Graphs & Combinatorial Problems

- A new part of the course – will cover the more theoretical aspects required in later lectures
  - Graphs, cliques, and colouring
  - Algorithms and intractability
  - Linear programming and integer linear programming
  - Shortest and longest path algorithms
- This lecture covers
  - Definition of graph (revision), clique, and clique number
  - Graph colouring, chromatic number
  - Interval graphs

Graphs

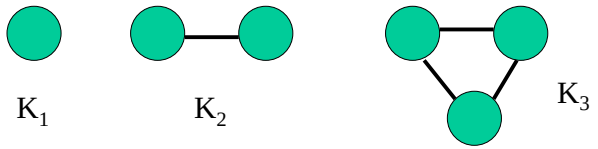
- Formal Definition:
  - A *graph G* is a finite nonempty set *V* together with an [irreflexive], symmetric relation *E* on *V*
- The relation *E* relates vertices to other vertices and is known as the edge relation, or "edge set"
- If relation *E* is symmetric, it means that
  - $(a,b) \in E \Rightarrow (b,a) \in E$
  - an edge has no concept of "direction"
- In mathematics, an edge relation is usually considered irreflexive:
  - $\neg \exists a : (a,a) \in E$
  - engineers often relax this constraint (hence the brackets)

Directed Graphs

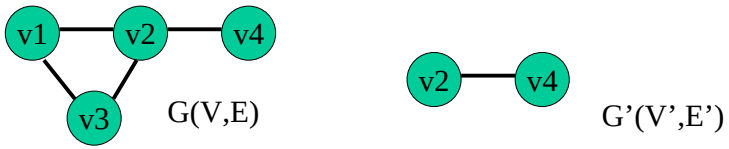
- Formal definition:
  - A *directed graph G* is a finite nonempty set V together with an [irreflexive] relation E on V
  - This time the concept of direction is implicit, as we *could* have $(a,b) \in E$ and $(b,a) \notin E$
- You may see directed graphs referred to as "digraphs"

Cliques

- A *complete graph* is a special type of graph where all possible edges are in the edge set

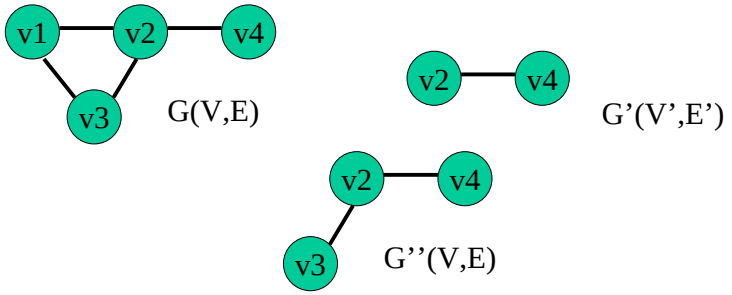


- A subgraph G'(V',E') of a graph G(V,E) is a graph whose vertex and edge sets obey
  - $V' \subseteq V$, $E' \subseteq E$



Cliques

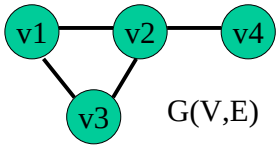
- A *clique* is a complete subgraph



- G' is a clique. G'' is not a clique (but it is a subgraph of G)

Clique Number

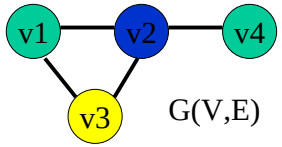
- The *clique number* $\omega(G)$ of a graph G is the size of the node set of its largest clique



- This graph has cliques with the following node subsets:
  - {v1}, {v2}, {v3}, {v4}, {v1,v2}, {v1,v3}, {v2,v3}, {v2,v4}, {v1,v2,v3}
- Its clique number is 3

Graph Colouring

- Graph colouring is the process of labelling each node of a graph such that no two connected nodes share the same label



- The graph above is coloured with three different colours
- Graph colouring can model many problems
  - e.g. colouring a conflict graph (Lecture 2) will result in a resource binding

A Colouring Algorithm

- A simple algorithm for colouring a graph is given below

```
Colour_Graph( G(V,E) )
begin
  foreach v ∈ V {
    c = 1;
    while ∃(v,v') ∈ E : v' has colour c
        c = c + 1;
    label v with colour c }
end
```

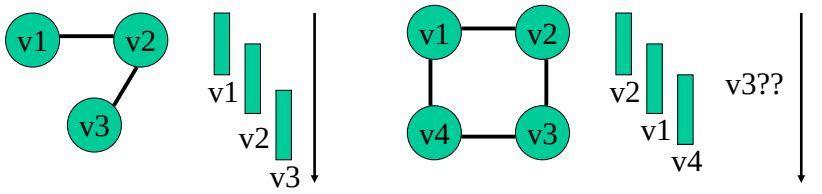
- This will always correctly colour a graph, but the number of distinct colours used depends on the order in which the nodes are visited

Chromatic Number

- The smallest number of colours with which it is possible to colour a graph G is called its chromatic number $\chi(G)$
- For a general graph, finding $\chi(G)$ is a "hard" problem
  - the algorithm presented does not guarantee a colouring with $\chi(G)$ colours
  - we'll be discussing "hard" problems next lecture
- In resource binding, the chromatic number tells us the minimum number of distinct resources required
- Since every node in a clique must be coloured differently to every other node in a clique,
  - $\omega(G) \leq \chi(G)$

Interval Graphs

- Luckily, not all graphs are "hard" to colour. One type of graph which is easy to colour with the minimum number of colours is an "interval graph"
- An *interval graph* is a graph whose vertices can be put in one-to-one correspondence with a set of intervals, such that two vertices are connected by an edge iff the corresponding intervals intersect



An interval graph NOT an interval graph

The Left Edge Algorithm

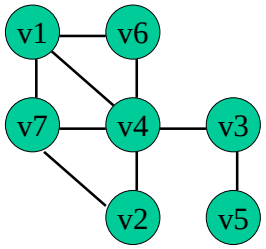
- The left-edge algorithm colours interval graphs optimally.
- Let us denote by l_i and r_i the left-most and right-most point of the interval corresponding to vertex v_i.

```
Left_Edge( G(V,E) )
begin
  sort nodes in ascending order of left edge – store in L
  c := 1;
  while( not all vertices have been coloured ) {
    r := 0;
    while( ∃ an element s in L with l_s > r ) {
      v_s := first node in L with l_s > r;
      r := r_s;
      label v_s with colour c
      L := L \ {v_s}; }
  c := c + 1; }
end
```

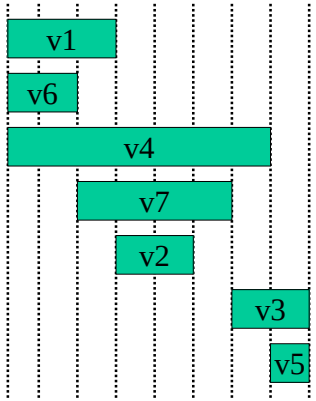

The Left Edge Algorithm

- Some set theory:
  - \ represents set subtraction
    - $X \setminus Y = \{ z : z \in X \wedge z \notin Y \}$
- The left edge algorithm tries to colour as many intervals as possible with one colour, before moving on the the next colour
- Left Edge was originally introduced to pack wire segments tightly on a VLSI layout. It is now used for many other purposes – particularly resource binding.

Left Edge - Example

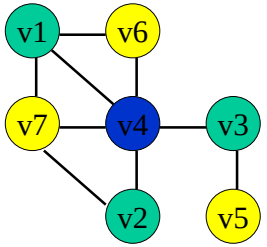


interval graph



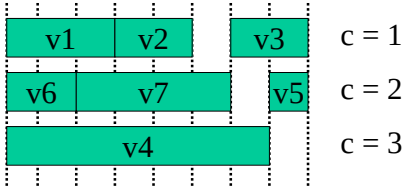
interval list *L*

Left Edge - Example



coloured graph

1 ↔
2 ↔
3 ↔



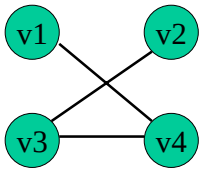
intervals packed into colours

Summary

- This lecture has covered
  - graphs and digraphs
  - cliques and clique number
  - colouring and chromatic number
  - interval graphs and the Left Edge algorithm
- Next lecture will examine the ideas behind designing “good” algorithms, and what it means for a problem to be “hard”

Suggested Problems

- For the graph below, apply the general colouring algorithm for the following two vertex orders. Compare and contrast your results. (*)
  - (a) (v1, v2, v3, v4)
  - (b) (v1, v4, v3, v2)
- By applying the left-edge algorithm, or otherwise, demonstrate that one of the two orders above results in an optimum colouring (*)



Algorithms and Intractability

- Part of our 4-lecture "theory break"
  - Graphs, cliques, and colouring
  - Algorithms and intractability
  - Linear programming and integer linear programming
  - Shortest and longest path algorithms
- This lecture covers
  - The definition of an "algorithm"
  - Polynomial-time and intractability
  - P and NP
  - Polynomial reduction, NP-completeness and NP-hardness

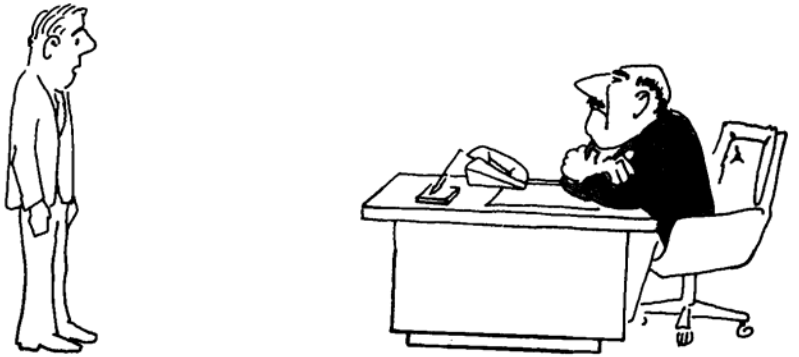
The Purpose of This Lecture

- Synthesis is all about writing algorithms to solve problems in digital design
- This lecture will consider some of the more theoretical aspects concerning
  - problems, algorithms, and complexity
- We will formalize what is meant by a "hard" problem
- You will **not** be required to prove the hardness of any unseen problems as part of this course
- You may be required to describe the ideas of hardness

Problems and Instances

- We have already discussed several problems and algorithms. We will now take a few minutes to formalize these concepts
- A *problem* is a general question to be answered, usually possessing several parameters, whose values are left unspecified
  - e.g. Can I schedule a DFG G(V,E) to complete within λ cycles using at most *n* multipliers?
- An *instance* of a problem is obtained by specifying particular values for all parameters
  - e.g. Can I schedule the DFG given in Lecture 1, slide 5, to complete within 10 cycles using at most 2 multipliers?

"Hard" Problems



"I can't find an efficient algorithm, I guess I'm just too dumb."

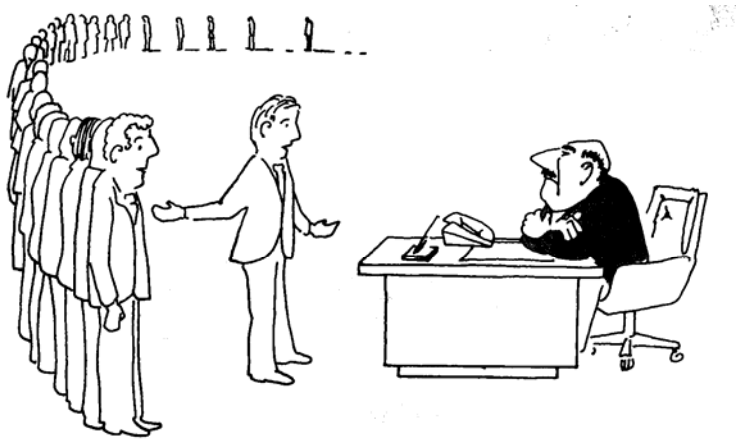
[Garey & Johnson 1979]

"Hard" Problems

"I can't find an efficient algorithm, because no such algorithm is possible!"

[Garey & Johnson 1979]

"Hard" Problems



"I can't find an efficient algorithm, but neither can all these famous people."

[Garey & Johnson 1979]

Algorithms and Efficiency

- An *algorithm* is a general step-by-step procedure for solving problems
- An algorithm is said to *solve* a problem Π if the algorithm can be applied to any instance of Π and is guaranteed to always produce a solution for that instance
- An *efficient* algorithm is one that solves the problem "quickly"
  - there are other factors such as memory usage, but we will ignore these

Complexity

- Usually, we can describe the worst-case performance of an algorithm as a function of the "size" *n* of the problem instance
- We generally are concerned with the "big picture" of how performance scales with size (especially for large sizes), rather than specific execution times
- The Big-Oh notation allows us to express this behaviour
  - $O(n)$, $O(n^2)$, $O(e^n)$
- An algorithm is O(f(n)) if its worst case performance is bounded by k f(n) for large n

Complexity

- Example: A (good) algorithm to add n numbers will be O(n)
- Example: An algorithm to sort n numbers in order. You may be familiar with
  - quicksort: O(n^2)
  - heapsort: O(n log n)
- Example: An algorithm which considers all possible k-colourings that a graph could have would be O(k^n)

Polynomial vs Exponential Time

- A polynomial-time algorithm is one which has O(p(n)) for some polynomial p(·).
- An exponential-time algorithm is any algorithm which is not polynomial-time.
- Clearly for large n, exponential-time algorithms take much longer than polynomial-time algorithms
  - the main distinction is thus: "is this algorithm exponential (bad) or polynomial (good)?"
  - the order of the polynomial is of secondary concern
- All problems which can be solved by polynomial algorithms are said to belong to the class *P*

Nondeterministic Polynomial Time

- To complicate matters, computer scientists have come up with another class, *NP* (nondeterministic polynomial).
- A problem is in *NP* if a *solution* to the problem can be *checked* in polynomial time
  - this doesn't mean it has to be solvable in polynomial time
- Example:
  - scheduling G(V,E) in time λ given resource constraints may or may not be solvable in polynomial time
  - it is clear that given a schedule, we could check in polynomial time that it is a valid schedule and it completes within λ cycles

Want to Earn Some Money?

- The problem "does *P* = *NP*?" is unsolved
- If you solve it you will
  - be famous
  - win $2,000,000 from www.claymath.org
- …but don't let it distract you from your degree!

Polynomial Reduction

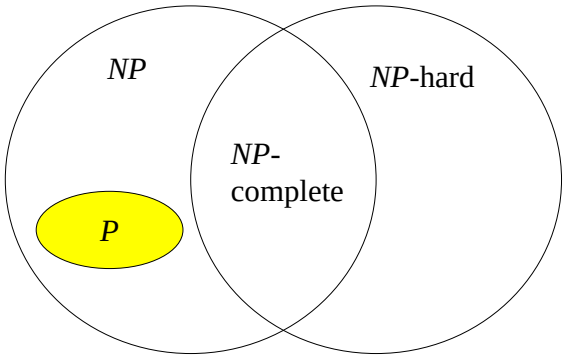
- Many interesting and difficult problems (like scheduling) are in *NP* but we don't know whether they're in *P*
- Since it is generally hard to prove that a given problem is not in *P*, we instead concentrate on proving that its "at least as hard" as a known hard problem
- If we can transform any instance of a hard problem Π^H into an instance of our problem Π, and that transformation can be done in polynomial time, then
  - if we can solve Π, we can solve $\Pi^H \Rightarrow \Pi$ is also hard!

NP-completeness & NP-hardness

- There are some problems which are in *NP* and which are known to be at least as hard as any other problem in *NP*.
  - these are called *NP*-complete
- *NP*-complete problems are of particular interest, as if a solution to any *NP*-complete problem can be found in polynomial time then *P* = *NP*
- A problem which is at least as hard as an *NP*-complete problem is called *NP*-hard
  - this is our formal definition: for "hard problem" read "NP-hard problem"

A Hierarchy of Problems

- Assuming P ≠ NP, this is how our "world of problems" looks



Proving Hardness

- Proving *NP*-hardness requires two stages
  - pick a known *NP*-hard problem
  - demonstrate a transformation from this problem to your problem
- There are some *NP*-complete problems which form the basis of many proofs. We will look at one: Partition
- Partition: Given a finite set *A* and a measure $s(a) \in Z^+$ for each $a \in A$, is there a subset $A' \subseteq A$ such that the following equation holds?

$$\sum_{a \in A'} s(a) = \sum_{a \in A - A'} s(a)$$

Proving Hardness

- An example instance of "partition":
  - A = {v1, v2, v3} with s(v1) = 1, s(v2) = 2, s(v3) = 1
  - for this instance, the answer is clearly "yes":
    - A' = {v2} or A' = {v1, v3}

Example: Scheduling is *NP*-hard

- To finish off, we'll prove the *NP*-hardness of an example problem (a simple form of scheduling)
- Our simple scheduling problem has no data dependencies and only one type of operation
- Remember that you won't be asked to do such a proof for an unseen problem, but this proof has been included
  - for completeness
  - to give a more "practical" end to a highly theoretical lecture
  - to justify past and future comments about scheduling being a "hard" task to perform

Scheduling is *NP*-hard

- Let's start by defining our problem:
  - given a finite set A of operations, a latency $d(a) \in Z^+$ for each $a \in A$, a number $m \in Z^+$ of resources, and a deadline $\lambda \in Z^+$
  - is there a schedule such that all operations complete within the deadline and no more than m resources are used?

Scheduling is *NP*-hard

- Let us rephrase the question:
  - is there a partition $A = A_1 \cup A_2 \cup \ldots \cup A_m$ of A into m disjoint subsets such that

$$\max_{1 \le i \le m} \left\{ \sum_{a \in A_i} d(a) \right\} \le \lambda$$

  - A_i represents the set of operations assigned to processor i, and no two operations can be executed at the same time on a single resource

Scheduling is *NP*-hard

- Let's consider a special case of our problem, for m=2 and $\lambda = \frac{1}{2}\sum_{a \in A} d(a)$
- Then the problem reduces to:
  - given a finite set A, and a value $d(a) \in Z^+$ for each $a \in A$
  - is there a partition into 2 disjoint subsets A' and A – A' such that

$$\max\left\{\sum_{a \in A'} d(a), \sum_{a \in A - A'} d(a)\right\} \leq \frac{1}{2}\sum_{a \in A} d(a)$$

Scheduling is *NP*-hard

- Rewriting, we require

$$\frac{1}{2}\max\left\{\sum_{a \in A'} d(a) - \sum_{a \in A - A'} d(a), \sum_{a \in A - A'} d(a) - \sum_{a \in A'} d(a)\right\} \leq 0$$

- But for any k, $\max(k,-k) \leq 0 \Rightarrow k = 0$, so we require

$$\sum_{a \in A'} d(a) = \sum_{a \in A - A'} d(a)$$

- But this is the "partition" problem. So "partition" is a special case of our problem and hence our problem is NP-hard

Summary

- This lecture has covered
  - The definition of an "algorithm"
  - Polynomial-time and intractability
  - P and NP
  - Polynomial reduction, NP-completeness and NP-hardness
- Next lecture we will look at the (NP-hard!) problem of Integer Linear Programming (ILP) and how we can use ILP solving software to help us optimize our hardware

[Integer] Linear Programming

- Part of our 4-lecture "theory break"
  - Graphs, cliques, and colouring
  - Algorithms and intractability
  - Linear programming and integer linear programming
  - Shortest and longest path algorithms
- This lecture covers
  - Mathematical programming, integer / mixed-integer programming, and linear programming
  - Slack variables
  - Application example: Capital budgeting

Mathematical Programming

- Mathematical "programming" is the name given to the branch of mathematics that considers the following optimization problem:

$$\max f(x), \quad x \in S \subseteq R^n$$

- Here R^n represents the set of n-dimensional vectors of real numbers, and f is a real-valued function defined on S. S is the *constraint set* and f is the *objective function*.
- By choosing f and S appropriately, we can model a wide variety of real-life problems in this way.

Feasibility and Optimality

- Any $x \in S$ is called a *feasible solution*
- If the there is an $x^o \in S$ such that

  $f(x) \leq f(x^o)$ for all $x \in S$

  then x^o is called an *optimal solution*

- The aim is to find an optimal solution for a given f and S

Integer Programming

- An integer programming problem is one where S is restricted to have only integer values

$$S \subseteq Z^n \subseteq R^n$$

- A mixed integer programming problem is one where some elements of S are restricted to integers
- Integer programming problems are typically harder than the equivalent real problem. You can gain an intuition why by considering the following problems
  - find the value of x minimizing $\cos(x/5)$
    - 5π
  - find the integer value of x minimizing $\cos(x/5)$
    - round(5π) ? round ($5\pi + 10\pi$) ? …

Linear Programming

- Problems where *f* and *S* are restricted to linear form are of particular interest

  $f(x) = c^T x, \qquad S = \{ x \mid Ax = b, x \geq 0 \}$

  – *c* is an *n* x 1 vector, *A* is an *m* x *n* matrix and *b* is an *m* x 1 vector
- Imposing the linearity constraints restricts the domain of problems, but allows us to use known solution techniques
- For general x, these problems can be solved exactly (e.g. Simplex technique). For integer x, the problem is *NP*-complete.

Why Are We Interested?

- We are interested in expressing problems as integer or mixed integer linear programs because
  – it provides a way to formalize the problem
  – we can apply known general techniques to solve the problem
  – lots of software exists to solve MILPs (e.g. lp_solve, available free from the web)
  – I will be introducing ILP formulations for scheduling and resource binding in later lectures

Modelling Complex Problems

- At first glance, linear constraints may seem very restrictive – this is not necessarily the case, if you build your model carefully.
- Here are three types of constraint that could be useful in synthesis
  – inequalities (e.g. x1 + x2 ≤ b1, rather than x1 + x2 = b1)
  – dichotomy (e.g. x1 + x2 ≤ b1 OR x3 + x4 ≤ b2)
  – conditionals (e.g. x1 + x2 ≤ b1 ⇒ x3 + x4 ≤ b2)
- We will only be considering the first in this brief introduction. If you wish to use the others,
  – R.S. Garfinkel and G.L. Nemhauser, “Integer Programming”, Wiley and Sons, 1972

Inequality

- Inequality constraints can easily be introduced by adding an extra variable
- For example, consider the program:

  max $2x_1 + 3x_2$ subject to $x_1 + x_2 \leq 10$

  This is the same as

  max $2x_1 + 3x_2$ subject to $x_1 + x_2 + x_3 = 10$
- For “≥”, we would insert ($-x_3$) into the constraint
- The extra variable is called a slack variable – it does not appear in the objective function
- Because this is so straight-forward, many ILP solving programs allow you to express constraints with inequality directly. From now on, we will use inequalities freely without considering slack variables explicitly

Example: Capital Budgeting

- From Garfinkel and Nemhauser (1972):
  - A firm has n projects that it would like to undertake, but due to budget limitations, not all can be selected. In particular, project j has a value of c_j, and requires an investment of a_{ij} in the time period i, i =1,…,m. The capital available in time period i is b_i.
  - Problem: Maximize the total value, subject to budget constraints

Example: Capital Budgeting

- Let's introduce a set of variables x_j, which we interpret as:
  - $x_j = 1 \Rightarrow$ project j is selected
  - $x_j = 0 \Rightarrow$ project j is not selected
- Then the objective function can be formulated as

$$\sum_{j=1}^{n} c_j x_j$$

- The constraints are

$$\sum_{j=1}^{n} a_{ij} x_j \le b_i, i = 1,...,m; \quad x_j \le 1, \; j = 1,...,n$$

Summary

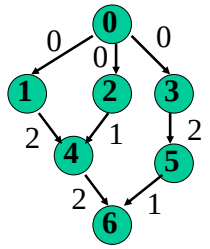
- This lecture has covered
  - Mathematical programming, integer / mixed-integer programming, and linear programming
  - Slack variables
  - Application example: Capital budgeting
- Next lecture (the last in our "theory break"), looks at finding the shortest and longest path through a graph

Path Problems and Algorithms

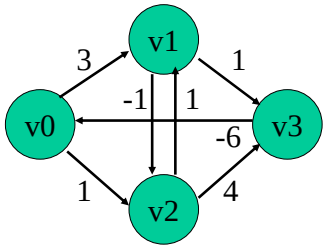
- Part of our 4-lecture "theory break"
  - Graphs, cliques, and colouring
  - Algorithms and intractability
  - Linear programming and integer linear programming
  - Shortest and longest path algorithms
- This lecture covers
  - Edge-weighted graphs, shortest and longest path problems
  - Longest path through a DAG
  - Longest path through a general graph: Liao-Wong
  - Longest path as a LP

Edge Weighted Graphs

- An edge-weighted graph is a graph $G(V,E)$ together with a weighting function w: $E \rightarrow$ R
- We can represent this graphically by annotating each edge $e \in E$ with its weight $w(e)$



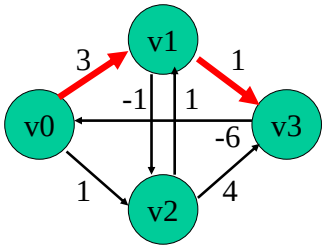
An edge weighted DAG



An edge-weighted graph with cycles

Shortest and Longest Path

- A *path* through a graph is an alternating sequence of vertices and edges



- A path between vertices v0 and v3, with total edge weight 3+1 = 4 has been highlighted

Shortest and Longest Path

- The *longest path* problem is to find a path of maximum total weight between a given "source" vertex and any other vertex in the graph
  - the shortest path problem is defined similarly
  - we will consider only longest path problems – shortest path can then be achieved by inverting all weights
    $w'(e) = -w(e)$
- Bellman's equations define the total weight of any vertex v

$$s_v = \max_{(u,v)\in E} (s_u + w(u,v))$$

Longest Path Through a DAG

- The longest path through a DAG is an easier problem than the equivalent for a general graph
- This is because we can find an order of nodes to visit such that the right-hand side of each Bellman's equation is known
- For our example DAG, let's choose vertex 0 as our source. Then $s_0 = 0$. If we now proceed to apply Bellman's equations in the order (s_1, s_2, s_3, s_4, s_5, s_6), we can determine the total weight for each node
  - $s_1 = 0$, $s_2 = 0$, $s_3 = 0$, $s_4 = 2$, $s_5 = 2$, $s_6 = 4$
- Note that this would not work with an arbitrary order. We must calculate s_v before s_u for all $(v,u) \in E$
- For a graph with cycles, it is not possible to find such an order

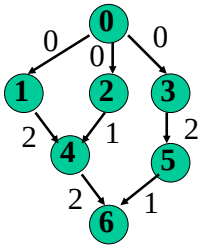
DAG Algorithm

- Below is one possible algorithm (apologies to the recursion-phobics)

```
Algorithm DAG_Longest_Path( G(V,E), source )
   Set s_source = 0;
   foreach v ∈ V
        Find_DAG_Path( G(V,E), v );
end DAG_Longest_Path

Algorithm Find_DAG_Path( G(V,E), v )
   if already know s_v
        return;
   else
        foreach (u,v) ∈ E
                Find_DAG_Path( G(V,E), u )
        Apply Bellman's equation to find s_v
end Find_DAG_Path
```


DAG example



- Let's assume the vertices are stored in V in an arbitrary order – say (4,1,2,3,5,0,6)
- A call to DAG_Longest_Path(G(V,E), 0) will set $s_0 = 0$, and then follow the following execution profile

1. Find_DAG_Path(G(V,E), 4)
   1. Find_DAG_Path(G(V,E), 1)
      1. Find_DAG_Path(G(V,E), 0)
      2. Calculate $s_1 = 0$
   2. Find_DAG_Path(G(V,E), 2)
      1. Find_DAG_Path(G(V,E), 0)
      2. Calculate $s_2 = 0$
   3. Calculate $s_4 = 2$

DAG Example

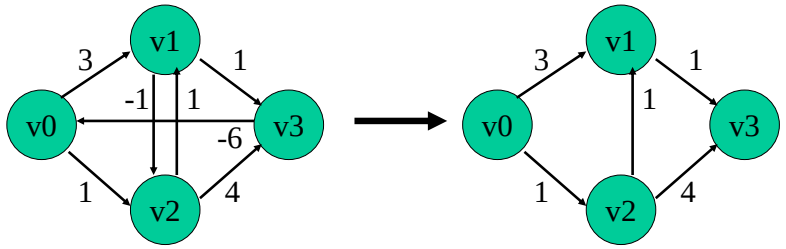
2. Find_DAG_Path(G(V,E), 1)
3. Find_DAG_Path(G(V,E), 2)
4. Find_DAG_Path(G(V,E), 3)
   1. Find_DAG_Path(G(V,E), 0)
   2. Calculate $s_3 = 0$
5. Find_DAG_Path(G(V,E), 5)
   1. Find_DAG_Path(G(V,E), 3)
   2. Calculate $s_5 = 2$
6. Find_DAG_Path(G(V,E), 0)
7. Find_DAG_Path(G(V,E), 6)
   1. Find_DAG_Path(G(V,E), 4)
   2. Find_DAG_Path(G(V,E), 5)
   3. Calculate $s_6 = 4$

General Longest Path

- Many algorithms to find the longest path of general graphs have been proposed in the literature
- We will consider Liao and Wong's algorithm as it is very efficient for cases where the graph edge set $E \cup F$ can be partitioned into a "forward" edge set E and a feedback edge set F where $G(V,E)$ is a DAG and $|E| >> |F|$
  - this is often the case with graphs arising in synthesis – we will consider some of these in future lectures

Example Edge Set Partition

- Consider our example graph. If we remove the edges labelled "-1" and "-6", we obtain a DAG



- The remaining edges form the set E, whereas the two we removed form the set F

General Algorithm

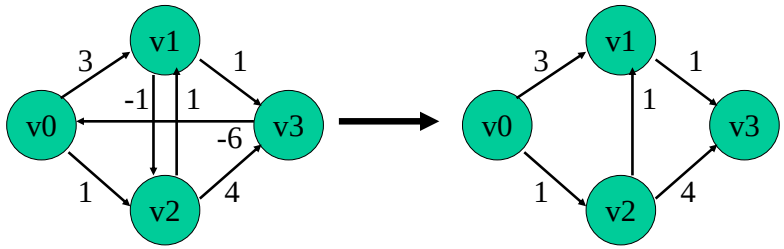
```
Algorithm Liao_Wong( G(V,E ∪ F), source )
  for j = 1 to | F | + 1 {
      DAG_Longest_Path( G(V,E), source);
      flag = TRUE;
      foreach (u,v) in F {
          if s_v < s_u + w(u,v) {
              flag = FALSE;
              E = E ∪ { (source, v) };
              w(source,v) = s_u + w(u,v);
          }
      }
      if( flag ) return;
  }
end Liao_Wong
```


Algorithm Description

- Liao Wong first applies the DAG algorithm on the forward edges only. If no feedback edge provides a longer path alternative, the algorithm terminates
- If a longer path alternative is found, the algorithm models this as an extra forward edge directly from the source
- This process is repeated, until no more changes to the edge set are necessary
- It is provable that if the graph contains no cycles where the sum of weights around the cycle is positive, the outer loop need only be executed at most $|F|+1$ times.

General Example

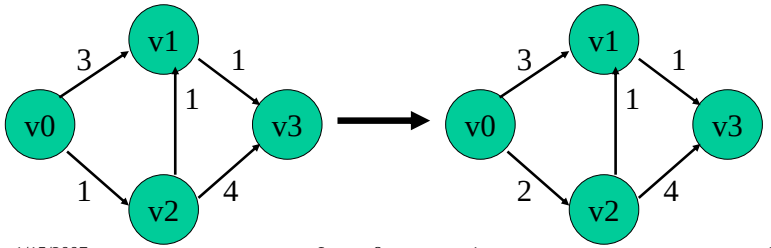
- Let us examine our example graph



- Performing our initial DAG longest path, with v0 as the source, leads to
  - $s_{v0} = 0$, $s_{v1} = 3$, $s_{v2} = 1$, $s_{v3} = 5$

General Example

- We now examine each of the feedback edges in turn
  - for edge (v3,v0), $s_{v0} \geq s_{v3} - 6$ $(0 \geq -1)$, so no change needs to be made
  - for edge (v1,v2), $s_{v2} < s_{v1} - 1$ $(1 < 2)$, so we must insert a new forward edge (v0,v2) with weight 2 [in this example, (v0,v2) is already in E, so we just modify the weight]



General Example

- Calculating the longest path on the modified DAG leads to
  - $s_{v0} = 0$, $s_{v1} = 3$, $s_{v2} = 2$, $s_{v3} = 6$
- Examining each feedback edge in turn
  - for edge (v3,v0), $s_{v0} \geq s_{v3} - 6$ $(0 \geq 0)$, so no change needs to be made
  - for edge (v1,v2), $s_{v2} \geq s_{v1} - 1$ $(2 \geq 2)$, so no change needs to be made
- At this point, the algorithm terminates as no changes are necessary

Longest Path as a LP

- To keep up our interest in LP, let's formulate the longest path problem as a LP
- Let's revisit Bellman's equations:

$$s_v = \max_{(u,v)\in E} (s_u + w(u,v))$$

- A necessary condition for satisfaction is:

$$\forall (u,v) \in E, \quad s_v \geq s_u + w(u,v) \quad (*)$$

- The minimum values of s_v that satisfy (*) are the solutions to Bellman's equations

Longest Path as a LP

- We can write this as:

  minimize $\sum_{v \in V} s_v$ subject to:

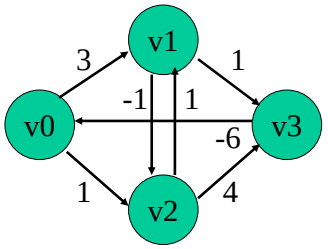
  $$s_v \geq s_u + w(u,v) \text{ for all } (u,v) \in E$$

  and $s_{source} = 0$

- This is a standard LP formulation (c.f. lecture 7), which can easily be cast in matrix notation A**x** ≥ **b** if required

LP Example

- For our general graph example, the LP objective function and constraints are given below



- minimize $s_0 + s_1 + s_2 + s_3$

  subject to:

  s1 ≥ s0 + 3; s1 ≥ s2 + 1

  s2 ≥ s0 + 1; s2 ≥ s1 – 1

  s3 ≥ s1 + 1; s3 ≥ s2 + 4

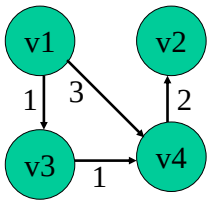
  s0 ≥ s3 – 6; s0 = 0

Some Applications

- Longest and shortest path problems have many real-life applications, including
  - Circuits: Determining the critical path in a circuit, and hence the performance of that circuit
  - Transport: Finding the (shortest/cheapest/least fuel) route between two places
  - Networking and Comms: Shortest path through a network

Worked Example

- Consider the edge-weighted graph shown below



- (a) determine the longest path from v1 to all other vertices in the graph
- (b) if an edge (v2,v3) with weight w(v2,v3) = -4 were added, how would this affect the longest paths?

Worked Example

(a) It should be easy to see that $s_{v1} = 0$, $s_{v2} = 5$, $s_{v3} = 1$, $s_{v4} = 3$ (verify by applying Bellman's equations in the order (v1, v3, v4, v2)

(b) This edge would close a cycle {v3, v4, v2}. We therefore use Liao-Wong to determine whether any change has occurred to the longest paths

Examining the feedback edge (v2,v3), we see that $s_{v3} \geq s_{v2} - 4$ ($1 \geq 5 - 4$) and therefore the extra edge has not affected the longest paths

Summary

- This lecture has covered
  - Edge-weighted graphs, shortest and longest path problems
  - Longest path through a DAG
  - Longest path through a general graph: Liao-Wong
  - Longest path as a LP
- This brings us to the end of our "theory break". Next lecture will look at scheduling digital circuits.

Suggested Problems

- Find the *shortest* path through the DAG used as an example in this lecture (*)
- Try to apply the Liao-Wong algorithm to find the *shortest* path through the cyclic graph example. Does it work? If not, why not? (***)
- In the cyclic example, change the weight of edge (v3,v0) to –4. Now apply Liao-Wong to the *shortest* path problem. (*)