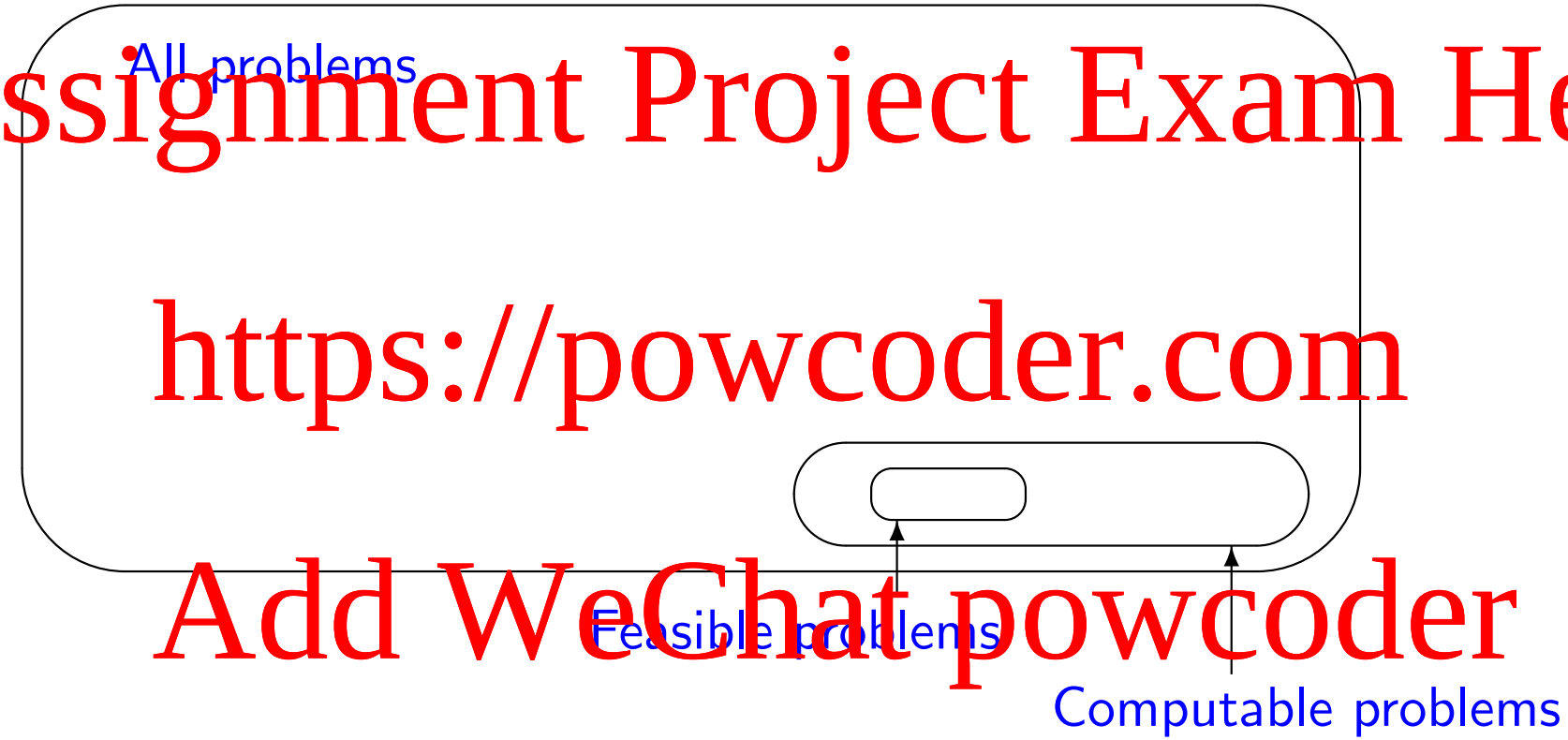
Turing Machines: Limits of Decidability

COMP1600 / COMP6260

Dirk Pattinson Victor Rivera
Australian National University

Semester 2, 2021

Efficient Computation



- Each of the inner sets is a tiny proportion of the set that contains it.

Time Complexity

Q. It's good about solving problems in general. How about efficiency?

Inverting booleans. Easy: constant time.

Multiplication of integers. Easy: polynomial time.

- takes time that is proportional to the square of the total number of digits
- if we double the digits, it takes *4 times* as long.
- if n are the total number of digits, multiplication has complexity *of the order* n^2.

Matrix Multiplication. Polynomial of order $n^{2.376}$

Theorem Proving. Hard, sometimes of order 2^{2^n} (or even undecidable)

Feasible Problems.

- can be solved in *polynomial time*, i.e. in time of the order n^k, for some k
- n is the length of (the representation of) the input.

P vs. NP: Signpost

Polynomial Time. The complexity class P

- problems (languages) that can be decided in the order of n^k steps
- usually considered feasible

Non-deterministic polynomial time. The complexity class NP

- problems that can be decided *with guessing* in polynomial time
- alternatively, problems whose solutions can be verified in polytime

Example. Boolean satisfiability is in NP

- given boolean formula A, can A evaluate to T?
- can guess solution (assignment)
- alternatively, can verify correctness of assignment in polynomial time

As a slogan. Coming up with a solution within a time bound seems intuitively harder than checking someone else's solution in that time.

Big Open Problem. Is $P = NP$?

- most important open problem in our discipline
- $1,000,000$ prize by the Clay maths foundation

More Detail

Computational Problem.

- given by a *Language* L, say
- Question: for a string w, is $w \in L$?

Solution. A Turing machine M that

- always halts
- and accepts w if and only if $w \in L$.

Time Complexity

- Given w, can count the number of steps of M to termination
- this defines a function $f(w)$ *dependent on the input*

Time Complexity – Abstraction

Problem. Number of steps function usually *very complicated*

- for example, $n^{17} + 23n^2 - 5$
- and hard to find in the first place

Solution. Consider *approximate* number of steps

- focus on *asymptotic* behaviour
- as we are only interested in *large* problems

Landau Symbols. for f and g functions on natural numbers

- $f \in \mathcal{O}(g)$ if $\exists c.\exists n_0.\forall n \geq n_0(f(n) \leq c \cdot g(n))$
- "for large n, g is an upper bound to f up to a constant."

Idea. Abstract details away by just focussing on upper bounds

- e.g. $n^{17} + 23n^2 - 5 \in \mathcal{O}(n^{17})$

Landau Symbols: Examples

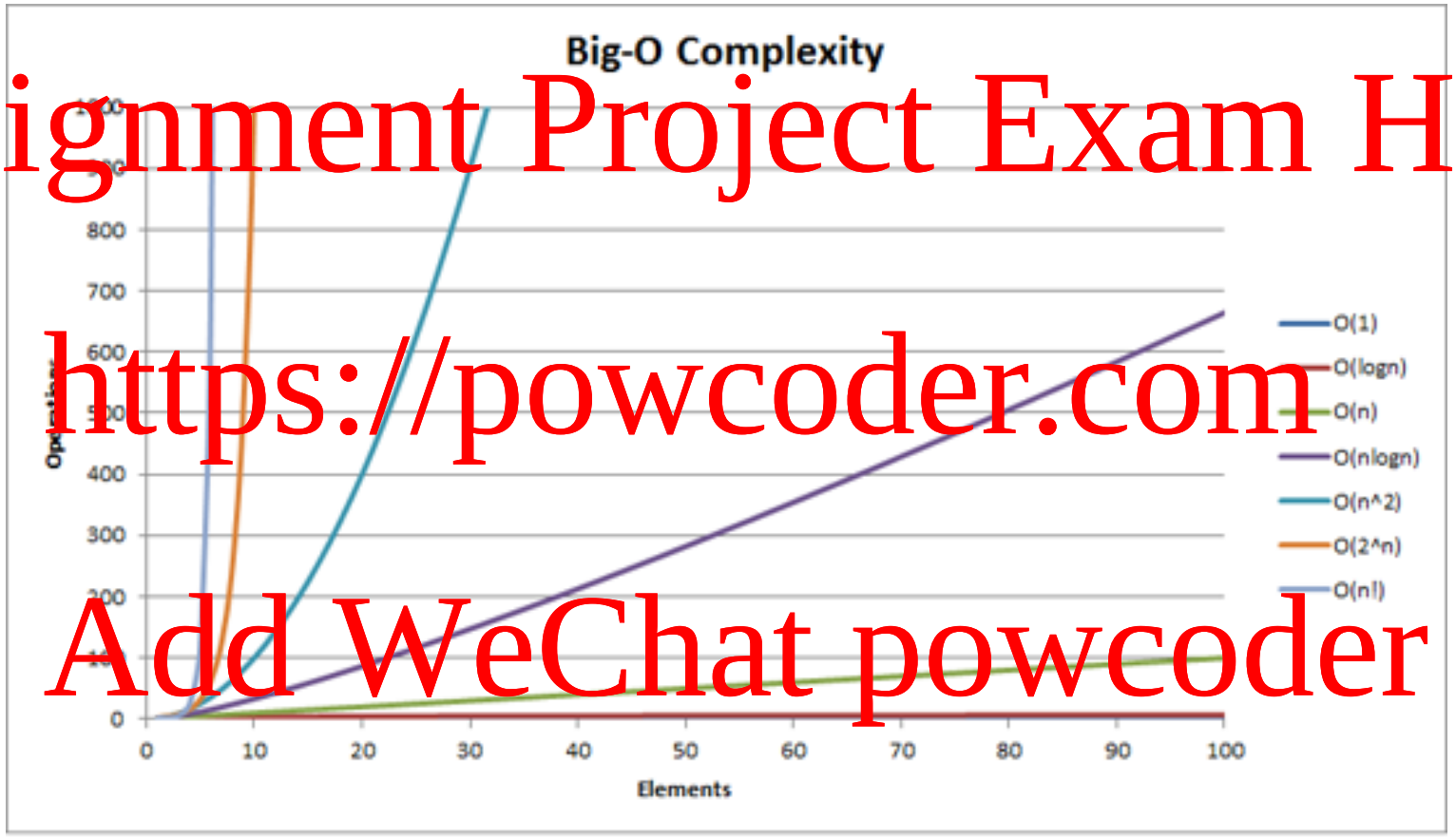
Examples.

- Polynomials: leading exponent dominates
  - e.g. $x^n +$ lower powers of $x \in \mathcal{O}(x^n)$
- Exponentials: dominate polynomials
- e.g. $2^n +$ polynomial $\in \mathcal{O}(2^n)$

Important Special Cases.

- *linear*. f is linear if $f \in \mathcal{O}(n)$
- *polynomial*. f is polynomial if $f \in \mathcal{O}(n^k)$, for some k
- *exponential*. f is exponential if $f \in \mathcal{O}(2^n)$

Important Special Cases, Graphically



(Image copyright Lauren Kroner)

Application to Computational Problems

Definition.

A computational problem (given by a language L) is *in* $\mathcal{O}(f)$ if there is a Turing machine M that

- always terminates, and accepts precisely all strings in L
- on every input string of length n, terminates in $g(n)$ steps and $g \in \mathcal{O}(f)$

Example: Regular Languages

- Given: regular language L
- Question: what's the complexity of deciding whether $w \in L$?

More Detail.

- need to construct a Turing machine that decides whether $w \in L$ or not.
- how many steps (depending on length of input string) does M take?

Application to Computational Problems

Definition.

A computational problem (given by a language L) is *in* $\mathcal{O}(f)$ if there is a Turing machine M that

- always terminates, and accepts precisely all strings in L
- on every input string of length n, terminates in $g(n)$ steps and $g \in \mathcal{O}(f)$

Example: Regular Languages

- Given: regular language L
- Question: what's the complexity of deciding whether $w \in L$?

More Detail.

- need to construct a Turing machine that decides whether $w \in L$ or not.
- how many steps (depending on length of input string) does M take?

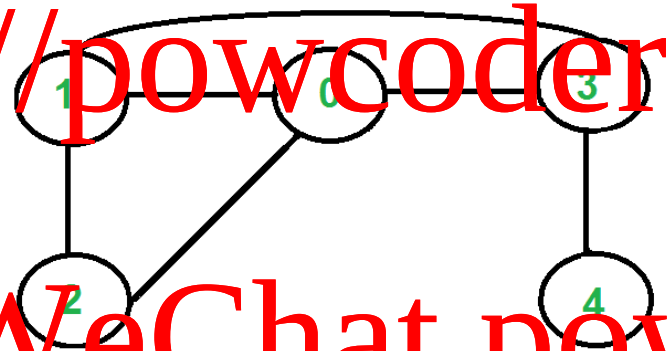
A. This is *linear*. Think of finite automata.

Example: Graph Connectedness

Reminder. A *graph* is a pair $G = (V, E)$ where

- V is the set of *vertices* of the graph
- E is the set of *edges*, a collection of two-element subsets of V

Example.



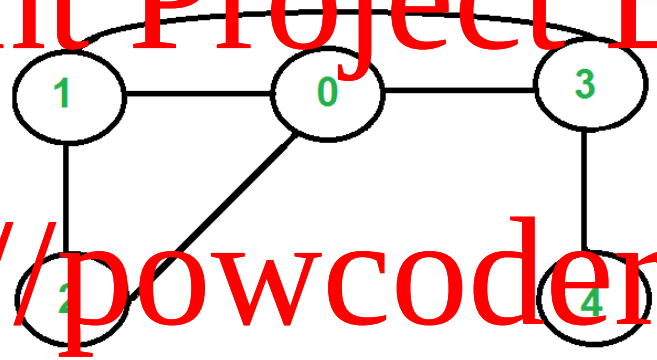
Formally. $G = (V, E)$ with

- $V = \{0, 1, 2, 3, 4\}$
- E consisting of $\{0, 2\}$, $\{0, 1\}$, $\{0, 3\}$, $\{1, 2\}$, $\{1, 3\}$, and $\{3, 4\}$.
- Note: Edges are *not directional*.

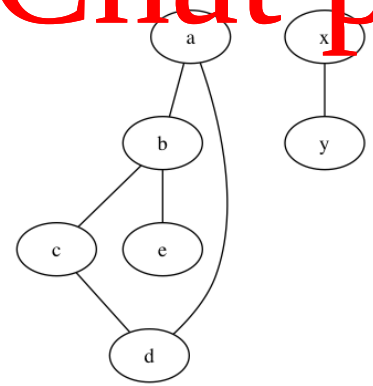
Connected and non-connected Graphs

Definition. A graph is *connected* if there is a path between any two nodes.

Connected Graph Example.



Non-Connected Graph Example



The Connected Graph Problem

Problem. Given a graph $G = (V, E)$

- is G connected?
- what is the complexity of deciding whether G is connected?

Algorithm.

- Pick a vertex v in G as starting vertex
- do a breadth-first search and remember vertices encountered
- if the total number of vertices found equals the number of vertices in the graph, we know that G is connected.

By Church-Turing Thesis. The problem "Is G connected?" is clearly computable.

Q. How many steps does an algorithm take to figure this out?

- number of steps refers to "on a Turing machine"
- but input to TMs are strings, not graphs ...

Coding of Graphs as Strings

Recall. We have coded TM transition tables as strings.

Coding of graphs:

- vertices: numbered 0 to n, can be coded by n in binary
- single edge: pair (n, k), can be coded by $\underbrace{01\ldots0}_{n}\#\underbrace{11\ldots1}_{k}$.
- set of edges: can be coded by $e_1\#\#e_2\ldots e_{l-1}\#\#e_l$

Complete Graph

$10\ldots1\#\#1\ldots1\#1\ldots0\#\#\ldots\#\#11\ldots0\#1\ldots0$

- green number of vertices
- blue set of edges
- black separators

Question, reloaded.

Computational Problem. Given a graph $G = (V, E)$, how many steps does a TM take to determine whether the *encoding* of G is a connected graph?

- the encoded graph is now a string
- number of steps relative to the *length of encoding*

Difficulty. Exact answers require *way too much* bookeeping.

- Landau symbols $\mathcal{O}(\cdot)$ allow us to be "generous"
- required answer of the form "in $\mathcal{O}(f)$".

Complexity Analysis

Algorithm in Turing machine form.

- pick vertex 0 initially.
- designate vertex 0 as "to explore" (e.g by writing its binary encoding to the right of the input)
- for every vertex v that is (still) to explore:
  - search through the edges to find other vertices it connects with
  - if a connected vertex is neither explored nor marked "to explore", mark it as "to explore" (e.g. by writing its binary encoding to the right of the input, with a special separator)
  - mark the vertex v as "explored"
- check that the number of vertices found is equal to the number of vertices in the graph.

Worst Case Analysis

Worst Case for a given graph G with n vertices

- When exploring a vertex, need to check n^2 edges
- For every edge checked, one more vertex to explore
- this needs to be done for every vertex

High Level Analysis. Of complexity $\mathcal{O}(n^3)$ – *polynomial*

- need to do n^2 checks, at most n times

Overhead of a Turing implementation.

- checking whether two vertices match: *polynomial*
  - at most n (in fact $\log n$) bitwise comparisons
  - and going back and forth over the tape, at most $n^2 \cdot n$ *times*
- adding another vertex to the list: *polynomial*
  - at most n bits to add
  - and going back and forth over the tape, at most $n^2 \cdot n$ steps

Summary. Polynomial Complexity

- polynomially many "high level" steps
- each of which takes polynomial time

Other Problems: Propositional Satisfiability

Given. A propositional formula, constructed from $\wedge$, $\vee$, $\rightarrow$, $\neg$, T, F and variables.

Question. Is there a truth value assignment to the propositional variables such that the formula evaluates to T?

Naive Algorithm. Truth tables

- loop through all possible assignments and evaluate the formula

Questions.

1. How many truth assignments do we need to check?
2. How do we measure the size of the input?
3. What is the worst case complexity of this algorithm?

Complexity Class: Polynomial Time

Definition. The class **P** of *polynomial time* decision problems consists of all problems that can be answered in time *polynomial* in the input

- of order $\mathcal{O}(n), \mathcal{O}(n^2), \mathcal{O}(n^3), \ldots$

Examples

- check whether a graph is connected
- check whether a list is sorted
- check whether a propositional formula is true for a given valuation

Last Problem. Have *two* inputs

- need only *one line* of the truth table
- according to the valuation given

Other Problems: Propositional Satisfiability

Given. A propositional formula, constructed from $\wedge$, $\vee$, $\rightarrow$, $\neg$, T, F and variables.

Coding.

- have new tape symbols for $\wedge$, $\rightarrow$, etc.
- assume that variables are numbered, encode in binary

Worst Case.

- variables proportional to length of formula (e.g. $p_1 \wedge p_2 \wedge p_3 \wedge \ldots$)
- *exponentially* many valuations

This Algorithm

- *at least* exponential, e.g. $\mathcal{O}(2^n)$
- and in fact exponential

Q. Can we do better?

Other Problems: Propositional Satisfiability

Given. A propositional formula, constructed from $\wedge$, $\vee$, $\rightarrow$, $\neg$, T, F and variables.

Coding.

- have new tape symbols for $\wedge$, $\rightarrow$, etc.
- assume that variables are numbered, encode in binary

Worst Case.

- variables proportional to length of formula (e.g. $p_1 \wedge p_2 \wedge p_3 \wedge \ldots$)
- *exponentially* many valuations

This Algorithm

- *at least* exponential, e.g. $\mathcal{O}(2^n)$
- and in fact exponential

Q. Can we do better?

A. Probably not . . . this is the $1,000,000 Clayton math problem

Propositional Satisfiability

Verifying whether a formula evaluates to T for an assignment

- takes polynomial time

Determining whether a satisfying assignment exists

- takes exponential time
- *but* if we could *guess* an assignment, it would be fast (polynomial)

Observation. The (coding of) a valuation is *polynomially large*

- in fact, shorter than the formula, a sequence of 0s and 1s

Non-Deterministic Machines (informally)

- like non-deterministic finite automata: more than one transition possible
- propositional satisfiability: *guess* a valuation, then check

Complexity Class: Nondeterministic Polynomial Time

Definition. The class **NP** of *non-deterministic polynomial time* decision problems consists of all decision problems L that can be solved by a *non-deterministic* Turing machine in polynomial time

- we don't define this type of machine formally
- idea: can make *guesses* at every stage
- accepts if the machine *can* move into final state

Alternative Characterisation L is in **NP** if, for every string $w \in L$ there exists a *certificate* c such that

- c is of polynomial length (in the length of w)
- determining whether c is a certificate for $w \in L$ is in **P**

Example. Propositional Satisfiability

- certificates are valuations
- checking the formula under a valuation is polynomial

More Problems

The Independent Set Problem Assume you want to throw a party. But you know that some of your friends don't get along. You only want to invite people that *do* get along.

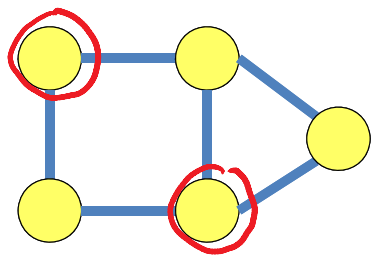
As a Graph:

- vertices are your mates
- draw an edge between two vertices if people don't get along

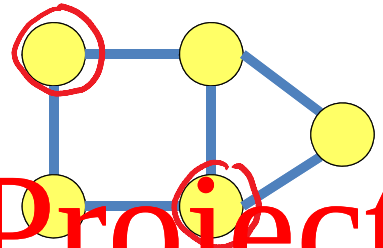
Problem. Given $k \geq 0$, is there an *independent set*, i.e. a subset I of $\geq k$ vertices so that

- no two elements of I are connected with an edge.
- i.e. everybody in I gets along

Example of an independent set of size 2



Independent Set: Naive Algorithm



- loop through all subsets of size $\geq k$
- and check whether they are independent sets

Alternative Formulation using guessing:

- *guess* a subset of vertices of size $\geq k$
- *check* whether it is an independent set

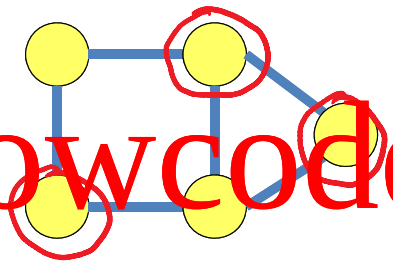
Complexity. Independent Set is in **NP**

- represent subsets as bit-vectors (certificates)
- checking is polynomial

Vertex Cover

Vertex Cover. Given a graph $G = (V, E)$, a *vertex cover* is a set C of vertices such that every edge in G has at least one vertex in C.

Example.



Vertex Cover Problem. Given a graph $G = (V, E)$ and $k \geq 0$, is there a vertex cover of size $\leq k$?

Naive Algorithm.

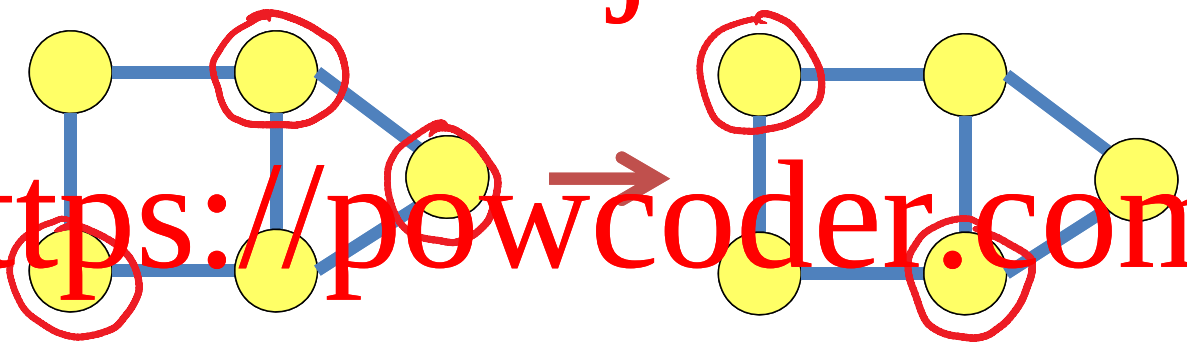
- search through all subsets of size $\leq k$
- check whether it's a vertex cover

From Independent Set to Vertex Cover

Reductions. Use solutions of one problem to solve another problem

Observation. Let G be a graph with n vertices and $k \geq 0$.

- G has a v. c. of size $\leq k$ iff G has an i. s. of size $n - k$



Reduction. A *polynomial reduction* from decision problem A to decision problem B is a function f that transforms A-instances to B-instances and

- $w \in A \iff f(w) \in B$ and f is computable in polynomial time

Example. Vertex cover to independent set

$$(G, k) \mapsto (G, n - k)$$

where n is the number of vertices of G

Reductions and Difficulty

Recall. A reduction from decision problem A to B is a polytime function f such that $w \in A \iff f(w) \in B$.

Informally. If we can solve B, then we can also solve A.

- Given w, is $w \in A$?
- Compute $f(w)$, and decide whether $f(w) \in B$

Q. If A is reducible to B, which of A and B is more "difficult"?

NP-Completeness

Q. What is the "hardest" or most difficult **NP**-problem?

A. It's a problem that all other **NP**-problems can be reduced to

- a solution would yield solutions to *all* **NP**-problems
- recall that B is more difficult than A if A can be reduced to B

NP-Hardness and completeness

- A decision problem L is NP-hard if all other **NP**-problems can be reduced to it.
- A decision problem is NP-complete if it is NP-hard and in **NP**.

Hard Theorem. (Stephen Cook 1974) The propositional satisfiability problem is NP-complete

- have seen that satisfiability is in **NP**
- hard part: reduce *all* **NP**-problems to satisfiability.

The P vs NP Problem

Big Open Question. is $\mathbf{P} = \mathbf{NP}$ or not?

- Given that propositional satisfiability is NP-complete
- "all" it takes is to solve *one* problem efficiently!

Ramifications If $\mathbf{P} = \mathbf{NP}$ ...

- could break public-key cryptography
- this includes https-protocol!
- could solve optimisation problems efficiently

- lots of AI (learning) problems have *fast* solutions

Summary.

Undecidable Problems.

- Problems for which we cannot find an algorithmic answer
- most famous: halting problem – determine whether a computation terminates

Efficiently Solvable Problems.

- usually identified with polynomial time, i.e. **P**

More difficult Problems. Polynomial time with guessing

- complexity class **NP**, not considered feasible
- **NP**-complete problems, like propositional satisfiability

Open Problem. Is **P** = **NP** or not?

- neither have proof nor counter-example
- most important open problem in the discipline

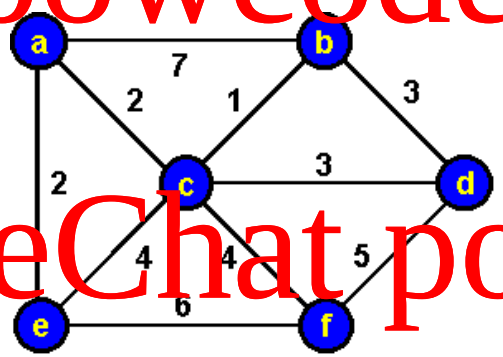
Weighted Graphs

Definition. A *weighted graph* is an undirected graph where every edge is (additionally) labelled with a non-negative integer.

Formally: A *weighted graph* is a triple $G = (V, E, l)$ where

- (V, E) is an undirected graph (with vertices V and edges E)
- $l : E \to \mathbb{N}$ is a labelling function that assigns a *weight* (or cost) to each edge.

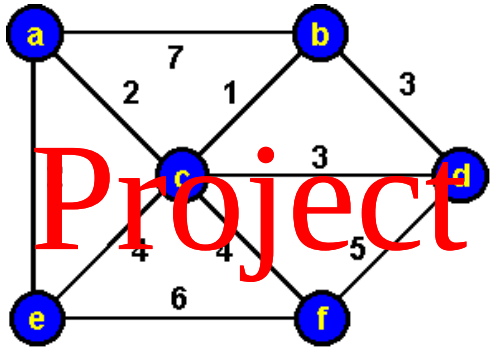
Example.



Intuition.

- the vertices can be locations
- the edges indicate whether we can travel between two locations
- the labels indicate the cost of travelling between two locations

Travelling Between Two Nodes



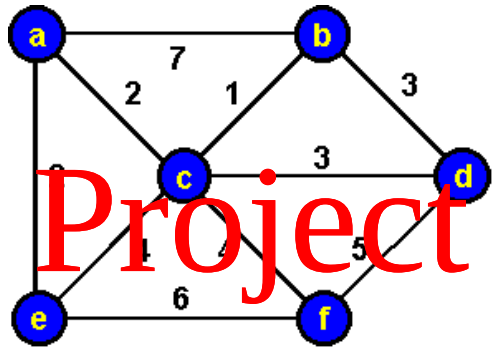
Q. Given two vertices v_1 and v_2, what is the 'cheapest' way of getting from v_1 to v_2?

In More detail.

- as a computation problem: given vertices v_1 and v_2, find a cheapest path that connects v_1 and v_2
- as a decision problem: given graph G and $k \geq 0$, is there a path that connects vertices v_1 and v_2 of total cost $\leq k$?

Q. Easy or hard? Solvable in polynomial time?

Dijkstra's Algorithm



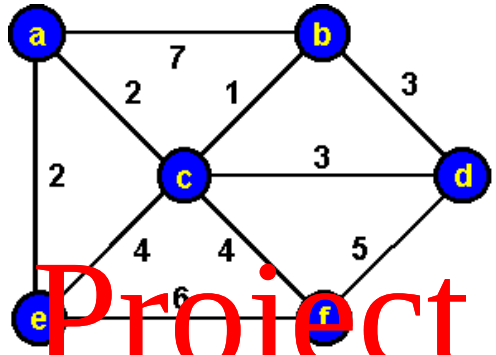
Main Idea.

- to find a shortest path between v_1 and v_2, need to consider intermediate nodes
- more general problem: shortest path between v_1 and *any* vertex v_2
- a bit like breadth-first search

Data Structures. Given a start vertex s

- `cheapest[v]`: For every vertex v, the "price" of the "cheapest" path from s to v
- `explored[v]`: a boolean marker whether `v` has been explored

Algorithm Detail for source vertex s



Initialisation.

- set `cheapest[s] = 0`, `cheapest[v] = undef` for `v` $\neq$ `s`
- set `explored[v] = false`, all $v \in V$

Iteration.

1. Select a vertex in v that is *not explored* and *minimal*:
   - `explored[v]` $=$ `false`
   - `cheapest[v]` $\leq$ `cheapest[w]` for all `w` with `explored[w] = false`
2. for each vertex w that is adjacent to `v`:
   - update, if the path $s \rightsquigarrow v \rightarrow w$ is cheaper, that is
   - if `cheapest[v] + cost(v, w)` $\leq$ `cheapest[w]`
   - then `cheapest[w] = cheapest[v] + cost(v, w)`
3. mark `v` as explored: `explored[v] = true`
   - once a node `v` is marked explored, `cheapest[v]` is the price of the cheapest path from source to `v`

Dijkstra's Algorithm: Correctness

Invariant. if E is the set of explored nodes, then

- for $e \in E$, `cheapest[e]` is the cost of cheapest path from source to `e`
- for $u \in V \setminus E$, `cheapest[u]` is the cost of the cheapest path to u that only visits explored nodes.

True after Initialisation.

- trivial, as all nodes are marked as unexplored

Invariant holds after every iteration.

- pick minimal and unexplored node u
- cheapest path from source to u *cannot* traverse unexplored nodes
- after (possible) update: `cheapest` is still minimal for paths through explored vertices

At End of Iteration.

- `cheapest[v]` is cost of cheapest path from source to `v`

Recognise the While-Rule in Hoare Logic?

- *could* formalise this in Hoare logic
- here: Hoare-Logic as informal guidance principle

Dijkstra's Algorithm: Complexity

Worst Case Focus.

- need to explore all nodes

In Every Iteration

- possibly update `cheapest[v]`, for all vertices `v`

Overall Complexity for a graph with n vertices

- run through n iterations, in each iteration:
  - find minimal unexplored vertex – n steps
  - update `cheapest[v]` – n steps
- so overall $\mathcal{O}(n^2)$ steps

Low-Level Operations are "harmless" (polynomial)

- comparing two n-bit binary numbers
- marking / checking explored status

Shortest Paths

Decision Problem. Given G, v_1, v_2 and $k \geq 0$ is there a path of cost $\leq k$ from v_1 to v_2?

- run Dijkstra's algorithm and then check whether $\texttt{cheapest}[\texttt{v}_2] \leq k$

Computational Problem. Given G, v_1 and v_2, *find* the shortest path from v_1 to v_2

- Dijkstra's algorithm only gives cost
- paths needs to be re-constructed
- idea: remember *penultimate* nodes

Computing Shortest Paths

Initialisation.

- set `cheapest[s] = 0`, `cheapest[v] = undef` for `v` $\neq$ `s`
- set `explored[v] = false` all $v \in V$
- set `penultimate[v] = undef`, all $v \in V$

Iteration.

1. Select a vertex in v that is *not explored* and *minimal*
   - `explored[v]` $=$ `false`
   - `cheapest[v]` $\leq$ `cheapest[w]` for all `w` with `explored[w]` $=$ `false`
2. for each vertex w that is adjacent to v:
   - update, if the path $s \rightsquigarrow v \rightarrow w$ is cheaper, that is
   - if `cheapest[v] + cost(v, w)` $\leq$ `cheapest[w]`
   - then `cheapest[w] = cheapest[v] + cost(v, w)`
   - and put `penultimate[w] = v`
3. mark `v` as explored: `explored[v] = true`
   - once a node `v` is marked explored, `cheapest[v]` is the price of the cheapest path from source to `v`

Path Reconstruction

Idea.

- `penultimate[v]` is the *penultimate* node of a cheapest path from `source` to `v`

Reconstruction of a path from `source` to `v`

- initialisation: the *last* node is `v`
- iteration: path expansion *on the left*
  - if partial path $v_1, \ldots, v_n$ is already constructed
  - extend it to $\texttt{penultimate}[v_1], v_1, \ldots, v_n$
- termination: if the constructed path is of the form $\texttt{source}, \ldots, v_n$

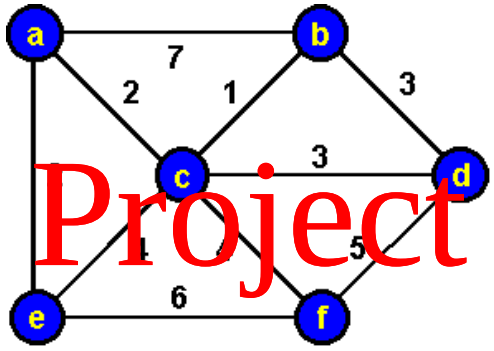
Complexity

- reconstruction phase: at most n additional steps
- iteration phase: adds constant overhead
- so overall, still polynomial

How About Coding Graphs as Strings?

- our analysis in terms of *number of vertices*
- bounded above by length of encoding

Travelling Salesman Problem



Given. A weighted undirected graph

- vertices represent cities
- edges represent travel time

Q. Find a path $v_1 \to v_2 \to \cdots \to v_n = v_1$

- that begins and ends in the same city
- that visits each city *exactly once*
- for which the overall travel time is minimal.

Q. Easy or hard? Solvable in Polytime?

Travelling Salesman: Naive Approach

Naive Algorithm Given n cities, and their distances $\texttt{dist}(c, d)$

- initialise: construct set S consisting of all possible sequences of nodes
  - sequences may not have repeated vertices
  - sequences must contain all vertices of the graph
- computation phase:
- for each $s = (c_1, \ldots, c_n) \in S$, compute the total distance $\sum_i \texttt{dist}(c_i, c_j)$
- report the smallest such distance

Q. What is the complexity of this algorithm?

Travelling Salesman: Naive Approach

Counting Permutations of n vertices

- n possibilities for 1st city, $n-1$ for 2nd, $n-2$ for third ...
- overall: $n!$ different paths to check

Estimate on a machine that does 1,000,000 steps per second

- simplified: this does not include overhead for moving on Turing tapes

size	time
20	7
40	$2 \cdot 10^{31}$
80	$2.3 \cdot 10^{102}$

Q. What is the unit of time in the right-hand column?

(we have seen this before)

Travelling Salesman as Decision Problem

Travelling Salesman as Decision Problem. Given weighted graph G and $k \geq 0$, is there a path that

- visits each vertex in G exactly once
- is of total cost $\leq k$

Guessing and Checking.

- Certificate for existence of path is path itself
- of polynomial size (if G is encoded "reasonably", i.e at least one bit per vertex)

Certificate Checking. As for Hamiltonian paths, plus

- check that the total cost of the path is $\leq k$
- $n - 1$ additions, one comparison – *polynomial*
- so checking is polynomial, overall

Corollary. Travelling Salesman *as a decision problem* is in NP.

The Knapsack Problem

Given: A knapsack with maximal capacity C and items with weights $w_1, \ldots, w_n$ and values $v_1, \ldots v_n$

What's the best way to fill the knapsack?

- sum of the weight of the items should not exceed capacity
- sum of the values of the items should be maximal.

Assumptions.

- all weights are strictly positive, i.e. $w_i > 0$ for all $i = 1, \ldots, n$
- there is an unlimited supply of each item

Knapsack: Main Idea

Construct a Table

0	1	2	...	C
$m(0)$	$m(1)$	$m(2)$	...	$m(C)$

- $m(w)$ = value of the "best" knapsack with weight w
- $m(0) = 0$ (empty knapsack)
- $m(w) = \max\{v_i + m(w - w_i) \mid w_i \leq w\}$

Solution. $m(C)$ = value of the best knapsack with capacity C

Correctness Argument.

- the "best" knapsack with weight w must contain an item – say item i
- removing this item, we get the best knapsack with weight $w - w_i$ (otherwise, can have better knapsack with weight w)

Solution.

- iteratively compute $m(0), m(1), \ldots, m(C)$
- store already computed values in a table (don't recompute)

Knapsack: Complexity Analysis

Complexity Analysis given capacity C and n items

- need to construct a table with $C + 1$ entries (starting at zero)
- for each > 0 entry for weight: need to check n items, compute maximum
- Overall complexity: $n \cdot C$

Q. Does that mean that knapsack is *linear* or *quadratic*?

Knapsack: Encoding

Recall. Complexity depends on *encoding* of input data

- usual encoding for numbers: as binary strings

For Knapsack:

- Capacity: C as a binary integer ($\log C$ bits)
- number of items n as binary integer ($\log n$ bits)
- weights as n-element lists of binary integers
- values as n-element lists of binary integers

Q. How large is C as a function of the *length of the encoding* of the knapsack problem *in the worst case*?

Knapsack: Encoding

Recall. Complexity depends on *encoding* of input data

- usual encoding for numbers: as binary strings

For Knapsack:

- Capacity: C as a binary integer ($\log C$ bits)
- number of items n as binary integer ($\log n$ bits)
- weights as n-element lists of binary integers
- values as n-element lists of binary integers

Q. How large is C as a function of the *length of the encoding* of the knapsack problem *in the worst case*?

A. In the worst case:

- one item, value 1, weight 1: 3 bits (plus separators)
- encoding of C has $\log C$ many bits, so $C = 2^{\log C}$ is *exponential*

Overall Complexity. *Exponential* in the size of the problem

- if numbers are coded in binary
- only polynomial if C is coded in unary – also called *pseudo polytime*

Summary

Bad News.

- there are *lots* of problems that are undecidable about TMs
- Rice's theorem gives plenty – any property of languages

But ...

- specific *instances* may be decidable
- often useful to *search* for solutions

Example. Provability in First-Order Logic

- not decidable, but there are plenty of implementations
- implementations may not terminate ...
- goal: try and solve *as many problems as possible*
- problem of interest: usually human-generated

Course Summary

Alignment

- you can already do programming (functional / imperative)

Questions.

- What do programs *really* do?
- What is computation in general? What are the limits?

Answers.

- use *logic* to formally describe systems
- functional programs: induction proofs
- imperative programs: Hoare logics
- computation in general: Turing machines