Lecture Notes on Machine Representation of a Graph

Jim Newton

September 22, 2026

1 Naive Implementation, Bad Idea

We might attempt to represent a graph programmically similar to its mathematical definition.

Definition 1.1. A *simple graph*, (V, E), is an object consisting of two finite sets V and E, such that $V \neq \emptyset$ and E is a set of two-element subsets of V. V is called the *vertex set*, and each element of V is called a *vertex* or a *node*. E is called the *edge set*, and every element of E is called an *edge* or an *arc*.

In keeping with this definition, one might attempt a programmatic definitions somewhat like the following.

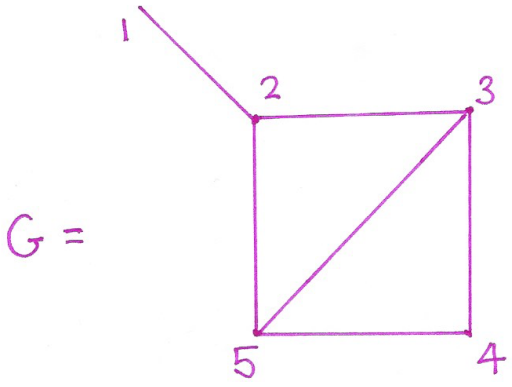
```
class Graph[T] {
  type Edge = UnorderedPair[T]
  def V:Set[T]
  def E:Set[Edge]
}
```

With such a class definition we would be able to represent a graph whose vertices are Integers as `Graph[Integer]`, similarly `Graph[String]`, or `Graph[Array[Boolean]]`. Unfortunately, this is rarely done because of performance. Such a representation does not facilitate common operations. For example: given a vertex, find the adjacent vertices.

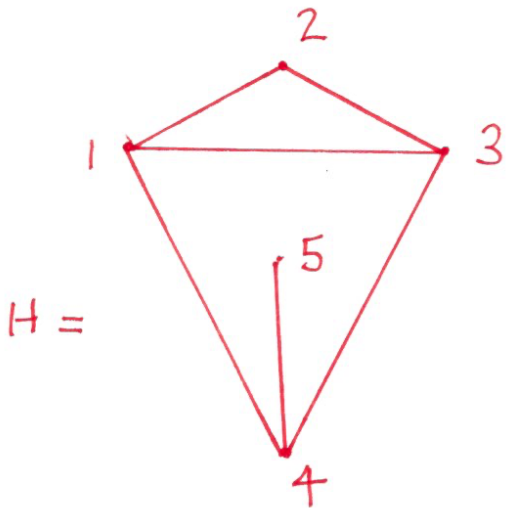
Two common representations are *Adjacency Matrix* and *Adjacency List*.

2 The Adjacency Matrix

The following example is taken from Sandra Herke, in her YouTube video *Graph Theory FAQ03, Isomorphisms using Adjacency Matrix.*



$$A_G = \begin{array}{c} \begin{matrix} ① & ② & ③ & ④ & ⑤ \end{matrix} \\ \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix} \end{array} \begin{matrix} ① \\ ② \\ ③ \\ ④ \\ ⑤ \end{matrix}$$



$$A_H = \begin{array}{c} \begin{matrix} ① & ② & ③ & ④ & ⑤ \end{matrix} \\ \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array} \begin{matrix} ① \\ ② \\ ③ \\ ④ \\ ⑤ \end{matrix}$$

Definition 2.1. The *adjacency matrix* of a graph G is the matrix with

1 at row i column j whenever G has an edge connecting vertices v_i and v_j, and 0 otherwise.

$$A_G = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix} \quad A_H = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

QUESTION: Recall the complement matrix, $\overline{G}$ which connects vertices v_i and v_j if and only if G does not. What are adjacency matrices for $\overline{G}$ and $\overline{H}$ above; i.e., what are $A_{\overline{G}}$ and $A_{\overline{H}}$? ANSWER: just exchange 1 for 0 and vice versa everywhere they appear, except maintain 0s on the main diagonal.

$$A_{\overline{G}} = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix} \quad A_{\overline{H}} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \end{bmatrix}$$

Note also that the adjacenty matrix for a simple (undirected) graph is symmetric. This is not the case in a directed graph.

3 Adjacency Matrix and Isomorphism

G and H are isomorphic if and only if there exists an orthogonal matrix P_{HG} such that

$$P_{HG} A_H P_{HG}^{-1} = A_G \, .$$

In such a case P_{HG} is called a *permutation matrix* or the *permutation matrix* going from H to G.

Definition 3.1. An *orthogonal matrix*, M, has orthogonal unit vectors as rows and columns and has the special property that

$$M^{-1} = M^T$$

its inverse is identical to its transpose.

So to calculate the inverse of P_{HG}, we simply transpose it. Finding P_{HG} is difficult, but verifying it is straightforward.

$$P_{HG} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Let's check whether $P_{HG} A_H P_{HG}^T = A_G$.

$$P_{HG}A_HP_{HG}^T = \overbrace{\begin{bmatrix}0&0&0&0&1\\0&0&0&1&0\\1&0&0&0&0\\0&1&0&0&0\\0&0&1&0&0\end{bmatrix}\begin{bmatrix}0&1&1&1&0\\1&0&1&0&0\\1&1&0&1&0\\1&0&1&0&1\\0&0&0&1&0\end{bmatrix}}^{\text{Associative, so we can multiply these first.}}\begin{bmatrix}0&0&1&0&0\\0&0&0&1&0\\0&0&0&0&1\\0&1&0&0&0\\1&0&0&0&0\end{bmatrix}$$

$$= \begin{bmatrix}0&0&0&1&0\\1&0&1&0&1\\0&1&1&1&0\\1&0&1&0&0\\1&1&0&1&0\end{bmatrix}\begin{bmatrix}0&0&1&0&0\\0&0&0&1&0\\0&0&0&0&1\\0&1&0&0&0\\1&0&0&0&0\end{bmatrix}$$

$$= \begin{bmatrix}0&1&0&0&0\\1&0&1&0&1\\0&1&0&1&1\\0&0&1&0&1\\0&1&1&1&0\end{bmatrix} = A_G$$

In addition, given $P_{HG} = \begin{bmatrix}0&0&1&0&0\\0&0&0&1&0\\0&0&0&0&1\\0&1&0&0&0\\1&0&0&0&0\end{bmatrix}$, we can *extract* the isomorphism. To do so, we note that any 1 in row i column j means that vertex v_i of A_H maps to vertex v_j of A_G.

	row		col
$v_{i,j}$	A_H	$\to$	A_G
$v_{5,1}$	5	$\to$	1
$v_{4,2}$	4	$\to$	2
$v_{1,3}$	1	$\to$	3
$v_{2,4}$	2	$\to$	4
$v_{3,5}$	3	$\to$	5

Looking at P_{HG} we conclude that $1 \to 3$, $3 \to 5$, $5 \to 1$, $2 \to 4$, and $4 \to 2$.

Provided the set of vertices of G and H are the same, $V = \{1, 2, 3, 4, 5\}$, in our case, we can represent an isomorphism as permutation, which can be denoted in so-called *cycle notation*. Notice that $1 \to 5 \to 3 \to 1$ is a cycle and so is $2 \to 4 \to 2$. This permutation in terms of its constituent cycles is denoted $A_{HG} = (1\ 3\ 5)(2\ 4)$.

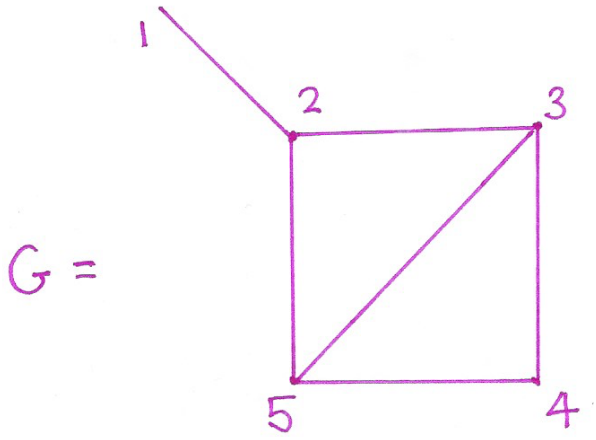
Note that an alternate permutation would be $A_{HG} = (1\ 5)(3)(2\ 4)$, because vertices 1 and 3 in graph H (corresponding to 3 and 5 in graph G) are indistinguishable.

To summarize, the adjacency matrix can be used to *verify* a suspected isomorphism. Given either the permutation cycles or the permutation matrix, we can easily calculate the other, but to calculate either of them given only the graphs is difficult.

4 Adjacency Matrix and k-step Walks

Another use of the adjacency matrix is to count the number of k-step walks between two given vertices.

If A is the adjacency matrix, then $(A^k)_{i,j} = (A^k)_{j,i}$ is precisely the number of k-step walks from vertex v_i to vertex v_j, and consequently from vertex v_j to vertex v_i.



Let's look again at .

$$A_G{}^0 = A_G \cdot A_G{}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$A_G{}^1 = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

The two trivial cases: from $A_G{}^0$, the identity matrix, we see that there is one 0-step walk from every vertex to itself, and from $A_G{}^1$ we see that there is exactly one 1-step walk from every vertex v_i to each vertex v_j if and only if $\{v_i, v_j\}$ is an edge of G.

$$A_G{}^2 = A_G \cdot A_G = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 \\ 0 & 3 & 1 & 2 & 1 \\ 1 & 1 & 3 & 1 & 2 \\ 0 & 2 & 1 & 2 & 1 \\ 1 & 1 & 2 & 1 & 3 \end{bmatrix}$$

From ${A_G}^2$ we see something more interesting. We see 3's in at $(2,2)$, $(3,3)$, and $(5,5)$; thus there are exactly 3 2-step walks from v_2 to itself, namely

1. $v_2 \to v_1 \to v_2$,
2. $v_2 \to v_3 \to v_2$, and
3. $v_2 \to v_5 \to v_2$;

from v_3 to itself, namely

1. $v_3 \to v_2 \to v_3$,
2. $v_3 \to v_4 \to v_3$, and
3. $v_3 \to v_5 \to v_3$;

and from v_5 to itself, namely

1. $v_5 \to v_2 \to v_5$,
2. $v_5 \to v_3 \to v_5$, and
3. $v_5 \to v_4 \to v_5$.

Likewise, there is a 2 at row 4 column 2 (and at row 2 column 4) meaning that there are 2 2-step walks from vertex v_2 to vertex v_4, namely

1. $v_2 \to v_3 \to v_4$, and
2. $v_2 \to v_5 \to v_4$.

$$ {A_G}^3 = {A_G}^2 \cdot A_G = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 \\ 0 & 3 & 1 & 2 & 1 \\ 1 & 1 & 3 & 1 & 2 \\ 0 & 2 & 1 & 2 & 1 \\ 1 & 1 & 2 & 1 & 3 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 3 & 1 & 2 & 1 \\ 3 & 2 & 6 & 2 & 6 \\ 1 & 6 & 4 & 5 & 5 \\ 2 & 2 & 5 & 2 & 5 \\ 1 & 6 & 5 & 5 & 4 \end{bmatrix} $$

The calculation of ${A_G}^3$ predicts that there are 6 3-step walks between v_2 and v_5, namely

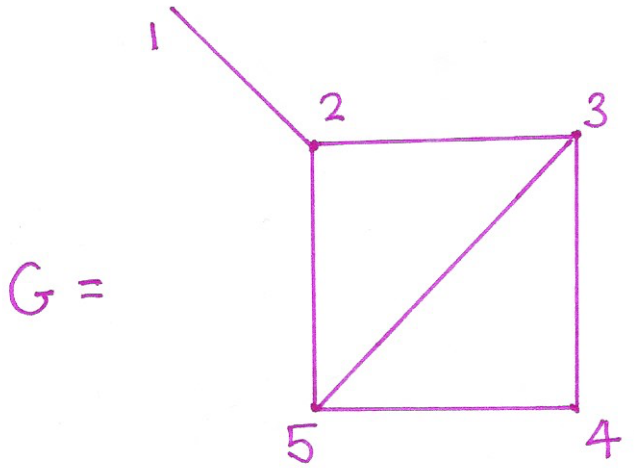
1. $v_2 \to v_1 \to v_2 \to v_5$,
2. $v_2 \to v_3 \to v_2 \to v_5$,
3. $v_2 \to v_5 \to v_2 \to v_5$,
4. $v_2 \to v_3 \to v_4 \to v_5$,
5. $v_2 \to v_5 \to v_3 \to v_5$, and
6. $v_2 \to v_5 \to v_4 \to v_5$.

Space requirement for the adjacency matrix is $\Theta(|V|^2)$, sometimes written $\Theta(V^2)$. This can become $\Theta(|V| + |E|)$ if a sparse matrix is used.

An advantage of the adjacency matrix which we have not yet mentioned is that given A_G, we can easily determine whether v_i connects to v_j in constant time, and we can easily determine the list of connecting vertices of v_i in linear time.

5 The Adjacency List

The so-called *adjacency list* requires $\Theta(|V| + |E|)$ space, and is efficient when $|E| << |V|^2$.



Let's look again at .

Start with an array indexed by vertex. The value associated with index i is a list of the vertices connected to v_i.

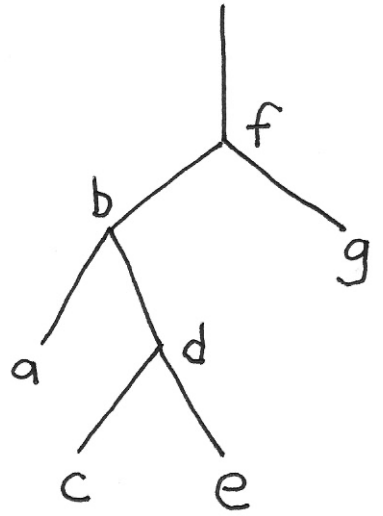
$$A[1] = \{2\}$$
$$A[2] = \{1, 3, 5\}$$
$$A[3] = \{2, 4, 5\}$$
$$A[4] = \{3, 5\}$$
$$A[5] = \{2, 3, 4\}$$

If the graph is un-directed, then whenever $i \in A[j]$, then also $j \in A[i]$. To determine whether v_i connects to v_j we must do a membership check, $i \in V[j]$ or $j \in V[i]$. If the vertices are integers from 0 to $n-1$ or from 1 to n, then A can be a simple array. If not you may have to apply some mapping strategy such as:

- renumbering the vertices
- mapping of the type of vertex to unsigned integer. e.g., String $\rightarrow$ Unsigned.
- hash table

6 Depth First Search, DFS

To understand DFS, first think of a walk over a binary tree without loops and without diamonds.



Algorithm 1: DFS-binary-tree(v,f)

Input : v vertex of a binary tree
Input : f function to apply to each vertex

```
1 if v.left then
2 |   DFS-binary-tree(v.left,f)
3 f(v)
4 if v.right then
5 |   DFS-binary-tree(v.right,f)
```

We'd like to extend this concept to an arbitrary graph, which:

- may not have a designated root,
- maybe has loops, or
- maybe has multiple edges (> 2) per vertex.

We first look at `DFS-visit` which is almost what we want but has some deficiencies which we will note.

Algorithm 2: DFS-visit(A,v,f,parent)

Input : A the adjacency list, mapping each vertex to list of adjacent vertices
Input : v a starting vertex, $v \in V$
Input : f a function to be applied to each vertex
Input : *parent* maps each vertex to a *parent* vertex or None, $V \to V \cup \{None\}$

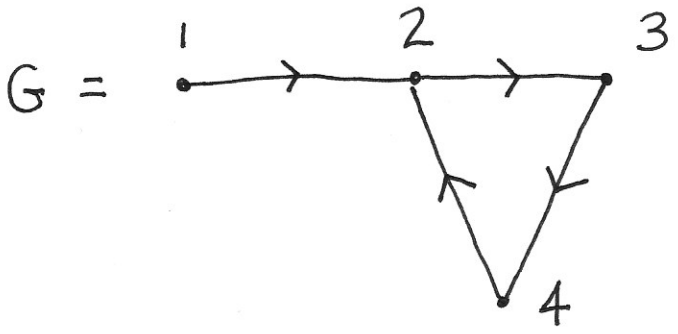
```
1 f(v)
2 foreach neighbor ∈ A[v] do
3 |   if not parent[neighbor] then
4 |   |   parent[neighbor] ← v
5 |   |   DFS-visit(A,neighbor,f,parent)
```

As an optimization and abuse of semantics, the `parent[]` array serves both as the back-pointer from a vertex to the vertex we arrived from, and also as a Boolean to indicate whether the vertex has already been visited, thus avoiding infinite loops. But we must set `parent[s]` to `None` before calling `DFS-visit(...s...)` from external.

What is `None`? It depends on the programming lanugage you are using? Does your programming langauge support heterogeneous arrays? Can you set `parent[3]="None"` and `parent[4] = 3` ? If so, you can use the string `"None"`. Otherwise you may have to use `0` for `None` and be careful that your vertices are numbered starting at `1`. Or perhaps use `-1` for `None`.

However, be careful that you can distinguish `parent[i]` having not yet been assigned, vs `parent[i]` having been assigned `None`.

If the graph is not connected, `DFS-visit` will fail to visit some vertices. Also if the graph is connected but directed, there is a chance of missing some vertices.



In this graph `DFS-visit(...1...)` would visit every vertex, but `DFS-visit(...3...)` would never visit vertex 1. We need an outer loop, and we can make the code cleaner converting `DFS-visit` to a local function.

Algorithm 3: DFS(A,f)

Input : A the adjacency list, mapping each vertex to list of adjacent vertices

Input : f a function to be applied to each vertex

```
1  parent ← ∅ /* empty map                                   */
2  local function DFS-visit(v):
3  |  f(v)
4  |  foreach n ∈ A[v] do
5  |  |  if not parent[n] then
6  |  |  |  parent[n] ← v
7  |  |  |  DFS-visit(n)
   /* main loop of DFS                                      */
8  foreach s ∈ A do
   |  /* indices/keys of A                                  */
9  |  if not parent[s] then
   |  |  /* if s has not yet been visited                   */
10 |  |  parent[s] ← None
11 |  |  DFS-visit(s)
```

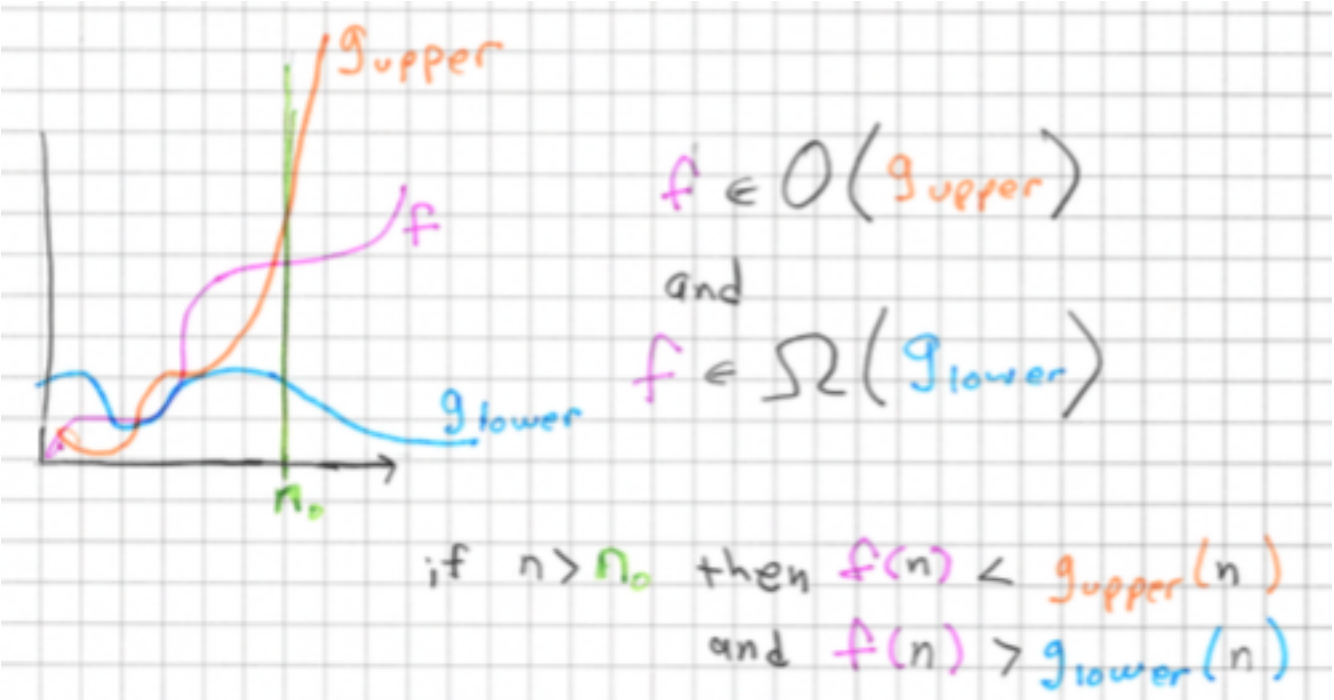
To analyze the complexity of `DFS` we know that `DFS-visit` is called exactly once per vertex, and for vertex v_i we perform $O(1) + O(1) \times$

$degree(v_i)$ amount of time assuming $f(i)$ is constant time.

$$\begin{aligned} Complexity &= \sum_{i=v_1}^{v_n} O(1) + O(1) \times degree(v_i) \\ &= |V| \times O(1) + O(1) \times \sum_{i=v_1}^{v_n} degree(v_i) \\ &= |V| \times O(1) + O(1) \times 2 \times |E| \\ &= O(|V| + |E|) \end{aligned}$$

7 Big-O and Big-Ω Notation

Recall that if $f \in O(g)$, then $f(n)$ is eventually bounded above by $cg(n)$ for $n >> 0$, and if $f \in \Omega(g)$, then $f(n)$ is eventually bounded below by $cg(n)$ for $n >> 0$.



8 Topological Sort

Topological sort is a way to order a set which is constrained by pairwise orderings. In terms of a graph, topological sort is a way of selecting an order of the vertices of an directed graph such that if (a, b) is an edge of the directed graph, then a precedes b in the sequence of ordered vertices. The problem is normally stated as a set of constraints, and you are asked to produce an ordering which does not violate any of the constraints.

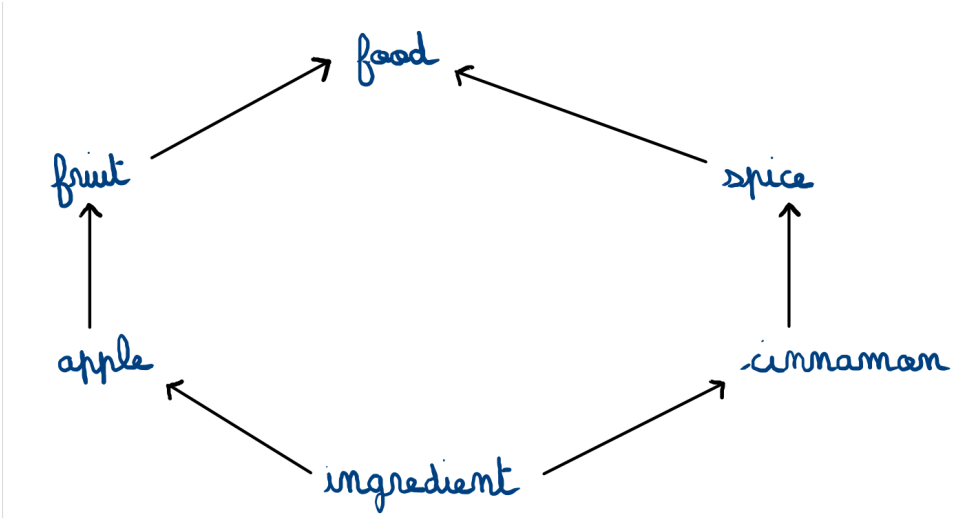
Here is an example from the Common Lisp (CL) programming language. CL provides multiple inheritance. A class is defined by `defclass`; the first argument names to class to be defined, and the second argument lists the direct superclasses.

```
(defclass food () ...)
(defclass fruit (food) ...)
(defclass spice (food) ...)
(defclass apple (fruit) ...)
(defclass cinnamon (spice) ...)
(defclass ingredient (apple cinnamon) ...)
```

In CL, classes are modeled as sets, and subclasses as subsets. If `fruit` is a subclass of `food`, as indicated in the above definition, we write $fruit \subset food$. This leads to the following subset relations.

$$\begin{aligned} ingredient &\subset apple \\ ingredient &\subset cinnamon \\ apple &\subset fruit \\ cinnamon &\subset spice \\ fruit &\subset food \\ spice &\subset food \end{aligned}$$

This leads to a class graph as follows.

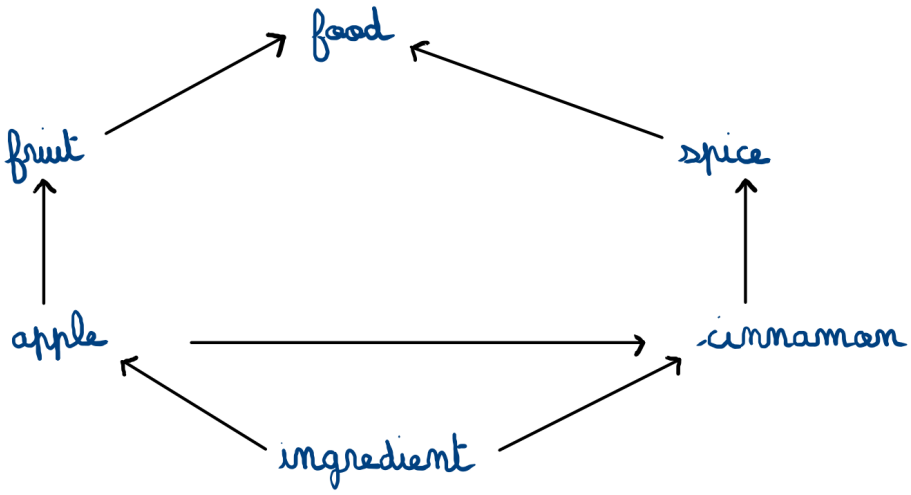


We would like to linearize these classes, respecting all the subclss relations. There are six such orders.

$$ingredient \to apple \to cinnamon \to fruit \to spice \to food$$
$$ingredient \to apple \to cinnamon \to spice \to fruit \to food$$
$$ingredient \to apple \to fruit \to cinnamon \to space \to food$$
$$ingredient \to cinnamon \to apple \to fruit \to spice \to food$$
$$ingredient \to cinnamon \to apple \to spice \to fruit \to food$$
$$ingredient \to cinnamon \to spice \to apple \to fruit \to food$$

It is important when compiling such a program that the CPL (class precedence list) be deterministic, otherwise a running program may have unpredictable behavior. For this reason CL defines additional constraints on the CPL as well as a so-called tie-breaker function.

Since the class definition of `ingredient` above mentions `apple` before `cinnamon`, CL requires that `apple` appear before `cinnamon` in the CPL. This adds an additional constraint in the graph.



However, even with this additional constraint, there are still leaves 3 possible orderings.

$$ingredient \to apple \to cinnamon \to fruit \to spice \to food$$
$$ingredient \to apple \to cinnamon \to spice \to fruit \to food$$
$$ingredient \to apple \to fruit \to cinnamon \to spice \to food$$

To alleviate this ambiguity, CL requires that whenever there is an ambiguity, such as $apple \to cinnamon$ or $apple \to fruit$, we must choose the element whose subclass is *farthest to the right* in the list computed so far. So when selecting between `cinnamon` and `fruit` we ask which of these two has a direct subclass farthest to the right in the list computed so far. The list computed so far is $ingredient \to apple$. In this case $apple \subset fruit$ and $ingredient \subset cinnamon$. As `fruit` is further right than `ingredient`, we select $apple \to fruit$, thus leaving only one possibility.

$$ingredient \to apple \to fruit \to cinnamon \to spice \to food$$

8.1 Kahn Algorithm (1962)

The following is the Kahn algorithm from wikipedia
`https://en.wikipedia.org/wiki/Topological_sorting`

Algorithm 4: Topological Sort (Kahn)

```
1  L → ∅ list that will contain the sorted elements;
2  S → Set of all nodes with no incoming edge ;
3  while S ≠ ∅ do
4  |   remove a node n from S ;
5  |   concatenate n to tail of L ;
6  |   foreach node m with an edge e from n to m do
7  |   |   remove edge e from the graph;
8  |   |   if m has no other incoming edges then
9  |   |   |   insert m into S ;
10 if graph has edges then
11 |   return error (graph has at least one cycle) ;
12 else
13 |   return L (a topologically sorted order)
```

The original Kahn algorithm did not explicitly mention a user defined tie-breaker function.

In terms of the CL Topological sort algorithm for calculating a deterministic CPL, the Kahn algorithm proceeds as follows.

List the constraints:

$$\begin{aligned} ingredient &\rightarrow apple \\ ingredient &\rightarrow cinnamon \\ apple &\rightarrow cinnamon \\ apple &\rightarrow fruit \\ cinnamon &\rightarrow spice \\ fruit &\rightarrow food \\ spice &\rightarrow food \end{aligned}$$

Start with the class being defined. This class is unconstrained, nothing is inheriting from it by definition, because we are in the process of defining it. I.e., initialize

$$CPL = ingredient$$

and mark all constraints $ingredient \rightarrow x$ as satisfied.

~~$ingredient \rightarrow apple$~~	SATISFIED
~~$ingredient \rightarrow cinnamon$~~	SATISFIED
$apple \rightarrow cinnamon$	
$apple \rightarrow fruit$	
$cinnamon \rightarrow spice$	
$fruit \rightarrow food$	
$spice \rightarrow food$	

Now find the set of unconstrained class(es). The only such class is `apple`, so add it to the *end* of the CPL.

$$CPL = ingredient \rightarrow apple$$

And remove all constraints on `apple`.

~~$ingredient \rightarrow apple$~~	SATISFIED
~~$ingredient \rightarrow cinnamon$~~	SATISFIED
~~$apple \rightarrow cinnamon$~~	SATISFIED
~~$apple \rightarrow fruit$~~	SATISFIED
$cinnamon \rightarrow spice$	
$fruit \rightarrow food$	
$spice \rightarrow food$	

Now find the set of unconstrained classes. They are $\{cinnamon, fruit\}$. Kahn's original algorithm didn't consider tie breaking. So by that original algorithm, you could take either `cinnamon` or `fruit` as you prefer.

However, we will apply the tie breaker rule. Look at the satisfied constraint arrows leading to `cinnamon` and to `fruit`, but only those corresponding to subclass relationships: $ingredient \rightarrow cinnamon$ and $apple \rightarrow fruit$. Which one of these originates from the class furthest to the right in the thus far computed list $ingredient \rightarrow apple$? The answer is $apple \rightarrow fruit$, so we take `fruit` rather than `ingredient`, and eliminate constraints on `fruit`.

$$CPL = ingredient \rightarrow apple \rightarrow fruit$$

And remove all constraints on `fruit`.

~~$ingredient \rightarrow apple$~~	SATISFIED
~~$ingredient \rightarrow cinnamon$~~	SATISFIED
~~$apple \rightarrow cinnamon$~~	SATISFIED
~~$apple \rightarrow fruit$~~	SATISFIED
$cinnamon \rightarrow spice$	
~~$fruit \rightarrow food$~~	SATISFIED
$spice \rightarrow food$	

Now, the only unconstrained class is `cinnamon`, so we update the CPL

$$CPL = ingredient \rightarrow apple \rightarrow fruit \rightarrow cinnamon$$

And remove all constraints on `cinnamon`.

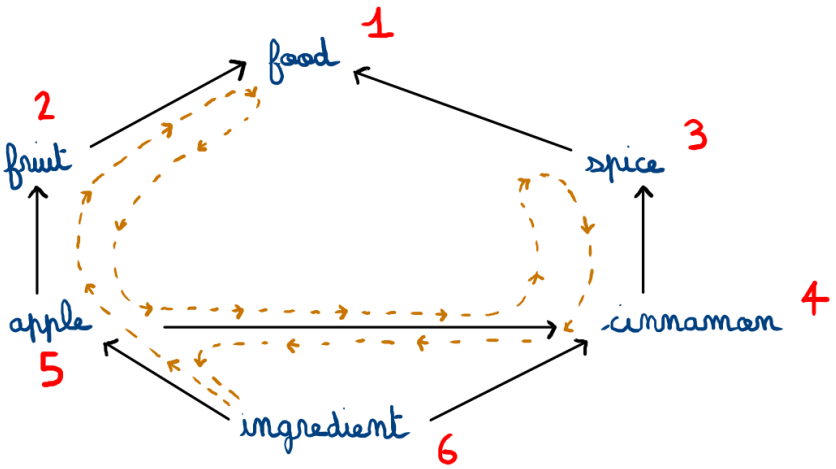
~~$ingredient \rightarrow apple$~~	SATISFIED
~~$ingredient \rightarrow cinnamon$~~	SATISFIED
~~$apple \rightarrow cinnamon$~~	SATISFIED
~~$apple \rightarrow fruit$~~	SATISFIED
~~$cinnamon \rightarrow spice$~~	SATISFIED
~~$fruit \rightarrow food$~~	SATISFIED
$spice \rightarrow food$	

We are now only left with `spice` and `food`, so we append them to the CPL.

$$CPL = ingredient \rightarrow apple \rightarrow fruit \rightarrow cinnamon \rightarrow spice \rightarrow food$$

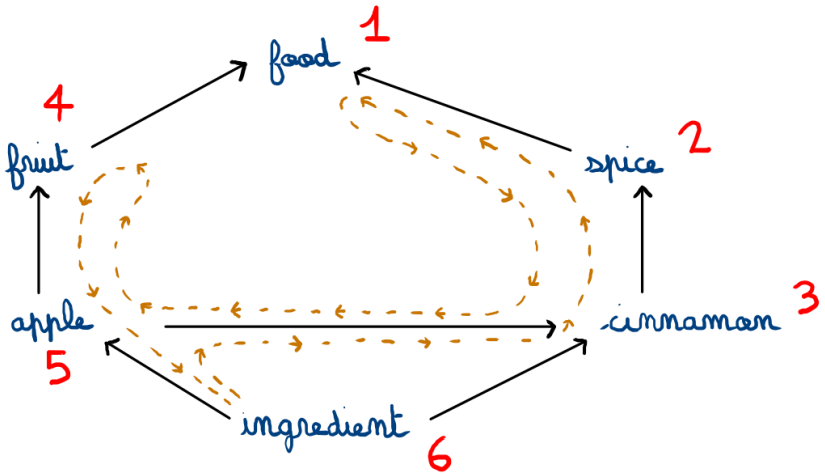
8.2 Tarjan Algorithm (1976)

To obtain a topologically sorted list of vertices of a graph, we may perform a DFS and annotate the verticies after visiting all the neighbors. Depending on the semantics of the arrow directions, either the resulting order or its reverse is the topological order. Since the neighbors are unordered, multiple topological orders are possible.



Now reverse the numbered vertices

6 *ingredient* → 5 *apple* → 4 *fruit* → 3 *cinnamon* → 2 *spice* → 1 *food*



6 *ingredient* → 5 *apple* → 4 *cinnamon* → 3 *spice* → 2 *fruit* → 1 *food*

8.3 Kahn more general than Tarjan

Consider the directed graph $G = (V, E)$ whose vertices are

$$V = (\{a, b, c, d, e, f\}\,.$$

There are many ways to topological sort the vertices of this graph subject to the constraints $\{(a, b), (a, c), (b, c), (b, d), (d, f), (c, e), (e, f)\}$. I.e., a precedes b, a precedes c, b precedes c etc.

- a b c d e f
- a b c e d f
- a b d c e f

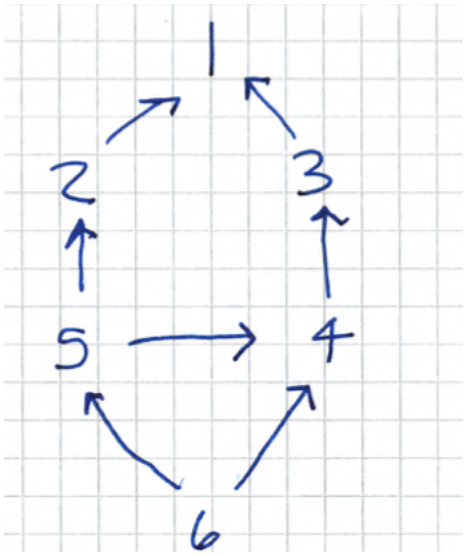
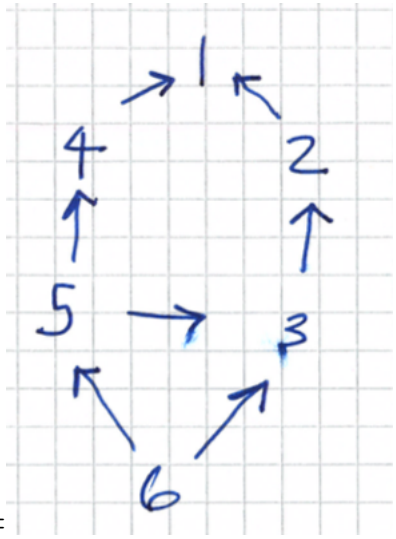
But we would like to find the order of the vertices which comes first in lexographical order. Using the Kahn algorithm we can use a tie-breaker rule that when given a choice of multiple verticies to select, we choose the one that comes first in alphabetical order. This gives

- a b c d e f

However, this order is not obtainable from the Tarjan algorithm because in every DFS of the graph, c and e are adjacent.

8.4 Topological sort as graph isomorphism

If we look at the two graphs in Section 8.2, the act of numbering the nodes is equivalent to finding an isomorphic graph.

$G =$  and $H =$  .

If we examine the adjacency matrix of G and H we see that they are both lower triangular.

$$A_G = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

$$A_H = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

We see that another way to calculate a topological sort is transform the adjacency matrix into a lower triangular matrix by swapping rows and columns.

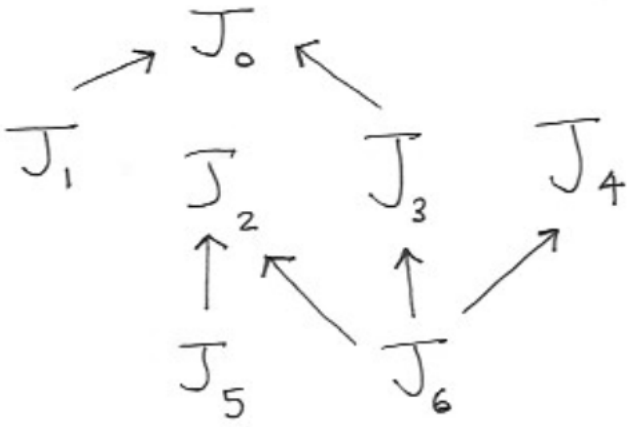
8.5 Example with job dependencies

Given a set of interdependent jobs you wish to run. How can we calculate which order to run the jobs so that all dependencies are met? Suppose that jobs $\{J_0, J_1, J_2, J_3, J_4, J_5, J_6\}$ need to run. But some are dependent on others. We cannot start J_0 until both J_1 and J_3 are finished. I.e., J_1 and J_3 depend on J_0.

Suppose $a \to \{b_1, b_2, \ldots b_n\}$ means a cannot start before $b_1 \ldots b_n$ all finish, or a depends on b. Consider a set of constraints as follows:

$$\begin{aligned} J_1 &\to \{J_0\} \\ J_3 &\to \{J_0\} \\ J_6 &\to \{J_2, J_3, J_4\} \\ J_5 &\to \{J_2\} \end{aligned}$$

The graph looks as follows:

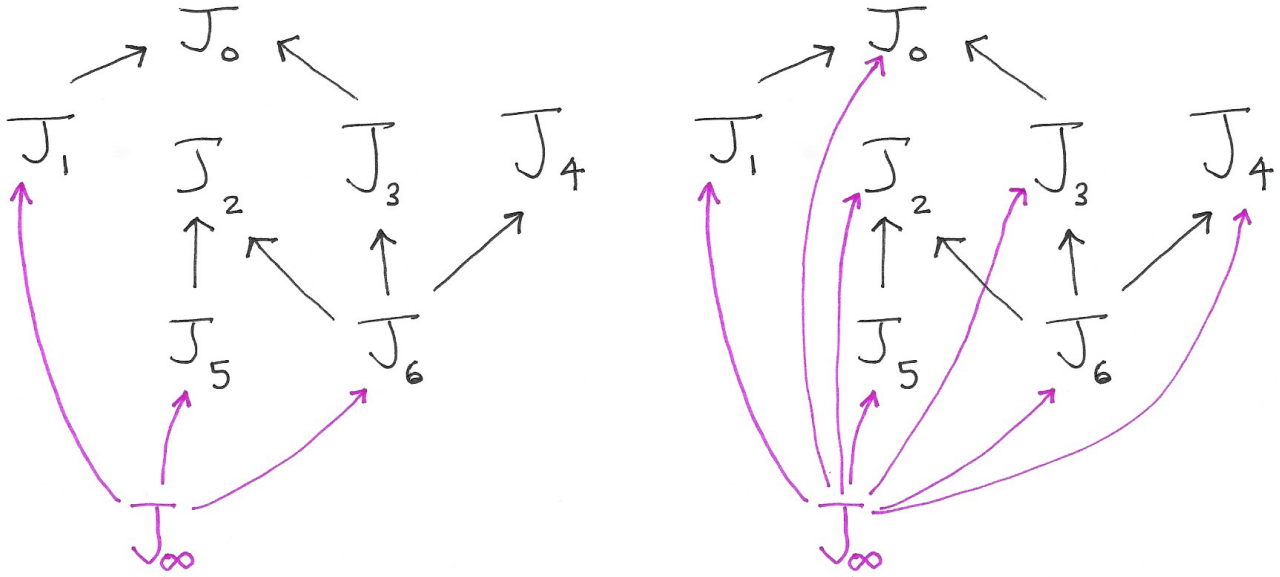


We may use DFS to label the vertices of this graph into a topological order, obeying the constraints.

If the jobs must execute sequentially, i.e., we cannot run multiple jobs at once, then any valid order is just as good as any other. So any topologically sorted order will suffice.

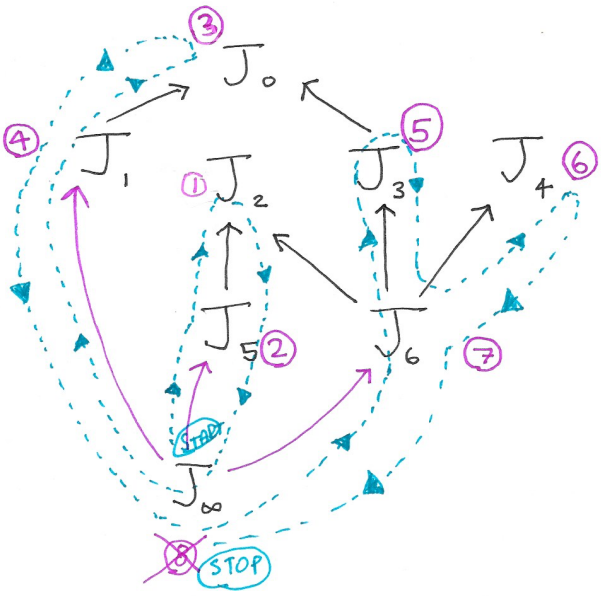
Since we have multiple starting (unconstrained) verticies $\{J_1, J_5, J_6\}$ we don't know where to start the DFS. We may create an *artificial* vertex,

J_∞ connected either to the vertices in $\{J_1, J_5, J_6\}$, or to all the verticies in the graph.



We can then perform the simple DFS starting at vertex J_∞. Labeling each node after all its neighbors have been visited.

$$\{J_2, J_5, J_0, J_1, J_3, J_4, J_6\}$$



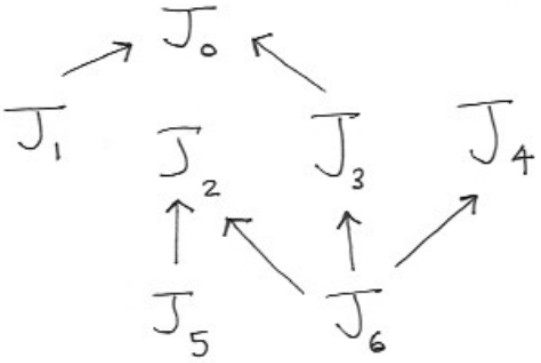
This may result in the following order, depending on the order neighbors are visited.

8.6 Job scheduling with multiple threads

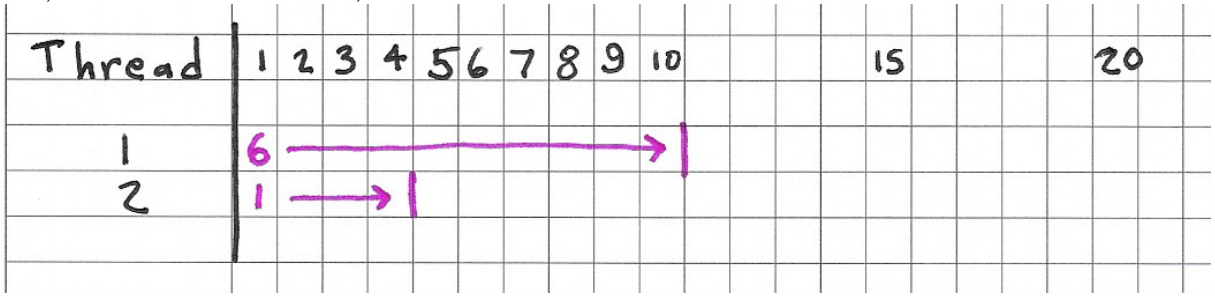
Now we revisit the scheduling problem assuming we have two threads to use–we may run two (but no more) jobs simultaneously, as long as we never violate the inter-job dependencies stated earlier. We further assume that we have estimates on how long the jobs will each take, and we can use this prediction to decide which order to visit the neighbors, and thus which order the jobs are launched .

Job No.	Estimated Time (sec)
0	3
1	4
2	6
3	2
4	5
5	3
6	10

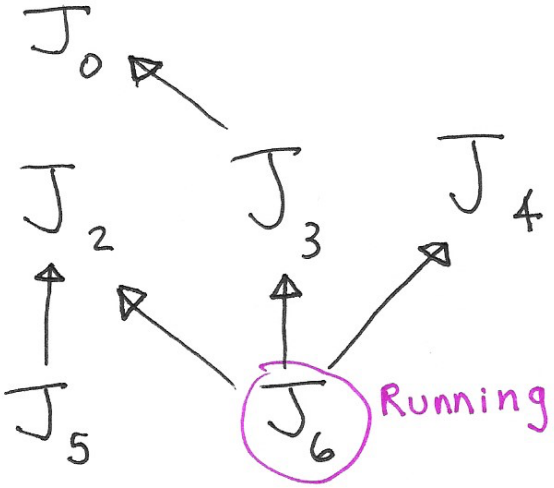
Whenever we have the choice of 2 or more jobs to start, we will start the one projected to run slowest.



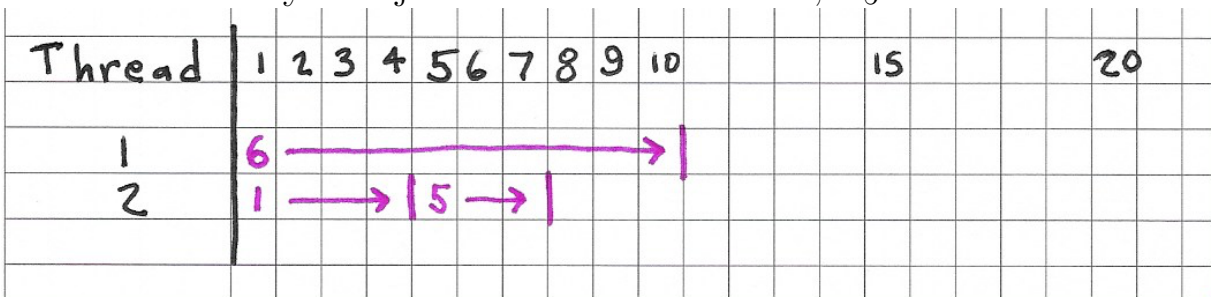
Initially, we can choose among J_1, J_5, and J_6. The slowest two are $J_6, \Delta t = 10$ and $J_1, \Delta t = 4$.



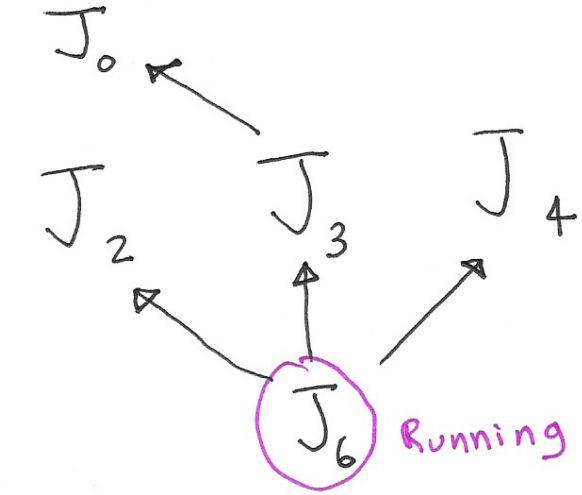
J_1 finishes at $t = 4$, leaving J_6 running and the following graph.



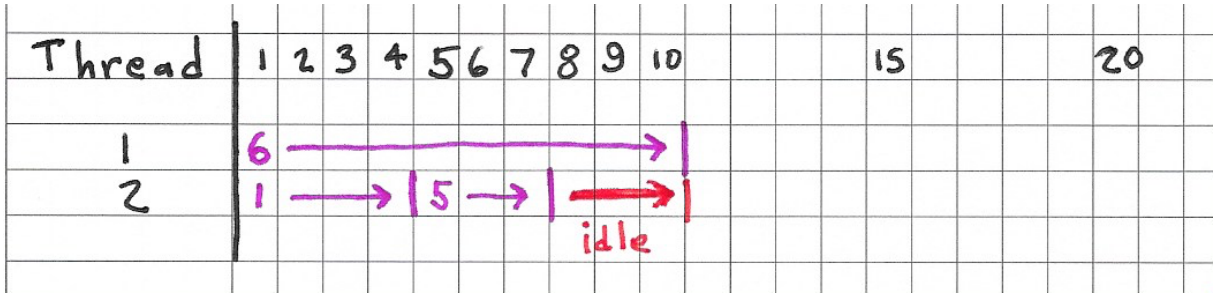
There is only one job which we can start, J_5.



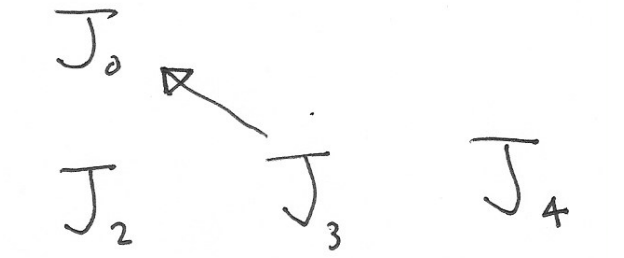
At after $\Delta t = 3$, at time $t = 7$, job J_5 finishes, but J_6 is still running.



At this point we cannot start any new job. We must wait with thread 2 idle until J_6 finishes at time $t = 10$.



After job J_6 finishes at time $t = 10$ the remaining dependency graph looks like the following.



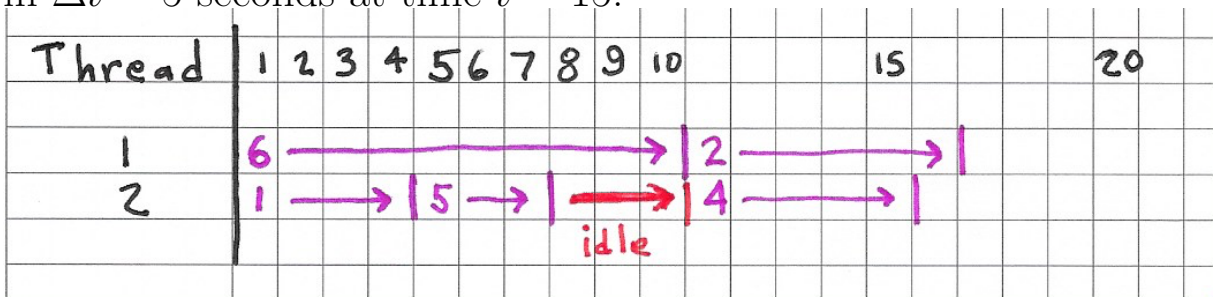
At this point there are two threads free, so we can start two jobs. There are three candidates:

J_2 $\Delta t = 6$

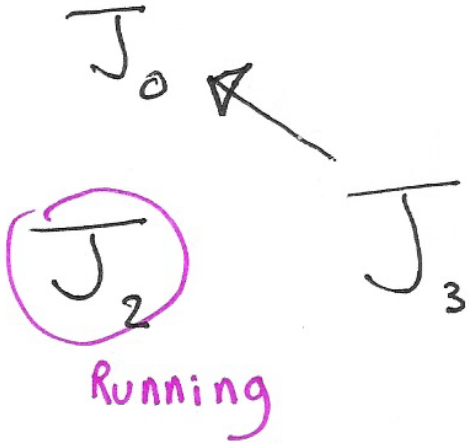
J_3 $\Delta t = 2$

J_4 $\Delta t = 5$

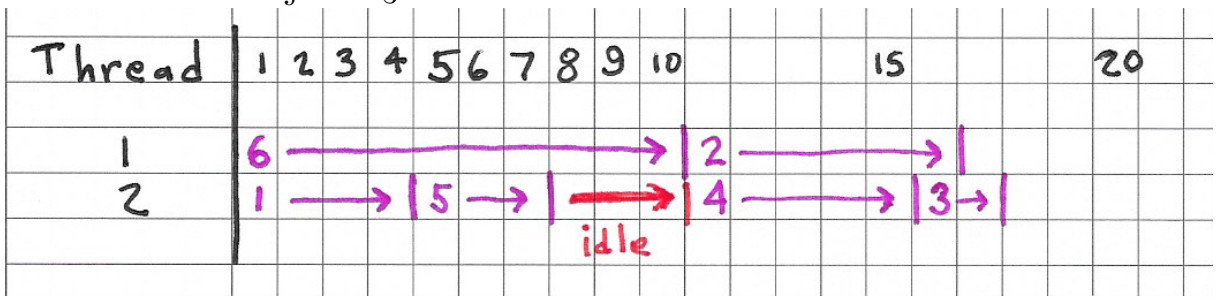
Choosing the slowest two of the three: i.e., J_2 and J_4. Job J_4 finishes in $\Delta t = 5$ seconds at time $t = 15$.



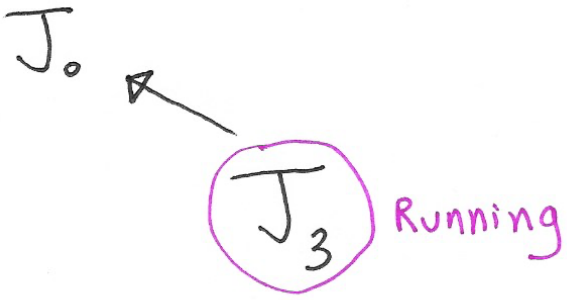
This leaves J_2 running, but J_3 can be launched.



So we start job J_3 at time $t = 15$.

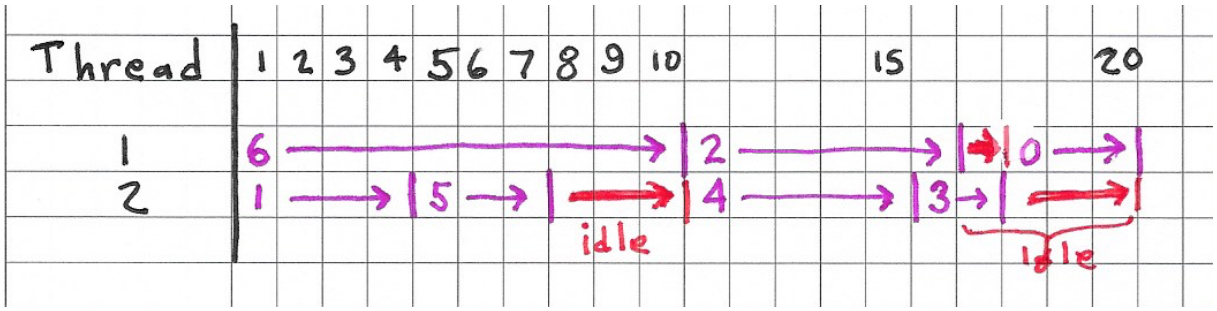


At time $t = 16$, job J_2 finishes, having run $\Delta t = 6$ seconds. This leaves the following dependency graph.



Since job J_3 is still running, we cannot launch any more jobs until J_3 finishes, leaving thread 1 idle for one second until time $t = 17$.

At time $t = 17$ we may launch the final job J_0, leaving one thread idle for $\Delta t = 3$ seconds.



The total wall time of execution is $\Delta t = 20$ seconds, including 7 seconds of idle time.

8.7 Exercises for the student

1. Rework the previous job schedule example using fast before slow as a tie-breaker. What is the wall time for everything to finish? Is it better or worse? How much time is wasted as idle time?

2. What if we consider not the time of the individual job, but the total time of that job and its dependent jobs? For J_6 the time estimate is

$$\max\{J_6 + J_2, J_6 + J_3 + J_0, J_6 + J_4\} = \max\{16, 15, 15\} = 16\,,$$

   and then use the tie-breaker where we start the job with the maximum estimated time. What changes?

3. Rework the example using three threads, with any tie breaker you wish or invent a new one.