Chapter 1

DAGs, sinks, extraction order, and dynamic programming

By Sariel Har-Peled, October 27, 2024[1]

> During a native rebellion in German East Africa, the Imperial Ministry in Berlin issued the following order to its representatives on the ground: "The natives are to be instructed that on pain of harsh penalties, every rebellion must be announced, in writing, six weeks before it breaks out."
>
> – – Dead Funny: Humor in Hitler's Germany, Rudolph Herzog.

Version: 0.21

1.1. DAGs

Definition 1.1.1. A ***DAG*** (***directed acyclic graph***) G is a directed graph with no cycles. A vertex $v \in \mathsf{V}(\mathsf{G})$ with no incoming edges into it is a ***source***. A vertex with only outgoing edges is a ***sink***.

Lemma 1.1.2. *A DAG G with $n \geq 1$ vertices, always has at least one source vertex, and at least one sink vertex.*

Proof: If there is a vertex $v \in \mathsf{V}(\mathsf{G})$ with no edges coming into it or outgoing out of it, then it is both a sink and a source vertex. This is the case if $n = |\mathsf{V}(G)| = 1$ (as no self loops are allowed).

Assume the claim is true for all DAGs with k vertices, and consider a DAG G, with $n = k + 1 \geq 2$ vertices, and pick an arbitrary vertex $v_1 \in \mathsf{V}(\mathsf{G})$. If v_1 is no outgoing edges, then it is a sink, and we can declare victory. Otherwise, it has at least one outgoing edge, say from $v_1 \to u$. We set $v_2 = u$. We continue in this fashion – in the ith iteration, we have a vertex v_i. If v_i has no outgoing edges, then it is a sink. Otherwise, it must have at least one outgoing edge, and we set the target of this edge to be the next vertex v_{i+1}. We repeat this process till success.

To see why the process must succeed and stop at a sink, consider the sequence $v_1, v_2, \ldots$ – the vertices in this sequence must all be different, as otherwise the graph G would have a cycle (contradiction). But G is a finite graph with n vertices, and thus, this sequence must be finite, which means that the process stopped at a sink.

Proving that G has a source follows a similar argument. ∎

Observation 1.1.3. *If G is a DAG, and we delete a vertex v from it (and all the edges adjacent to it), then the remaining graph $\mathsf{G} - v$ is a DAG.*

Definition 1.1.4. An ***extraction order*** for a DAG is an ordering of the vertices of G: $v_1, v_2, \ldots, v_n$, such that v_i is a source in $\mathsf{G} - \{v_1, \ldots, v_{i-1}\}$.

This is the topological ordering of the DAG, but this direct definition is simpler.

1.1.1. Computing the extraction order in linear time

Let G be a DAG with n vertices, and m edges. For convenience assume the vertices $\mathsf{V}(\mathsf{G}) = \{1, 2, \ldots, n\}$. The edges of G are a list of m directed edges $e_1, \ldots, e_m$, where an edge is just a pair (i, j) (sometime denoted by $i \to j$). (I.e.., the graph G is specified by the number n, and the list of edges.)

As a preprocessing stage, we create for each vertex $i \in \mathsf{V}(\mathsf{G})$ a list of all the outgoing edges from i, and all the incoming edges into i, for all i. Clearly, this can be done by scanning the edges of the graph once, and for each edge, add it to the two lists it should be listed on.

In particular, for each vertex we now compute its incoming degree $d_{\text{in}}(v)$ (i.e., the length of its incoming edges), and its outgoing degree $d_{\text{out}}(v)$ (i.e., length of the list of all the outgoing edges).

We create a working list W of all the vertices that have incoming degree 0 (i.e., these are the current set of sinks). The list W is initially not empty, by the above lemma. In the ith iteration, for $i = 1, \ldots, n$, the algorithm now extracts an arbitrary vertex v_i from W, output it as the ith vertex in the extraction order, and "delete" it from G (and all its outgoing edges). Deleting it from G is no more than scanning its outgoing edges, and decreasing the incoming degree counters for all the targets of the edges. Specifically, if $v_i \to u$ is an edge, then we decrease $d_{\text{in}}(u)$ by one. If such a counter becomes zero, the algorithm adds it to W.

Running time. Creating the lists takes $O(n + m)$ time, where n and m are the number of vertices and edges in G. Deleting a vertex takes time proportional to its out degree. Overall, the running time is $O(n + m)$.

Correctness. While we do not delete the edges from the adjacency lists, we simulate doing it by decreasing the degree counters. As such, the list W contains in any point in time the list of all the sources in the surviving DAG, and thus the computed extraction order is correct.

1.2. Computing longest distance (# of edges) in a DAG

Given a DAG G with n vertices and m edges, the problem is to compute the longest path in G in the number of edges.

To this end, we compute the extraction order, and renumber its vertices to be $1, \ldots, n$, such that all the edges in the DAG of the form $i \to j$ have the property that $i < j$. In this DAG G, we would like to compute the longest path in number of edges in the path. If $g(i)$ is the length of the longest path starting at i, then we have the recurrence:

$$g(i) = \begin{cases} 0 & i = n \\ 0 & d_{out}(i) = 0 \\ 1 + \max_{i \to j \in E_{\text{out}}(i)} g(j) & \end{cases}$$

Where $E_{\text{out}}(i)$ is the list/set of all outgoing edges from i. This immediately leads to the following DP:

```
LongestPathLen(G: DAG)
  Compute extraction order for G.
  Renumber vertices of G to 1,...,n, s.t.  i → j ∈ E(G) ⟹ i < j
  D[1...n] ← 0
  for i ∈ n−1, n−2, ..., 1 do
    if d_out(i) > 0
      D[i] ← 1 + max_{i→j∈E_out(i)} D[j]
  return max(D[1], ..., D[n−1])
```

Clearly, this algorithm runs in linear time in the size of the graph. Namely, we can compute the longest path length (and the path itself) in the graph in linear time. Contrast this with the general case – computing the longest path in a graph is NP-HARD, and believed to require exponential time.

1.3. Computing longest path between s and t in a DAG

Given a DAG G with n vertices and m edges, and weights on the edges (positive or negative), the problem is to compute the longest path from s to t in G (taking into account the weights of the edges).

We compute the extraction order (or topological ordering), and renumber its vertices to be $1, \ldots, n$, such that all the edges in the DAG of the form $i \to j$ have the property that $i < j$. For simplicity, let s (resp. t) be the number assigned to the source (resp. t) vertex in this numbering.

In this DAG G, we would like to compute the longest path in the weight of edges from s to t. If $h(i)$ is the (weighted) length of the longest path starting at i that arrives to t, then we have the recurrence:

$$h(i) = \begin{cases} -\infty & i < s \text{ or } i > t \\ 0 & i = t \\ \infty & d_{out}(i) = 0 \\ \max_{i \to j \in E_{\text{out}}(i)} \Big(w(i \to j) + h(j) \Big) & otherwise. \end{cases}$$

Where $E_{\text{out}}(i)$ is the list/set of all outgoing edges from i. This immediately leads to the following DP:

```
//Compute longest path from v_s to v_t in G
LongestPath(G: DAG, v_s, v_t)
  Compute extraction order for G.
  Renumber vertices of G as 1,...,n according to extraction order
  // Now:  ∀ i → j ∈ E(G) ⟹ i < j
  s ← number of v_s in numbering
  t ← number of v_t in numbering
  D[1...n] ← −∞
  D[t] = 0
  for i ∈ t−1, ..., s do
    E_i ← {(i → j) ∈ E_out(i) | j ≤ t}.
    if |E_i| > 0
      D[i] ← max_{(i→j)∈E_i} ( w(i → j) + D[j] )
  return D[s]
```

Running time. Initialization, topological ordering (or equivalently extraction order) takes $O(n+m)$ to compute. The later parts of the algorithm spends $O(1)$ time per vertex/edge, and as such the total running time is $O(n+m)$.

Correctness.

Lemma 1.3.1. *In the end of the algorithm execution, for $i \in [\![s \,..\, t]\!]$, we have that $D[i]$ is the length of the longest path from i to t in G.*

Proof: The proof is by induction.

Induction base: For $i = t$ the claim is immediate as this distance is initialized to zero.

Induction hypothesis: Assume the claim is correct for all $i \geq k$.

Induction step: We now prove the claim for $i = k - 1$. If i has not outgoing edges than there is no path from i to t, and $D[i]$ is initialized to $-\infty$, and this distance is never updated, so it is the correct value.

Otherwise, let (say) there are ℓ edges outgoing from i with index $\leq t$: $U = \{i \to j_1, i \to j_2, \ldots, i \to j_\ell\}$. Assume the longest path from i to t goes through the first edge. Clearly, by induction $D[j_1]$ is computed correctly, and thus the value $\alpha = w(i \to j_1) + D[j_1]$ would considered by the algorithm. Furthermore, any value computed that is not $-\infty$ computed by the algorithm corresponds to a real path that is realizable. It follows, that the algorithm would set $D[i]\alpha$ as desired, as all other values considered must be smaller. ∎

Summary.

Theorem 1.3.2. *Given a DAG G with n vertices and m edges, and weights on the edges (positive or negative), the longest path from s to t in G (taking into account the weights of the edges) can be computed in $O(n+m)$ time.*