Theorem (Efficiency of Ford’s Algorithm)

If \( \mathbf{c} \) is integer-valued and \( G = (V, A) \) has no negative-cost directed circuit, then Ford’s Algorithm terminates after at most \( Cn^2 \) steps, where \( n = |V| \) and

\[
C = m(\mathbf{c}) = 1 + 2\|\mathbf{c}\|_{\infty} \quad \text{with} \quad \|\mathbf{c}\|_{\infty} = \max\{|c_{(a,b)}| : (a,b) \in A\}.
\]

This is a typical statement of an efficiency result: We establish an upper bound for (some abstraction of) the running time, with a simple formula. Note we do not predict the precise running time; the algorithm could be much faster than that.

Proof.

- The first correction step, leading to a finite potential value \( y_v \) of a vertex, gives \( y_v \) that is at most \( \|\mathbf{c}\|_{\infty}(n - 1) \), because it is the cost of a simple directed path (with at most \( n - 1 \) arcs!) from the root.
- In every later correction step, \( y_v \) gets reduced by at least 1, because \( \mathbf{c} \) is integral.
- In the end, the potential value \( y_v \) is the cost of a least-cost path; this may be negative, but we certainly have \( y_v \geq -\|\mathbf{c}\|_{\infty}(n - 1) \).
- Thus at most \( 1 + 2 \cdot \|\mathbf{c}\|_{\infty}(n - 1) \leq Cn \) steps per vertex.
- Hence, at most \( Cn^2 \) correction steps in total.
The theorem establishes that Ford’s Algorithm is a **pseudo-polynomial algorithm**, i.e., its running time is bounded above by a polynomial expression in the “dimensions” (such as \( n_k \)) and the absolute values of its input data.

Because the bound is monotonous in \( C \) and \( n \), it is convenient to interpret this bound as an upper bound on the running time of the **worst case** that can happen among all problems \((G,c)\) with \(|V| \leq n\) and and \( m(c) \leq C \).

In a homework exercise, you show that there is a one-parameter family \([ (G_k = (V_k,A_k), c^k) : k \in \mathbb{N} ] \) of networks with \( n_k = |V_k| = 2k + 4 \) vertices and \( C_k = 2^k \), such that Ford’s Algorithm (with a specific, clever, evil way of choosing which incorrect arc should be corrected) takes more than \( 2^k \) steps.

This shows that the upper bound is “not too far off” from the worst case.

**Better efficiency classes:**

- We are not happy with pseudo-polynomial algorithms, because for the same graph \( G \), the running time might grow quickly if we just use “large numbers” (it might grow **exponentially in the number of digits** of the data \( c_{(a,b)} \)).
- Better are **polynomial algorithms**, where the running time is allowed to grow polynomially in the “dimensions” (such as \( n = |V| \) and \( m = |A| \)), but only **polynomially in the number of digits** of the data (such as \( c_{a,b} \)).
- Even better are **strongly polynomial algorithms**, where the worst-case running time (#steps) is allowed to depend only on the dimensions, not on the data.
In a homework exercise, we saw that there are examples, in which a particular order of correcting arcs leads to very bad performance (many iterations). Let’s try to find an order that is better.

Let’s rewrite the body of the while loop like this:

1. Choose an arc \((v, w)\).
2. If \((v, w)\) is incorrect, then correct it, updating predecessor information.

We call this verifying arc \((v, w)\).

We denote by \(S = ((v_1, w_1), (v_2, w_2), \ldots, (v_k, w_k))\) a sequence of arcs that we verify during Ford’s Algorithm.

Important observation:

**Lemma**

In Ford’s Algorithm, after verifying the sequence \(S\) of arcs, for all directed paths \(P\) from \(r\) to \(v\) that are embedded in \(S\), i.e.,

the arcs of \(P\) appear as a subsequence of \(S\) (i.e., in the right order, but not necessarily consecutively)

we have \(y_v \leq c(P)\).
Proof.

Let $P = (v_0, a_0, v_1, a_1, v_2, \ldots, a_K, v_k)$ with $v_0 = r$ and $v_k = v$ be a directed path that is embedded in $S$.

- After verifying $a_0$ in some iteration $q_0$, we have
  \[ y_{v_1}^{(q_0)} \leq y_{v_0}^{(q_0 - 1)} + c_{v_0, v_1} = c_{v_0, v_1}. \]

- Then, after verifying $a_1$ in iteration $q_1 > q_0$, we have
  \[ y_{v_2}^{(q_1)} \leq y_{v_1}^{(q_1 - 1)} + c_{v_1, v_2} \quad \text{(verification)} \]
  \[ \leq y_{v_1}^{(q_0)} + c_{v_1, v_2} \quad \text{($y_{v_1}$ possibly decreased between $q_0$ and $q_1 - 1$)} \]
  \[ \leq c_{v_0, v_1} + c_{v_1, v_2}. \quad \text{(per above)} \]

- and so on: induction yields $y_v \leq c(P)$.

\[ \square \]
Now let us design a sequence $S$ of arcs such that every possible minimum-cost path is embedded in $S$.

Minimum-cost paths are simple directed paths, so they contain at most $n - 1$ arcs (where $n = |V|$).

Simple construction: Let $S_i$ be any ordering of the arcs $A$. Then the sequence

$$S = (S_1, \ldots, S_{n-1})$$

has the desired property. We say that we make $n - 1$ passes through the graph.

We call this refined algorithm the **Ford–Bellman algorithm**.
Improving Ford’s Algorithm: Ford–Bellman [1958]

Ford–Bellman algorithm

**Input:** A digraph \( G = (V, A) \) with arc costs, starting node \( r \)

**Output:** If \( G \) has a negative cycle, output “negative cycle!”; otherwise output a predecessor vector \( p \), which encodes minimum-cost paths from \( r \) to all other nodes.

1. Initialize \( y \) and \( p \)
2. Set \( i := 0 \)
3. While \( i < n \) and \( y \) is not a feasible potential:
   - Set \( i := i + 1 \)
   - For \((v, w) \in A\) (in arbitrary order):
     - If \((v, w)\) is incorrect, then correct it, updating predecessor information.
4. If \( i = n \), return “negative cycle!”; otherwise, return \( p \).
Theorem (Correctness and efficiency of Ford–Bellman)

The Ford–Bellman algorithm is correct. It terminates after at most \( m \cdot n \) arc verifications.

Proof.

Correctness follows from the above lemma:

- If there is no negative cycle, after the arc-verification sequence \( S \), for every minimum-cost path \( P_v \) we have \( y_v \leq c(P_v) \) because \( P_v \) is embedded in \( S \). Thus the \textit{while} loop terminates with \( i < n \).

- If there is a negative cycle, we know there does not exist a feasible potential, so the \textit{while loop} terminates because of \( i = n \).

The bound on the number of verifications is obvious.

- Thus it is a strongly combinatorial algorithm.
- In the general case, no algorithm with a better running time bound is known.
The case of topologically sortable graphs

- Suppose that we can order the vertices of the directed graph $G = (V, A)$ “from left to right”, so that all arcs go from left to right.
- In other words, suppose there is an ordering $v_1, \ldots, v_n$ of $V$ such that for any arc $(v_i, v_j) \in A$ we have $i < j$.
- Such an ordering is called a topological sort of $G$.

Observation:
- All directed paths in $G$ are embedded in the arc-correction sequence $S = (L_1, \ldots, L_n)$ where $L_i$ is an arbitrary ordering of the arcs leaving vertex $v_i$.
- Therefore, Ford’s Algorithm has the correct answer after running this arc-correction sequence $S$.

Where do topologically sortable graphs come from?
- In some applications, the graphs have a natural topological sort because the vertices are layered, for instance by “time”, and there are only arcs that go from “now” to “later”.
- This is related to the idea of dynamic programming (with respect to time or other “increasing” parameters)
- Which (other) directed graphs have a topological sort? Complete answer on the next slide.
Characterization of Topological Sortability

**Lemma (Topological Sortability Lemma)**

A directed graph has a topological sort if and only if it is **acyclic** (has no directed circuit)

**Proof of the Topological Sortability Lemma.**

1. If there is a topological sort \( v_1, \ldots, v_n \), there clearly is no directed circuit.
2. For the converse, we first show that there is a suitable choice for \( v_1 \), i.e., a vertex with no predecessor, i.e., no incoming arc.
   - Suppose, to the contrary, that every vertex has a predecessor.
   - Let \( w_1 \in V \) be arbitrary; pick a predecessor of \( w_1 \) and call it \( w_2 \).
   - Pick a predecessor of \( w_2 \) and call it \( w_3 \).
   - This produces an infinite sequence \( w_1, w_2, \ldots \in V \).
   - However, \( V \) is finite, so there is some \( i < j \) with \( w_i = w_j \).
   - Thus there is a directed circuit \((w_j, w_{j-1}, \ldots, w_{i+1}, w_i)\) in \( G \), a contradiction.
3. Continue inductively on a graph \( G_1 \) where we have removed \( v_1 \) (and the arcs originating from \( v_1 \)).

This proof suggests an efficient algorithm that constructs a topological sort or detects a directed circuit (homework).
Bellman’s Algorithm for the Acyclic Case

Bellman’s Algorithm ("Dynamic Programming")

Input: A digraph $G = (V, A)$ with arc costs, starting node $r$

Output: If $G$ has a cycle, output “cycle!”; otherwise output a predecessor vector $p$, which encodes minimum-cost paths from $r$ to all other nodes.

1. Find a topological sort $v_1, \ldots, v_n$ of $G$; if there is none, return “cycle!”.
2. Initialize $y$ and $p$.
3. For $i = 1$ to $n$:
   - **Scan** vertex $v_i$, i.e., do for all arcs $(v_i, w) \in A$:
     - If $(v_i, w)$ is incorrect, then correct it, updating predecessor information.

This is still a label-correcting algorithm; but it’s a **one-pass algorithm**.

Theorem (Correctness and Efficiency of Bellman’s Algorithm)

Bellman’s algorithm is correct. It terminates after $m = |A|$ arc verification steps.
Another important special case is to allow directed cycles, but to require that all arc costs are nonnegative.

- Again, we use an arc-correction sequence that corresponds to the idea of scanning the vertices in some ordering $v_1, v_2, \ldots, v_n$ (i.e., first verifying all arcs leaving $v_1$, then all arcs leaving $v_2$, etc.)

- This time, however, we do not determine this ordering a priori

- Rather, when $v_1, v_2, \ldots, v_i$ have been determined and scanned, we choose $v_{i+1}$ as an unscanned vertex $v$ with minimum potential value $y_v$ (at that time).

The resulting algorithm is called Dijkstra’s Algorithm.
Dijkstra’s Algorithm [1959]

Dijkstra’s Algorithm

Input: A digraph $G = (V, A)$ with nonnegative arc costs, starting node $r$

Output: A predecessor vector $p$, encoding minimum-cost paths from $r$ to all nodes.

1. Initialize $y, p$. 
2. Set $S := V$. 
3. While $S \neq \emptyset$:
   - Choose $v \in S$ with $y_v$ minimum. 
   - Set $S := S \setminus \{v\}$. 
   - Scan vertex $v$, i.e., do for all arcs $(v, w) \in A$: 
     - If $(v, w)$ is incorrect, then correct it, updating predecessor information.
Dijkstra’s Algorithm: Correctness

- We use the notation $v_1, v_2, \ldots, v_n$ for the ordering of the nodes.
- We denote by $y^{(i)}$ the value of $y$ at the point when $v_i$ is chosen to be scanned.

**Lemma (Monotonicity of potentials of scanned nodes)**

*For all $i < k$ we have $y^{(i)}_{v_i} \leq y^{(k)}_{v_k}$.*

**Proof.**

- Suppose the contrary, i.e., there exist $i < k$ with $y^{(i)}_{v_i} > y^{(k)}_{v_k}$.
- Fix such an $i$ and choose $k$ minimal with this property, i.e., $v_k$ is the **earliest-chosen vertex after** $v_i$ that, at the time of its scanning, had a smaller potential than the vertex $v_i$ at the time of its scanning.
- But by the minimal choice in the algorithm, we have $y^{(i)}_{v_i} \leq y^{(i)}_{v_k}$.
- So $y_{v_k}$ must have been lowered while scanning some vertex $v_j$ with $i < j < k$.
- This arc correction made $y^{(k)}_{v_k} = y^{(j+1)}_{v_k} = y^{(j)}_{v_j} + c_{v_j, v_k}$.
- Because $c_{v_j, v_k} \geq 0$, we have $y^{(j)}_{v_j} \leq y^{(k)}_{v_k} < y^{(i)}_{v_i}$.
- This is a contradiction to the definition of $k$. 
Dijkstra’s Algorithm: Correctness, II

**Theorem**

*Dijkstra’s Algorithm is correct.*

**Proof.**

We prove that, after all vertices have been scanned, we have a feasible potential $y^{n+1}$:

- Suppose not, i.e., for some $(v_i, v_k) \in A$, we have $y_{v_i}^{(n+1)} + c_{v_i,v_k} < y_{v_k}^{(n+1)}$.
- But directly after scanning vertex $v_i$, we certainly did have $y_{v_i}^{(i+1)} + c_{v_i,v_k} \geq y_{v_k}^{(i+1)}$.
- Since we never increase the potentials, $y_{v_i}$ must have been lowered afterwards! Say, it was lowered the last time when scanning vertex $v_j$ (with $i < j$).
- Thus $y_{v_i}^{(i+1)} > y_{v_i}^{(n+1)} = y_{v_i}^{(j+1)} = y_{v_j}^{(j)} + c_{v_j,v_i} \geq y_{v_j}^{(j)}$.
- On the other hand, by the Lemma, because $v_j$ was scanned after $v_i$, we have $y_{v_j}^{(j)} \geq y_{v_i}^{(i)}$, a contradiction ($y_{v_i}^{(i+1)} > y_{v_i}^{(i)}$).
Dijkstra’s Algorithm: Efficiency

Theorem (Efficiency of Dijkstra’s Algorithm)

*Dijkstra’s Algorithm terminates after* \( m = |A| \) *arc verification steps.*

- Let’s try out Dijkstra’s Algorithm in practice; we expect that the running time essentially only depends, linearly, on the number of arcs.
- We try on examples with the same number of arcs, but different numbers of vertices.
- Result: There is a great dependence on the number of vertices, and we are not happy with the running time for large, sparse graphs (many vertices, few arcs).
- Where is the running time spent? Our coarse abstraction of running time (number of arc verification steps) does not give the answer.
- To find this out in the practical program, it is strongly recommended to find this out by measuring time, rather than thinking or guessing.
- Every modern, reasonable programming system has a facility for measuring how much running time is spent in parts of the program; this is called a *(time)* profiler.
- In the case of C, the GCC toolchain (compiler/linker option `-pg`) and the gprof tool provide a (sampling) time profiler.