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

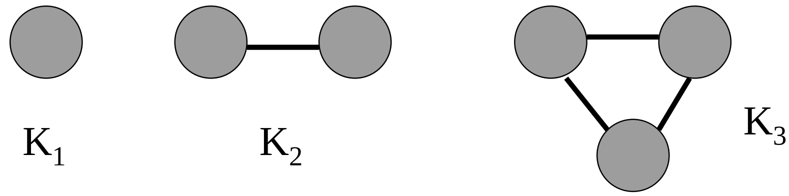
- Formal Definition:
  - A *graph G* is a finite nonempty set *V* together with an [irreflexive], symmetric relation *E* on *V*
- The relation *E* relates vertices to other vertices and is known as the edge relation, or “edge set”
- If relation *E* is symmetric, it means that
    - $(a,b)\in E \Rightarrow (b,a)\in E$
    - 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

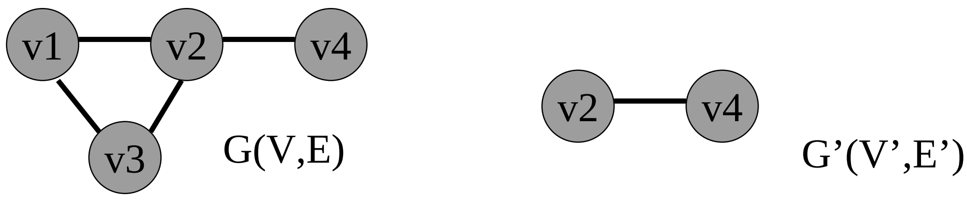
- Formal definition:
  – A *directed graph G* is a finite nonempty set V together with an [irreflexive] relation E on V
  – This time the concept of direction is implicit, as we *could* have $(a,b) \in E$ and $(b,a) \notin E$
- You may see directed graphs referred to as “digraphs”

Cliques

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

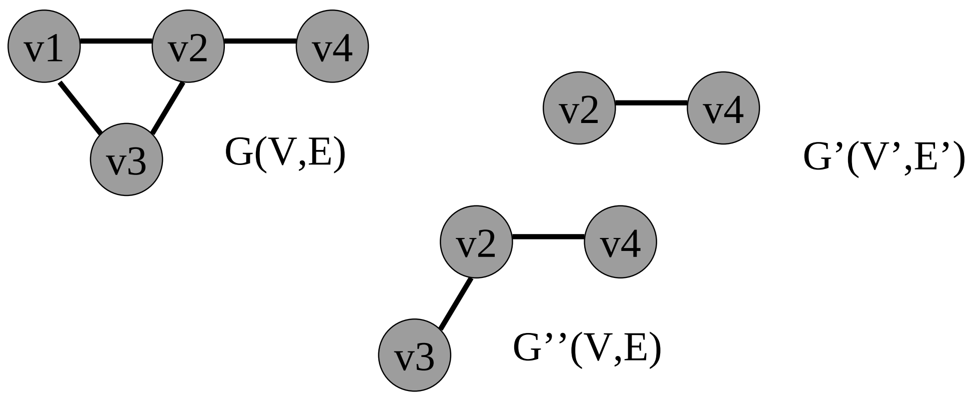


- A subgraph G’(V’,E’) of a graph G(V,E) is a graph whose vertex and edge sets obey
  - $V' \subseteq V$, $E' \subseteq E$



Cliques

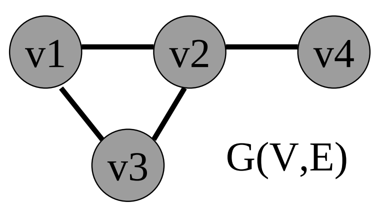
- A *clique* is a complete subgraph



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

Clique Number

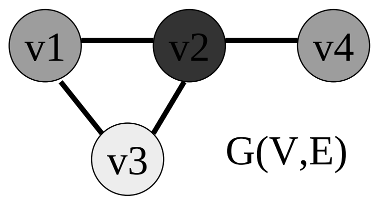
- The *clique number* $\omega(G)$ of a graph *G* is the size of the node set of its largest clique



- 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*** $\in$ ***V*** **{**
    c **= 1;**
    while $\exists$**(*v*,*v*')** $\in$ ***E*** **: *v*' has colour *c***
      c **=** ***c*** **+ 1;**
    label *v* with colour *c* }
end

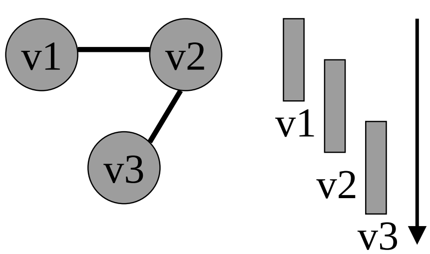
- This will always correctly colour a graph, but the number of distinct colours used depends on the order in which the nodes are visited

Chromatic Number

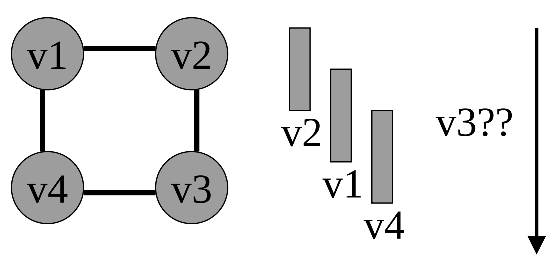
- The smallest number of colours with which it is possible to colour a graph *G* is called its chromatic number $\chi(G)$
- For a general graph, finding $\chi(G)$ is a "hard" problem
  – the algorithm presented does not guarantee a colouring with $\chi(G)$ colours
  – we'll be discussing "hard" problems 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

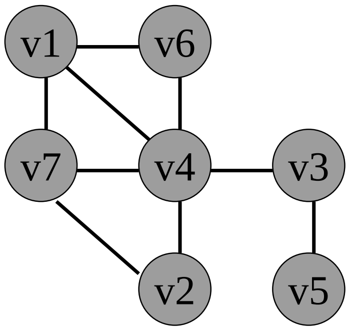
- The left-edge algorithm colours interval graphs optimally.
- Let us denote by l_i and r_i the left-most and right-most point of the interval corresponding to vertex v_i.

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;**
    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

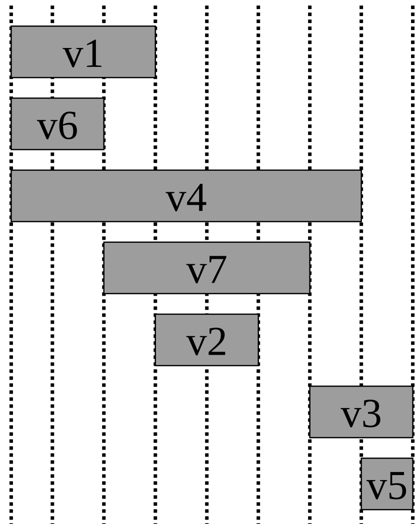
The Left Edge Algorithm

- 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

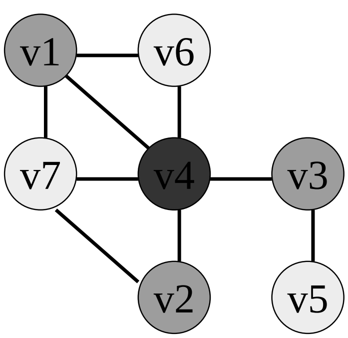


interval graph

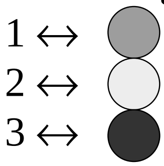


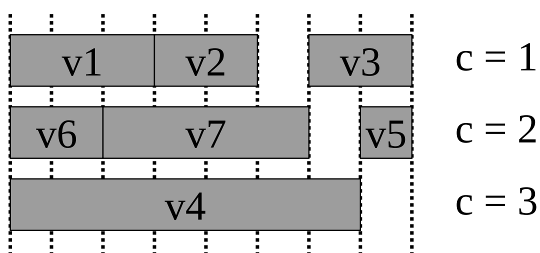
interval list *L*

Left Edge - Example



coloured graph





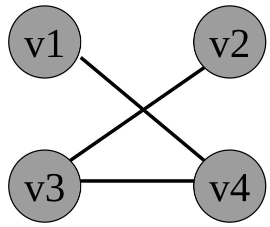
intervals packed into colours

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

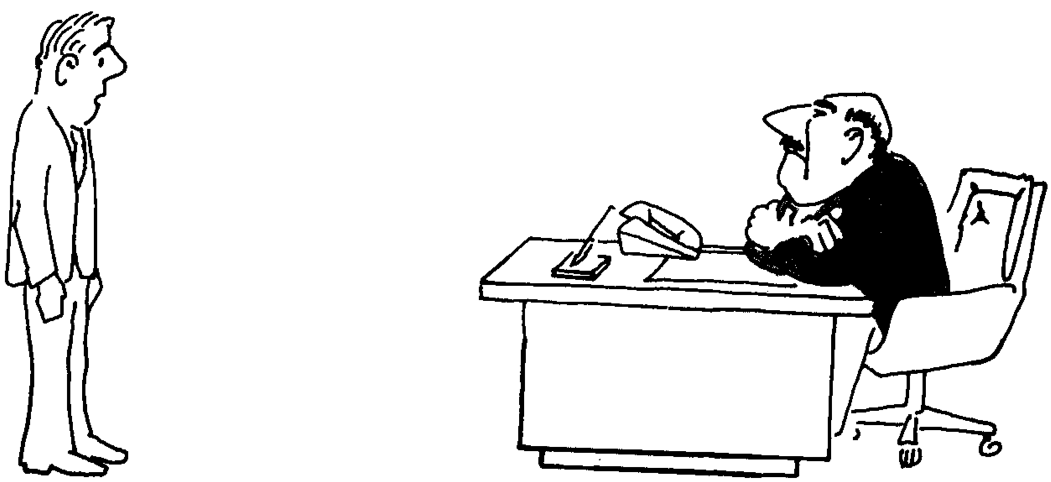
The Purpose of This Lecture

- 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 problem as part of this course
  - don't go to sleep!
- 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?

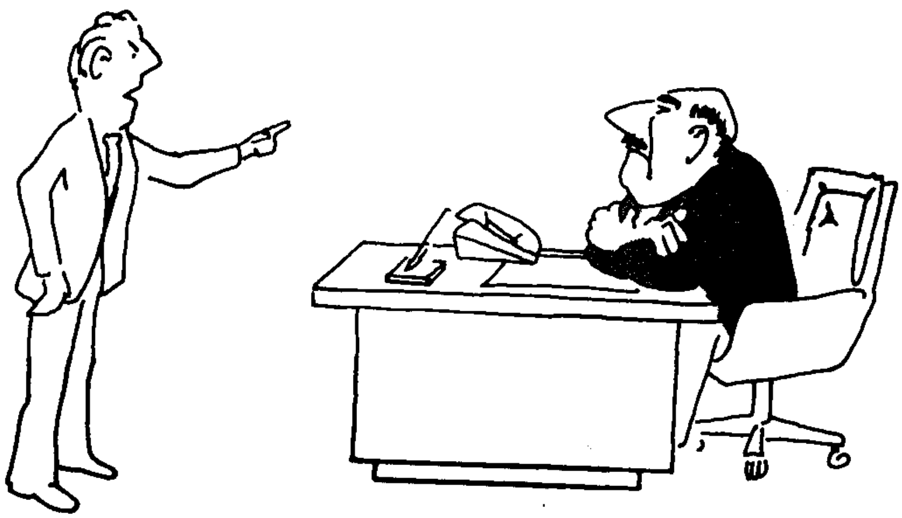
"Hard" Problems



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

[Garey & Johnson 1979]

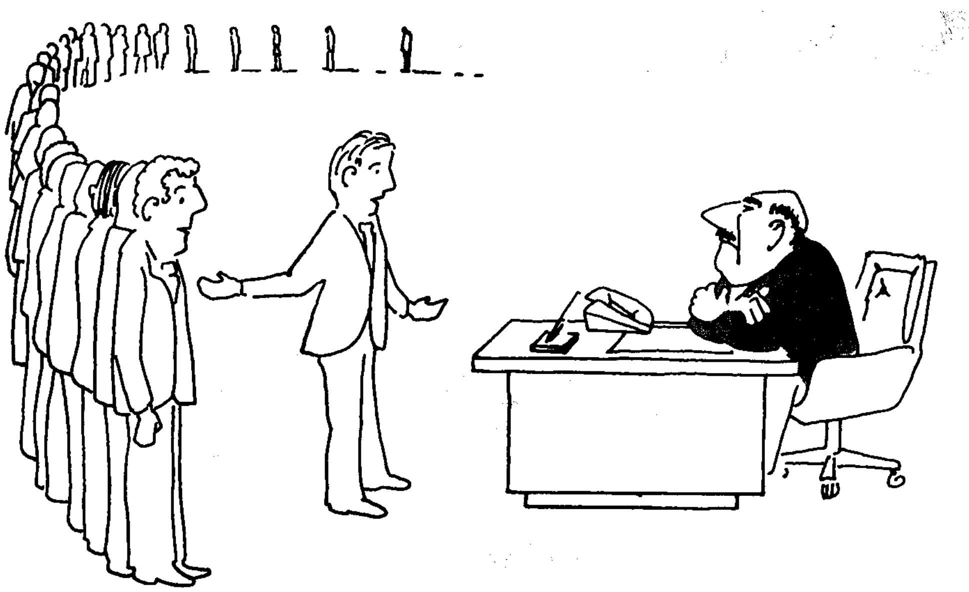
“Hard” Problems



“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 colourings a graph could have would be $O(e^n)$

Polynomial vs Exponential Time

- A polynomial-time algorithm is one which has $O(\ p(n)\ )$ for some polynomial $p(\cdot)$.
- 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

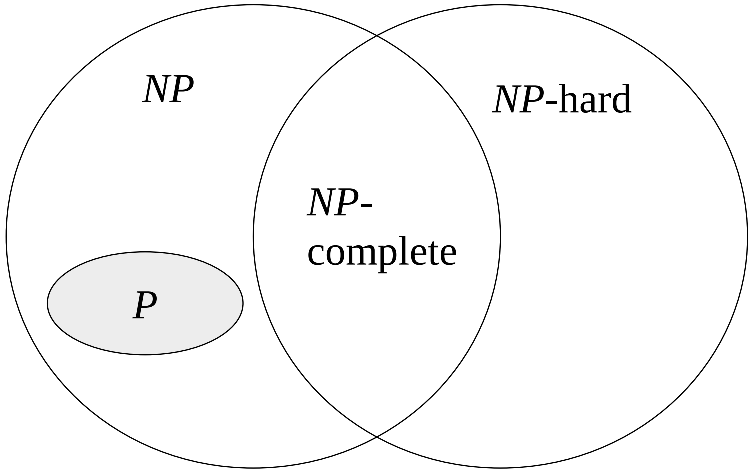
- Many interesting and difficult problems (like scheduling) are in *NP* but we don't know whether they're in *P*
- Since it is generally hard to prove that a given problem is not in *P*, we instead concentrate on proving that its "at least as hard" as a known hard problem
- If we can transform any instance 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, 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

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

Scheduling is *NP*-hard

- Rewriting, we require

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

- But for any *k*, max(*k*,-*k*) ≤ 0 ⇒ *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

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

  – 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. $x1 + x2 \leq b1$, rather than $x1 + x2 = b1$)
  - dichotomy (e.g. $x1 + x2 \leq b1$ OR $x3 + x4 \leq b2$)
  - conditionals (e.g. $x1 + x2 \leq b1 \Rightarrow x3 + x4 \leq b2$)
- 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

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

Summary

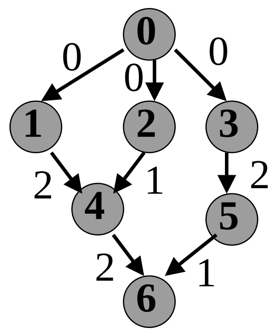
- This lecture has covered
  - Mathematical programming, integer / mixed-integer programming, and linear programming
  - Slack variables
  - Application example: Capital budgeting
- Next lecture (the last in our “theory break”), looks at finding the shortest and longest path through a graph

Path Problems and Algorithms

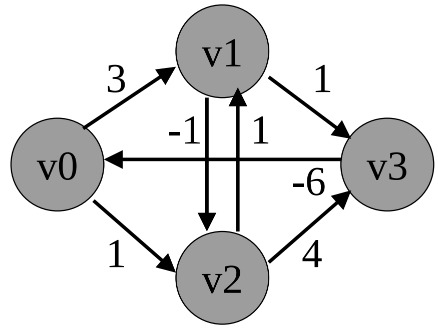
- Part of our 4-lecture “theory break”
  - Graphs, cliques, and colouring
  - Algorithms and intractability
  - Linear programming and integer linear programming
  - Shortest and longest path algorithms
- This lecture covers
  - Edge-weighted graphs, shortest and longest path problems
  - Longest path 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 $w(e)$



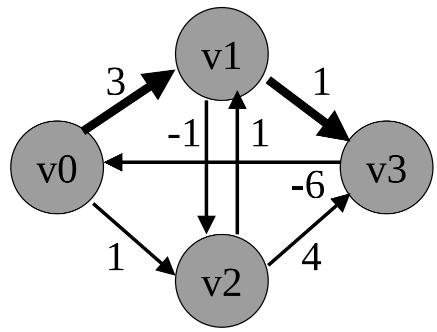
An edge weighted DAG



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

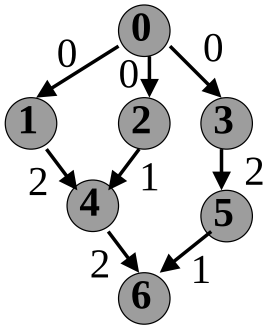
DAG Algorithm

- 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

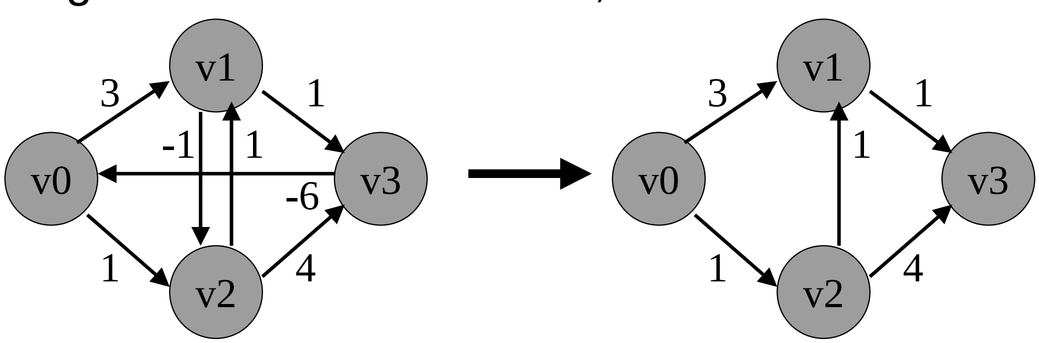
2. **Find_DAG_Path(G(V,E), 1)**
3. **Find_DAG_Path(G(V,E), 2)**
4. **Find_DAG_Path(G(V,E), 3)**
   1. **Find_DAG_Path(G(V,E), 0)**
   2. **Calculate s_3 = 0**
5. **Find_DAG_Path(G(V,E), 5)**
   1. **Find_DAG_Path(G(V,E), 3)**
   2. **Calculate s_5 = 2**
6. **Find_DAG_Path(G(V,E), 0)**
7. **Find_DAG_Path(G(V,E),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

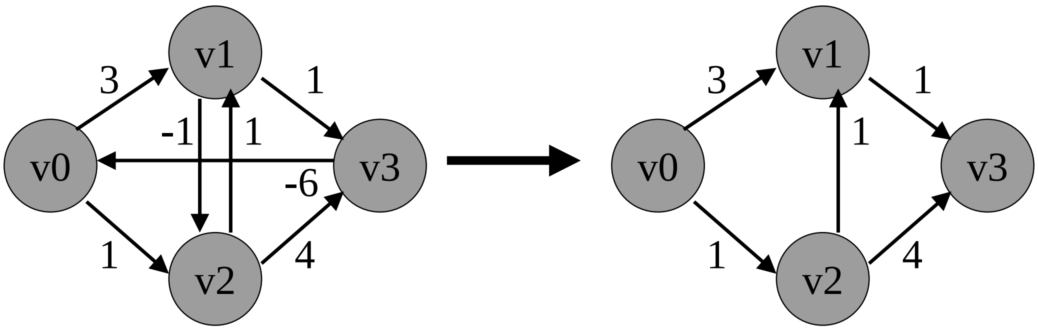
```
Algorithm Liao_Wong( G(V,E ∪ F), source )
   for j = 1 to | F | + 1 {
         foreach v in V
               s_v = DAG_Longest_Path( G(V,E), source);
         flag = TRUE;
         foreach (u,v) in F {
               if s_v < s_u + w(u,v) {
                    flag = FALSE;
                    E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

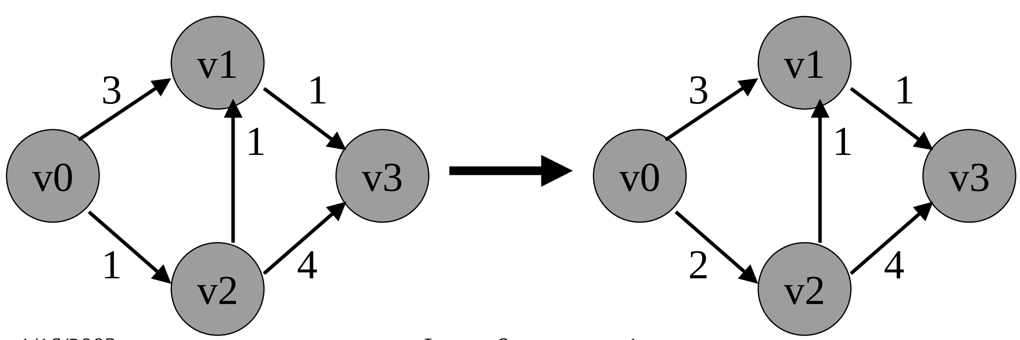
- Let us examine our example graph



- 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:

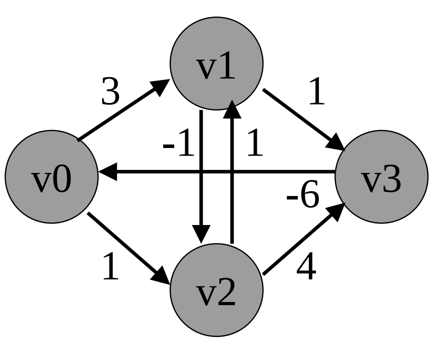
  $$s_v \geq s_u + w(u,v) \text{ for all } (u,v) \in E$$

  $$\text{and } s_{source} = 0$$

- 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

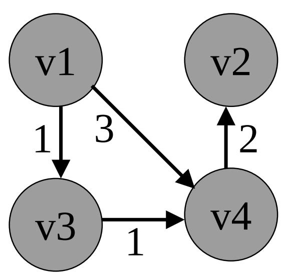
  s0 ≥ s3 – 6; s0 = 0

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
  - Shortest path through a DAG
  - Shortest path through a general graph: Liao-Wong
  - Shortest path as a LP
- This brings us to the end of our "theory break". Next lecture will look at pipelining 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. (*)

ASAP and ALAP scheduling

- We're now entering the final portion of the course
  - Scheduling algorithms
  - Resource sharing algorithms
  - Module selection
  - Perspectives for the future
- This lecture covers
  - The ASAP scheduling algorithm
  - The ALAP scheduling algorithm and operation slack
  - Introducing timing constraints into schedules

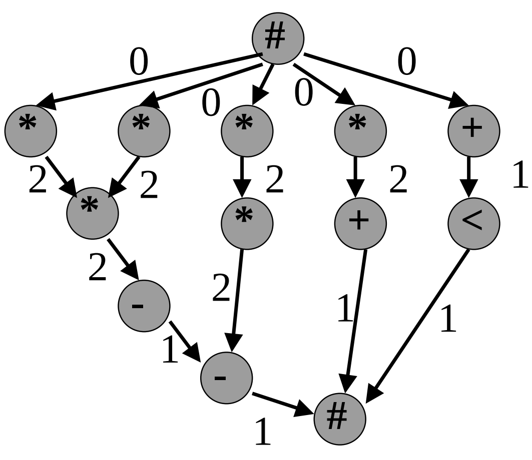
ASAP Scheduling

- The simplest type of scheduling occurs when we wish to optimize the overall latency of the computation and do not care about the number of resources required
- This can be achieved by simply starting each operation in a CDFG as soon as its predecessors have completed
- This strategy gives rise to the name ASAP for "As Soon As Possible"

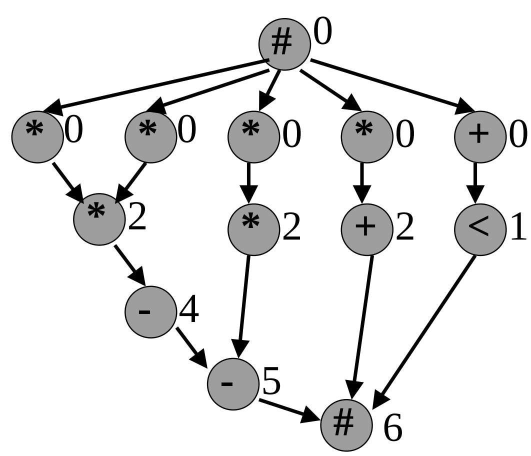
ASAP Scheduling

- Let’s label each edge in the CDFG with the latency of the node producing that edge
- Then scheduling under ASAP is equivalent to finding the longest path between each operation and the source node
- Since a CDFG is a DAG, we can use the DAG longest path algorithm presented in Lecture 8
- Consider the original example from Lecture 1, and assume that multiplication takes two cycles, whereas addition and comparison take one cycle

ASAP Scheduling



Edge weighted CDFG



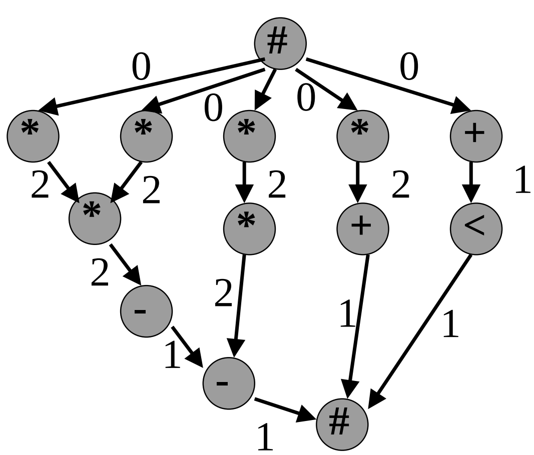
Scheduled start times

- Applying the DFG algorithm to finding the longest path between the start and end nodes leads to the scheduled start times on the right-hand diagram

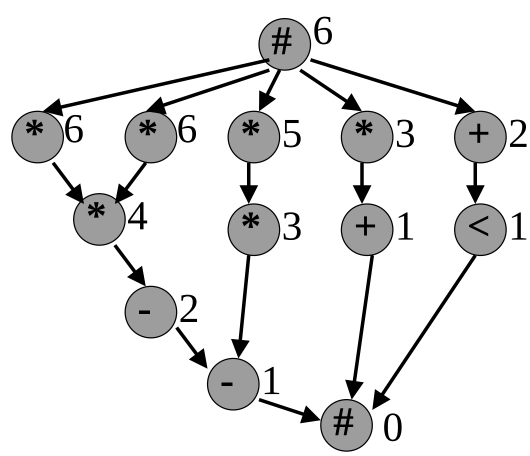
ALAP Scheduling

- The ASAP algorithm schedules each operation at the earliest opportunity. Given an overall latency constraint, it is equally possible to schedule operations at the latest opportunity.
- This leads to the concept of As-Late-As-Possible (ALAP) scheduling.
- ALAP scheduling can be performed by seeking the longest path between each operation and the end or “sink” node.
- We will re-examine the example, under the same delay assumptions, with an overall latency constraint of 6 clock cycles.

ALAP Scheduling



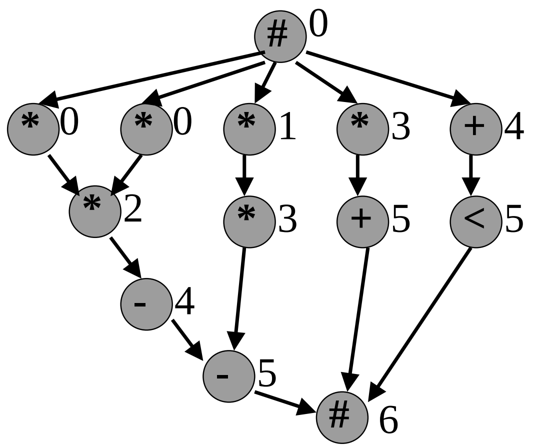
Edge-weighted CDFG



Longest paths to sink node

- The ALAP schedule start times can be derived by subtracting the longest path time from the desired overall latency constraint

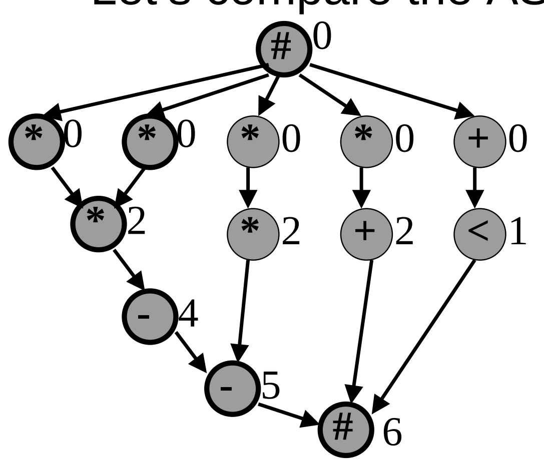
ALAP Scheduling

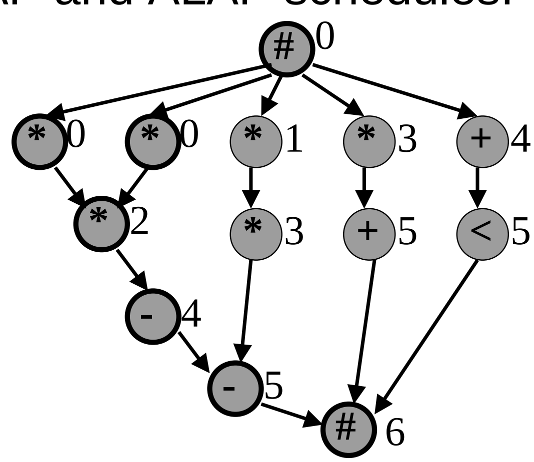


- Here are the ALAP start times. You can see that each operation starts at the latest opportunity possible to still meet 6 cycles overall

Mobility

- Let’s compare the ASAP and ALAP schedules:





- The highlighted nodes have equal ASAP and ALAP times. For all others there is a difference of at least once cycle.

Mobility

- The difference between the ALAP and ASAP times for an operation is called the *operation mobility* or *slack*.
- Mobility measures how free we are to move the operation into different time-slots.
- Operations with zero mobility are *critical operations*, and together form the *critical path*, which determines how fast our circuit will run.
- More sophisticated scheduling algorithms will take advantage of positive mobility to balance the resource requirements over time.

Types of Timing Constraint

- As well as an overall latency constraint, other types of timing constraint are important
- Consider these examples [DeMicheli94]
  – A circuit reads data from a bus, performs a computation, and writes the result back onto the bus. The bus interface specifies that the data must be written exactly three cycles after the read
  – A circuit has two independent streams of operations, constrained to communicate simultaneously to external circuits by providing two pieces of data at two interfaces. The cycle in which the data are made available is irrelevant, although the simultaneity of the data is essential.

Types of Timing Constraint

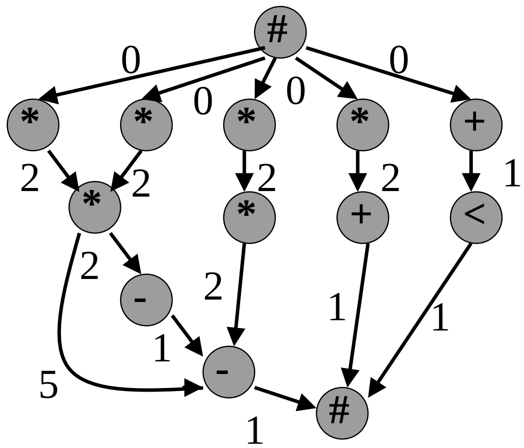
- We will consider two types of constraint
  - a minimum timing constraint l_{ij} between operations v_i and v_j: $S(v_j) \geq S(v_i) + l_{ij}$
  - a maximum timing constraint u_{ij} between operations v_i and v_j : $S(v_j) \leq S(v_i) + u_{ij}$
- These constraints are sufficient to model the situations on the previous slide, in addition to many others. Solutions for previous slide:
  - set both min and max of 3 cycles between read and write
  - set both min and max of 0 cycles between the two writes

Modelling Timing Constraints

- How can we incorporate these timing constraints within our sequencing graph-based model, and how do they affect the schedule?
- From the sequencing graph $G(V,E)$, we construct an edge-weighted *constraint graph* $G_C(V,E_C)$, where $E \subset E_C$:
  - the edge weights for edges in E are the same as before (i.e. the delay of the node producing that edge)
  - we add extra edges to model the timing constraints

Modelling Timing Constraints

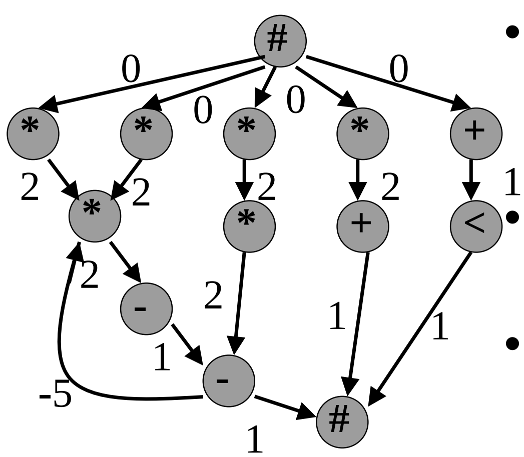
- Minimum timing constraints can simply be modelled by adding an extra edge (v_i, v_j) with weight l_{ij}



- By adding the curved edge with weight 5, the subtraction operation cannot start for at least 5 cycles after the multiplication starts

Modelling Timing Constraints

- Maximum timing constraints can be modelled by adding an extra edge (v_j, v_i) with weight $-u_{ij}$



- Now the multiplication cannot occur before -5 cycles after the subtraction starts
- $S(\text{mult}) \geq S(\text{sub}) - 5$, i.e. $S(\text{sub}) \leq S(\text{mult}) + 5$
- The subtraction cannot occur later than five cycles after the multiplication starts

Scheduling with timing constraints

- ASAP / ALAP scheduling can still be performed on constraint graphs through the longest path technique, BUT…
  – the graph may no longer be a DAG (e.g. on the previous slide)
  – we may need to use Liao-Wong to find the longest path

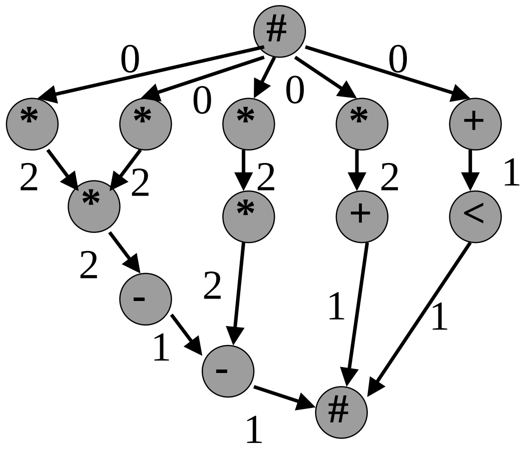
Summary

- This lecture has covered
  – The ASAP scheduling algorithm
  – The ALAP scheduling algorithm and operation slack
  – Introducing timing constraints into schedules

- Next lecture will look at list scheduling, an heuristic method to find a short schedule given constraints on the number of each type of resource available

Suggested Problem

- Consider again the differential equation example from Lecture 1, repeated again below.



- It is required that the subtraction operation marked (α) begin no later than 3 cycles after the addition operation marked (β)
- Compare the ALAP schedules with and without this constraint

More Suggested Problems

- DeMicheli, Chapter 5, Problems 2 and 3 (note that DeMicheli refers to a combined min and max constraint between the source vertex and an operation as a “release time” constraint)

List Scheduling

- The final portion of the course covers
  - Scheduling algorithms
  - Resource sharing algorithms
  - Module selection
  - Perspectives for the future
- This lecture covers
  - resource constrained scheduling and latency constrained scheduling
  - the resource-constrained list-scheduling algorithm
  - the latency-constrained list-scheduling algorithm

Resource Constrained Scheduling

- The following problem is given the name "resource constrained scheduling":
  - Given a library of resources, and a constraint on the maximum number of each type of resource to be used in the implementation, find a schedule of minimum latency
- This problem is NP-hard (proof in Lecture 6), so generally heuristics are used to attack the problem
  - we will also be looking at a way to find an optimum solution next lecture

Resource Constrained Scheduling

- Let R denote the set of resource types,
  - e.g. R = {add, mult, ALU}
- Let the bound on the number of each resource type $r \in R$ be a_r
- In list scheduling, we schedule operations by considering each clock-cycle in turn
  - $U_{t,r}$ is used to denote the set of operations of type r whose predecessors have already completed by cycle t – the candidate set
  - $T_{t,r}$ is used to denote the set of operations of type r started, but not completed by cycle t

Resource Constrained Algorithm

```
Algorithm RC_ListSchedule( G(V,E), R, a ) {
 set t = 0;
 repeat {
   foreach r ∈ R {
     determine U_{t,r};
     determine T_{t,r};
     select Y ⊆ U_{t,r}, s.t. |Y| + |T_{t,r}| ≤ a_r;
     set S(v) = t for all v ∈ Y;
   }
   set t = t+1;
 } until all nodes scheduled
 return( S );
}
```

Resource Constrained Algorithm

- At each clock cycle, the candidate set represents those operations we *could* schedule
- From the candidate set, we select a subset Y, which we *do* schedule
- The constraint on selection of Y is that we can never have more than a_r operations of type r executing simultaneously

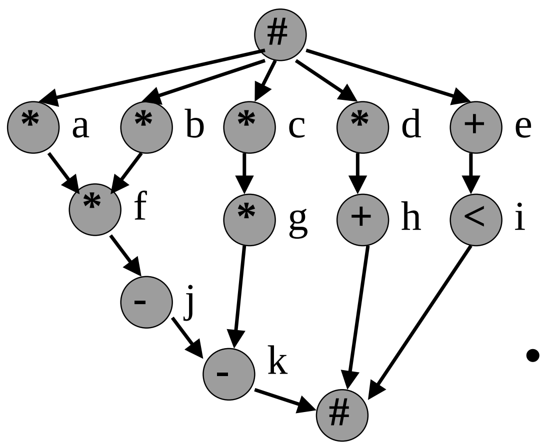
- Notice that as $a_r \rightarrow \infty$ for all $r \in R$, the list schedule approaches an ASAP schedule

Resource Constrained Algorithm

- Notice that the algorithm is not fully defined, as we haven't said how to pick Y
- The most common way to pick Y is to prefer to schedule the most urgent operations first
- Urgency is typically defined in terms of the minimum latency ALAP schedule time – the lower the ALAP time, the more urgent the operation is

Resource Constrained Example

- Let's re-visit our familiar differential equation example



- Consider scheduling under the resource set R = {*, +/-, <}, where the delay of +/- and < is 1 cycle, and the delay of * is 2 cycles
- We will perform a list-schedule with a_*=2, $a_{+/-}$=2, $a_<$=1

Resource Constrained Example

- $t = 0$
  - $U_{0,*}$ = {a,b,c,d}, $U_{0,+/-}$ = {e}, $U_{0,<} = \varnothing$
  - $T_{0,*} = \varnothing$, $T_{0,+/-} = \varnothing$, $T_{0,<} = \varnothing$
  - For +/-, easy to select Y = {e}
  - For *, we have a choice. ALAP times for a,b,c,d are 0,0,1,3, respectively (see Lecture 9). So most urgent are Y = {a,b}
  - For <, there is nothing to schedule $Y = \varnothing$
  - S(a) = 0, S(b) = 0, S(e) = 0

Resource Constrained Example

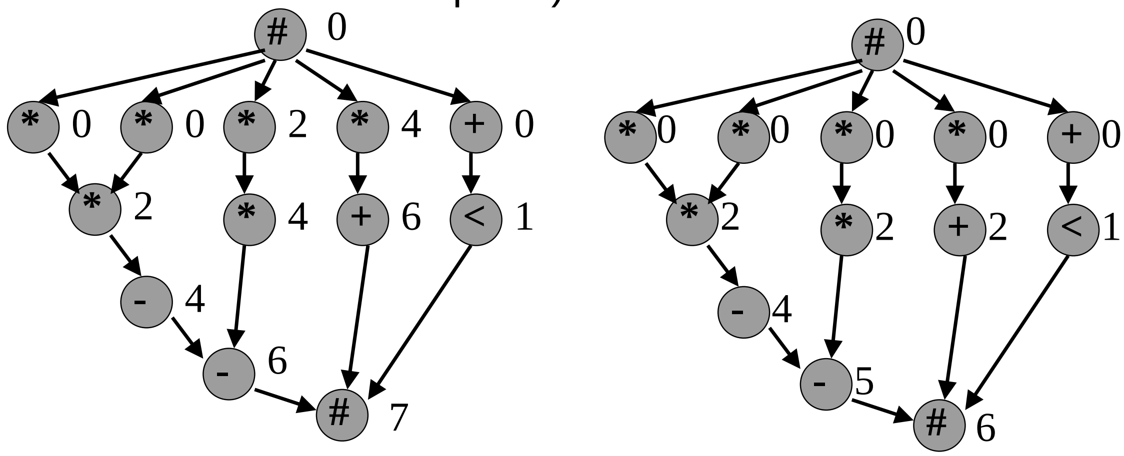
- $t = 1$
  - $U_{1,*} = \{c,d\}$, $U_{1,+/-} = \varnothing$, $U_{1,<} = \{i\}$
  - $T_{1,*} = \{a,b\}$, $T_{1,+/-} = \varnothing$, $T_{1,<} = \varnothing$
  - For +/-, $Y = \varnothing$
  - For *, $Y = \varnothing$ (all resources busy)
  - For <, $Y = \{i\}$
  - $S(i) = 1$

Resource Constrained Example

- t = 2
  - $U_{2,*} = \{c,d,f\}$, $U_{2,+/-} = \varnothing$, $U_{2,<} = \varnothing$
  - $T_{2,*} = \varnothing$, $T_{2,+/-} = \varnothing$, $T_{2,<} = \varnothing$
  - For +/-, $Y = \varnothing$
  - For *, ALAP times for c,d,f are 1,3,2 respectively. $Y = \{c,f\}$
  - For <, $Y = \varnothing$
  - $S(c) = 2$, $S(f) = 2$

Resource Constrained Example

- If we continue this process until the algorithm terminates
  - we take once cycle longer than ASAP (but can use half the number of multipliers)



List-scheduled times ASAP times from Lect 9

Latency Constrained Scheduling

- The dual problem is "latency constrained scheduling":
  - Given a library of resources, and a constraint on the maximum overall latency of the schedule, find a schedule using the minimum number of resources of each type
- This problem is also NP-hard (the same proof holds), so again heuristics are used to attack the problem
- Let λ denote the desired maximum latency

Latency Constrained Algorithm

Algorithm LC_ListSchedule(*G*(*V*,*E*), *R*, λ) {
perform ALAP(G(V,E), λ);
set a_r = 1 for all $r \in R$;
set t = 0;
repeat {
foreach $r \in R$ {
determine $U_{t,r}$;
determine $T_{t,r}$;
determine slack $s_v = ALAP_v - t$ for all $v \in U_{tr}$;
set $Y_1 = \{v \in V: s_v = 0\}$;
set $a_r = \max(a_r , |Y_1| + |T_{tr}|)$;
select $Y_2 \subseteq U_{tr}$, s.t. $|Y_1 \cup Y_2| + |T_{tr}| \leq a_r$;
set $S(v) = t$ for all $v \in Y_1 \cup Y_2$;
}
set $t = t+1$;
} until all nodes scheduled
return(*S, a*);
}

Latency Constrained Algorithm

- This algorithm works by constantly refining the "maximum" number of resources it allows
  - we start with one resource of each type
  - this is changed if the desired latency is not achievable
- For each cycle, we calculate the *slack* of the candidate operations
  - slack is the difference between the last cycle an operation could be scheduled in and the current cycle
  - if the slack of an operation is zero, it must clearly be scheduled immediately, even if that means increasing the number of resources allowed

Latency Constrained Algorithm

- Such "forced" scheduled nodes are placed in set Y_1
- It may also be possible to schedule additional nodes, without increasing the resource requirements further. These are placed in Y_2, and selected on the basis of urgency, as with the resource-constrained algorithm

Latency Constrained Example

- As an example, we will again consider the differential equation CDFG
  - The ASAP schedule gave a minimum schedule length of 6 cycles. It had up to 4 "*", 1 "+" and 1 "<" operating in parallel
  - Let's see whether latency constrained list scheduling can do better than that
- We will execute LC_ListSchedule(G(V,E), R, 6)
- The ALAP times for this example have already been determined in Lecture 9, and are:
  - a: 0, b: 0, c: 1, d: 3, e: 4, f: 2, g: 3, h: 5, i: 5, j: 4, k: 5

Latency Constrained Example

- t = 0
  - $U_{0,*}$ = {a,b,c,d}, $U_{0,+/-}$ = {e}, $U_{0,<} = \emptyset$
  - $T_{0,*} = \emptyset$, $T_{0,+/-} = \emptyset$, $T_{0,<} = \emptyset$
  - $s_a = 0$, $s_b = 0$, $s_c = 1$, $s_d = 3$, $s_e = 4$
  - For *, Y_1 = {a,b}; for +/-, $Y_1 = \emptyset$; for <, $Y_1 = \emptyset$
  - $a_* = 2$; others unchanged
  - For *, $Y_2 = \emptyset$; for +/-, Y_2 = {e}; for <, $Y_2 = \emptyset$
  - S(a) = 0, S(b) = 0, S(e) = 0

Latency Constrained Example

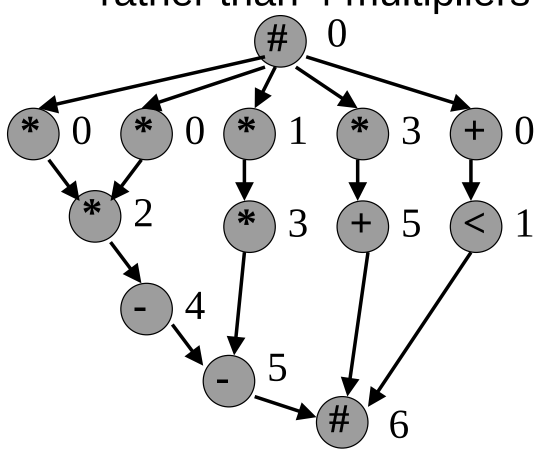
- t = 1
  - $U_{1,*}$ = {c,d}, $U_{1,+/-} = \emptyset$, $U_{1,<}$ = {i}
  - $T_{1,*}$ = {a,b}, $T_{1,+/-} = \emptyset$, $T_{1,<} = \emptyset$
  - $s_c = 0$, $s_d = 2$, $s_i = 4$
  - For *, Y_1 = {c}; for +/-, $Y_1 = \emptyset$; for <, $Y_1 = \emptyset$
  - $a_* = 3$; others unchanged
  - For *, $Y_2 = \emptyset$; for +/-, $Y_2 = \emptyset$; for <, Y_2 = {i}
  - S(c) = 1, S(i) = 1

Latency Constrained Example

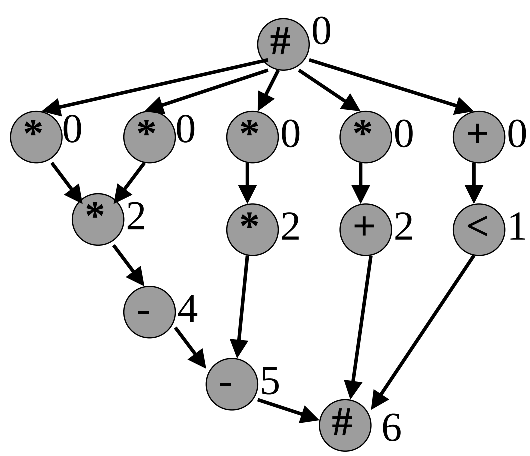
- t = 2
  - $U_{2,*}$ = {f}, $U_{2,+/-} = \varnothing$, $U_{2,<} = \varnothing$
  - $T_{2,*}$ = {c}, $T_{2,+/-} = \varnothing$, $T_{2,<} = \varnothing$
  - $s_f = 0$
  - For *, Y_1 = {f}; for +/-, $Y_1 = \varnothing$; for <, $Y_1 = \varnothing$
  - all resource constraints unchanged
  - For *, $Y_2 = \varnothing$; for +/-, $Y_2 = \varnothing$; for <, $Y_2 = \varnothing$
  - S(f) = 2

Latency Constrained Example

- If we continue this process until the algorithm terminates
  - schedule has the same latency as ASAP, but requires 3 rather than 4 multipliers



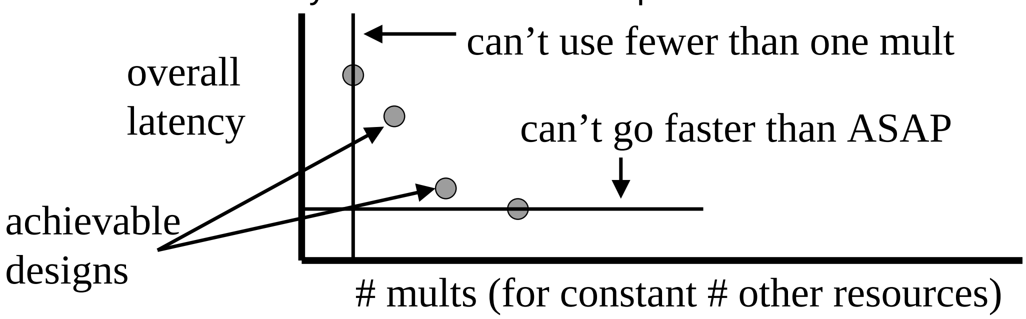
List-scheduled times



ASAP times from Lect 9

Area / Speed Tradeoffs

- In general, if we allow more resources, the schedule may have a shorter latency
- Similarly, if we allow a longer latency, the schedule may require fewer resources
- This leads to the concept of an area / speed tradeoff
  - one of a designers most important jobs is to explore this curve – and architectural synthesis tools can help



Summary

- This lecture has covered
  - resource constrained scheduling and latency constrained scheduling
  - the resource-constrained list-scheduling algorithm
  - the latency-constrained list-scheduling algorithm
  - area / speed tradeoffs

- Next lecture will look at optimum scheduling methods, using Integer Linear Programming

Suggested Problems

1. Re-visit the differential equation example. For two +/- resources and one < resource, draw the complete Area / Speed tradeoff curves achieved by applying
   - resource-constrained list-scheduling
   - latency-constrained list-scheduling

   Are they the same? Account for any differences (**)
2. Write a program to perform one of the list-scheduling algorithms and test it on some CDFGs of your own invention (***)

Optimum Scheduling

- The final portion of the course covers
  - Scheduling algorithms
  - Resource sharing algorithms
  - Module selection
  - Perspectives for the future
- This lecture covers
  - Optimum scheduling: why ILP?
  - Integer linear program model
  - Example ILP and solution

Optimum Scheduling

- Last lecture we looked at an heuristic scheduling technique: list scheduling
- We may also wish to know the optimum result for a given scheduling problem
  - optimum results are only achievable for small problems, as resource-constrained scheduling is *NP*-hard
  - if we design a heuristic, and it achieves near-optimal schedules for small problems, we are usually more confident it will do well for large problems
  - optimum results form a “baseline” against which we can compare heuristics

Why ILP?

- Integer Linear Programming is useful to achieve optimum results because
  - it lets us formalize the problem
  - it gives a structure to the problem: what is the objective function, what are the constraints, how many are there, what are their nature?
  - we can use ILP solvers such as lp_solve (`ftp://ftp.es.ele.tue.nl/pub/lp_solve/`) to solve problems once they are in ILP format

Notation

- We will use the following notation, mainly carried over from previous lectures
  - $S(v)$: the scheduled start time of node v
  - d_v: the delay (latency) of node v
  - a_r: the maximum number of resources of type r
  - $T(v)$: the type of node v
  - R: the set of resource types
  - λ: the maximum overall latency
  - ASAP_v (ALAP_v): the ASAP time (ALAP time) under overall latency λ
  - x_{vt}: binary decision variable (see next slide)
  - c_r: the cost of a resource of type r

Binary Decision Variables

- We will use a trick often used in ILP formulations: to introduce binary decision variables
- We will use x_{vt} ($v \in V$, $t \in$ {ASAP$_v$, ASAP$_v$+1, …, ALAP$_v$}, with $x_{vt} = 1$ iff node v is scheduled to start at time t, i.e. $x_{vt} = 1 \Leftrightarrow S(v) = t$
- These will allow us to formulate the resource constraints as *linear* functions of x_{vt}
- Note that if we are doing resource-constrained scheduling, we may not know λ. Since it is an upper bound, we can use RC list scheduling to obtain it.

Ensuring a Unique Start Time

- Our first constraint needs to be to ensure that each operation starts at only one time

$$\forall v \in V : \sum_{t=ASAP_v}^{ALAP_v} x_{vt} = 1$$

- Because x_{vt} are constrained to be binary variables, this means that exactly one time-index is true for each operation

Specifying Data Dependencies

- Of course we can't allow operations to start before their predecessors in the CDFG have completed

$$\forall (v', v) \in E: \sum_{t=ASAP_v}^{ALAP_v} t \cdot x_{vt} \geq \sum_{t=ASAP_{v'}}^{ALAP_{v'}} t \cdot x_{v't} + d_{v'}$$

- Each edge in the CDFG defines one of these constraints
- Each summation represents the start time of the particular node (*v* on the LHS, *v'* on the RHS)

Specifying Resource Constraints

- No more than a_r operations of type *r* can simultaneously execute

$$\forall r \in R, \forall t \in \{0, \ldots, \lambda\},$$
$$\sum_{v \in V : T(v) = r} \sum_{t' \in \{t - d_v + 1, \ldots, t\} \cap \{ASAP_v, \ldots, ALAP_v\}} x_{vt'} \leq a_r$$

- The first summation is over all nodes of type *r*
- The second summation is over a time "window" covering all start cycles *t'* for which the operation would still be executing by cycle *t*

Resource-Constrained Objective Function

- Under these constraints, the resource-constrained scheduling problem can be solved by minimizing the overall latency (we fix a_r)

$$\min : \sum_{t=ASAP_{v_z}}^{ALAP_{v_z}} x_{v_z t}$$

- Here, v_z represents the “end” or “sink” node in the CDFG

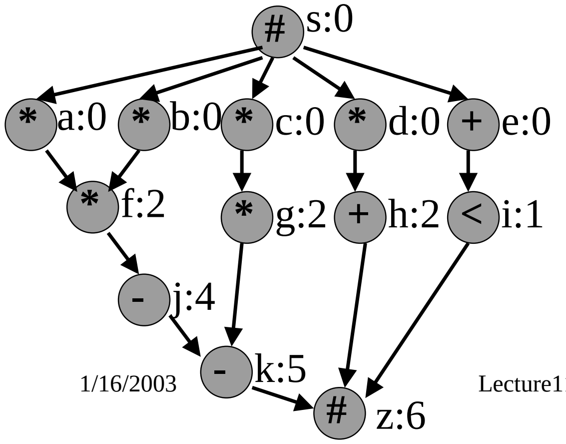
Latency-Constrained Objective Function

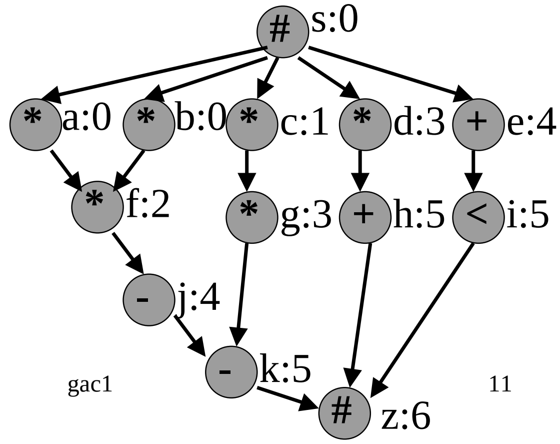
- Under the same constraints, the latency-constrained scheduling problem can be solved by minimizing the cost of the resources required (we fix λ)

$$\min : \sum_{r \in R} c_r a_r$$

Example ILP

- We will build an ILP for the differential equation solver as an example
- We will formulate the latency-constrained problem for $\lambda = 6$, the minimum possible latency
- To refresh your memories, here are the ASAP and ALAP times for $\lambda = 6$ from Lecture 9





Example ILP

- First, lets examine what variables we have:

$$\{x_{s0}, x_{a0}, x_{b0}, x_{c0}, x_{c1}, x_{d0}, x_{d1}, x_{d2},$$
$$x_{d3}, x_{e0}, x_{e1}, x_{e2}, x_{e3}, x_{e4}, x_{f2}, x_{g2},$$
$$x_{g3}, x_{h2}, x_{h3}, x_{h4}, x_{h5}, x_{i1}, x_{i2}, x_{i3},$$
$$x_{i4}, x_{i5}, x_{j4}, x_{k5}, x_{z6}\}$$

- Operations with large mobility give rise to a large number of variables

Example ILP

- The first constraints are unique-start-time constraints:

$$x_{s0} = 1$$
$$x_{a0} = 1$$
$$x_{b0} = 1$$
$$x_{c0} + x_{c1} = 1$$
$$x_{d0} + x_{d1} + x_{d2} + x_{d3} = 1$$
$$x_{e0} + x_{e1} + x_{e2} + x_{e3} + x_{e4} = 1$$

$$x_{f2} = 1$$
$$x_{g2} + x_{g3} = 1$$
$$x_{h2} + x_{h3} + x_{h4} + x_{h5} = 1$$
$$x_{i1} + x_{i2} + x_{i3} + x_{i4} + x_{i5} = 1$$
$$x_{j4} = 1$$
$$x_{k5} = 1$$
$$x_{z6} = 1$$

Example ILP

- The next constraints are dependency constraints:

$$0 \cdot x_{a0} \geq 0 \cdot x_{s0} + 0$$
$$0 \cdot x_{b0} \geq 0 \cdot x_{s0} + 0$$
$$0 \cdot x_{c0} + 1 \cdot x_{c1} \geq 0 \cdot x_{s0} + 0$$
$$0 \cdot x_{d0} + 1 \cdot x_{d1} + 2 \cdot x_{d2} + 3 \cdot x_{d3} \geq 0 \cdot x_{s0} + 0$$
$$0 \cdot x_{d0} + 1 \cdot x_{d1} + 2 \cdot x_{d2} + 3 \cdot x_{d3} + 4 \cdot x_{d4} \geq 0 \cdot x_{s0} + 0$$
$$2 \cdot x_{f2} \geq 0 \cdot x_{a0} + 2$$
$$2 \cdot x_{f2} \geq 0 \cdot x_{b0} + 2$$
$$2 \cdot x_{g2} + 3 \cdot x_{g3} \geq 0 \cdot x_{c0} + 1 \cdot x_{c1} + 2$$
$$2 \cdot x_{h2} + 3 \cdot x_{h3} + 4 \cdot x_{h4} + 5 \cdot x_{h5} \geq 0 \cdot x_{d0} + 1 \cdot x_{d1} + 2 \cdot x_{d2} + 3 \cdot x_{d3} + 2$$
$$1 \cdot x_{i1} + 2 \cdot x_{i2} + 3 \cdot x_{i3} + 4 \cdot x_{i4} + 5 \cdot x_{i5} \geq 0 \cdot x_{e0} + 1 \cdot x_{e1} + 2 \cdot x_{e2} + 3 \cdot x_{e3} + 4 \cdot x_{e4} + 1$$

Example ILP

- Dependency constraints continued…

$$4 \cdot x_{j4} \geq 2 \cdot x_{f2} + 2$$

$$5 \cdot x_{k5} \geq 2 \cdot x_{g2} + 3 \cdot x_{g3} + 2$$

$$6 \cdot x_{z6} \geq 2 \cdot x_{h2} + 3 \cdot x_{h3} + 4 \cdot x_{h4} + 5 \cdot x_{h5} + 1$$

$$6 \cdot x_{z6} \geq 1 \cdot x_{i1} + 2 \cdot x_{i2} + 3 \cdot x_{i3} + 4 \cdot x_{i4} + 5 \cdot x_{i5} + 1$$

$$5 \cdot x_{k5} \geq 4 \cdot x_{j4} + 1$$

$$6 \cdot x_{z6} \geq 5 \cdot x_{k5} + 1$$

Example ILP

- Resource constraints:

$$r = <, t = 1 : x_{i1} \leq a_{<}$$

$$r = <, t = 2 : x_{i2} \leq a_{<}$$

$$r = <, t = 3 : x_{i3} \leq a_{<}$$

$$r = <, t = 4 : x_{i4} \leq a_{<}$$

$$r = <, t = 5 : x_{i5} \leq a_{<}$$

$$r = +/-, t = 0 : x_{e0} \leq a_{+/-}$$

$$r = +/-, t = 1 : x_{e1} \leq a_{+/-}$$

$$r = +/-, t = 2 : x_{e2} + x_{h2} \leq a_{+/-}$$

$$r = +/-, t = 3 : x_{e3} + x_{h3} \leq a_{+/-}$$

$$r = +/-, t = 4 : x_{e4} + x_{h4} + x_{j4} \leq a_{+/-}$$

$$r = +/-, t = 5 : x_{h5} + x_{k5} \leq a_{+/-}$$

Example ILP

- More resource constraints:

$$r = *, t = 0: x_{a0} + x_{b0} + x_{c0} + x_{d0} \le a_*$$

$$r = *, t = 1: x_{a0} + x_{b0} + x_{c0} + x_{c1} + x_{d0} + x_{d1} \le a_*$$

$$r = *, t = 2: x_{c1} + x_{d1} + x_{d2} + x_{f2} + x_{g2} \le a_*$$

$$r = *, t = 3: x_{d2} + x_{d3} + x_{f2} + x_{g2} + x_{g3} \le a_*$$

- Objective function:
  – let's assume the cost of a mult is "2", and that of an adder and comparator is "1" and the cost of an add is "1":

$$\min: 2a_* + a_{+/-} + a_<$$

Example ILP

- This (rather long!) example contains 29 binary decision variables and 3 resource allocation variables (total = 32) and 44 constraints
- For even this small example, the ILP model is quite sizable
  – ILP is only really practical for solving small problems

Summary

- This lecture has covered
  – Optimum scheduling: why ILP?
  – Integer linear program model
  – Example ILP and solution
- Next lecture will move off the subject of scheduling, and start to consider algorithms for resource sharing

Suggested Problems

- Download a copy of lp_solve from the website given at the start of the lecture, and solve the ILP example
  – what is the minimum possible cost?
  – how many adders, multipliers, and comparators does it use?
  – how does that compare with a latency-constrained list-schedule?

Resource Sharing

- The final portion of the course covers
  - Scheduling algorithms
  - Resource sharing algorithms
  - Module selection
  - Perspectives for the future
- This lecture covers
  - Non-hierarchical CDFGs
  - Hierarchical CDFGs

Introduction

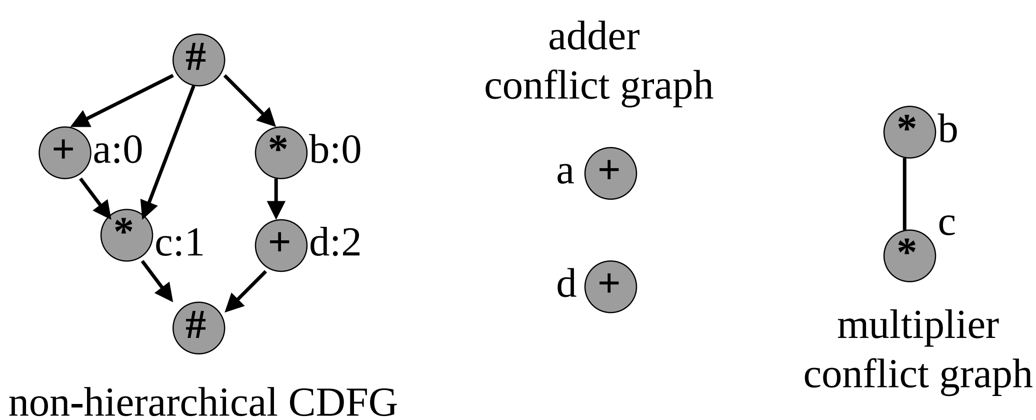
- We will consider some approaches for sharing resources between operations
- Non-hierarchical and hierarchical CDFGs will be considered separately
  - problem has different complexity
- Remember that hierarchical CDFGs can be used to represent the following (Lecture 1)
  - conditionals
  - loops
  - function calls

Resource Conflict Graph

- The one fundamental restriction on sharing resources:
  - two operations executing simultaneously cannot be executed on the same resource
- This leads to the concept of "resource conflict"
- Two operations are in resource conflict if they overlap in execution time
- A resource conflict graph uses the same node set as the CDFG, but uses a set of undirected edges such that: (Lecture 2)
  - two operations are joined by an edge iff they are in resource conflict

Non-Hierarchical CDFGs

- For non-hierarchical CDFGs (i.e. those with just one level of hierarchy), such a conflict graph is simple



Graph Structure

- Conflict graphs for non-hierarchical CDFGs are *interval graphs*
- Recall from Lecture 5 that an interval graph is one 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
- Also recall from Lecture 5 that such graphs are colourable easily in polynomial time using the *left-edge* algorithm

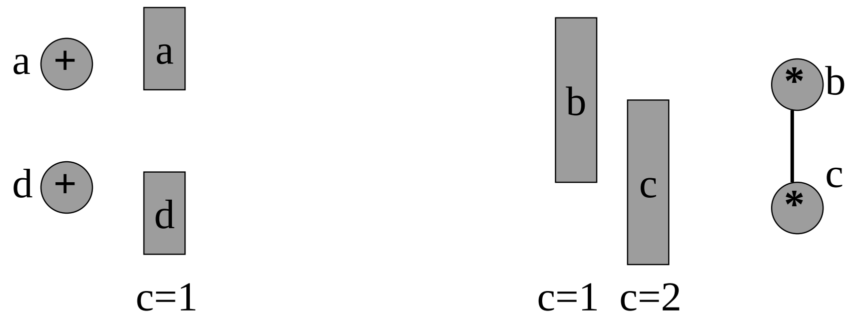
Solution via Left-Edge

- We can therefore find an optimum binding using left-edge, reproduced below from Lecture 5
  - use the scheduled start and end times as the left and right "edges", respectively

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;
    v_s := first node in *L* with $l_s > r$;
    r: = r_s;
    label v_s with colour c
    $L := L \setminus \{v_s\}$; }
  c := c + 1;
end

Left-Edge: Example

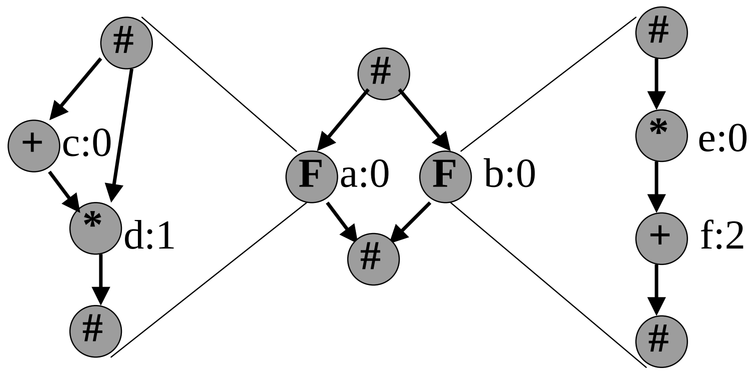
- Taking the previous example:



- So use one adder to do both a and d, but different multipliers to do b and c
- Formally, Y(a) = (+,1); Y(b) = (*,1); Y(c)=(+,1); Y(d)=(*,2)

Hierarchical CDFGs

- Consider a simple hierarchical CDFG with function calls, performing the same function as the previous example



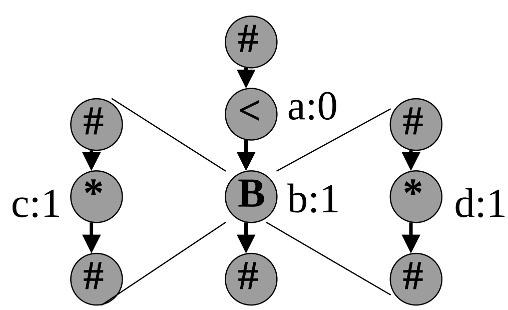
Hierarchical CDFGs

- How do we perform resource sharing?
  - a naïve approach would be to perform resource sharing on each level of the hierarchy in turn
  - for our example, this would lead to one multiplier and one adder for each function: one more adder than we needed for the non-hierarchical version
- We should try to share resources across the levels of hierarchy

Conditionals

- Conditionals help us share resources, as the two branches ("if" and "else") are never needed simultaneously

```
a = b < c;
if (a) then
  d = b * b;
else
  d = c * c;
```



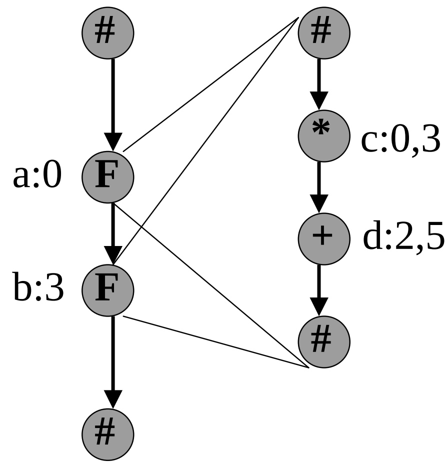
- Operations c and d are not in resource conflict, although they have the same type and "overlap" in time

Multiple Function Calls

- Multiple calls to the same function complicate matters, as operations can have several execution times

```
a = fun(x);
b = fun(a);
```

```
fun(p) {
 return p*p + 5;
}
```



Graph Properties

- Conditionals and multiple function calls change the structure of the conflict graph
  – it no longer must be an interval graph
  – the left-edge algorithm is therefore no longer applicable
- We need an heuristic approach to colouring the graph
  – one such algorithm is given in Lecture 5

Colouring Heuristic

- Here is the colouring heuristic from Lecture 5:

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

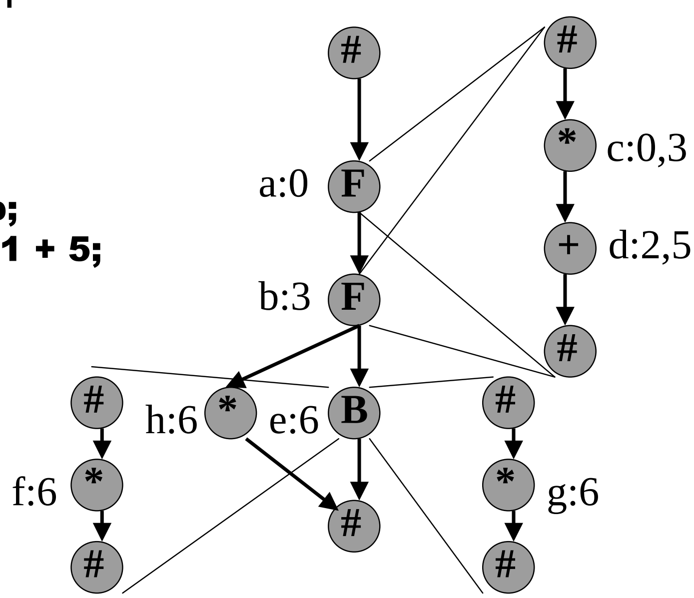
- We will apply it to an example with conditionals and multiple function calls

Hierarchical Example

- Here is a more complex scheduled CDFG

```
a = fun(x);
b = fun(a);
if (y) then
  c = b * b;
else
  c = 2 * b;
d = 3*b;
```

```
fun(p) {
 t1 = p*p;
 return t1 + 5;
}
```



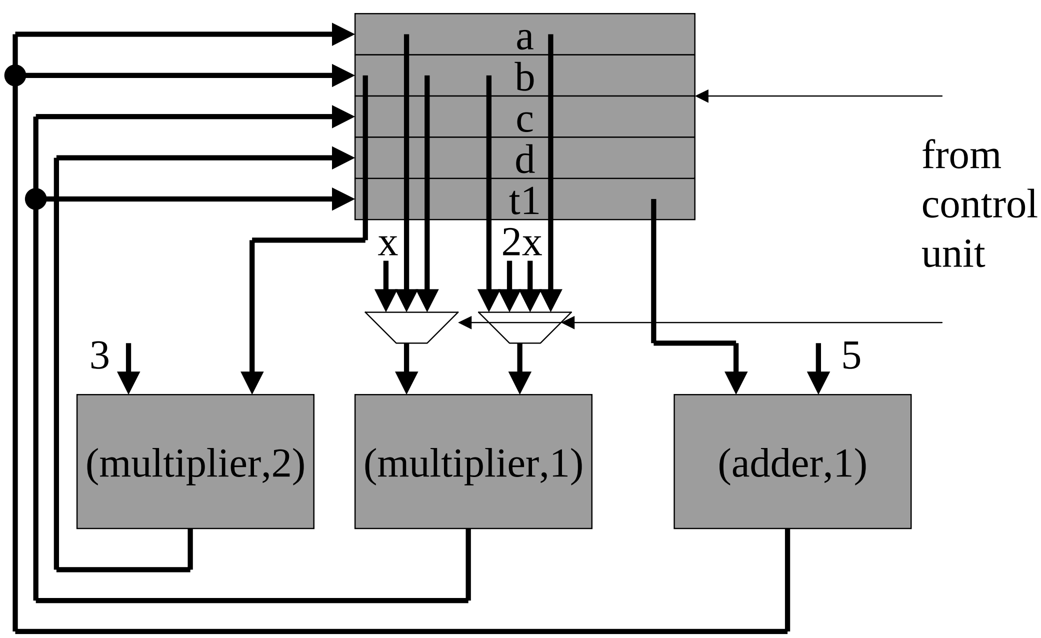
Hierarchical Example

- Remember f and g don't conflict (if / else)

multiplier conflict graph   adder conflict graph

- Let's colour the multiplier nodes in the order: c, f, g, h
  - c gets colour 1; f gets colour 1; g gets colour 1; h gets colour 2
  - we need two mults and an add

Example Datapath



Summary

- We have investigated resource sharing for both
  - Non-hierarchical CDFGs
  - Hierarchical CDFGs
- Next lecture we will look at register sharing

Suggested Problems

- Perform a resource binding for the list-scheduled differential equation example from Lecture 10 and draw the completed datapath (*)
- Design a controller for this datapath (*)
- Discuss resource binding for conditionals within conditionals (****)
- Discuss a possible approach to resource binding for loops (****)
- De Micheli, Problems 6.11, No. 1 (conflict graphs only) (*)