Lecture 6

• Algorithmic complexity

• Directed graphs, DAGs

• DAG structure

• Dynamic programming to find highest weight paths in WDAGs
Algorithmic Complexity

- Basic questions about an algorithm:
  - how long does it take to run?
  - how much space (RAM or disk space) does it require?
- Would like precise function $f(N)$, e.g.
  $$f(N) = .05 N^3 + 50.7 N^2 + 6.03 N$$
  for
  - running time in secs, or
  - space in kbytes,
  as function of the size $N$ of input data set.
- But
  - tedious to derive &
  - depends on (often uninteresting – though important!) hardware & software implementation details.
Instead, more customary to give “the” **asymptotic complexity**, i.e. expression \( g(N) \) such that

\[
C_1 g(N) < f(N) < C_2 g(N)
\]

for some constants \( C_1 \) and \( C_2 \), and \( N \) large enough.

This is written \( O(g(N)) \), where notation \( O() \) means “up to an unspecified multiplicative constant”.

– e.g. for the \( f(N) \) above, the dominating term for large \( N \) is \( .05 N^3 \), so

  • can take \( g(N) = N^3 \)

  • asymptotic complexity = \( O(N^3) \).
• Can be misleading, since
  – for small $N$ a different term may dominate
    • (e.g. 2$^d$ term in above example much more important for $N < 1000$)
  – size of constant may be quite important
    • (big difference between .05 and 5,000,000!)
    • e.g. BLAST and Smith-Waterman both $O(N^2)$, but size of constant enormously different
• but very useful as rough guide to performance.
• Cache misses (non-cache memory accesses) and disk accesses often dominate running time, yet are ‘invisible’ to complexity analysis (because affect constant factor only)
• Another limitation to complexity analysis:
  – time or space requirement may depend on specific characteristics of input data.

• Usually give “worst case” complexity
  – applies to the worst data set of a given size,

  \[\textit{but}\]

  – in biological situations the \textit{average biologically occurring case} is
    • more relevant
    • often much easier than worst case (which may never arise in practice), or even “average case” in some idealized sense.
• Proof that a problem is $NP$-hard
  – (has complexity very likely greater than any polynomial function of $N$ and therefore effectively unsolvable for large $N$)

  can be useful in guiding search for more efficient algorithms

  but can also be misleading, since

  – we need *some* solution anyway, for data sets occurring in practice

  – average *biologically relevant* case may be quite manageable
Directed Graphs

• A directed graph is a pair \((V, E)\) where
  – \(V\) is a finite set of vertices, or nodes.
  – \(E\) is a set of ordered pairs (called edges) of vertices in \(V\).

• An edge \((v_i, v_j)\) is said to leave \(v_i\) and to enter \(v_j\).
  – \((v_i\) and \(v_j\) are vertices)

• in-degree of a vertex = \# edges entering it;
• out-degree = \# edges leaving it.
Example:

- $V = \{1,2,3,4,5,6\}$,
- $E = \{(1,2), (1,3), (2,4), (4,1), (5,3), (3,1)\}$
- Vertex 3 has in-degree 2 and out-degree 1.
Paths and Cycles

- A **path** of **length** $k$ in $G$ from $u$ to $u'$ (vertices) is
  - a sequence $P$ of vertices $(v_0, v_1, \ldots, v_k)$ such that
    - $v_0 = u$,
    - $v_k = u'$, and
    - $(v_{i-1}, v_i)$ is an edge for $i = 1,2, \ldots, k$.
- A path can have length 0.
- We write $|P| = k$.
- A **cycle** is a path of length $\geq 1$ from a vertex to itself.
- In example at right, 
  - $(1,2,4)$ is a path,
  - $(1,3,5)$ is not, and
  - $(1,2,4,1)$ and $(1,3,1)$ are cycles.
• Can join
  – any path \((u, \ldots, v)\) from \(u\) to \(v\), to
  – any path \((v, \ldots, w)\) from \(v\) to \(w\)

to get a path \((u, \ldots, v, \ldots, w)\) from \(u\) to \(w\).
DAGs

- A *directed acyclic graph* (DAG) is a directed graph with no cycles.
- In a DAG, for distinct nodes $v_i$ and $v_j$, we say
  - $v_i$ is a *parent* of $v_j$, and $v_j$ is a *child* of $v_i$, if
    - there is an edge $(v_i, v_j)$
  - $v_i$ is an *ancestor* of $v_j$, and $v_j$ is a *descendant* of $v_i$, if
    - there is a path from $v_i$ to $v_j$
- In a DAG the length of a path cannot exceed $|V| - 1$,
  - (where $|V|$ = total # vertices in graph)
  because
  - in a path of length $\geq |V|$,
    - at least one vertex $v$ would have to appear twice in the path;
  - but then there would be a path from $v$ to $v$, i.e. a cycle.
Structure of DAGs

• Define the *depth* of a node $v$ in $V$ as:
  – the length of the longest path ending at $v$;
  by above, the depth is well-defined and $\leq |V| - 1$.

• *Every descendant $w$ of a node $v$ has higher depth than $v$:*
  If
  – $(u, ..., v)$ is path of length $n = \text{depth}(v)$ ending at $v$, and
  – $(v, ..., w)$ is path from $v$ to $w$,
  then $(u, ..., v, ..., w)$ is a path of length $> n$ ending at $w$, so $\text{depth}(w) > n$. 
• **Every node \( v \) of positive depth has a parent of depth exactly one less:**
  
  – Let \((u, ..., v', v)\) be path of length \( n = \text{depth}(v) \) ending at \( v \).
  
  – Then \( v' \) is a parent of \( v \).
  
  – Since \((u, ..., v')\) has length \( n - 1 \), \( \text{depth}(v') \geq n - 1 \).

  – Since also \( \text{depth}(v') < n \) (because \( v \) is a descendant of \( v' \)), \( \text{depth}(v') \) is exactly \( n - 1 \).

• **The nodes on any path are of increasing depth.**
Important special cases:

• A \textit{(rooted) tree} is a DAG which
  – has unique depth 0 node (the \textit{root}), \textit{and}
  – every other node has in-degree 1
    • (i.e. has a unique parent, of depth one less than that of the node).

• A \textit{binary tree} is a tree in which
  – every node has out-degree at most 2.

• A \textit{linked list} is a tree in which
  – every node has out-degree at most 1
  – or equivalently, a DAG in which \exists at most one node of each depth
binary tree

linked list

\begin{itemize}
  \item $v_0$
  \item $v_1$
  \item $v_2$
  \item $v_3$
  \item $v_4$
  \item $v_5$
  \item $v_6$
  \item $v_7$
  \item $v_8$
\end{itemize}
Remarks on Depth Structure

• For *dynamic programming* algorithm
  – we need an order \( v_1, v_2, \ldots, v_n \) for the vertices
    • (not a path!)
      in which parents appear before children.
  – From the above, *depth order*
    • (in which depth 0 nodes are listed first, then depth 1 nodes, etc.)
      is such an order.
  – In general there are many other such orders.

• We haven’t given constructive procedure for finding the depths of all vertices.
  – For an arbitrary DAG, can be done in \( O(|V| + |E|) \) time;
  – we omit algorithm, since for DAGs related to sequence analysis, the depth structure is obvious.
Weighted Directed Graphs

- A **weighted directed graph** is
  - a directed graph \((V, E)\) together with
  - a function \(w\) from \(E\) to the real numbers,
    - i.e. with a numerical **weight** \(w(e)\) (which may be positive, negative, or 0) associated to each edge \(e\).

A weighted DAG is called a WDAG.

- The (**sum**) **weight of a path** is defined to be the sum of the weights on the edges of the path.
- Similarly, the **product weight of a path** is the product of the edge weights
  - usually only consider this when all weights are non-negative.

- weight of a path \(P\) is written \(w(P)\)
- For a path of length 0 (i.e. consisting of a single vertex):
  - the sum weight is 0
  - the product weight is 1
Highest Weight Paths on WDAGs

- **Problem**: find a path with the highest possible weight.

- **Solution**:
  - “Brute force” approach
    - i.e. simply enumerating all possible paths and comparing their weights
    - is usually impractical (too many paths!)
  - Instead, use the method of *dynamic programming* (‘The Fundamental Algorithm of Computational Biology’).
• Let $P_n = (v_0, v_1, \ldots, v_n)$ be a path of highest weight.
• Then for each $k < n$, the sub-path $P_k = (v_0, v_1, \ldots, v_k)$ must have highest weight of all paths ending at $v_k$, because
  – if $Q = (u_0, u_1, \ldots, v_k)$ were another path ending at $v_k$ and having higher weight than $P_k$,
  – then the path $(Q, v_{k+1}, \ldots, v_n)$ would have weight
    \[
    w((Q, v_{k+1}, \ldots, v_n)) = w(Q) + w((v_k, \ldots, v_n))
    > w(P_k) + w((v_k, \ldots, v_n)) = w(P_n),
    \]
    contradicting assumption that $P_n$ has highest weight.
Subpaths of a highest-weight path can’t be improved:

If this has highest weight of all paths ending at \( v_5 \) then...

This must have highest weight of all paths ending at \( v_4 \)
• So generalize the problem as follows:
  • find, for *each* vertex $v$, the highest weight of all paths ending at $v$ – call this $w(v)$
• Can find $w(v)$ in single pass through $V$, as follows:
  – process the $v$ in depth order (or any order in which parents precede children)
  – if $v$ has no parents, $w(v) = 0$ (the only path ending at $v$ is $(v)$).
  – for any other $v$, except for the path $(v)$ (which has weight 0), any path ending at $v$ is of form $(v_0, v_1, \ldots, v_k, u, v)$. Then
  – $u$ is a parent of $v$, so $w(u)$ has already been computed, and

    $$w((v_0, v_1, \ldots, v_k, u, v)) \leq w(u) + w((u,v))$$

    with equality for an appropriate choice of $v_i$.
  – Therefore we may compute $w(v)$ as

    $$w(v) = \max(0, \max_{u \in \text{parents}(v)} (w(u) + w((u,v))))$$
Example
$w(v)$ – depth 0 nodes
$w(v)$ – depth 1 nodes
$w(v) - \text{depth 2 nodes}$
$w(v) –$ depth 3 nodes
\( w(v) - \text{depth 4 nodes} \)
• To reconstruct best path, need “traceback” pointer to immediate predecessor of \( v \) in best path:

\[
T(v) = \begin{cases} 
  v & \text{if } w(v) = 0 \\
  \arg \max_{u \in \text{parents}(v)} (w(u) + w((u,v))) & \text{if } w(v) \neq 0 
\end{cases}
\]

– in preceding graph, \( T(v) \) is the \textit{parent} on \textit{red edge} coming into \( v \)
  • if more than one such edge, pick one at random;
  • if no such edge, \( T(v) = v \)

• Sometimes useful to record \textit{beginning} of best path:

\[
B(v) = \begin{cases} 
  v & \text{if } w(v) = 0 \\
  B(T(v)) & \text{if } w(v) \neq 0 
\end{cases}
\]
• Then highest weight of any path in graph is
  \[ \max_{v \in V} (w(v)) \]
  – updated as each node is visited
  • indicated by [] in preceding graph –
    and so doesn’t require additional pass through vertices
• if \( u = \arg\max_{v \in V} (w(v)) \), can reconstruct highest weight
  path by tracing back from \( u \), using \( T \):
  – path ends at \( u \);
  – immediate predecessor of \( u \) is \( T(u) \);
  – predecessor of \( T(u) \) is \( T(T(u)) \); etc.
  – stop when \( T(v) = v \).

• In preceding example, highest weight is 6 and \( u = v_{11} \)
Dynamic programming on WDAGs

Depth 0

Depth 1

Depth 2

Depth 3

Depth 4
Complexity of Dynamic Programming

- Time to find a best path is $O(|E| + |V|)$:
  - in initial pass, visit each node, and each edge into that node: $O(|E| + |V|)$
  - in traceback, visit subset of nodes, and unique edge from each node: $O(|V|)$

(Complexity to find all highest weight paths can be higher)

For very large graphs, even $O(|E| + |V|)$ may be unacceptable!
• Space requirements:
  – If only want *weight* of best path, and beginning and end, then
    – don’t need $T(v)$, and
    – only need retain $w(v)$ and $B(v)$ until have processed all children of $v$ (or when best path found so far ends at $v$).

  Space depends on graph structure, but usually $<< O(|V|)$.

  – If want path itself, must store $T(v) \forall v$
    – space $= O(|V|)$
    – $\exists$ algorithms (for some graphs) to reduce this, but may take more time.
Implementing Dynamic Programming in a Computer Program

• Storing entire graph has space complexity = \( O(|V| + |E|) \)

• If graph has regular structure, can often “create” and process vertices and edges on the fly, without storing in memory
  – cf. edit graph (to be defined later) for aligning sequences
Same dynamic programming approach can be used to find:

1. Highest product weight path (if weights are \( \geq 0 \))

2. Highest weight path that
   - \textit{starts} in particular subset \( V' \) of vertices,
     - don’t consider paths that start outside \( V' \):
       i.e. when computing \( w(v) \), don’t consider trivial path unless \( v \in V' \)
   - and/or \textit{ends} in particular subset \( V'' \)
     - only scan for the maximum \( w(v) \) over \( V'' \)

3. Sum of product weights of all paths ending at particular vertex
   - \textit{sum} over all edges coming into \( v \), instead of \textit{maximizing}
   - this useful for probability calculations

• Will use the above variants later!