Foundations of Computing 2

Introduction

- Comp. Sci. is the science of automatic processing of information.
- Comp. Sci. has its own fundamental questions:
  - What is an algorithm?
  - What are the fundamental capabilities / limitations of computers?
  - When should an algorithm be considered practically feasible?
  - What makes some problems computationally hard?

Turing Machines

- Basic features of an algorithm: (informal)
  - Clear, well-defined inputs and outputs
  - Organised structure
  - Reduces new problems to solved problems
    - eg. we can already solve division, squares, etc.
- These features are not always helpful though!
  - Example: Diophantic equations are well-defined, but do not readily reduce to solved problems / operations.
- Some problems (eg. Diophantic equations) are "solved" by proving that there is no algorithmic solution.
  - This "non-existence" statement cannot be made on the above, informal definition of an algorithm, so a formal definition is required.
- Formal definition of algorithms: Turing machine.
- A Turing machine has two parts:
  - A tape
  - A finite control unit

TM: Tape

- A sequence of cells, infinite in both directions.
- Each cell contains a symbol from a finite alphabet.
- There is a tape head that can:
  - Read the current symbol
  - Overwrite the current symbol
  - Move one cell left (L), one cell right (R), or stay at the current position (S).

TM: Finite State Control Unit

- Depending entirely on the current state and the current symbol in that state:
  1. Write a symbol into the current state
  2. Move according to L, R or S
  3. Enter and read the next state.

Decidability and Undecidability

- If we start a TM M on an input w, there are three possible outcomes:
  - M halts in an accepting state.
  - M halts in a rejecting state.
  - M loops and runs forever.
- A language is Turing-decidable if some TM decides it.
  - That is, some TM halts on all inputs.

The Acceptance Problem

- Testing whether a particular DFA accepts a string.

$$A_{DFA} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts } w\}$$

- Theorem: A_{DFA} is a decidable language.
- Proof: Construct a TM that decides A_{DFA}:

  M = "On input $\langle B, w\rangle$, simulate B on input w; if simulation ends in an accepting state, accept, otherwise reject."

REMEMBER: DFA = Deterministic Finite Automaton

Testing Equality

- Do two DFAs recognise the same language?

$$EQ_{DFA} = \{\langle A, B\rangle \mid A \text{ and } B \text{ are DFAs}, L(A) = L(B)\}$$

- **Theorem**: EQ_{DFA} is a decidable language

- **Proof**:
  - We use the previous acceptance proof.
  - Construct a new DFA C from A and B, where C only accepts strings accepted by A XOR B.
  - If $L(A) = L(B)$ then $L(C) = \emptyset$.

$$L(C) = \left(L(A) \cap \overline{L(B)}\right) \cup \left(\overline{L(A)} \cap L(B)\right)$$

The Halting Problem

- Some problems are algorithmically unsolvable
  - They are undecidable.

- Test whether a given TM accepts a string w.

$$A_{TM} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$$

- **Theorem**: A_{TM} is undecidable *

- **Proof**:
  - The following TM U recognises A_{TM}:

    U = "On input $\langle M, w\rangle$, simulate M on input w;
    if M ever enters an accepting state, accept
    " " " " a rejecting state, reject."

* In proving this theorem, we show that requiring a TM to halt on all inputs restricts the kinds of languages it can recognise.
∴ A general TM is more powerful than one that halts on all inputs.

Notes:
- U loops on input $\langle M, w\rangle$ if M loops on input w.
- If there was a way to determine that M is non-halting on w, U could reject.
- A_{TM} is called the halting problem.
- U is called the Universal TM, because it can simulate any other TM from its description.

A Proof Idea:

- Assume A_{TM} is decidable and H is a TM that decides it.

$$H(\langle M, w\rangle) = \begin{cases} \text{accept} & \text{if } M \text{ accepts } w \\ \text{reject} & \text{if } M \text{ does not accept } w \end{cases}$$

- Construct a second TM D that uses H as a sub-routine.

D = "On input $\langle M\rangle$, run H on input $\langle M, \langle M\rangle\rangle$ and return the opposite of what H outputs"

$$D(\langle M\rangle) = \begin{cases} \text{accept} & \text{if } M \text{ does not accept } \langle M\rangle \\ \text{reject} & \text{if } M \text{ accepts } \langle M\rangle \end{cases}$$

- If we now run D on input $\langle D\rangle$, we have:
  - accept if D does not accept $\langle D\rangle$
  - reject if D accepts $\langle D\rangle$

- This is an obvious contradiction.

TM Variants

- Multiple tapes
- Multiple heads
- Nondeterministic TMs
- Random Access TMs
- Two-dimensional tapes
- etc...

- None of these add power: we show equivalence by simulating one on the other.

Algorithms

- A well-defined, finite set of instructions to perform a calculation or solve a problem.
- Of an algo, we can ask... is it correct?
  is it efficient? → can use Big-O$_h$
  is it optimal?

Example: Fibonacci Numbers

- Simple, exponentially recursive solution:

```
fib(n) {
  if (n==0) return 0;
  if (n==1) return 1;
  return fib(n-1) + fib(n-2);
}
```

- Correct
- Very inefficient
- Runs in exponential time

- Dynamic Programming solution:

```
fib(n) {
  create array f[0...n]
  f[0] = 0;
  f[1] = 1;
  for i = 2...n:
    f[i] = f[i-1] + f[i-2]
  return f[n];
}
```

- Correct
- More efficient
- Runs in linear time

- Iterative solution:

```
fib(n) {
  if (n <= 1) return n;
  x=0; y=0; z=1;

  for (i=1; i<n; ++i) {
    x=y;
    y=z;
    z=x+y;
  }
  return z;
}
```

- Correct
- Uses less memory than prev. solution
- Runs in linear time

- Matrix solution:

$$\begin{pmatrix} F_{n-1} \\ F_n \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{(n-1)} \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

- Correct
- Efficiency depends on exponentiation method

Example: Exponentiation

- Basic method:

$$a^n = \underbrace{a \times a \times \dots \times a}_{n}$$

- Runs in linear time.

- By squaring:

$$a^n = \begin{cases} 1 & \text{if } n = 0 \\ a & \text{if } n = 1 \\ (a^2)^{n/2} & \text{if } n \text{ is even} \\ (a^2)^{\frac{n-1}{2}} \cdot a & \text{if } n \text{ is odd} \end{cases}$$

- Runs in $\log_2 n$ time
- Fast!

Primes

- Used in various areas of cryptography.
- Huge numbers must be generated - upwards of 500 digits.

The Sieve of Eratosthenes (250 BC)

- Input: integer n
- Output: a list of all primes $\leq n$
- Process:

```
P = ∅;
L = [2, 3, 4, ..., n];
while L ≠ [] do:
    a = head(L);
    P = P ∪ a;
    delete all multiples of a from L;
return P;
```

Prime Test: Naive Division Method

- Input: integer n
- Output: true if n is prime; false otherwise
- Process:

```
if (n == 2) return true;
if (n % 2 == 0) return false;
for (int i = 1; i ≤ ⌊√n/2⌋; ++i):
    if (n % (2i+1) == 0) return false;
return true;
```

- Performs $O(\sqrt{n}) = O(n^{1/2}) = O(2^{(\log_2 n)/2})$ divisions
  - ie. Exponential in $\log_2 n$
- Very slow - useless in practise
- Randomised algorithms can be a lot faster.

Fermat's Little Theorem

- Let p be a prime number.
- For all $a \not\equiv 0$, $a^{p-1} \bmod p \equiv 1$

Also stated as $a^{p-1} \equiv 1 \pmod{p}$

- Eg. $p = 7$, $a = 4$ — $a^6 = 4096$, $4096 \bmod 7 = 1$.

Quotient-Remainder Theorem

- Let a be an integer, and b be a positive integer.
- There exist unique integers q and r such that:
  - $0 \leq r < b$
  - $a = qb + r$
- q is the quotient: $a \operatorname{div} b$
- r is the remainder: $a \bmod b$

$7 \operatorname{div} 2 = 3$
$7 \bmod 2 = 1$

- For $a \geq 0$ and $b > 0$, we define:

$$a \bmod b = \begin{cases} a & a < b \\ (a-b) \bmod b & a \geq b \end{cases}$$

Modular Arithmetic

- Two or more integers may be congruent, modulo a number N.
  - eg. $7 \equiv 19 \pmod{2}$

because... $7 \bmod 2 \equiv 1$
$19 \bmod 2 = 1$.

- If $r = a \bmod N$, then $r \in \{0, 1, \ldots, N-1\}$
- We say two numbers congruent modulo n: $a \equiv b \pmod{N}$

- Let $a_1 \equiv b_1 \pmod n$ and $a_2 \equiv b_2 \pmod n$, then...
  - $a_1 + a_2 \equiv b_1 + b_2 \pmod n$
  - $a_1 - a_2 \equiv b_1 - b_2 \pmod n$
  - $a_1 a_2 \equiv b_1 b_2 \pmod n$

Randomised Primality Testing

- From Fermat Little Theorem we know that if p is prime then:

  $\forall a \in \{1, \dots, p-1\} : a^{p-1} \equiv 1 \pmod p$

- Process:
  - Given a number n
  - Randomly select $a \in \{1, \dots, n-1\}$
  - Check whether $a^{n-1} \equiv 1 \pmod n$ holds
  - If not, then we know n is not prime
    - We say a is a witness
  - $a^{n-1} \equiv 1 \pmod n$ does not guarantee n is prime
- Checking $a^{n-1} \equiv 1 \pmod n$ can be done in $O(\log n)$.
- So the above algo can run in $O(n \log n)$.

Carmichael Numbers

- The Fermat-Test can only prove with certainty that a number is not prime.
- A positive result is not guaranteed to be correct.
- CNs are numbers that are not prime, but have very few witnesses.
- They have this property:

  $\forall (a \text{ with } \gcd(a,n)=1) : a^{n-1} \equiv 1 \pmod n$

gcd: greatest common divisor

Non-Trivial Square Roots

- Let p be prime and $a \in \{1, \ldots, p-1\}$
- For $a^2 \equiv 1 \pmod{p}$ we only have $a = 1$ and $a = p-1$
- If there is an $a \in \{2, \ldots, n-2\}$ such that $a^2 \equiv 1 \pmod{n}$, then a is a **non-trivial square root modulo n**
- Eg. Let $a = 6$, $n = 35$

  $6 \in \{2, \ldots, 33\}$ $\quad 6^2 \equiv 1 \pmod{35}$

  $\therefore$ 6 is a non-trivial square root modulo 35

  $\therefore$ 35 is **not prime**
- This can be used to improve the randomised algorithm.

Modular Exponentiation

- From the last lecture:

$$a^n = \begin{cases} a^{n/2} \times a^{n/2} & \text{if } n \text{ is even} \\ a^{\frac{n-1}{2}} \times a^{\frac{n-1}{2}} \times a & \text{if } n \text{ is false} \end{cases}$$

- This algo will compute an exponentiation mod n, and give an indication of primality:
  - Input: integers a, p, n
  - Output: $r = a^p \bmod n$, prime = false if there is an NTSR, true otherwise
  - Process: power(a,p,n):

```
prime = true;
if (p == 0) {r = 1;}
else {
    x = power(a, int(p/2), n);
    r = (x * x) mod n;
    if (r == 1 and x ≠ 1 and x ≠ n-1) prime = false;
}
if (p is odd) r = r * a;
return r, prime;
```

Miller-Rabin Algorithm

- This combines the Fermat-Test with the test for NTSRs.
- Input: integer n
- Output: false → n is not prime
  true → n is probably prime
- Process:

```
a = rand(2, n-1);
r, prime = power(a, n-1, n);
if (r ≠ 1 or !prime) return false;
return true;
```

- If n is not prime, there are at least $(n-1)/2$ numbers $a \in \{2, \ldots, n-1\}$ that are witnesses for n being a non-prime in the MR algo test.
  - The probability of the MR test giving true for a non-prime is $1/2$.
  - Run the test 50 times, and this probability is $1/2^{50}$ or 2^{-50}.

Public key Encryption

Private keys:

- Alice and Bob have a shared private key, k.
- Alice computes $V(Msg, k) = C$
- C is sent to Bob
- Bob computes $V^{-1}(C, k) = Msg$
- Problems: ○ how do they share k?
  ○ multiple people → $n(n-1)/2$ keys!

Public keys:

- Each party has two keys: public and private (secret).
- Bob gets P_A and encrypts his message: $P_A(Msg) = C$
- C is sent to Alice
- Alice uses her private key: $S_A(C) = Msg$.

P_A = Alice's public key
S_A = Alice's private key.

Key Properties

- Let D be the set of all possible messages (all finite binary strings).
- Each party has two bijective functions:

$$P_A: D \to D \qquad S_A: D \to D$$

that satisfy...

- P_A and S_A can be efficiently computed.
- $\forall m \in D: P_A(S_A(m)) = m$
- $\forall m \in D: S_A(P_A(m)) = m$
- S_A is hard to get from P_A

- The third property is used for **sending encrypted messages!**
- The second property can be used to **sign a file**

Digital Signatures

- Alice computes $\sigma = S_A(f)$ and sends f to Bob
- Anyone can prove with $P_A(\sigma) = P_A(S_A(f)) = f$ that the file really was sent by Alice
- Note: σ depends on f and is only valid for one document.
- Note: it is usually sufficient to compute $\sigma = S_A(h(f))$ where h is some hash function.

RSA Encryption Algorithm

gcd = greatest common divisor

- To generate keys:
  - Pick two large primes p and q
  - Let $n = p \times q$
    $s = (p-1) \times (q-1)$
    e = a small number w/ $\gcd(e, s) = 1$.
  - Pick d such that $d \times e \pmod{s} = 1$. (mult. inverse)
  - Publish $P(e, n)$ as the public key
  - keep $S(d, n)$ as the private key.

- Sender encrypts m: $P(m) = m^e \bmod n = c$
- Receiver decrypts m: $S(c) = c^d \bmod n = m$

Computing GCD

- $\gcd(a, b)$ = the largest integer divisor of a and b.
- Computing w/ factorisation:
  - Factorise a and b
  - $\gcd(a, b)$ = product of common factors.
- But factorisation is computationally hard.
- Euclids Algorithm:

$$\gcd(a, 0) = a$$
$$\gcd(a, b) = \gcd(b, a \bmod b) \qquad b > 0$$

- Note: if $\gcd(a, b) = d$, there exist integers m and n such that:

$$ma + nb = d$$

- Proof of Euclidi Algorithm

  ◦ Let $c = gcd(a,b)$ and $d = gcd(b,r)$.

  ◦ Note that $a = qb + r$

  ◦ From $a = qb + r$, and $d|b$, $d|r \rightarrow d|a$

  because $d|a$, $d|b$ and $c = gcd(a,b) \rightarrow d|c$

  $\therefore c \geq d$

  ◦ From $r = a - qb$ and $c|a$, $c|b \rightarrow c|r$

  because $c|b$, $c|r$ and $d = gcd(b,r) \rightarrow c|d$

  $\therefore d \geq c$

  ◦ Therefore, $c \equiv d$

Computing Multiplicative Inverse.

- Say we have $e = 5$ and $s = 6$.
- We need to pick a number d such that $5d \pmod 6 = 1$.
- Suitable options: 5, 11, 17, etc..

Full example:

- Pick $p = 13$ and $q = 7$
- $n = 91$
- $s = 72$
- $e = 5$
- $d = 29$
- $P(5, 91)$
- $S(29, 91)$

e Proof = $\gcd(5,72)$
$= \gcd(5,2)$
$= \gcd(2,1)$
$= \gcd(1,1)$
$= \gcd(1,0)$
$= 1$

d Proof = $5 \times 29 = 145$

$145 \bmod 72 = 1$.

- Encrypt:

Message = 6

Ciphertext = $6^5 \bmod 91 = 41$

- Decrypt:

Message = $41^{29} \bmod 91 = 6$

Correctness of RSA

- Does RSA satisfy the three conditions from earlier?

1. Can $P(m)$ and $S(m)$ be efficiently computed?

- Yes! m^e and m^d can be computed via exponentiation modulo n in $O(\log e) = O(\log d) = O(\log n)$.

2. Are $P(m)$ and $S(m)$ mutually inverse?

- p and q are prime, so Fermat's little theorem tells us:

$m^{p-1} = 1 \bmod p$ $\quad$ $m^{q-1} = 1 \bmod q$

- The Chinese Remainder Theorem tells us:

$m^{(p-1)(q-1)} = 1 \bmod p$ and $m^{(p-1)(q-1)} = 1 \bmod q$

- So... $S(P(m)) = (m^e)^d \pmod n$

$= m^{ed} \pmod n$

$= m^{1+r(p-1)(q-1)} \pmod n$

$= m \pmod n$

- Because we picked $e \times d = 1 \bmod s$, and $s = (p-1)(q-1)$.
- $\therefore e \times d = 1 + r \times s$ for some r,
- The same can prove the inverse, $P(S(m))$.
- So yes, property two is satisfied.

3. Is $S()$ hard to guess from $P()$?
   - Computing d is easy if we know p and q.
   - But we only know e and n, which $= pq$, but we don't know them.
   - To "crack" a message, we must factorise n.
   - This is very hard! So yes, this passes.

Graphs

- A graph G is defined by the tuple (V, E)
  - V = a non-empty set of vertices
  - E = a set of edges (pair of vertices)

- There are several types of graph:
  - Undirected graphs - edges are unordered pairs (v, w)
  - Directed graphs - edges are ordered pairs $\langle v, w \rangle$
  - Multi-graphs - multiple edges between the same two vertices, and/or "looping" edges.
  - Multi-graphs can be directed or undirected.

Terms

- Adjacent vertices: v is adj. to w iff $\exists (v,w) \in E$
- Degree: $deg(v)$ = # of vertices adjacent to v.
- Path: a sequence of vertices $v_0, v_1, \ldots v_k$ with $\exists (v_i, v_{i+1}) \in E$ for all $0 \leq i < k$
- Simple Path: path with no repeated vertices
- Length of a path = # of edges
- Cycle: a "closed" path with $v_0 = v_k$
- Simple cycle: a "closed" simple path.

Connectivity

- Connected graph: for any two vertices, a path can be found that joins them
- Subgraph: $H = (V', E')$ is a subgraph of $G = (V, E)$ if $V' \subseteq V$ and $\forall e \in E', e = (v, w) : v, w \in V'$ and $(v, w) \in E$
- Spanning subgraph: a subgraph where $V' = V$
- Induced subgraph: a subgraph where no edges have been "lost" from the vertices that remain.
- Connected component: a maximal induced subgraph that is connected.

Complete Graphs

"Most connected"

- A complete graph has an edge connecting each pair of vertices
- Complete graph of n vertices $= k_n$
- aka. "Cliques"

Tree Graphs

"Least Connected"

- No cycles
- One path between any pair of vertices
- Removing any edge disconnects the graph

Edges in Complete Graphs

- $k_1 = 0$, $k_2 = 1$, $k_3 = 3$, ... $k_n = \frac{n(n-1)}{2}$
- Proof: $k_{n+1} = k_n + n$ (from k_n we have to add n edges to draw k_{n+1})

$$k_{n+1} = \frac{n(n-1)}{2} + n = \frac{n(n-1)+2n}{2} = \frac{n(n-1+2)}{2} = \frac{n(n+1)}{2}$$

Edges in Tree Graphs

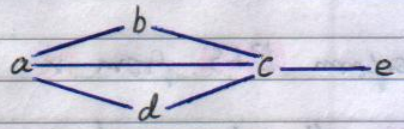
- A tree graph with n vertices has $n-1$ edges.

Spanning Trees

- A spanning tree of a graph G is a spanning subgraph that is also a tree.
- A complete graph k_n has n^{n-2} distinct spanning trees.
- Spanning trees can be found by starting at a node and applying depth-first or breadth-first searches, recording the edges that lead to unvisited nodes.

Graph Representations

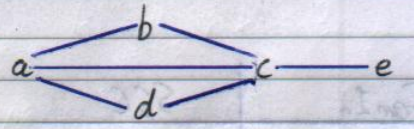
- **Adjacency List**
  - For all $v \in V$, $L(v)$ lists all w such that $(v,w) \in E$.
  - Eg.



$\Rightarrow$

v	$L(v)$
a	b c d
b	a c
c	a b d e
d	a c
e	c

- **Adjacency Matrix**
  - An $n \times n$ matrix such that $A[v,w] = 1$ means $(v,w) \in E$.
  - Eg.



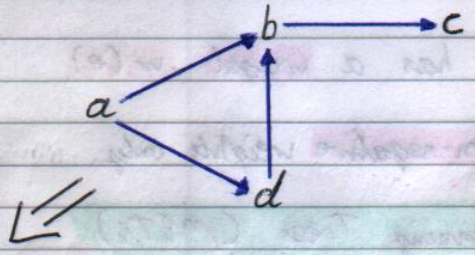
$\Rightarrow$

	a	b	c	d	e
a	0	1	1	1	0
b	1	0	1	0	0
c	1	1	0	1	1
d	1	0	1	0	0
e	0	0	1	0	0

Symmetric for a simple graph.

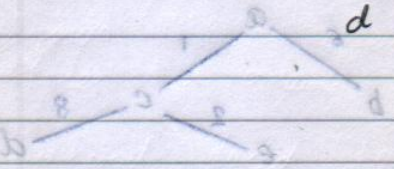
Digraphs

- Edges are ordered pairs: $\langle v,w \rangle \Leftrightarrow v$ is adj. to w
- $\langle v,w \rangle$ does not imply $\langle w,v \rangle$
- Eg.



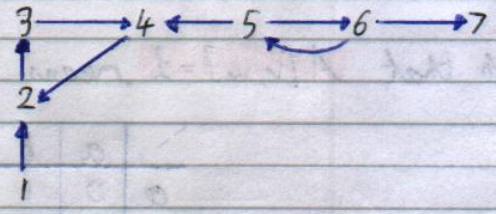
v	$L(v)$
a	b d
b	c
c	
d	b

	a	b	c	d
a	0	1	0	1
b	0	0	1	0
c	0	0	0	0
d	0	1	0	0



- FanOut (v) : set of all vertices reachable from v (incl. v)
- FanIn (v) : set of all vertices from which v is reachable (incl. v)
- SCC (v) : strongly connected component containing v

  scc (v) = FanIn (v) ∩ FanOut (v)
- To find FanOut (v), perform BFS from v
- To find FanIn (v), reverse all edges and perform BFS from v
- Eg:



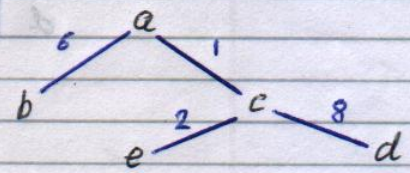
Node	FanOut	FanIn	SSC
1	1 2 3 4	1	1
2	2 3 4	2 3 4 5 6 1	2 3 4
3	3 4 2	3 2 4 5 6 1	2 3 4
4	4 2 3	4 1 2 3 5 6	2 3 4
5	5 6 7 4 2 3	5 6	5 6
6	6 5 7 4 2 3	6 5	5 6
7	7	7 6 5	7

Weighted Graphs

- Each edge e has a weight w (e)
- We consider non-negative weights only.

Minimum Cost Spanning Trees (MSTs)

- An MST for the graph G is a spanning tree of G such that the combined sum of all edges is minimised.
- Eg. T_a =



$W(T_a) = 6 + 1 + 2 + 8$
$= 17$

- Prim's Algorithm for finding MSTs

  - Input: graph G with vertices V and edges E
  - Output: T, a subset of E inducing MST
  - Process:

```
S = {}                                  (set of seen vertices)
Start at some vertex v
S = {v}
Until T spans G, repeat:
    Add the cheapest edge e joining an unvisited vertex w to T
    T = T ∪ e
    S = S ∪ w
Return T
```

- Kruskal's Algorithm for finding MSTs (Growing Forests)

  - Grows a forest of trees and finds the cheapest edge to connect two disjoint trees.
  - Input: graph G with vertices V and edges E
  - Output: T, a subset of E inducing MST
  - Process:

```
T = {}                                  (set of edges belonging to MST)
Start w/ all vertices and no edges
Sort all edges in order by weight
T = {e1}
Until T spans G, repeat:
    Does the next cheapest edge (v,w) join two distinct connected
    components of T?
    T = T ∪ {(v,w)}
Return T
```

Single Source Shortest Paths

- The shortest path P from nodes v to w is a path from v to w such that the cost of edges in that path is minimised.
- One approach to find this path is Dijkstra's Algorithm.

- Dijkstra's Algorithm

* We set $w(e) = \infty$ for all $e \notin E$

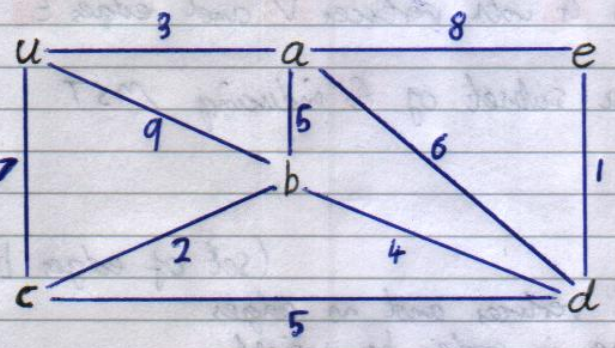
- Input: Graph $G = (V, E)$ and vertex $u \in V$
- Output: shortest path from u to all other vertices
- Process:

Initial Phase
- $S = \{u\}$ (set of seen vertices)
- $\ell(u) = 0$ (cost of reaching this vertex)
- for all $y \in V$ do $\ell(y) = w(u, y)$ *

Looping Phase
while $S \neq V$:
- Choose $x \in V - S$ with smallest $\ell(x)$ and add x to S
- for all $y \in V - S$:
  - $\ell(y) = \min(\ell(y), \ell(x) + w(x, y))$

return paths according to ℓ

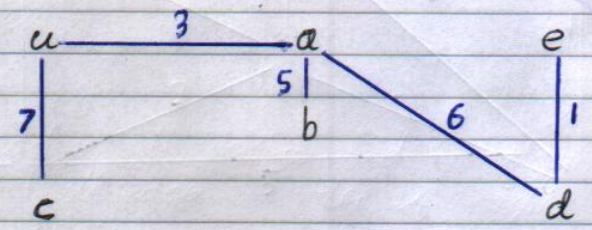
- Dijstra Example Graph:



Problem: Find the shortest path from u to all other nodes.

	x	V − S	"old l(·)" l(y)	l(x) + w(x,y)	"new l(·)" min	from	edge
Initial Phase	u	a	3	-	3 *	u	u, a
		b	9	-	9		
		c	7	-	7		
		d	∞	-	∞		
		e	∞	-	∞		
Looping Phase	a	b	9	3 + 5 = 8	8		
		c	7	3 + ∞ = ∞	7 *	u Δ1	u, c
		d	∞	3 + 6 = 9	9		
		e	∞	3 + 8 = 11	11		
	c	b	8	7 + 2 = 9	8 *	a Δ2	a, b
		d	9	7 + 5 = 12	9		
		e	11	7 + ∞ = ∞	11		
	b	d	9	8 + 4 = 12	9 *	a Δ3	a, d
		e	11	8 + ∞ = ∞	11		
	d	e	11	9 + 1 = 10	10 *	d Δ4	d, e

Result:



* = "choose an x with the lowest l(x)"

Δ1 c was given 7 during the u round

Δ2 b was given 8 during the a round

Δ3 d was given 9 during the a round

Δ4 e was given 10 during the d round.