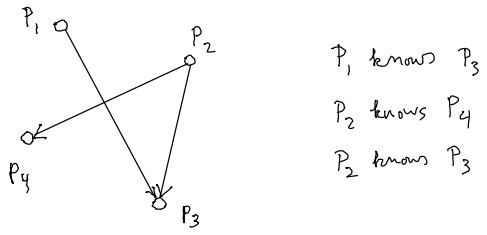
Graph Introduction Lecture 24

Graphs model binary relations $R \subseteq V \times V$

- Objects are represented by vertices
- Related objects are connected by edges

Example: Relation R with

$$(x, y) \in R \Leftrightarrow \text{Person } x \text{ knows person } y$$



Relations are not necessarily symmetric:

$$(x, y) \in R \not\Rightarrow (y, x) \in R$$

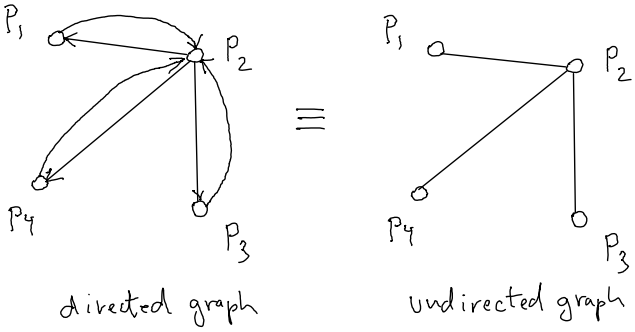
Therefore we need directed edges ("arcs")

For symmetric relations undirected edges suffice

Example:

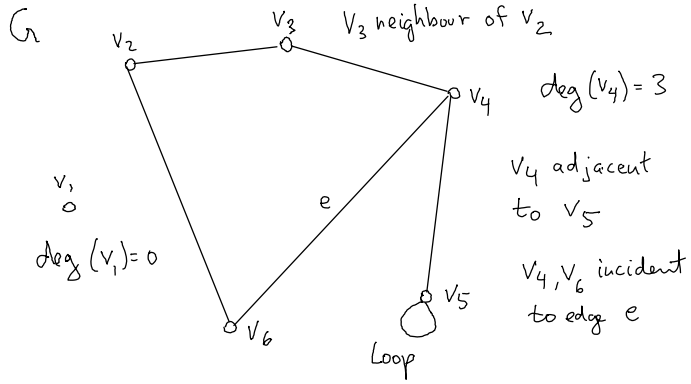
$$(x, y) \in F \Leftrightarrow x \text{ is a friend of } y$$

Then (usually) $(x, y) \in F \Leftrightarrow (y, x) \in F$



Definition:

An **(undirected) graph** is a pair $G = (V, E)$ that is composed of a non-empty vertex (or node) set V and an edge set E, such that each edge contains one or two vertices, i.e. for all $e \in E$, $e \subseteq V$ and $1 \leq |e| \leq 2$.



$G = (V, E) \quad V = \{v_1 \dots v_6\}$

$E = \{ \{v_2, v_3\}, \{v_3, v_4\}, \{v_4, v_5\}, \{v_5\}, \{v_4, v_6\}, \{v_2, v_6\} \}$

The order of G is 6 (number of nodes), its size is also 6 (number of edges)

The **order** of $G = (V, E)$ is $|V|$.

The **size** of $G = (V, E)$ is $|E|$.

We say that edge $e = \{u, v\}$ **connects** node u with node v. In case $e = \{u\}$, vertex u is connected to itself and forms a **loop**.

A graph is called **simple** if it does not contain loops.

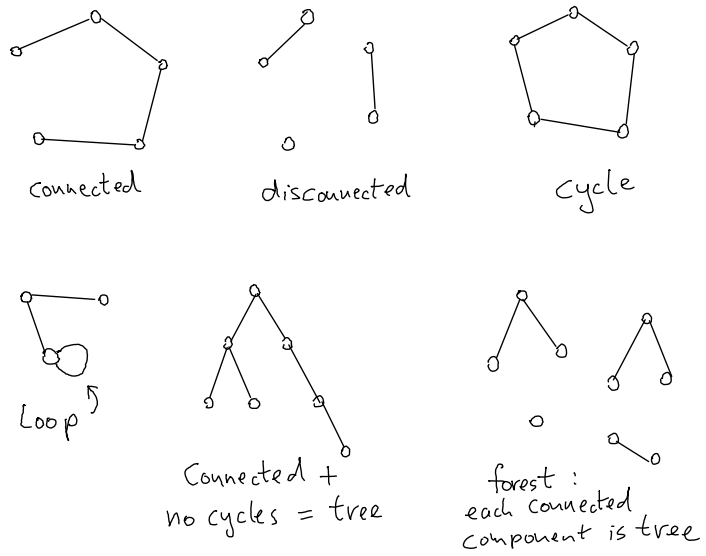
A vertex u is called a **neighbour** of vertex v iff $\{u, v\} \in E$.

The number of neighbours of a vertex u is called its **degree**, denoted $\deg(u)$. A loop contributes twice to the degree.

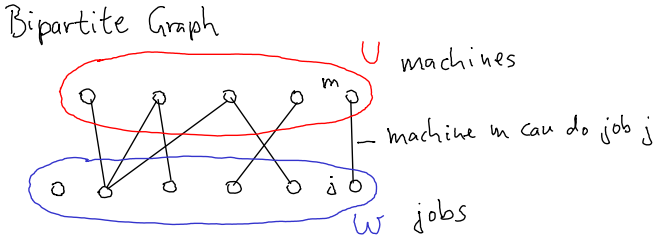
$u, v \in V$ are **adjacent** iff $\{u, v\} \in E$

$u, v \in V$ are **incident** to edge $e \in E$ iff $e = \{u, v\}$

Fundamental Graph Classes and Properties



Formal definitions coming up.



$V = U \cup W$ and $U \cap W = \emptyset$ (node partition)

All edges of form $e = \{u, w\}$ with $u \in U$ and $w \in W$

Definition:

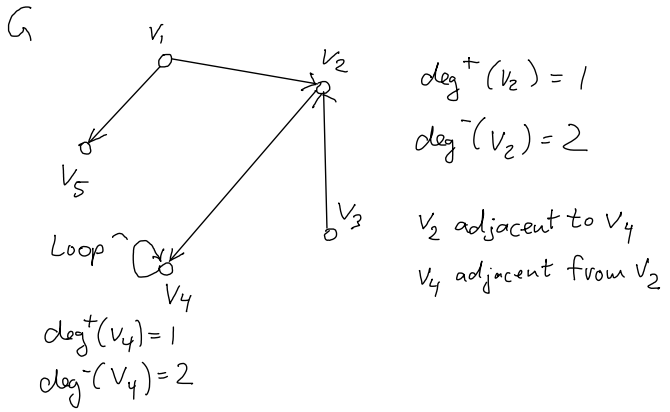
A **directed graph** is a pair $G = (V, E)$ that is composed of a non-empty vertex set V and an edge set $E \subseteq V \times V$.

Edge $e = (u, v)$ connects vertex u with vertex v (u is said to be adjacent to v, and v is adjacent from u). u is called the **initial vertex** of e, and v is called the **terminal vertex** of e.

The **in-degree** of a vertex v, denoted $\deg^-(v)$, is the number of edges with v as their terminal vertex.

The **out-degree** of a vertex v, denoted $\deg^+(v)$, is the number of edges with v as their initial vertex.

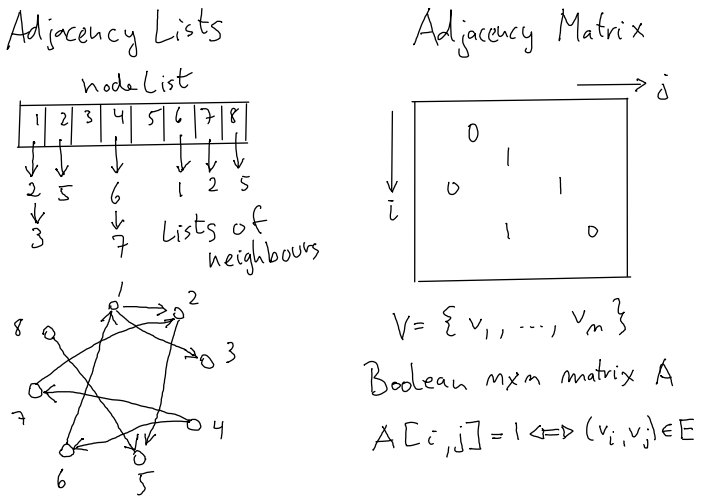
Loops contribute 1 to both the in- and out-degree.



$G = (V, E),\ V = \{v_1 \dots v_6\}$

$E = \{(v_1, v_2), (v_1, v_5), (v_2, v_4), (v_3, v_2), (v_4, v_4)\}$

Graph Data Structures



Adjacency lists are smaller if G is **sparse** ($|E| \in O(|V|)$)
If G is **dense** ($|E|$ quadratic in $|V|$), adjacency matrices save space and allow direct access to edge information.

Which representation is better depends on algorithm used

Connectivity

Many real-world problems can be modeled with paths formed by traveling along the edges of graphs.

Problems of efficiently planning routes for mail delivery, garbage pickup, diagnostics in computer networks, etc. can be solved using models that involve paths in graphs.

Informally, a path is a sequence of edges that begins at a vertex of a graph and travels from vertex to vertex along edges of the graph.

Definition

Let $n \in \mathbb{N}$ and $G = (V, E)$ a graph. A **path** of length n from u to v in G is a sequence of n edges

$$(e_1, e_2, \dots, e_n)$$

– written as $(e_i)_{i=1}^n$ – such that $e_i = \{x_{i-1}, x_i\} \in E$ for all i, with $u = x_0$ and $x_n = v$.

In case of directed graphs, $e_i = (x_{i-1}, x_i) \in E$.

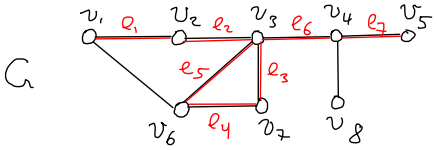
Sometimes it is convenient to define a path just by the sequence of visited vertices, which implies the edges:

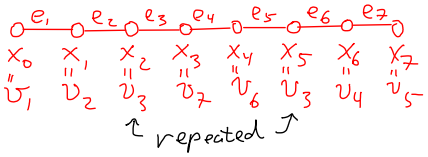
$$(x_0, x_1, \dots x_n)$$

The path is a **circuit** (or cycle) iff $x_0 = x_n$ and $n > 0$.

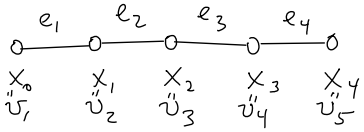
A path or circuit is **simple** iff it does not contain the same node more than once (except for the necessary repetition of the start and end vertex).

Examples



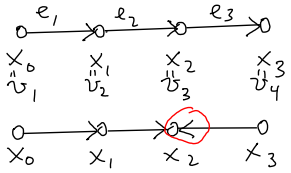


Path from v_1 to v_5



Simple path from v_1 to v_5

Directed path examples:



Simple directed path

Not a directed path!

Lecture 25

A graph is called **connected** iff there is a path between every pair of distinct vertices of the graph.

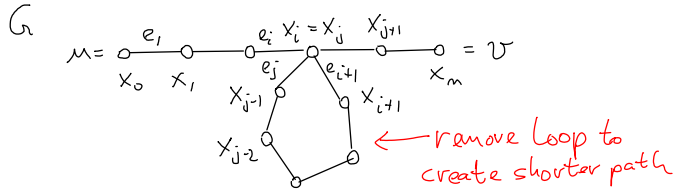
Theorem

There is a simple path between every pair of distinct vertices of a connected graph.

Proof

Let $G = (V, E)$ be a connected graph and $u, v \in V$. Then there exists a path $P_0 = (e_i)_{i=1}^n$ that connects u to v, with $e_i = \{x_{i-1}, x_i\} \in E$ for all i, with $u = x_0$ and $x_n = v$.

If no vertex is repeated, then P_0 is a simple path and we are done. Otherwise, let i, j be distinct integers with $i < j$ and $x_i = x_j$.



If we delete the edges $e_{i+1}, \ldots, e_j$ from P_0 we obtain a shorter path P_1 from u to v and has fewer repeated nodes. If P_1 is a path we are done. Otherwise, we repeat the process.

Since P_0 is a finite sequence, eventually we must reach stage k where no vertices are repeated and the resulting path P_k is simple. □

Given a graph $G = (V, E)$, being reachable defines a relation C on V:

$(u, v) \in C \Leftrightarrow u = v$ or there is a path from u to v in G

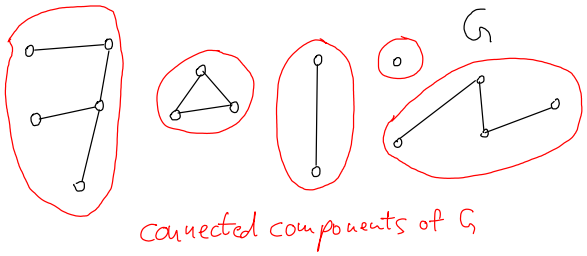
Then, C is

- reflexive: $\forall v \in V : (v, v) \in C$
- symmetric: $\forall u, v \in V : (u, v) \in C \Rightarrow (v, u) \in C$
- and transitive: $\forall x, y, z \in V :$
  $[(x, y) \in C \wedge (y, z) \in C] \Rightarrow (x, z) \in C$

(Why?)

This means that C is an equivalence relation, and as such the set of all element equivalence classes (= sets of nodes that are related to each other) forms a partition of V.

Each such equivalence class together with the set of edges that are incident to those vertices are called **connected component**.



Observations:

Connected components of graph G are connected subgraphs of G that contain vertices and all incident edges of G, such that no other vertex can be added to create larger connected subgraphs.

[Graph $H = (W, F)$ is a **subgraph** of $G = (V, E)$ iff $W \subseteq V$ and $F \subseteq E$]

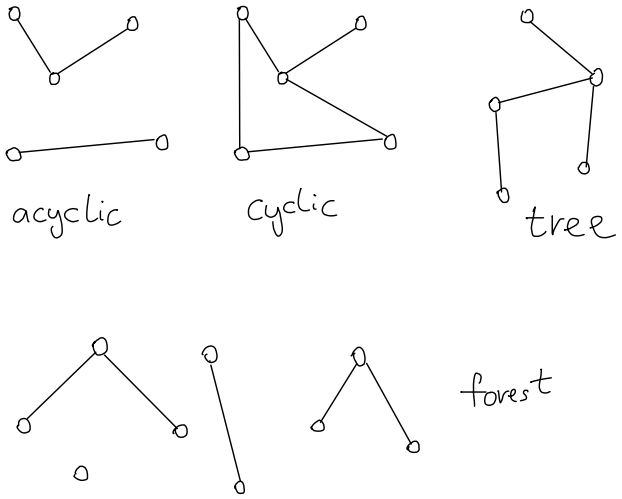
A graph is connected iff it has one connected component.

Definition

A graph is called **acyclic** iff it does not contain a cycle.

A connected acyclic graph is called a **tree**.

An acyclic graph is also called **forest**.



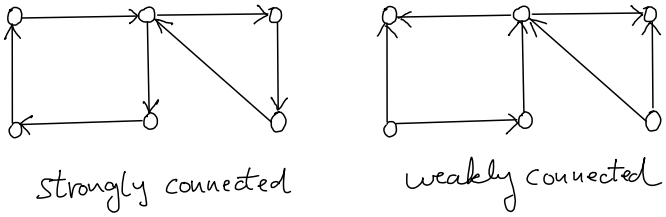
Connectivity in Directed Graphs

There are two notions of connectedness in directed graphs, depending on whether the directions of the edges are considered.

Definition

A directed graph (V, E) is **strongly connected** iff for all distinct vertices $u, v \in V$ there is a path from u to v and from v to u.

A directed graph is **weakly connected** iff the underlying undirected graph, that is constructed by turning every directed edge into an undirected edge, is connected.

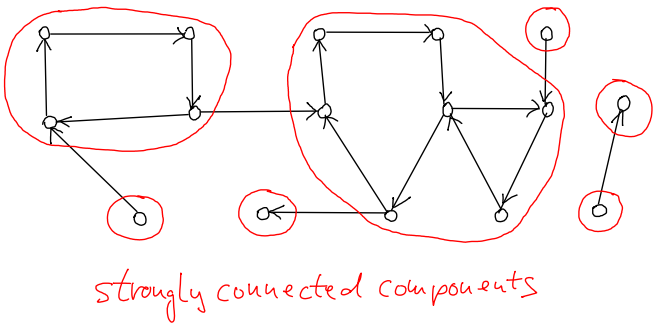


Similar to the reachability relation on graphs, the reachability relation C on digraphs given by

$$(u, v) \in C \Leftrightarrow$$

$u = v$ or there is a path from u to v and from v to u

is an equivalence relation and therefore induces a vertex partition. The subgraphs induced by the equivalence classes are called **strongly connected components** (SCCs).

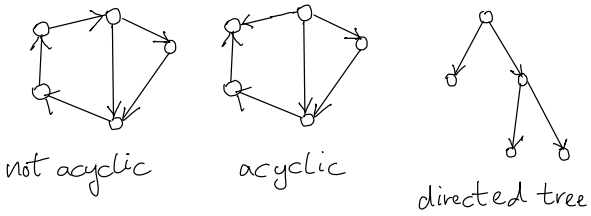


In each SCC, every node can be reached from any other by a directed path, and like connected components in graphs, they can't be enlarged by adding a node.

Definition

A directed graph is **acyclic** iff it does not contain a directed cycle.

A directed graph is a **directed tree** iff it would be a tree if all directed edges are turned into undirected edges and in-degree $\deg^-(v) \leq 1$ for all vertices.



The All-Pairs Shortest Path Problem

In this section we shall see a dynamic programming formulation to solve the all-pairs shortest-path problem on a directed graph $G = (V, E)$, which seeks to find shortest paths between any two nodes in V.

Finding shortest paths has many applications.

For example:

- Minimize travel time between cities
- Find shortest paths for mobile objects in video games
- More generally, minimize the cost of doing an activity
  - Vertices are states of the world
  - Edge weights are costs of moving between states

For studying distance related graph problems we augment graphs by a weight function $w : E \to \mathbb{R}$, that assigns weights (or distances) to edges. Formally, we define a **weighted** digraph as triple $G = (V, E, w)$.

For convenience we assume that the vertices are numbered $1, 2, \ldots, |V| = n$, so that the input is an $n \times n$ maxtrix W representing the edge weights. That is, $W = (w_{ij})$, where

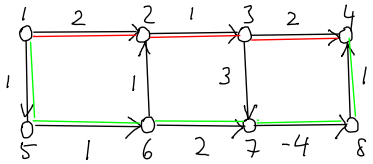
$$w_{ij} = \begin{cases} 0 & \text{if } i = j \\ \text{weight of edge } (i,j) & \text{if } i \neq j \text{ and } (i,j) \in E \\ \infty & \text{if } i \neq j \text{ and } (i,j) \notin E \end{cases}$$

The weight $w(e)$ of an edge $e = (i, j)$ equals w_{ij} and the weight $w(P)$ of a path $P = (e_k)_{k=1}^n$ is defined as

$$w(P) := \sum_{k=1}^{n} w(e_k)$$

We allow negative-weight edges, but we assume that the input graph contains no negative-weight cycles, i.e. cycles whose edge weight sum is < 0 (why?).

The goal of the all-pairs shortest-path computation is to determine for each pair of vertices (i, j) the weight $\boldsymbol{\delta_{ij}}$ of a minimum-weight (i.e. shortest) path from i to j.



Consider two paths from vertex 1 to vertex 4

w_{ij}

0	2	–	–	1	–	–	–
–	0	1	–	–	–	–	–
–	–	0	2	–	–	3	–
–	–	–	0	–	–	–	–
–	–	–	–	0	1	–	–
–	1	–	–	–	0	2	–
–	–	–	–	–	–	0	-4
1	–	–	–	–	–	–	0

–: ∞

P_1 path induced by $(1, 2, 3, 4)$

P_2 path induced by $(1, 5, 6, 7, 8, 4)$

$w(P_1) = 2 + 1 + 2 = 5$

$w(P_2) = 1 + 1 + 2 - 4 + 1 = 1$

$\rightsquigarrow$ P_2 is shorter than P_1

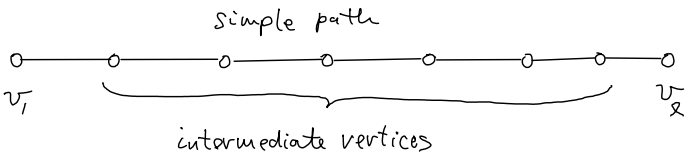
In fact, P_2 is the shortest path: $\delta(1,4) = 1$

Dynamic Programming Approach Lecture 26

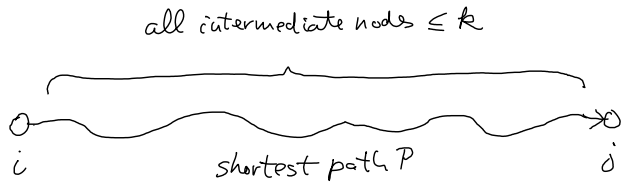
Without loss of generality (w.l.o.g.) shortest paths are simple.

If vertices are repeated we can simply iteratate removing cycle edges to obtain a path that is not longer than the orgiginal one, like in the undirected case we have seen.

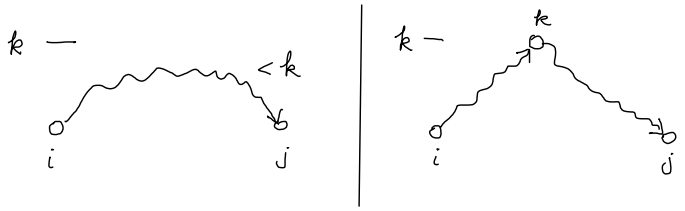
We define an **intermediate vertex** of a simple path P that is induced by vertex sequence $v_1, \ldots, v_l$ as any vertex along P other than v_1 and v_l.



To create a sequence of subproblems of increasing complexity we consider vertex subset $\{1 \ldots k\}$ and restrict the intermediate vertices of paths from i to j to that set.



Two cases:



Consider a shortest path P from i to j subject to this restriction and call its weight $\delta_{ij}^{(k)}$. Then

- If k is not an intermediate vertex of P, then all intermediate vertices of P are in the set $\{1 \ldots k-1\}$ and $\delta_{ij}^{(k)} = \delta_{ij}^{(k-1)}$.
- If k is an intermediate vertex of P, then we decompose P into two paths: P_1 from i to k and P_2 from k to j.

Then P_1 and P_2 are shortest paths with intermediate nodes in $\{1 \ldots k-1\}$, because repeating k is not beneficial and if shorter paths P_1' or P_2' existed, they could be used to shorten P.

- If $k = 0$ then no intermediate vertex is allowed, and therefore $\delta_{ij}^{(0)} = w_{ij}$

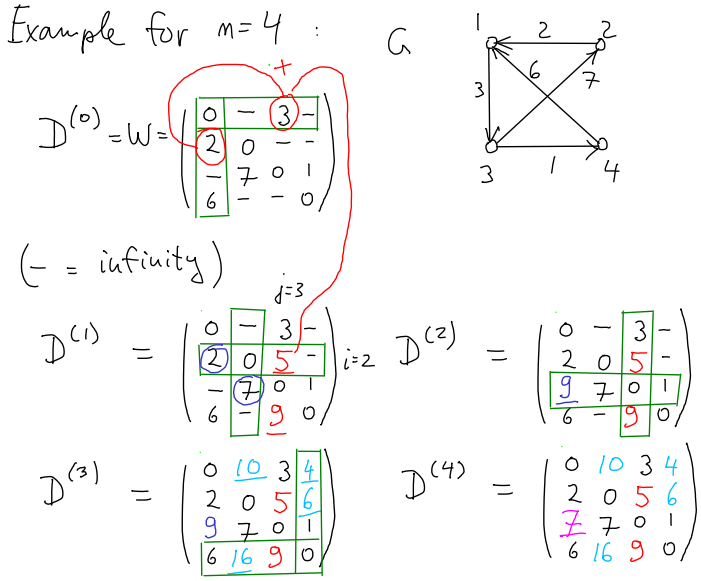
From this we obtain the following recurrence relation:

$$\delta_{ij}^{(k)} = \begin{cases} w_{ij} & k = 0 \\ \min(\delta_{ij}^{(k-1)}, \delta_{ik}^{(k-1)} + \delta_{kj}^{(k-1)}) & k > 0 \end{cases}$$

Because for any path all intermediate vertices are in $\{1, \ldots, n\}$ the final answer is: $\delta_{ij} = \delta_{ij}^{(n)}$ for all $i, j \in V$.

The Floyd–Warshall algorithm (1962) uses this observation to iteratively compute matrices $D^{(k)} := (\delta_{ij}^{(k)})$ for $k = 0, \ldots n$, thereby computing all shortest distances between all node pairs (i, j).

Example for $n = 4$:



```
function FloydWarshall(W[1..n,1..n])
D <- W  // matrix containing deltas(k)
for k <- 1 to n do
  // previous matrix is D, current one is E
  for i <- 1 to n do
    for j <- 1 to n do
      E[i,j] <- min(D[i,j], D[i,k]+D[k,j])
    end
  end
  D <- E // new matrix complete, copy over
end
return D
```

Runtime: $\Theta(n^3)$ Space: $\Theta(n^2)$

There is actually no need for matrix E as D can be updated in-place.

How to reconstruct optimal paths?

Can this algorithm be used to detect negative cycles?

(Exercises)

Transitive Closure of Directed Graphs

Given a directed graph $G = (V, E)$ with $V = \{1, 2 \dots n\}$ we might wish to determine whether there is a path in G from i to j for all vertex pairs $i, j \in V$.

For this purpose we define the **transitive closure** of G as graph $G^* = (V, E^*)$ where

$E^* = \{(i, j) \mid i = j$ or there is a path from i to j in $G\}$

How can we compute E^*?

One option is to use the Floyd-Warshall algorithm:

We set $w(e) = 1$ for all $e \in E$ with $e \neq (i, i)$ for all $i \in V$

Then it is easy to verify that

There is a path from i to j in G iff $\delta_{ij} < n$

This solves the transitive closure problem in time $\Theta(n^3)$

By revisiting the derivation of the recurrence relation for the Floyd-Warshall algorithm, we can decrease the storage and runtime of the transitive closure algorithm in practice.

Intuition: Rather than computing path lengths we only need to update Boolean values that indicate reachability.

For $k = 1, 2, \dots, n$ we define $t_{ij}^{(k)}$ to be 1 if there is a path from i to j with all intermediate nodes in the set $\{1, \dots, k\}$, and 0 otherwise.

Our goal: $E^* = \{(i, j) \mid t_{ij}^{(n)} = 1\}$

The base case $k = 0$ is trivial:

$$t_{ij}^{(0)} = \begin{cases} 0 \text{ if } i \neq j \text{ and } (i, j) \notin E \\ 1 \text{ otherwise} \end{cases}$$

How to update values when going from $k - 1$ to k?

We can reach j from i stepping over nodes $\leq k$ iff we can reach j over intermediate nodes $\leq k - 1$ or we can reach k from i and j from k using nodes $\leq k - 1$.

This leads to the following recurrence relation:

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$$

where $\vee, \wedge$ are the Boolean connectives "or" and "and", respectively.

```
function TransitiveClosure(E[1..n,1..n])
Initialize T from E (k=0)
for k <- 1 to n do
  for i <- 1 to n do
    for j <- 1 to n do
      T[i,j] <- T[i,j] v (T[i,k] ^ T[k,j])
    end
  end
end
return T
```

Here we omitted the creation of intermediate matrices $T^{(k)}$ for each value of k. Why does this work? (Exercise)

Although, the runtime of `TransitiveClosure` is still $\Theta(n^3)$, its innermost loop runs faster than Floyd-War-

shall's on modern computers. In addition, we need less space.

The computation speed can be increased further by adjusting the code such that modern vector operations can be used, that apply Boolean operations in bit-parallel fashion.

Depending on the computer architecture this can speed up the innermost loop by factors of 32 or greater!

Lecture 27

Is there a way to compute the transitive closure of a directed graphs faster than $\Theta(n^3)$?

Yes. Based on fast matrix multiplication!

Consider matrix multiplication $\odot$ over algebraic structure $\mathcal{S} = (\mathbb{B}, \vee, \wedge, 0, 1)$, where

- $\mathbb{B} = \{0, 1\}$ is the value set (Boolean false, true),
- $\vee$ is the Boolean OR operation,
- $\wedge$ is the Boolean AND operation,
- 0 is the neutral element for $\vee$, and
- 1 is the neutral element for $\wedge$

$$(A \odot B)_{ij} = \bigvee_{k=1}^{n} (a_{ik} \wedge b_{kj}),$$

for Boolean $n \times n$ matrices A, B.

Note that this is the standard matrix multiplication with $+$ replaced by $\vee$ and $\cdot$ replaced by $\wedge$

Define $A^{\odot k} := \underbrace{A \odot A \cdots \odot A}_{A \text{ } k\text{-times}}$

Theorem: Let $T := (t_{ij}^{(0)})$, i.e.

$T_{ii} = 1$ for all $i \in V$, and for $i \neq j$, $T_{ij} = 1$ iff $(i, j) \in E$.

Then there exists a directed path from i to j of length $\leq l$ iff $(T^{\odot l})_{ij} = 1$.

Proof: Induction on l.

Suppose $l = 1$. From the definition of T we know $T_{ij} = 1$ iff j can be reached from i with 0 or 1 step.

Now assume the claim holds for $l - 1$ with $l \geq 2$.

Consider the last multiplication step:

$$(T^{\odot l})_{ij} = ((T^{\odot l-1}) \odot T)_{ij} = \bigvee_{k=1}^{n} (T^{\odot l-1})_{ik} \wedge T_{kj}$$

Therefore, $(T^{\odot l})_{ij} = 1$ iff there exists a node k such that there is a path from i to k in $\leq l - 1$ steps and j can be reached from k in 0 or 1 step.

This means that $(T^{\odot l})_{ij} = 1$ iff j can be reached from i in $\leq l$ steps. $\square$

The theorem suggests an alternative way of computing the transitive closure:

$$E^* = \{(i, j) \mid (T^{\odot n})_{ij} = 1\}$$

This works, because if j can be reached from i in G, then it can be reached in $\leq n$ steps. For if we need $> n$ steps, a vertex is repeated and we can shorten the path. This argument also shows

$$T^{\odot n+i} = T^{\odot n} \text{ for all } i \geq 0 \qquad (*)$$

What is the runtime?

Using the standard matrix multiplication method and applying it $n - 1$ times will take $\Theta(n \cdot n^3) = \Theta(n^4)$ bit operations, which is worse than the Floyd-Warshall-based method.

However, using equation $(*)$ and the doubling trick we have seen when computing powers we can reduce the runtime:

```
function PowerTC(T[1..n,1..n])
i <- 1
while i < n do
  T <- T o T  // Boolean matrix multipl.
  i <- i*2
end
return T
```

This computes $T^{\odot k}$ for some $k \geq n$ (which equals $T^{\odot n}$) in $\approx \log n$ iterations.

Total number of bit operations: $\Theta((\log n) \cdot n^3)$

Still not good enough.

Can we apply the Strassen-Winograd algorithm?

Not directly, because $\mathcal{S}$ is not a ring (it has no inverse element for $\vee$, which would require $(-a) \vee a = 0$ for all $a \in \{0, 1\}$)

However, there is a way around this problem:

$$(A \odot B)_{ij} = \bigvee_{k=1}^{n} (a_{ik} \wedge b_{kj}),$$

can be computed using regular arithmetic:

$$(A \odot B)_{ij} = \{[\sum_{k=1}^{n} (a_{ik} \cdot b_{kj})] > 0\},$$

where $\{x > 0\}$ is 1 if $x > 0$ and 0 otherwise. $(**)$

So the algorithm becomes

```
function StrassenTC(T[1..n,1..n])
i <- 1
while i < n do
  T <- Strassen(T,T) // regular multipl.
  T <- ScaleBack(T)  // using (**)
  i <- i*2
end
return T
```

How fast is it in terms of bit operations?

Based on the equations on page 19 in part 5 it can be proved that intermediate results in the Strassen-Winograd algorithm when multiplying two $(n \times n)$-0/1-matrices lie in range $-6n \ldots 6n$.

Using standard quadratic time multiplication each ring operation can be performed with $O((\log n)^2)$ bit operations.

Therefore, each iteration uses $O((\log n)^2 n^{\log 7})$ bit operations, and the total runtime is

$$O((\log n)^3 n^{\log 7}) \subseteq O(n^{2.81})$$

— which is better than $\Theta(n^3)$.

In the Strassen-Winograd algorithm one can also do all operations modulo $(n+1)$, i.e. taking the remainder when dividing by $n+1$ after each addition, subtraction, and multiplication.