FIT2004: Supplementary notes for lecture 10

Graphs: Directed Acyclic Graphs

DANIEL ANDERSON [1]

A directed acyclic graph (DAG) is a directed graph that contains no cycles. DAGs have some special properties that allow us to find interesting information about them, and often admit simple solutions to problems that are very difficult to solve on general graphs. We'll focus on two important properties of DAGs, topological ordering and critical (longest) paths.

Summary: Directed Acyclic Graphs

In this lecture, we cover:

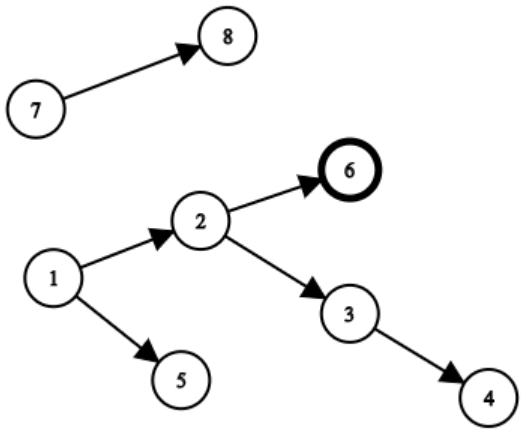
- Directed acyclic graphs (DAGs) and their properties
- The topological sorting problem
- The critical path problem
- The relationship between DAGs and dynamic programming

Recommended Resources: Directed Acyclic Graphs

- CLRS, Introduction to Algorithms, Sections 22.4 and 24.2
- Weiss, Data Structures and Algorithm Analysis, Sections 9.2 and 9.3.4
- `https://visualgo.net/en/sssp`
- `http://users.monash.edu/~lloyd/tildeAlgDS/Graph/DAG/`

Directed Acyclic Graphs

A directed acyclic graph (DAG) is a directed graph that contains no cycles.



A directed acyclic graph.

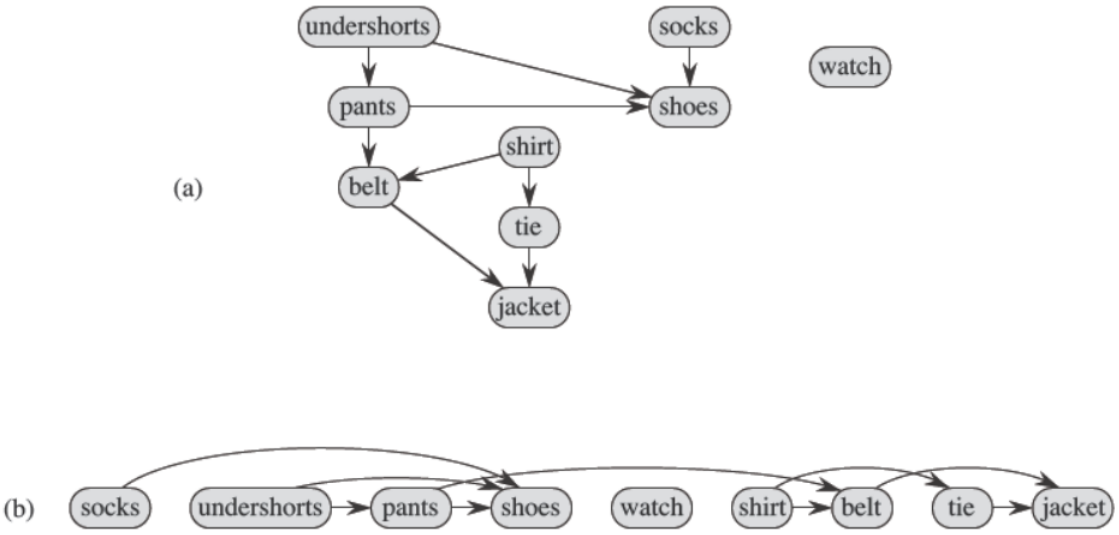
DAGs are useful in particular for representing activities that have prerequisites. For example, the units in your degree and their prerequisites form a DAG, where each edge denotes a unit that has another as a requirement. In project management, the tasks in a project might form a DAG which indicates which tasks must be performed

[1] Faculty of Information Technology, Monash University: `daniel.anderson@monash.edu`. These notes are based on the lecture slides developed by Arun Konagurthu for FIT2004.

before others. The two most fundamental and interesting problems on DAGs are the topological sorting problem and the critical path problem.

The Topological Sorting Problem

Given a DAG $G = (V, E)$, a topological ordering is a permutation of the vertices such that for any directed edge $e = (u, v)$, the vertex u occurs before the vertex v. In other words, if the edges of the graph represent prerequisites, then a topological ordering represents a valid order in which to complete the tasks such that every prerequisite is satisfied. The topological sorting problem is the problem of producing a valid topological ordering for a given DAG.



A DAG (a) representing prerequisites for getting dressed and a correct topological ordering of the DAG (b).
Image source: CLRS

The most well-known algorithm for topological sorting is Kahn's algorithm, which maintains a queue of vertices that are ready to be completed and inserts them one by one into the topological ordering. A vertex is considered ready to be completed if it has no incoming edges. Once a vertex is taken and inserted into the ordering, any outgoing edges that it has are removed and its descendants are checked to see whether they are now ready too.

Algorithm: Topological Sorting using Kahn's Algorithm

```
1: function TOPOLOGICAL_SORT(G = (V[1..N], E[1..N]))
2:     Set order = []
3:     Set ready =  queue of all vertices with no incoming edges
4:     while ready is not empty do
5:         u = ready.pop()
6:         order.append(u)
7:         for each edge (u, v) adjacent to u do
8:             Remove (u, v) from G
9:             if v has no remaining incoming edges then
10:                ready.push(v)
11:            end if
12:        end for
13:    end while
14:    return order
15: end function
```

The advantage of Kahn's algorithm is that the queue used to maintain the ready vertices can be easily swapped out for a priority queue if a particular topological ordering is required. Since the topological ordering is not

necessarily unique, you might want to find one where the vertices are listed in lexicographical order in the case of a tie. This can be done by using a priority queue keyed on the index of the vertices. Since Kahn's algorithm visits each vertex once and removes every edge once, its time complexity is $O(|V| + |E|)$.

Another useful algorithm for computing topological orderings is via a depth-first search. The key idea is that by performing a depth first search, we can visit all of a node's descendants and add them to the result before adding the node in question. Since each node is added strictly after all of its descendants, this will result in us constructing a reverse topological ordering, which can then be reversed to obtain a correct ordering.

Algorithm: Topological Sorting using DFS

```
1: function TOPOLOGICAL_SORT(G = (V[1..N], E[1..N]))
2:     Set order = []
3:     Set visited[1..N] = False
4:     for v = 1 to N do
5:         if visited[v] = False then
6:             dfs(v)
7:         end if
8:     end for
9:     return reverse(order)
10: end function
11:
12: function DFS(u)
13:     visited[u] = True
14:     for each vertex v adjacent to u do
15:         if visited[v] = False then
16:             dfs(v)
17:         end if
18:     end for
19:     order.append(v)
20: end function
```

Since we are just performing a depth-first search and appending $|V|$ items to an array, the time complexity of this algorithm is also $O(|V| + |E|)$. Topological sorting, and in particular, the depth-first search algorithm for topological sorting have a strong underlying relationship to the dynamic programming paradigm, which will be made evident in the next example.

The Critical Path Problem

The critical path problem is the problem of finding the **longest** path in a directed acyclic graph. This problem arises frequently in project management and operations research, where the vertices of a graph represent task completions and the edges represent tasks that must be completed, weighted by the amount of time that it will take to do so. The edges of the graph dictate the prerequisites of the tasks in the project. The longest path in the DAG therefore corresponds to the minimum amount of time in which the project can be completed if all prerequisites are obeyed.

The critical path problem can be formulated as a simple dynamic programming problem. Just like shortest paths, longest paths also admit substructure, any sub-path of a longest path must also be a longest path.

For all vertices $\texttt{v} \in V$, let `longest[v]` denote the length of the longest path that starts at `v`. Our base cases are all of the vertices with no outgoing edges, where `longest[v] = 0`. To determine the longest path that begins at some arbitrary vertex `u`, we simply consider all of the outgoing edges of `u`, and see which one yields the longest total path. In other words, we compute the dynamic program

$$\texttt{longest[u]} = \max_{v \in \texttt{adj[u]}} (w(u, v) + \texttt{longest[v]})$$

The dependencies of the sub-problems are clear from the graph, we must compute the longest path for all descendants of a node before we can compute the result for that node. In other words, the sub-problems are dependant in a reverse topological order.

Algorithm: Longest Path in a DAG

```
1: function LONGEST_PATH(G = (V[1..N], E[1..N]))
2:     Set longest[1..N] = 0
3:     Set order = reverse(topological_sort(G))
4:     for each vertex u in order do
5:         for each outgoing edge (u, v) adjacent to u do
6:             longest[u] = max(longest[u], w(u, v) + longest[v])
7:         end for
8:     end for
9:     return longest[1..N]
10: end function
```

Since we are required to compute the topological order, we might as well solve the problem top-down recursively which removes the need to know the order of dependencies of the sub-problems.

Algorithm: Longest Path in a DAG (Recursive)

```
1: function LONGEST_PATH(G = (V[1..N], E[1..N]), u)
2:     if u has no outgoing edges then
3:         return 0
4:     else
5:         if longest[u] has been computed before then
6:             return longest[u]
7:         else
8:             Set longest[u] = 0
9:             for each outgoing edge (u, v) adjacent to u do
10:                longest[u] = max(longest[u], w(u, v) + longest_path(v))
11:            end for
12:            return longest[u]
13:        end if
14:    end if
15: end function
```

Relationship with Dynamic Programming

Although it might not be immediately obvious, this problem actually illuminates a very interesting fact about dynamic programming that we saw when comparing top-down vs. bottom-up solutions. When computing a dynamic programming bottom-up, it is necessary to know the order in which the sub-problems are dependent on each other so that all of the necessary values are available when needed to compute each subsequent sub-problem. When implementing a solution top-down however, we did not need to know the order of the sub-problems since they would be computed precisely when needed.

The sub-problems can not depend on each other cyclically, so their dependencies on each other actually form a DAG, and the order in which we compute them must be a valid topological order of that DAG. Whenever we compute a solution to a dynamic program using top-down recursion, we are actually performing a **depth-first search on the sub-problem graph**, which means that we are actually producing a reverse topological order as a by-product! This explains why top-down solutions do not need to know a valid order in advance, because they are performing precisely the algorithm that computes one, even if we did not realise it.

Indeed, many dynamic programming problems can be rephrased as path problems on DAGs. Recall the coin change problem for instance. If we construct a graph G where each node corresponds to a dollar amount $\$V$, with edges present between $\$u$ and $\$v$ if and only if $u - v = c_i$ for one of the denominations c_i, then the coin change problem is precisely the problem of finding the shortest path from the desired dollar amount in this DAG to the vertex of \$0.