Kolkata Algorithms Short Course: II. “Expanding” Algorithms

Kenneth W. Regan
University at Buffalo (SUNY)

University of Calcutta, 3 August 2016
Breadth-First Search—Brief Review

- Solves search problem, “is node \( f \) reachable from \( s \)” (GAP)
Breadth-First Search—Brief Review

- Solves search problem, “is node $f$ reachable from $s$?” (GAP)
- BFS expands a set of FOUND nodes until no further change.
Breadth-First Search—Brief Review

- Solves search problem, “is node $f$ reachable from $s$?” (GAP)
- BFS expands a set of FOUND nodes until no further change.
- Economizes time but FOUND takes up much space.
Breadth-First Search—Brief Review

- Solves search problem, “is node $f$ reachable from $s$?” (GAP)
- BFS expands a set of FOUND nodes until no further change.
- Economizes time but FOUND takes up much space.
- Needs random access to look up whether $v \in$ FOUND.
Breadth-First Search—Brief Review

- Solves search problem, “is node $f$ reachable from $s$?” (GAP)
- BFS expands a set of FOUND nodes until no further change.
- Economizes time but FOUND takes up much space.
- Needs random access to look up whether $v \in$ FOUND.
- Theoretical distinction: the search problem is can be “solved” by NTM in $O(\log n)$ space,
Breadth-First Search—Brief Review

- Solves search problem, “is node \( f \) reachable from \( s \)?” (GAP)
- BFS expands a set of FOUND nodes until no further change.
- Economizes time but FOUND takes up much space.
- Needs random access to look up whether \( v \in \) FOUND.
- Theoretical distinction: the search problem is can be “solved” by NTM in \( O(\log n) \) space, meaning with finitely many pointers (“fingers”) into a read-only data structure where they move at-will.
Breadth-First Search—Brief Review

- Solves search problem, “is node $f$ reachable from $s$?” (GAP)
- BFS expands a set of FOUND nodes until no further change.
- Economizes time but FOUND takes up much space.
- Needs random access to look up whether $v \in$ FOUND.
- Theoretical distinction: the search problem is can be “solved” by NTM in $O(\log n)$ space, meaning with finitely many pointers (“fingers”) into a read-only data structure where they move at-will. Shows NL $\subseteq$ P.
Breadth-First Search—Brief Review

- Solves search problem, “is node $f$ reachable from $s$?” (GAP)
- BFS expands a set of FOUND nodes until no further change.
- Economizes time but FOUND takes up much space.
- Needs random access to look up whether $v \in$ FOUND.
- Theoretical distinction: the search problem is can be “solved” by NTM in $O(\log n)$ space, meaning with finitely many pointers (“fingers”) into a read-only data structure where they move at-will. Shows NL $\subseteq$ P.
- Example: Maze “dungeon” problem (and string-matching problem) looked more complex but obeyed this distinction so in the same “class” of algorithms.
Breadth-First Search—Brief Review

- Solves search problem, “is node \( f \) reachable from \( s \)?” (GAP)
- BFS expands a set of FOUND nodes until no further change.
- Economizes time but FOUND takes up much space.
- Needs random access to look up whether \( v \in \) FOUND.
- Theoretical distinction: the search problem is can be “solved” by NTM in \( O(\log n) \) space, meaning with finitely many pointers (“fingers”) into a read-only data structure where they move at-will. Shows \( \text{NL} \subseteq \text{P} \).
- Example: Maze “dungeon” problem (and string-matching problem) looked more complex but obeyed this distinction so in the same “class” of algorithms.

And Depth-First Search economizes memory but not time, shows \( \text{NP} \subseteq \text{PSPACE} \).
Is this problem in the “BFS class”? 

- Given a graph $G$ and a node $h$ deemed a “health risk,”
Is this problem in the “BFS class”? 

- Given a graph $G$ and a node $h$ deemed a “health risk,”
- If $v$ is a health risk and $u \rightarrow v$ then $u$ is a health risk.
Is this problem in the “BFS class”?

- Given a graph $G$ and a node $h$ deemed a “health risk,”
- If $v$ is a health risk and $u \rightarrow v$ then $u$ is a health risk.
- Is the start node $s$ a health risk?
Is this problem in the “BFS class”? 

- Given a graph $G$ and a node $h$ deemed a “health risk,"
- If $v$ is a health risk and $u \rightarrow v$ then $u$ is a health risk.
- Is the start node $s$ a health risk?

Yes, problem is in BFS class. It is the same as GAP but “thinking backwards.” Answer is still yes iff there is a path from $s$ to $h$. 
Is this problem in the “BFS class”?  

- Given a graph $G$ and a node $h$ deemed a “health risk,”  
- If $v$ is a health risk and $u \rightarrow v$ then $u$ is a health risk.  
- Is the start node $s$ a health risk?

Yes, problem is in BFS class. It is the same as GAP but “thinking backwards.” Answer is still yes iff there is a path from $s$ to $h$.

Solved by BFS working forwards from $s$—or more intuitively, by working backwards from $h$ and expanding the set nodes known to be “health risks.” In the latter case it is BFS in the “reversed graph.”
A much harder example

- A 2-clause is a logical formula \((x \lor y)\) or \(((\neg x) \lor y)\) or \((x \lor (\neg y))\) or \((\neg x) \lor (\neg y))\).
A much harder example

- A 2-clause is a logical formula \((x \lor y)\) or \(((\neg x) \lor y)\) or \((x \lor (\neg y))\) or \((\neg x) \lor (\neg y))\).

- We can write the four possible 2-clauses more economically as \((x \lor y)\) or \((\bar{x} \lor y)\) or \((x \lor \bar{y})\) or \((\bar{x} \lor \bar{y})\).
A much harder example

- A 2-clause is a logical formula \((x \lor y)\) or \(((\neg x) \lor y)\) or \((x \lor (\neg y))\) or \((\neg x) \lor (\neg y))\).

- We can write the four possible 2-clauses more economically as \((x \lor y)\) or \((\bar{x} \lor y)\) or \((x \lor \bar{y})\) or \((\bar{x} \lor \bar{y})\).

- Consider logical formulas \(f\) that are ANDs of such clauses.
A much harder example

- A 2-clause is a logical formula \((x \lor y)\) or \(((\neg x) \lor y)\) or \((x \lor (\neg y))\) or \((\neg x) \lor (\neg y)\).

- We can write the four possible 2-clauses more economically as \((x \lor y)\) or \((\bar{x} \lor y)\) or \((x \lor \bar{y})\) or \((\bar{x} \lor \bar{y})\).

- Consider logical formulas \(f\) that are ANDs of such clauses. Called “2-Conjunctive Normal Form” (2CNF).
A much harder example

- A \textit{2-clause} is a logical formula \((x \lor y) \text{ or } ((\neg x) \lor y) \text{ or } (x \lor (\neg y)) \text{ or } (\neg x) \lor (\neg y))\).

- We can write the four possible 2-clauses more economically as \((x \lor y) \text{ or } (\bar{x} \lor y) \text{ or } (x \lor \bar{y}) \text{ or } (\bar{x} \lor \bar{y})\).

- Consider logical formulas \(f\) that are ANDs of such clauses. Called "2-Conjunctive Normal Form" (2CNF).

- The \textit{problem} is, given an \(f\), is there a way to make it true—or must it always be false?
A much harder example

- A 2-clause is a logical formula \((x \lor y)\) or \(((\neg x) \lor y)\) or \((x \lor (\neg y))\) or \((\neg x) \lor (\neg y))\).

- We can write the four possible 2-clauses more economically as \((x \lor y)\) or \((\bar{x} \lor y)\) or \((x \lor \bar{y})\) or \((\bar{x} \lor \bar{y})\).

- Consider logical formulas \(f\) that are ANDs of such clauses. Called “2-Conjunctive Normal Form” (2CNF).

- The problem is, given an \(f\), is there a way to make it true—or must it always be false?

Example:

\[ f = (u \lor v) \land (\bar{u} \lor w) \land (\bar{u} \lor x) \land (\bar{w} \lor \bar{x}).\]
A much harder example

- A 2-clause is a logical formula \((x \lor y)\) or \(((\neg x) \lor y)\) or \((x \lor (\neg y))\) or \((\neg x) \lor (\neg y))\).

- We can write the four possible 2-clauses more economically as \((x \lor y)\) or \((\bar{x} \lor y)\) or \((x \lor \bar{y})\) or \((\bar{x} \lor \bar{y})\).

- Consider logical formulas \(f\) that are ANDs of such clauses. Called “2-Conjunctive Normal Form” (2CNF).

- The problem is, given an \(f\), is there a way to make it true—or must it always be false?

Example:

\[ f = (u \lor v) \land (\bar{u} \lor w) \land (\bar{u} \lor x) \land (\bar{w} \lor \bar{x}). \]

If we set \(u = \text{true}\) then we must set \(w, x = \text{true}\) as well, but then the last clause fails.
A much harder example

- A 2-clause is a logical formula \((x \lor y)\) or \(((\neg x) \lor y)\) or \((x \lor (\neg y))\) or \((\neg x) \lor (\neg y))\).
- We can write the four possible 2-clauses more economically as \((x \lor y)\) or \((\bar{x} \lor y)\) or \((x \lor \bar{y})\) or \((\bar{x} \lor \bar{y})\).
- Consider logical formulas \(f\) that are ANDs of such clauses. Called “2-Conjunctive Normal Form” (2CNF).
- The problem is, given an \(f\), is there a way to make it true—or must it always be false?

Example:

\[
f = (u \lor \nu) \land (\bar{u} \lor w) \land (\bar{u} \lor x) \land (\bar{w} \lor \bar{x}).
\]

If we set \(u = \text{true}\) then we must set \(w, x = \text{true}\) as well, but then the last clause fails. However, we can set \(u = 0, \nu = 1, \) and either \(w\) or \(x\) false—then we satisfy \(f\).
Second Example and Key Idea

\[ f' = (u \lor v) \land (\bar{u} \lor w) \land (\bar{u} \lor x) \land (\bar{w} \lor \bar{x}) \land (\bar{v} \lor w) \land (\bar{v} \lor x). \]

This burdens \( f \) with two more clauses.
Second Example and Key Idea

\[ f' = (u \lor v) \land (\overline{u} \lor w) \land (\overline{u} \lor x) \land (\overline{w} \lor \overline{x}) \land (\overline{v} \lor w) \land (\overline{v} \lor x). \]

This burdens \( f \) with two more clauses. Now if we set \( u = 0 \) and \( v = 1 \), the two new clauses force us to make \( w = x = 1 \).
Second Example and Key Idea

\[ f' = (u \lor v) \land (\bar{u} \lor w) \land (\bar{u} \lor x) \land (\bar{w} \lor \bar{x}) \land (\bar{v} \lor w) \land (\bar{v} \lor x). \]

This burdens \( f \) with two more clauses. Now if we set \( u = 0 \) and \( v = 1 \), the two new clauses force us to make \( w = x = 1 \). But then the fourth clause \( (\bar{w} \lor \bar{x}) \) fails.

- So there is no way. But how can we convincingly prove it?
Second Example and Key Idea

\[ f' = (u \lor v) \land (\bar{u} \lor w) \land (\bar{u} \lor x) \land (\bar{w} \lor \bar{x}) \land (\bar{v} \lor w) \land (\bar{v} \lor x). \]

This burdens \( f \) with two more clauses. Now if we set \( u = 0 \) and \( v = 1 \), the two new clauses force us to make \( w = x = 1 \). But then the fourth clause \( (\bar{w} \lor \bar{x}) \) fails.

- So there is no way. But how can we convincingly prove it?
- **Idea**: \( x \rightarrow y \) is equivalent to \( ((\neg x) \lor y) \).
Second Example and Key Idea

\[ f' = (u \lor v) \land (\overline{u} \lor w) \land (\overline{u} \lor x) \land (\overline{w} \lor \overline{x}) \land (\overline{v} \lor w) \land (\overline{v} \lor x). \]

This burdens \( f \) with two more clauses. Now if we set \( u = 0 \) and \( v = 1 \), the two new clauses force us to make \( w = x = 1 \). But then the fourth clause \((\overline{w} \lor \overline{x})\) fails.

- So there is no way. But how can we convincingly prove it?
- **Idea:** \( x \rightarrow y \) is equivalent to \((\neg x) \lor y\).
- So \((x \lor y) \equiv \overline{x} \rightarrow y\) and \((\overline{x} \lor y) \equiv x \rightarrow y\).
Second Example and Key Idea

\[ f' = (u \lor v) \land (\bar{u} \lor w) \land (\bar{u} \lor x) \land (\bar{w} \lor \bar{x}) \land (\bar{v} \lor w) \land (\bar{v} \lor x). \]

This burdens \( f \) with two more clauses. Now if we set \( u = 0 \) and \( v = 1 \), the two new clauses force us to make \( w = x = 1 \). But then the fourth clause \( (\bar{w} \lor \bar{x}) \) fails.

- So there is no way. But how can we convincingly prove it?
- **Idea:** \( x \rightarrow y \) is equivalent to \( ((\neg x) \lor y) \).
- So \( (x \lor y) \equiv \bar{x} \rightarrow y \) and \( (\bar{x} \lor y) \equiv x \rightarrow y \).
- And \( (x \lor \bar{y}) \equiv \bar{x} \rightarrow \bar{y} \) and \( (\bar{x} \lor \bar{y}) \equiv x \rightarrow \bar{y} \).
Second Example and Key Idea

\[ f' = (u \lor v) \land (\bar{u} \lor w) \land (\bar{u} \lor x) \land (\bar{w} \lor \bar{x}) \land (\bar{v} \lor w) \land (\bar{v} \lor x). \]

This burdens \( f \) with two more clauses. Now if we set \( u = 0 \) and \( v = 1 \), the two new clauses force us to make \( w = x = 1 \). But then the fourth clause \((\bar{w} \lor \bar{x}) \) fails.

- So there is no way. But how can we convincingly prove it?
- **Idea:** \( x \rightarrow y \) is equivalent to \((\neg x) \lor y\).
- So \((x \lor y) \equiv \bar{x} \rightarrow y \) and \((\bar{x} \lor y) \equiv x \rightarrow y\).
- And \((x \lor \bar{y}) \equiv \bar{x} \rightarrow \bar{y} \) and \((\bar{x} \lor \bar{y}) \equiv x \rightarrow \bar{y}\).
- Also \((x \lor y) \equiv (y \lor x) \) so include \( \bar{y} \rightarrow x \) etc.
Second Example and Key Idea

\[ f' = (u \lor v) \land (\bar{u} \lor w) \land (\bar{u} \lor x) \land (\bar{w} \lor \bar{x}) \land (\bar{v} \lor w) \land (\bar{v} \lor x). \]

This burdens \( f \) with two more clauses. Now if we set \( u = 0 \) and \( v = 1 \), the two new clauses force us to make \( w = x = 1 \). But then the fourth clause \( (\bar{w} \lor \bar{x}) \) fails.

- So there is no way. But how can we convincingly prove it?
- Idea: \( x \rightarrow y \) is equivalent to \( ((\neg x) \lor y) \).
- So \( (x \lor y) \equiv \bar{x} \rightarrow y \) and \( (\bar{x} \lor y) \equiv x \rightarrow y \).
- And \( (x \lor \bar{y}) \equiv \bar{x} \rightarrow \bar{y} \) and \( (\bar{x} \lor \bar{y}) \equiv x \rightarrow \bar{y} \).
- Also \( (x \lor y) \equiv (y \lor x) \) so include \( \bar{y} \rightarrow x \) etc.
- Make a graph \( G_f \) with these nodes and all these edges.
Second Example and Key Idea

\[ f' = (u \lor v) \land (\bar{u} \lor w) \land (\bar{u} \lor x) \land (\bar{w} \lor \bar{x}) \land (\bar{v} \lor w) \land (\bar{v} \lor x). \]

This burdens \( f \) with two more clauses. Now if we set \( u = 0 \) and \( v = 1 \), the two new clauses force us to make \( w = x = 1 \). But then the fourth clause \((\bar{w} \lor \bar{x})\) fails.

- So there is no way. But how can we convincingly prove it?
- **Idea:** \( x \rightarrow y \) is equivalent to \((\neg x) \lor y\).
- So \((x \lor y) \equiv \bar{x} \rightarrow y\) and \((\bar{x} \lor y) \equiv x \rightarrow y\).
- And \((x \lor \bar{y}) \equiv \bar{x} \rightarrow \bar{y}\) and \((\bar{x} \lor \bar{y}) \equiv x \rightarrow \bar{y}\).
- Also \((x \lor y) \equiv (y \lor x)\) so include \(\bar{y} \rightarrow x\) etc.
- Make a graph \( G_f \) with these nodes and all these edges.
- **Lemma:** \( f \) is unsatisfiable \( \iff \) \( G_f \) has a “vicious cycle” involving some node \( u \) and its negation \( \bar{u} \). [Draw \( G_f \), show example.]
If there is a path from $u$ to $w$ in $G_f$, then $u \implies w$ logically.
Analysis and Algorithm

- If there is a path from \( u \) to \( w \) in \( G_f \), then \( u \rightarrow w \) logically.
- Same for any combination of \( u, \bar{u} \) and \( w, \bar{w} \).
Analysis and Algorithm

- If there is a path from $u$ to $w$ in $G_f$, then $u \leftrightarrow w$ logically.
- Same for any combination of $u, \bar{u}$ and $w, \bar{w}$.
- So if $u$ and $\bar{u}$ are on a cycle, then $u \leftrightarrow \neg u$ and $\neg u \leftrightarrow u$. 

This contradiction means there is no consistent truth assignment, so $f$ is unsatisfiable.

If there is no cycle involving both $u$ and $\bar{u}$, for any $u$, then how can we satisfy $f$ and prove the Lemma?

Granting the Lemma, a nondeterministic TM $N$ can "solve" $f$ being unsatisfiable by guessing a contradictory $u; \bar{u}$, putting two fingers there ("batsmen") and walking each in $G_f$. If and when the "batsmen" change places, we have the cycle. So this is BFS class. We can get clean BFS by converting $N$ to its "ID graph." Can you find a more efficient algorithm directly?
Analysis and Algorithm

- If there is a path from \( u \) to \( w \) in \( G_f \), then \( u \equiv w \) logically.
- Same for any combination of \( u, \bar{u} \) and \( w, \bar{w} \).
- So if \( u \) and \( \bar{u} \) are on a cycle, then \( u \equiv \neg u \) and \( \neg u \equiv u \).
- This *contradiction* means there is no consistent truth assignment, so \( f \) is *unsatisfiable*. 
Analysis and Algorithm

- If there is a path from \( u \) to \( w \) in \( G_f \), then \( u \leftrightarrow w \) logically.
- Same for any combination of \( u, \bar{u} \) and \( w, \bar{w} \).
- So if \( u \) and \( \bar{u} \) are on a cycle, then \( u \leftrightarrow \neg u \) and \( \neg u \leftrightarrow u \).
- This *contradiction* means there is no consistent truth assignment, so \( f \) is *unsatisfiable*.
- If there is no cycle involving both \( u \) and \( \bar{u} \), for any \( u \), then how can we satisfy \( f \) and prove the Lemma?
Analysis and Algorithm

- If there is a path from \( u \) to \( w \) in \( G_f \), then \( u \implies w \) logically.
- Same for any combination of \( u, \bar{u} \) and \( w, \bar{w} \).
- So if \( u \) and \( \bar{u} \) are on a cycle, then \( u \implies \neg u \) and \( \neg u \implies u \).
- This *contradiction* means there is no consistent truth assignment, so \( f \) is *unsatisfiable*.
- If there is no cycle involving both \( u \) and \( \bar{u} \), for any \( u \), then how can we satisfy \( f \) and prove the Lemma?
- Granting the Lemma, a *nondeterministic* TM \( N \) can “solve” \( f \) being unsatisfiable by *guessing* a contradictory \( u, \bar{u} \),
If there is a path from $u$ to $w$ in $G_f$, then $u \implies w$ logically.

Same for any combination of $u, \bar{u}$ and $w, \bar{w}$.

So if $u$ and $\bar{u}$ are on a cycle, then $u \implies \neg u$ and $\neg u \implies u$.

This contradiction means there is no consistent truth assignment, so $f$ is unsatisfiable.

If there is no cycle involving both $u$ and $\bar{u}$, for any $u$, then how can we satisfy $f$ and prove the Lemma?

Granting the Lemma, a nondeterministic TM $N$ can “solve” $f$ being unsatisfiable by guessing a contradictory $u, \bar{u}$, putting two fingers there (“batsmen”) and walking each in $G_f$. If and when the “batsmen” change places, we have the cycle.
If there is a path from \( u \) to \( w \) in \( G_f \), then \( u \equiv w \) logically.

Same for any combination of \( u, \bar{u} \) and \( w, \bar{w} \).

So if \( u \) and \( \bar{u} \) are on a cycle, then \( u \equiv \neg u \) and \( \neg u \equiv u \).

This contradiction means there is no consistent truth assignment, so \( f \) is unsatisfiable.

If there is no cycle involving both \( u \) and \( \bar{u} \), for any \( u \), then how can we satisfy \( f \) and prove the Lemma?

Granting the Lemma, a nondeterministic TM \( N \) can “solve” \( f \) being unsatisfiable by guessing a contradictory \( u, \bar{u} \), putting two fingers there (“batsmen”) and walking each in \( G_f \). If and when the “batsmen” change places, we have the cycle.

So this is BFS class. We can get clean BFS by converting \( N \) to its “ID graph.”
Analysis and Algorithm

- If there is a path from $u$ to $w$ in $G_f$, then $u \implies w$ logically.
- Same for any combination of $u$, $\bar{u}$ and $w$, $\bar{w}$.
- So if $u$ and $\bar{u}$ are on a cycle, then $u \implies \neg u$ and $\neg u \implies u$.
- This contradiction means there is no consistent truth assignment, so $f$ is unsatisfiable.
- If there is no cycle involving both $u$ and $\bar{u}$, for any $u$, then how can we satisfy $f$ and prove the Lemma?
- Granting the Lemma, a nondeterministic TM $N$ can “solve” $f$ being unsatisfiable by guessing a contradictory $u$, $\bar{u}$, putting two fingers there (“batsmen”) and walking each in $G_f$. If and when the “batsmen” change places, we have the cycle.
- So this is BFS class. We can get clean BFS by converting $N$ to its “ID graph.”
- Can you find a more efficient algorithm directly?
Another Example

Let’s picture BFS as “conquest” or “occupation” or “invasion”:
- If we have occupied $u$ and $u \rightarrow v$ is an edge and $v$ is undefended, then we conquer $v$. 
Another Example

Let’s picture BFS as “conquest” or “occupation” or “invasion”:

- If we have occupied \( u \) and \( u \to v \) is an edge and \( v \) is undefended, then we conquer \( v \).
- But if \( v \) is a “Fort,” say we conquer \( v \) only if we have occupied all “supply lines” \( u \) such that \( u \to v \).
Another Example

Let’s picture BFS as “conquest” or “occupation” or “invasion”:

- If we have occupied \( u \) and \( u \to v \) is an edge and \( v \) is undefended, then we conquer \( v \).
- But if \( v \) is a “Fort,” say we conquer \( v \) only if we have occupied \( \text{all} \)
  “supply lines” \( u \) such that \( u \to v \).

- Now given a graph \( G \) where we occupy \( s \), and a node \( t \) with some
  forts in-between, the question is, can we conquer \( t \)?
Another Example

Let’s picture BFS as “conquest” or “occupation” or “invasion”:

- If we have occupied $u$ and $u \rightarrow v$ is an edge and $v$ is undefended, then we conquer $v$.
- But if $v$ is a “Fort,” say we conquer $v$ only if we have occupied all “supply lines” $u$ such that $u \rightarrow v$.
- Now given a graph $G$ where we occupy $s$, and a node $t$ with some forts in-between, the question is, can we conquer $t$?
- [Show examples on board.]
Another Example

Let’s picture BFS as “conquest” or “occupation” or “invasion”:

- If we have occupied \( u \) and \( u \rightarrow v \) is an edge and \( v \) is undefended, then we conquer \( v \).
- But if \( v \) is a “Fort,” say we conquer \( v \) only if we have occupied all “supply lines” \( u \) such that \( u \rightarrow v \).
- Now given a graph \( G \) where we occupy \( s \), and a node \( t \) with some forts in-between, the question is, can we conquer \( t \)?
- [Show examples on board.]
- We can straightforwardly modify the previous BFS algorithm to solve this. So everything the same?
Another Example

Let’s picture BFS as “conquest” or “occupation” or “invasion”:
- If we have occupied \( u \) and \( u \rightarrow v \) is an edge and \( v \) is undefended, then we conquer \( v \).
- But if \( v \) is a “Fort,” say we conquer \( v \) only if we have occupied all “supply lines” \( u \) such that \( u \rightarrow v \).
- Now given a graph \( G \) where we occupy \( s \), and a node \( t \) with some forts in-between, the question is, can we conquer \( t \)?
- [Show examples on board.]
- We can straightforwardly modify the previous BFS algorithm to solve this. So everything the same?
- The kind of question where you gain insight from theory is:
Another Example

Let’s picture BFS as “conquest” or “occupation” or “invasion”:

- If we have occupied \( u \) and \( u \to v \) is an edge and \( v \) is undefended, then we conquer \( v \).
- But if \( v \) is a “Fort,” say we conquer \( v \) only if we have occupied *all* “supply lines” \( u \) such that \( u \to v \).
- Now given a graph \( G \) where we occupy \( s \), and a node \( t \) with some forts in-between, the question is, can we conquer \( t \)?

[Show examples on board.]

We can straightforwardly modify the previous BFS algorithm to solve this. So everything the same?

The kind of question where you gain insight from *theory* is:

Does this problem belong to the BFS class?
set<Node> CONQUERED = {s}, POPPED = {};  
bool novel = true;  //fort: v_strength = indeg(v)  
while (novel) {
    novel = false;  
    foreach (u in CONQUERED \ POPPED) {
        foreach (v: u→v) {
            if (v not in CONQUERED) {
                novel = true;  
                v_hits++;  
                if (v_hits >= v_strength) {
                    CONQUERED += {v};
                }
            }
        }
    }
    POPPED += {u};  //Can you ‘‘ND-do’’ this  
}  //using O(1)—many fingers?
Conquering Boolean Logic

- Let’s say we merely want to evaluate a Boolean formula $f$ on a given 0-1 truth assignment.
Conquering Boolean Logic

- Let’s say we merely want to *evaluate* a Boolean formula \( f \) on a given 0-1 truth assignment.
- Much easier in general than trying to tell whether \( f \) is satisfiable.
Let’s say we merely want to *evaluate* a Boolean formula $f$ on a given 0-1 truth assignment.

Much easier in general than trying to tell whether $f$ is satisfiable.

We may suppose $f$ uses AND, OR, and NOT gates only, and has variables $x_1, \ldots, x_n$. We think of $n$ as the “rough size” of $f$. 

Further, using DeMorgan’s Laws, we may suppose all negations are pushed inside:

\[
\neg (g \land h) = \neg g \lor \neg h, \\
\neg (g \lor h) = \neg g \land \neg h.
\]

So we make $f$ use $\land, \lor$ only with $2^n$ literals $x_1, \ldots, x_n, \neg x_1, \ldots, \neg x_n$. 

Conquering Boolean Logic
Let’s say we merely want to evaluate a Boolean formula $f$ on a given 0-1 truth assignment.

Much easier in general than trying to tell whether $f$ is satisfiable.

We may suppose $f$ uses AND, OR, and NOT gates only, and has variables $x_1, \ldots, x_n$. We think of $n$ as the “rough size” of $f$.

Further, using DeMorgan’s Laws, we may suppose all negations are pushed inside: $\neg (g \land h) = (\neg g) \lor (\neg h)$; $\neg (g \lor h) = (\neg g) \land (\neg h)$. 
Let’s say we merely want to *evaluate* a Boolean formula $f$ on a given 0-1 truth assignment.

Much easier in general than trying to tell whether $f$ is satisfiable.

We may suppose $f$ uses AND, OR, and NOT gates only, and has variables $x_1, \ldots, x_n$. We think of $n$ as the “rough size” of $f$.

Further, using DeMorgan’s Laws, we may suppose all negations are pushed inside: $\neg(g \land h) = (\neg g) \lor (\neg h)$; $\neg(g \lor h) = (\neg g) \land (\neg h)$.

So we make $f$ use $\land, \lor$ only with $2n$ literals $x_1, \ldots, x_n, \overline{x}_1, \ldots, \overline{x}_n$. 

From Formula (or Circuit) to a Graph

- Given $f$ using $\land, \lor$ and $x_1, \ldots, x_n, \bar{x}_1, \ldots, \bar{x}_n$;
From Formula (or Circuit) to a Graph

- Given $f$ using $\land, \lor$ and $x_1, \ldots, x_n, \overline{x}_1, \ldots, \overline{x}_n$;
- Any given truth assignment $a = (a_1, \ldots, a_n) \in \{0, 1\}^n$ sets $n$ literals true and $n$ of them false. They are $2n$ nodes in our graph.
From Formula (or Circuit) to a Graph

- Given $f$ using $\land, \lor$ and $x_1, \ldots, x_n, \overline{x}_1, \ldots, \overline{x}_n$;
- Any given truth assignment $a = (a_1, \ldots, a_n) \in \{0, 1\}^n$ sets $n$ literals true and $n$ of them false. They are $2n$ nodes in our graph.
- Conceptually we connect our start node to the $n$ made true—each is “conquered.”
From Formula (or Circuit) to a Graph

- Given $f$ using $\land, \lor$ and $x_1, \ldots, x_n, \overline{x}_1, \ldots, \overline{x}_n$;
- Any given truth assignment $a = (a_1, \ldots, a_n) \in \{0, 1\}^n$ sets $n$ literals true and $n$ of them false. They are $2n$ nodes in our graph.
- Conceptually we connect our start node to the $n$ made true—each is “conquered.”
- Now each $\land, \lor$ gate in $f$ is also a node, and has in-edges from its two arguments. [Show examples on board.]
From Formula (or Circuit) to a Graph

- Given $f$ using $\land, \lor$ and $x_1, \ldots, x_n, \overline{x}_1, \ldots, \overline{x}_n$;
- Any given truth assignment $a = (a_1, \ldots, a_n) \in \{0, 1\}^n$ sets $n$ literals true and $n$ of them false. They are $2n$ nodes in our graph.
- Conceptually we connect our start node to the $n$ made true—each is “conquered.”
- Now each $\land, \lor$ gate in $f$ is also a node, and has in-edges from its two arguments. [Show examples on board.]
- An AND gate is a fort—conquered iff both of its arguments are.
From Formula (or Circuit) to a Graph

- Given $f$ using $\land, \lor$ and $x_1, \ldots, x_n, \overline{x}_1, \ldots, \overline{x}_n$;
- Any given truth assignment $a = (a_1, \ldots, a_n) \in \{0, 1\}^n$ sets $n$ literals true and $n$ of them false. They are $2n$ nodes in our graph.
- Conceptually we connect our start node to the $n$ made true—each is “conquered.”
- Now each $\land, \lor$ gate in $f$ is also a node, and has in-edges from its two arguments. [Show examples on board.]
- An AND gate is a fort—conquered iff both of its arguments are.
- An OR gate is an undefended node—one “truth invader” suffices.
From Formula (or Circuit) to a Graph

- Given $f$ using $\land$, $\lor$ and $x_1, \ldots, x_n, \overline{x}_1, \ldots, \overline{x}_n$;
- Any given truth assignment $a = (a_1, \ldots, a_n) \in \{0, 1\}^n$ sets $n$ literals true and $n$ of them false. They are $2n$ nodes in our graph.
- Conceptually we connect our start node to the $n$ made true—each is “conquered.”
- Now each $\land, \lor$ gate in $f$ is also a node, and has in-edges from its two arguments. [Show examples on board.]
- An AND gate is a fort—conquered iff both of its arguments are.
- An OR gate is an undefended node—one “truth invader” suffices.
- $f(a) = \text{true} \iff$ we conquer the output gate of $f$. 
From Formula (or Circuit) to a Graph

- Given $f$ using $\land, \lor$ and $x_1, \ldots, x_n, \overline{x}_1, \ldots, \overline{x}_n$;
- Any given truth assignment $a = (a_1, \ldots, a_n) \in \{0, 1\}^n$ sets $n$ literals true and $n$ of them false. They are $2n$ nodes in our graph.
- Conceptually we connect our start node to the $n$ made true—each is “conquered.”
- Now each $\land, \lor$ gate in $f$ is also a node, and has in-edges from its two arguments. [Show examples on board.]
- An AND gate is a fort—conquered iff both of its arguments are.
- An OR gate is an undefended node—one “truth invader” suffices.
- $f(a) = \text{true} \iff$ we conquer the output gate of $f$.
- In a formula, each gate is argument to at most 1 other gate. Literals can be used as often as desired.
From Formula (or Circuit) to a Graph

- Given $f$ using $\land, \lor$ and $x_1, \ldots, x_n, \bar{x}_1, \ldots, \bar{x}_n$;
- Any given truth assignment $a = (a_1, \ldots, a_n) \in \{0, 1\}^n$ sets $n$ literals true and $n$ of them false. They are $2n$ nodes in our graph.
- Conceptually we connect our start node to the $n$ made true—each is “conquered.”
- Now each $\land, \lor$ gate in $f$ is also a node, and has in-edges from its two arguments. [Show examples on board.]
- An AND gate is a fort—conquered iff both of its arguments are.
- An OR gate is an undefended node—one “truth invader” suffices.
- $f(a) = \text{true} \iff$ we conquer the output gate of $f$.
- In a formula, each gate is argument to at most 1 other gate. Literals can be used as often as desired.
- In a (proper) circuit, some gates fan out to 2 or more other gates.
Theorem: Let $M$ be any deterministic Turing machine that runs in time $t(n)$ and space $s(n)$. Then for any $n$, we can build a Boolean logic circuit $C$ of size $O(t(n) \times s(n))$ with input nodes $x_1, \ldots, x_n$ (and their negations $\overline{x}_1, \ldots, \overline{x}_n$) such that for all inputs $x \in \{0, 1\}^n$,

$$M \text{ accepts } x \iff C(x) = 1.$$
Circuit Evaluation “Conquers” All of P

**Theorem:** Let $M$ be any deterministic Turing machine that runs in time $t(n)$ and space $s(n)$. Then for any $n$, we can build a Boolean logic circuit $C$ of size $O(t(n) \times s(n))$ with input nodes $x_1, \ldots, x_n$ (and their negations $\bar{x}_1, \ldots, \bar{x}_n$) such that for all inputs $x \in \{0, 1\}^n$,

$$M \text{ accepts } x \iff C(x) = 1.$$ 

[Show on board.] This embodies the slogan:

“Software Can be Efficiently Burned Into Hardware.”
Circuit Evaluation “Conquers” All of P

**Theorem:** Let $M$ be any deterministic Turing machine that runs in time $t(n)$ and space $s(n)$. Then for any $n$, we can build a Boolean logic circuit $C$ of size $O(t(n) \times s(n))$ with input nodes $x_1, \ldots, x_n$ (and their negations $\bar{x}_1, \ldots, \bar{x}_n$) such that for all inputs $x \in \{0, 1\}^n$,

$$M \text{ accepts } x \iff C(x) = 1.$$ 

[Show on board.] This embodies the slogan:

“Software Can be Efficiently Burned Into Hardware.”

**Consequence:** “Graph Conquest” is in the BFS class only if $P = NL$.  

More Non-BFS “Expanding” Algorithms

- Minimum Spanning Tree.
- Shortest Paths.
- Edit Distance and Other Dynamic Programming.
- How (Not) to Compute Fibonacci Numbers.
Minimum Spanning Tree

- Given an undirected $G$ and weights $w_e \geq 0$ on each edge $e$, find a spanning tree $T$ to minimize $w(T) = \sum_{e \in T} w_e$. 
Minimum Spanning Tree

- Given an *undirected* \( G \) and weights \( w_e \geq 0 \) on each edge \( e \), find a spanning tree \( T \) to minimize \( w(T) = \sum_{e \in T} w_e \).
- Motivating example: \( V(G) = \text{hubs } u, v, \ldots \) for electrification, \( w(u, v) = \text{cost of building electric lines between } u \text{ and } v \).
Minimum Spanning Tree

- Given an undirected $G$ and weights $w_e \geq 0$ on each edge $e$, find a spanning tree $T$ to minimize $w(T) = \sum_{e \in T} w_e$.
- Motivating example: $V(G) =$ hubs $u, v, \ldots$ for electrification, $w(u, v) =$ cost of building electric lines between $u$ and $v$.
- A useful idea: If $C \subset E(G)$ is a cutset, meaning a set of edges whose removal creates two (or more) islands—like bridges over a river—then $T$ must include a minimum-weight edge from $C$.

[Show diagram of why on board.]

Repeat until $T$ is built: add a minimum-weight edge $e$ that does not cause a cycle.

[Show example on board. Why is this correct? If “add” means “add to $T$” then we get Prim’s algorithm; if we allow $e$ to start a new tree and choose the minimum-available edge overall then Kruskal’s algorithm.]
Minimum Spanning Tree—new idea?

- In Prim’s algorithm we can choose any vertex $v$ to start building $T$. 
Minimum Spanning Tree—new idea?

- In Prim’s algorithm we can choose any vertex $v$ to start building $T$.
- With Kruskal’s the (or some) minimum-weight edge begins a first tree, but we may build up separate trees before joining them.
Minimum Spanning Tree—new idea?

- In Prim’s algorithm we can choose any vertex \( v \) to start building \( T \).
- With Kruskal’s the (or some) minimum-weight edge begins a first tree, but we may build up separate trees before joining them.
- Indeed Kruskal can regard the start as a forest of \( n \) trivial trees, each consisting of just one isolated node, then every good choice of edge joins two trees.
Minimum Spanning Tree—new idea?

- In Prim’s algorithm we can choose any vertex $v$ to start building $T$.
- With Kruskal’s the (or some) minimum-weight edge begins a first tree, but we may build up separate trees before joining them.
- Indeed Kruskal can regard the start as a forest of $n$ trivial trees, each consisting of just one isolated node, then every good choice of edge joins two trees.
- Idea (new?): Can we blend the two algorithms? Is that still correct?
Minimum Spanning Tree—new idea?

- In Prim’s algorithm we can choose any vertex $v$ to start building $T$.
- With Kruskal’s the (or some) minimum-weight edge begins a first tree, but we may build up separate trees before joining them.
- Indeed Kruskal can regard the start as a forest of $n$ trivial trees, each consisting of just one isolated node, then every good choice of edge joins two trees.
- Idea (new?): Can we blend the two algorithms? Is that still correct?
- That is, say we do a “Kruskal step” if we choose a least edge that has not already been used or rejected (because it causes a cycle).
Minimum Spanning Tree—new idea?

- In Prim’s algorithm we can choose any vertex \( v \) to start building \( T \).
- With Kruskal’s the (or some) minimum-weight edge begins a first tree, but we may build up separate trees before joining them.
- Indeed Kruskal can regard the start as a forest of \( n \) trivial trees, each consisting of just one isolated node, then every good choice of edge joins two trees.
- **Idea (new?):** Can we blend the two algorithms? Is that still correct?
- That is, say we do a “Kruskal step” if we choose a least edge that has not already been used or rejected (because it causes a cycle).
- In a “Prim step” we choose one (any) tree \( U \) from the forest and then add a least edge that touches \( U \).
Minimum Spanning Tree—new idea?

- In Prim’s algorithm we can choose any vertex $v$ to start building $T$.
- With Kruskal’s the (or some) minimum-weight edge begins a first tree, but we may build up separate trees before joining them.
- Indeed Kruskal can regard the start as a forest of $n$ trivial trees, each consisting of just one isolated node. Then every good choice of edge joins two trees.
- **Idea (new?):** Can we blend the two algorithms? Is that still correct?
- That is, say we do a “Kruskal step” if we choose a least edge that has not already been used or rejected (because it causes a cycle).
- In a “Prim step” we choose one (any) tree $U$ from the forest and then add a least edge that touches $U$.
- **Challenge:** Can this ‘liberal’ mix of the algorithms make a mistake?
In our code for BFS we iterated over FOUND nodes that were not yet POPPED in the graph-label order.

Instead, let us maintain for each node $v$ its currently-known distance $d(v)$ from $s$. Initially $d(s) = 0$; $d(v) = 1$ for all other $v$. At each step, choose $u$ from FOUND not yet POPPED with least $d(u)$. For each edge $e$ from $u$ to a neighbor $v$—even if $v$ already visited (but not popped)—if $d(u) + w(e) < d(v)$ then update $d(v) := d(u) + w(e)$, and make a pointer from $v$ point to $u$. Then pop $u$. Choose new $u$ with least $d(u)$; repeat until done. Following pointers back from $t$ then gives a shortest path $P$ from $s$. To prove correct, think of the first $e$ where a supposedly shorter path $P_0$ differs from $P$...
In our code for BFS we iterated over FOUND nodes that were not yet POPPED in the graph-label order.

Instead, let us maintain for each node \( v \) its currently-known distance \( d(v) \) from \( s \).
BFS and Shortest Paths (Dijkstra’s Algorithm)

- In our code for BFS we iterated over FOUND nodes that were not yet POPPED in the graph-label order.
- Instead, let us maintain for each node $v$ its currently-known distance $d(v)$ from $s$.
- Initially $d(s) = 0$; $d(v) = \infty$ for all other $v$. 

To prove correct, think of the first $e$ where a supposedly shorter path $P_0$ differs from $P$...[Show on board, note use of heaps.]
BFS and Shortest Paths (Dijkstra’s Algorithm)

- In our code for BFS we iterated over FOUND nodes that were not yet POPPED in the graph-label order.
- Instead, let us maintain for each node \( v \) its currently-known distance \( d(v) \) from \( s \).
- Initially \( d(s) = 0; \) \( d(v) = \infty \) for all other \( v \).
- At each step, choose \( u \in \text{FOUND} \setminus \text{POPPED} \) with least \( d(u) \).
BFS and Shortest Paths (Dijkstra’s Algorithm)

- In our code for BFS we iterated over FOUND nodes that were not yet POPPED in the graph-label order.
- Instead, let us maintain for each node $v$ its currently-known distance $d(v)$ from $s$.
- Initially $d(s) = 0$; $d(v) = \infty$ for all other $v$.
- At each step, choose $u \in \text{FOUND} \setminus \text{POPPED}$ with least $d(u)$.
- For each edge $e$ from $u$ to a neighbor $v$—even if $v$ already visited (but not popped)—if $d(u) + w(e) < d(v)$ then update $d(v) := d(u) + w(e)$, and make a pointer from $v$ point to $u$. 

Following pointers back from $t$ then gives a shortest path $P$ from $s$. To prove correct, think of the first $e$ where a supposedly shorter path $P_0$ differs from $P$...[Show on board, note use of heaps.]
BFS and Shortest Paths (Dijkstra’s Algorithm)

- In our code for BFS we iterated over FOUND nodes that were not yet POPPED in the graph-label order.
- Instead, let us maintain for each node $v$ its currently-known distance $d(v)$ from $s$.
- Initially $d(s) = 0$; $d(v) = \infty$ for all other $v$.
- At each step, choose $u \in \text{FOUND} \setminus \text{POPPED}$ with least $d(u)$.
- For each edge $e$ from $u$ to a neighbor $v$—even if $v$ already visited (but not popped)—if $d(u) + w(e) < d(v)$ then update $d(v) := d(u) + w(e)$, and make a pointer from $v$ point to $u$.
- Then pop $u$. Choose new $u'$ with least $d(u')$; repeat until done.

Following pointers back from $t$ then gives a shortest path $P$ from $s$.

To prove correct, think of the first $e$ where a supposedly shorter path $P_0$ differs from $P$...[Show on board, note use of heaps.]
In our code for BFS we iterated over FOUND nodes that were not yet POPPED in the graph-label order.

Instead, let us maintain for each node $v$ its currently-known distance $d(v)$ from $s$.

Initially $d(s) = 0$; $d(v) = \infty$ for all other $v$.

At each step, choose $u \in \text{FOUND} \setminus \text{POPPED}$ with least $d(u)$.

For each edge $e$ from $u$ to a neighbor $v$—even if $v$ already visited (but not popped)—if $d(u) + w(e) < d(v)$ then update $d(v) := d(u) + w(e)$, and make a pointer from $v$ point to $u$.

Then pop $u$. Choose new $u'$ with least $d(u')$; repeat until done.

Following pointers back from $t$ then gives a shortest path $P$ from $s$. 
BFS and Shortest Paths (Dijkstra’s Algorithm)

- In our code for BFS we iterated over FOUND nodes that were not yet POPPED in the graph-label order.
- Instead, let us maintain for each node \( v \) its currently-known distance \( d(v) \) from \( s \).
- Initially \( d(s) = 0 \); \( d(v) = \infty \) for all other \( v \).
- At each step, choose \( u \in \text{FOUND} \setminus \text{POPPED} \) with least \( d(u) \).
- For each edge \( e \) from \( u \) to a neighbor \( v \)—even if \( v \) already visited (but not popped)—if \( d(u) + w(e) < d(v) \) then update \( d(v) := d(u) + w(e) \), and make a pointer from \( v \) point to \( u \).
- Then pop \( u \). Choose new \( u' \) with least \( d(u') \); repeat until done.
- Following pointers back from \( t \) then gives a shortest path \( P \) from \( s \).
- To prove correct, think of the first \( e \) where a supposedly shorter path \( P' \) differs from \( P \)… [Show on board, note use of heaps.]
Edit Distance and Dynamic Programming

The term *dynamic programming* (DP) is IMHO misleading [tell story of 1950s “political correctness”].
The term *dynamic programming* (DP) is IMHO misleading [tell story of 1950s “political correctness”].
Really it means cleverly finding a way to compute a global function by incrementally building and updating a localized table.
Edit Distance and Dynamic Programming

- The term *dynamic programming* (DP) is IMHO misleading [tell story of 1950s “political correctness”].
- Really it means cleverly finding a way to compute a global function by incrementally building and updating a localized table.
- The size of the table is most important to the running time.
The term *dynamic programming* (DP) is IMHO misleading [tell story of 1950s “political correctness”].

Really it means cleverly finding a way to compute a global function by incrementally building and updating a localized table.

The size of the table is most important to the running time.

Dijkstra’s algorithm updates the table $d(v)$, but is more direct than what is usually called DP and the table has only $O(n)$ size (unless you want *all-pairs shortest paths*).
Edit Distance and Dynamic Programming

- The term *dynamic programming* (DP) is IMHO misleading [tell story of 1950s “political correctness”].
- Really it means cleverly finding a way to compute a global function by incrementally building and updating a localized table.
- The size of the table is most important to the running time.
- Dijkstra’s algorithm updates the table $d(v)$, but is more direct than what is usually called DP and the table has only $O(n)$ size (unless you want *all-pairs shortest paths*).
- In the *edit distance* problem, we wish to compute a certain distance $d(x, y)$ between a string $x$ of some length $m$ and $y$ of length $n$. 

We will build a table $D$ of size $O(mn)$—indeed dimension $(m+1)(n+1)$. If we number chars $x = x_1 \ldots x_m$ from 1, then we conveniently number the “fenceposts” between and around them by $0; \ldots; m$. The “dynamic” idea is $D(i; j) = d(x_1 \ldots x_i; y_1 \ldots y_j)$.
Edit Distance and Dynamic Programming

- The term *dynamic programming* (DP) is IMHO misleading [tell story of 1950s “political correctness”].
- Really it means cleverly finding a way to compute a global function by incrementally building and updating a localized table.
- The size of the table is most important to the running time.
- Dijkstra’s algorithm updates the table $d(v)$, but is more direct than what is usually called DP and the table has only $O(n)$ size (unless you want *all-pairs shortest paths*).
- In the *edit distance* problem, we wish to compute a certain distance $d(x, y)$ between a string $x$ of some length $m$ and $y$ of length $n$.
- We will build a table $D$ of size $O(mn)$—indeed dimension $(m + 1) \times (n + 1)$. 
The term *dynamic programming* (DP) is IMHO misleading [tell story of 1950s “political correctness”].

Really it means cleverly finding a way to compute a global function by incrementally building and updating a localized table.

The size of the table is most important to the running time.

Dijkstra’s algorithm updates the table $d(v)$, but is more direct than what is usually called DP and the table has only $O(n)$ size (unless you want *all-pairs shortest paths*).

In the *edit distance* problem, we wish to compute a certain distance $d(x, y)$ between a string $x$ of some length $m$ and $y$ of length $n$.

We will build a table $D$ of size $O(mn)$—indeed dimension $(m + 1) \times (n + 1)$.

If we number chars $x = x_1 \ldots x_m$ from 1, then we conveniently number the “fenceposts” between and around them by 0, $\ldots$, $m$. 
The term *dynamic programming* (DP) is IMHO misleading [tell story of 1950s “political correctness”].

Really it means cleverly finding a way to compute a global function by incrementally building and updating a localized table.

The size of the table is most important to the running time.

Dijkstra’s algorithm updates the table $d(v)$, but is more direct than what is usually called DP and the table has only $O(n)$ size (unless you want *all-pairs shortest paths*).

In the *edit distance* problem, we wish to compute a certain distance $d(x, y)$ between a string $x$ of some length $m$ and $y$ of length $n$.

We will build a table $D$ of size $O(mn)$—indeed dimension $(m + 1) \times (n + 1)$.

If we number chars $x = x_1 \cdots x_m$ from 1, then we conveniently number the “fenceposts” between and around them by 0, $\ldots$, $m$.

The “dynamic” idea is $D(i, j) = d(x_1 \cdots x_i, y_1 \cdots y_j)$. 
Example: editing Calcutta to Kolkata

The edits we are allowed to make are:

- Delete any character;

One way to do this is

\[ \text{Calcutta} \rightarrow \text{Kalcutta} \rightarrow \text{Kolcutta} \rightarrow \text{Kolkutta} \rightarrow \text{Kolkatta} \rightarrow \text{Kolkata} \]

This takes 5 steps. Is that minimum?

Well, think of building the city up from scratch...

\[ d(0; \text{Kolkata}) = 7 \] clearly 7 inserts needed.

Similarly \[ d(\text{Calcutta}; 0) = 8 \].

Thus for any strings we always initialize \[ D(0; j) = j \] and \[ D(i; 0) = i \].

A "Northeast" recurrence then expands the whole table.
Example: editing Calcutta to Kolkata

The edits we are allowed to make are:

- Delete any character;
- Insert any character (in a “fencepost”);

One way to do this is Calcutta -> Kalcutta -> Kolcutta -> Kolkutta -> Kolkatta -> Kolkata. This takes 5 steps. Is that minimum?

Well, think of building the city up from scratch... $d(\text{Kolkata};) = 7$: clearly 7 inserts needed. Similarly $d(\text{Calcutta};) = 8$.

Thus for any strings we always initialize $D(0; j) = j$ and $D(i; 0) = i$. A “Northeast” recurrence then expands the whole table.
Example: editing Calcutta to Kolkata

The edits we are allowed to make are:

- Delete any character;
- Insert any character (in a “fencepost”);
- Substitute any character $c$ by any letter $d$.
Example: editing Calcutta to Kolkata

The edits we are allowed to make are:

- Delete any character;
- Insert any character (in a “fencepost”);
- Substitute any character \( c \) by any letter \( d \).
- (The last is 1 step, rather than the 2 steps of deleting \( c \) and inserting \( d \).)

One way to do this is Calcutta -> Kalcutta -> Kolcutta -> Kolkutta -> Kolkatta -> Kolkata.
Example: editing Calcutta to Kolkata

The edits we are allowed to make are:

- Delete any character;
- Insert any character (in a “fencepost”);
- Substitute any character $c$ by any letter $d$.
- (The last is 1 step, rather than the 2 steps of deleting $c$ and inserting $d$.)

One way to do this is Calcutta $\rightarrow$ Kalcutta $\rightarrow$ Kolcutta $\rightarrow$ Kolkutta $\rightarrow$ Kolkatta $\rightarrow$ Kolkata. This takes 5 steps. Is that minimum?
Example: editing Calcutta to Kolkata

The edits we are allowed to make are:

- Delete any character;
- Insert any character (in a “fencepost”);
- Substitute any character $c$ by any letter $d$.
- (The last is 1 step, rather than the 2 steps of deleting $c$ and inserting $d$.)

One way to do this is Calcutta $\rightarrow$ Kalcutta $\rightarrow$ Kolcutta $\rightarrow$ Kolkutta $\rightarrow$ Kolkatta $\rightarrow$ Kolkata. This takes 5 steps. Is that minimum? Well, think of building the city up from scratch...
Example: editing Calcutta to Kolkata

The edits we are allowed to make are:

- Delete any character;
- Insert any character (in a “fencepost”);
- Substitute any character $c$ by any letter $d$.

(The last is 1 step, rather than the 2 steps of deleting $c$ and inserting $d$.)

One way to do this is Calcutta $\rightarrow$ Kalcutta $\rightarrow$ Kolcutta $\rightarrow$ Kolkutta $\rightarrow$ Kolkatta $\rightarrow$ Kolkata. This takes 5 steps. Is that minimum? Well, think of building the city up from scratch...

- $d(\lambda, \text{Kolkata}) = 7$: clearly 7 inserts needed.
Example: editing Calcutta to Kolkata

The edits we are allowed to make are:

- Delete any character;
- Insert any character (in a “fencepost”);
- Substitute any character $c$ by any letter $d$.

(The last is 1 step, rather than the 2 steps of deleting $c$ and inserting $d$.)

One way to do this is Calcutta -> Kalcutta -> Kolcutta -> Kolkutta -> Kolkatta -> Kolkata. This takes 5 steps. Is that minimum? Well, think of building the city up from scratch...

- $d(\lambda, \text{Kolkata}) = 7$: clearly 7 inserts needed.
- Similarly $d(\text{Calcutta}, \lambda) = 8$. 

The edits we are allowed to make are:

- Delete any character;
- Insert any character (in a “fencepost”);
- Substitute any character $c$ by any letter $d$.

(The last is 1 step, rather than the 2 steps of deleting $c$ and inserting $d$.)

One way to do this is Calcutta $\rightarrow$ Kalcutta $\rightarrow$ Kolcutta $\rightarrow$ Kolkutta $\rightarrow$ Kolkatta $\rightarrow$ Kolkata. This takes 5 steps. Is that minimum? Well, think of building the city up from scratch...

- $d(\lambda, \text{Kolkata}) = 7$: clearly 7 inserts needed.
- Similarly $d(\text{Calcutta}, \lambda) = 8$.
- Thus for any strings we always initialize $D(0, j) = j$ and $D(i, 0) = i$. 
Example: editing Calcutta to Kolkata

The edits we are allowed to make are:

- Delete any character;
- Insert any character (in a “fencepost”);
- Substitute any character $c$ by any letter $d$.

(The last is 1 step, rather than the 2 steps of deleting $c$ and inserting $d$.)

One way to do this is Calcutta -> Kalcutta -> Kolcutta -> Kolkutta -> Kolkatta -> Kolkata. This takes 5 steps. Is that minimum? Well, think of building the city up from scratch...

- $d(\lambda, \text{Kolkata}) = 7$: clearly 7 inserts needed.
- Similarly $d(\text{Calcutta}, \lambda) = 8$.
- Thus for any strings we always initialize $D(0, j) = j$ and $D(i, 0) = i$.
- A “Northeast” recurrence then expands the whole table.
**The Edit Distance Recursion**

**Lemma:** For any strings $x, y$ and $i, j$ with $1 \leq i \leq |x|$, $1 \leq j \leq |y|$: if $x_i = y_j$ then $D(i, j) = D(i - 1, j - 1)$, else

$$D(i, j) = 1 + \min\{D(i - 1, j - 1), D(i - 1, j), D(i, j - 1)\}.$$

- If $x_i = y_j$ then the least sequence converting $x_1 \cdots x_{i-1}$ to $y_1 \cdots y_{j-1}$ also converts $x_1 \cdots x_i$ to $y_1 \cdots y_j$ with no more edits.
The Edit Distance Recursion

**Lemma:** For any strings $x, y$ and $i, j$ with $1 \leq i \leq |x|$, $1 \leq j \leq |y|$: if $x_i = y_j$ then $D(i, j) = D(i - 1, j - 1)$, else

$$D(i, j) = 1 + \min\{D(i - 1, j - 1), D(i - 1, j), D(i, j - 1)\}.$$ 

- If $x_i = y_j$ then the least sequence converting $x_1 \cdots x_{i-1}$ to $y_1 \cdots y_{j-1}$ also converts $x_1 \cdots x_i$ to $y_1 \cdots y_j$ with no more edits.
- If not, then because $x_i$ and $y_j$ are the last chars in the respective (sub-)strings, at some point we have to change $x_i$ either by (a) substituting it, (b) deleting it, or (c) inserting $y_j$ someplace after it.
The Edit Distance Recursion

Lemma: For any strings $x, y$ and $i, j$ with $1 \leq i \leq |x|$, $1 \leq j \leq |y|$: if $x_i = y_j$ then $D(i, j) = D(i - 1, j - 1)$, else

$$D(i, j) = 1 + \min\{D(i - 1, j - 1), D(i - 1, j), D(i, j - 1)\}.$$  

- If $x_i = y_j$ then the least sequence converting $x_1 \cdots x_{i-1}$ to $y_1 \cdots y_{j-1}$ also converts $x_1 \cdots x_i$ to $y_1 \cdots y_j$ with no more edits.
- If note, then because $x_i$ and $y_j$ are the last chars in the respective (sub-)strings, at some point we have to change $x_i$ either by (a) substituting it, (b) deleting it, or (c) inserting $y_j$ someplace after it.
- So let $S$ be a minimum sequence of edits from $x' = x_1 \cdots x_i$ to $y' = y_1 \cdots y_j$. 

The Edit Distance Recursion

**Lemma:** For any strings $x, y$ and $i, j$ with $1 \leq i \leq |x|$, $1 \leq j \leq |y|$: if $x_i = y_j$ then $D(i, j) = D(i - 1, j - 1)$, else

$$D(i, j) = 1 + \min\{D(i - 1, j - 1), D(i - 1, j), D(i, j - 1)\}.$$

- If $x_i = y_j$ then the least sequence converting $x_1 \cdots x_{i-1}$ to $y_1 \cdots y_{j-1}$ also converts $x_1 \cdots x_i$ to $y_1 \cdots y_j$ with no more edits.
- If not, then because $x_i$ and $y_j$ are the last chars in the respective (sub-)strings, at some point we have to change $x_i$ either by (a) substituting it, (b) deleting it, or (c) inserting $y_j$ someplace after it.
- So let $S$ be a minimum sequence of edits from $x' = x_1 \cdots x_i$ to $y' = y_1 \cdots y_j$.
- If $y_j$ is already in $x_1 \cdots x_{i-1}$ then $S$ deletes $x_i$. We may as well do that first. So $D(i, j) \leq 1 + D(i - 1, j)$. 
The Edit Distance Recursion

**Lemma:** For any strings $x, y$ and $i, j$ with $1 \leq i \leq |x|$, $1 \leq j \leq |y|$: if $x_i = y_j$ then $D(i, j) = D(i - 1, j - 1)$, else

$$D(i, j) = 1 + \min\{D(i - 1, j - 1), D(i - 1, j), D(i, j - 1)\}.$$

- If $x_i = y_j$ then the least sequence converting $x_1 \cdots x_{i-1}$ to $y_1 \cdots y_{j-1}$ also converts $x_1 \cdots x_i$ to $y_1 \cdots y_j$ with no more edits.
- If note, then because $x_i$ and $y_j$ are the last chars in the respective (sub-)strings, at some point we have to change $x_i$ either by (a) substituting it, (b) deleting it, or (c) inserting $y_j$ someplace after it.
- So let $S$ be a minimum sequence of edits from $x' = x_1 \cdots x_i$ to $y' = y_1 \cdots y_j$.
- If $y_j$ is already in $x_1 \cdots x_{i-1}$ then $S$ deletes $x_i$. We may as well do that first. So $D(i, j) \leq 1 + D(i - 1, j)$.
- If not, and if $S$ does not delete $x_i$, then either it substitutes $x_i$ or inserts after $x_i$. 
Proof, continued...

- If $S$ does not delete $x_i$, then it substitutes $x_i$ or inserts after $x_i$. 
Proof, continued...

- If \( S \) does not delete \( x_i \), then it substitutes \( x_i \) or inserts after \( x_i \).
- If it substitutes \( x_i := y_j \) then we can do that first (or last), so
  \[ D(i, j) \leq 1 + D(i - 1, j - 1). \]
If $S$ does not delete $x_i$, then it substitutes $x_i$ or inserts after $x_i$.

If it substitutes $x_i := y_j$ then we can do that first (or last), so
$D(i, j) \leq 1 + D(i-1, j-1)$.

Else, we insert $y_j$ after the position occupied by $x_i$. Again we can
just as well do that last, having produced $y_1 \cdots y_{j-1}$. So
$D(i, j) \leq 1 + D(i, j-1)$ in that case. 

Proof, continued...
Proof, continued...

- If $S$ does not delete $x_i$, then it substitutes $x_i$ or inserts after $x_i$.
- If it substitutes $x_i := y_j$ then we can do that first (or last), so $D(i, j) \leq 1 + D(i - 1, j - 1)$.
- Else, we insert $y_j$ after the position occupied by $x_i$. Again we can just as well do that last, having produced $y_1 \cdots y_{j-1}$. So $D(i, j) \leq 1 + D(i, j - 1)$ in that case.
- One case must hold, so proved. □
Proof, continued...

- If $S$ does not delete $x_i$, then it substitutes $x_i$ or inserts after $x_i$.
- If it substitutes $x_i := y_j$ then we can do that first (or last), so $D(i, j) \leq 1 + D(i - 1, j - 1)$.
- Else, we insert $y_j$ after the position occupied by $x_i$. Again we can just as well do that last, having produced $y_1 \cdots y_{j-1}$. So $D(i, j) \leq 1 + D(i, j - 1)$ in that case.
- One case must hold, so proved. □

"Calcutta Example": Clearly $D(1, 1) = d(C, K) = 1$. So

\[
D(2, 1) = d(Ca, K) = 1 + \min\{D(1, 0), D(1, 1), D(2, 0)\} \\
= 1 + \min\{d(C, \lambda), d(C, K), d(Ca, \lambda)\} = 2.
\]
Proof, continued...

- If $S$ does not delete $x_i$, then it substitutes $x_i$ or inserts after $x_i$.
- If it substitutes $x_i := y_j$ then we can do that first (or last), so $D(i, j) \leq 1 + D(i - 1, j - 1)$.
- Else, we insert $y_j$ after the position occupied by $x_i$. Again we can just as well do that last, having produced $y_1 \cdots y_{j-1}$. So $D(i, j) \leq 1 + D(i, j - 1)$ in that case.
- One case must hold, so proved. □

“Calcutta Example”: Clearly $D(1, 1) = d(C, K) = 1$. So

$$D(2, 1) = d(Ca, K = 1 + \min\{D(1, 0), D(1, 1), D(2, 0)\}$$

$$= 1 + \min\{d(C, \lambda), d(C, K), d(Ca, \lambda)\} = 2.$$

Next $D(1, 2) = d(C, Ko) = 2$ and $D(2, 2) = d(Ca, Ko) = 2$ and

$$D(3, 3) = D(2, 2) = 2 \quad because \quad x_3 = y_3 = l.$$
Proof, continued...

- If $S$ does not delete $x_i$, then it substitutes $x_i$ or inserts after $x_i$.
- If it substitutes $x_i := y_j$ then we can do that first (or last), so $D(i, j) \leq 1 + D(i - 1, j - 1)$.
- Else, we insert $y_j$ after the position occupied by $x_i$. Again we can just as well do that last, having produced $y_1 \cdots y_{j-1}$. So $D(i, j) \leq 1 + D(i, j - 1)$ in that case.
- One case must hold, so proved. □

"Calcutta Example": Clearly $D(1, 1) = d(C, K) = 1$. So
\[
D(2, 1) = d(Ca, K = 1 + \min\{D(1, 0), D(1, 1), D(2, 0)\}
\]
\[
= 1 + \min\{d(C, \lambda), d(C, K), d(Ca, \lambda)\} = 2.
\]
Next $D(1, 2) = d(C, Ko) = 2$ and $D(2, 2) = d(Ca, Ko) = 2$ and
\[
D(3, 3) = D(2, 2) = 2 \quad \text{because} \quad x_3 = y_3 = l.
\]
Building up, we eventually get $D(8, 7) = 5$ (exercise).
Big Issue: Can We Improve the Time?

Can we improve the \( \Theta(mn) \) running time to \( O(m + n) \)?
Big Issue: Can We Improve the Time?

Can we improve the $\Theta(mn)$ running time to $O(m + n)$? or to $\tilde{O}(m + n)$ ignoring any factors of $\log(m + n)$? or at least to $O((m + n)^{2-\epsilon})$ for some $\epsilon > 0$ so the time is better than quadratic?
Can we improve the $\Theta(mn)$ running time to $O(m+n)$? or to $
abla O(m+n)$ ignoring any factors of $\log(m+n)$? or at least to $O((m+n)^{2-\epsilon})$ for some $\epsilon > 0$ so the time is better than quadratic?

- Would have huge impact in gene sequencing, for instance.
Big Issue: Can We Improve the Time?

Can we improve the $\Theta(mn)$ running time to $O(m + n)$? or to $\tilde{O}(m + n)$ ignoring any factors of $\log(m + n)$? or at least to $O((m + n)^{2-\epsilon})$ for some $\epsilon > 0$ so the time is better than quadratic?

- Would have huge impact in gene sequencing, for instance.
- Can we “jump the table,” as for Fibonacci Numbers $F_n$?
Big Issue: Can We Improve the Time?

Can we improve the $\Theta(mn)$ running time to $O(m + n)$? or to $\tilde{O}(m + n)$ ignoring any factors of $\log(m + n)$? or at least to $O((m + n)^{2-\epsilon})$ for some $\epsilon > 0$ so the time is better than quadratic?

- Would have huge impact in gene sequencing, for instance.
- Can we “jump the table,” as for Fibonacci Numbers $F_n$?
- The formula $F_n = F_{n-1} + F_{n-2}$ is a great definition...
Big Issue: Can We Improve the Time?

Can we improve the $\Theta(mn)$ running time to $O(m + n)$? or to $\tilde{O}(m + n)$ ignoring any factors of $\log(m + n)$? or at least to $O((m + n)^{2-\epsilon})$ for some $\epsilon > 0$ so the time is better than quadratic?

- Would have huge impact in gene sequencing, for instance.
- Can we “jump the table,” as for Fibonacci Numbers $F_n$?
- The formula $F_n = F_{n-1} + F_{n-2}$ is a great definition... but a lousy recursion.
Big Issue: Can We Improve the Time?

Can we improve the $\Theta(mn)$ running time to $O(m + n)$? or to $\tilde{O}(m + n)$ ignoring any factors of $\log(m + n)$? or at least to $O((m + n)^{2-\epsilon})$ for some $\epsilon > 0$ so the time is better than quadratic?

- Would have huge impact in gene sequencing, for instance.
- Can we “jump the table,” as for Fibonacci Numbers $F_n$?
- The formula $F_n = F_{n-1} + F_{n-2}$ is a great definition... but a lousy recursion.
- Better is $(F_n, F_{n-1}) = (2F_{n-2} + F_{n-3}, F_{n-2} + F_{n-3})$: $O(n)$ time.
Big Issue: Can We Improve the Time?

Can we improve the $\Theta(mn)$ running time to $O(m + n)$? or to $\tilde{O}(m + n)$ ignoring any factors of $\log(m + n)$? or at least to $O((m + n)^{2-\epsilon})$ for some $\epsilon > 0$ so the time is better than quadratic?

- Would have huge impact in gene sequencing, for instance.
- Can we “jump the table,” as for Fibonacci Numbers $F_n$?
- The formula $F_n = F_{n-1} + F_{n-2}$ is a great definition... but a lousy recursion.
- Better is $(F_n, F_{n-1}) = (2F_{n-2} + F_{n-3}, F_{n-2} + F_{n-3})$: $O(n)$ time.
- Filling table iteratively not recursively is simple and good.
Big Issue: Can We Improve the Time?

Can we improve the $\Theta(mn)$ running time to $O(m + n)$? or to $\tilde{O}(m + n)$ ignoring any factors of $\log(m + n)$? or at least to $O((m + n)^{2-\varepsilon})$ for some $\varepsilon > 0$ so the time is better than quadratic?

- Would have huge impact in gene sequencing, for instance.
- Can we “jump the table,” as for Fibonacci Numbers $F_n$?
- The formula $F_n = F_{n-1} + F_{n-2}$ is a great definition... but a lousy recursion.
- Better is $(F_n, F_{n-1}) = (2F_{n-2} + F_{n-3}, F_{n-2} + F_{n-3})$: $O(n)$ time.
- Filling table iteratively not recursively is simple and good.
- But can we compute $F_n$ without computing $F_{n-1}$ or $F_{n-2}$—and without any fancy arithmetic like powers of the golden ratio?
Big Issue: Can We Improve the Time?

Can we improve the $\Theta(mn)$ running time to $O(m + n)$? or to $\tilde{O}(m + n)$ ignoring any factors of $\log(m + n)$? or at least to $O((m + n)^{2-\epsilon})$ for some $\epsilon > 0$ so the time is better than quadratic?

- Would have huge impact in gene sequencing, for instance.
- Can we “jump the table,” as for Fibonacci Numbers $F_n$?
- The formula $F_n = F_{n-1} + F_{n-2}$ is a great definition... but a lousy recursion.
- Better is $(F_n, F_{n-1}) = (2F_{n-2} + F_{n-3}, F_{n-2} + F_{n-3})$: $O(n)$ time.
- Filling table iteratively not recursively is simple and good.
- But can we compute $F_n$ without computing $F_{n-1}$ or $F_{n-2}$—and without any fancy arithmetic like powers of the golden ratio?

- Surprise(?) yes: keep squaring $M = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$. 
Big Issue: Can We Improve the Time?

Can we improve the $\Theta(mn)$ running time to $O(m + n)$? or to $\tilde{O}(m + n)$ ignoring any factors of $\log(m + n)$? or at least to $O((m + n)^{2-\epsilon})$ for some $\epsilon > 0$ so the time is better than quadratic?

- Would have huge impact in gene sequencing, for instance.
- Can we “jump the table,” as for Fibonacci Numbers $F_n$?
- The formula $F_n = F_{n-1} + F_{n-2}$ is a great definition... but a lousy recursion.
- Better is $(F_n, F_{n-1}) = (2F_{n-2} + F_{n-3}, F_{n-2} + F_{n-3})$: $O(n)$ time.
- Filling table iteratively not recursively is simple and good.
- But can we compute $F_n$ without computing $F_{n-1}$ or $F_{n-2}$—and without any fancy arithmetic like powers of the golden ratio?

- Surprise(?) yes: keep squaring $M = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$.
- But for ED, new “Puzzling Evidence” that $\Theta(mn)$ cannot be avoided.
Original Third Lecture Day...

Shorter, done from board:

- Sorting is a “Good Guy.”
Original Third Lecture Day...

Shorter, done from board:

- Sorting is a “Good Guy.”
- Parallel Prefix Sum
Original Third Lecture Day...

Shorter, done from board:

- Sorting is a "Good Guy."
- Parallel Prefix Sum
- Map-Reduce in the Abstract.
Shorter, done from board:

- Sorting is a “Good Guy.”
- Parallel Prefix Sum
- Map-Reduce in the Abstract.
- Log-Depth Circuits and Cloud-Friendly Algorithms.