CSCI 5451 Fall 2015

Week 4 Notes

Professor Ellen Gethner

September 7, 2015

Dynamic Programming: Second Big Example

- We now study a special case of *sequence comparisons*, a problem that was inspired by molecular biology.

Dynamic Programming: Second Big Example

- We now study a special case of *sequence comparisons*, a problem that was inspired by molecular biology.

- **Problem.** Find the minimum number of *edit steps* required to change one string to another.

Dynamic Programming: Second Big Example

- We now study a special case of *sequence comparisons*, a problem that was inspired by molecular biology.

- **Problem.** Find the minimum number of *edit steps* required to change one string to another.

- The strings need not be the same length.

Dynamic Programming: Second Big Example

- We now study a special case of *sequence comparisons*, a problem that was inspired by molecular biology.
- **Problem.** Find the minimum number of *edit steps* required to change one string to another.
- The strings need not be the same length.
- **Set-up.** Let $A = a_1 a_2 a_3 \ldots a_n$ and let $B = b_1 b_2 b_3 \ldots b_m$.

Dynamic Programming: Second Big Example

- We now study a special case of *sequence comparisons*, a problem that was inspired by molecular biology.
- **Problem.** Find the minimum number of *edit steps* required to change one string to another.
- The strings need not be the same length.
- **Set-up.** Let $A = a_1 a_2 a_3 \dots a_n$ and let $B = b_1 b_2 b_3 \dots b_m$.
- Assume the universe of characters is finite.

Dynamic Programming: Second Big Example

- We now study a special case of *sequence comparisons*, a problem that was inspired by molecular biology.
- **Problem.** Find the minimum number of *edit steps* required to change one string to another.
- The strings need not be the same length.
- **Set-up.** Let $A = a_1 a_2 a_3 \ldots a_n$ and let $B = b_1 b_2 b_3 \ldots b_m$.
- Assume the universe of characters is finite.
- We want to change A, character by character, to B.

Operations on Strings

- We'll allow the following *edit steps* (and one more later).

1. **Insert.** Insert one character in the string.
2. **Delete.** Delete one character from the string.
3. **Replace.** Replace one character with another.

Operations on Strings

- We'll allow the following *edit steps* (and one more later).

  1. **Insert.** Insert one character in the string.
  2. **Delete.** Delete one character from the string.
  3. **Replace.** Replace one character with another.

- Each edit step above is assigned a cost of 1.

Operations on Strings

- We'll allow the following *edit steps* (and one more later).

  1. **Insert.** Insert one character in the string.
  2. **Delete.** Delete one character from the string.
  3. **Replace.** Replace one character with another.

- Each edit step above is assigned a cost of 1.

- **Example.** Transform *abbc* into *babb*.

Transform *abbc* into *babb*

- **Here goes.**

Transform *abbc* into *babb*

- **Here goes.**

- $abbc \to abc$ by deleting a b.

Transform *abbc* into *babb*

- **Here goes.**
- $abbc \to abc$ by deleting a b.
- $abc \to babc$ by inserting a b at the beginning of the string.

Transform *abbc* into *babb*

- **Here goes.**
- $abbc \to abc$ by deleting a b.
- $abc \to babc$ by inserting a b at the beginning of the string.
- $babc \to babb$ by replacing the c with a b.

Transform *abbc* into *babb*

- **Here goes.**
- $abbc \rightarrow abc$ by deleting a b.
- $abc \rightarrow babc$ by inserting a b at the beginning of the string.
- $babc \rightarrow babb$ by replacing the c with a b.
- The cost of the above set of edit steps is 3.

Transform *abbc* into *babb*

- **Here goes.**
- $abbc \rightarrow abc$ by deleting a b.
- $abc \rightarrow babc$ by inserting a b at the beginning of the string.
- $babc \rightarrow babb$ by replacing the c with a b.
- The cost of the above set of edit steps is 3.
- But were we efficient?

Transform *abbc* into *babb*

- **Here goes.**
- *abbc* $\rightarrow$ *abc* by deleting a *b*.
- *abc* $\rightarrow$ *babc* by inserting a *b* at the beginning of the string.
- *babc* $\rightarrow$ *babb* by replacing the *c* with a *b*.
- The cost of the above set of edit steps is 3.
- But were we efficient?
- No: *abbc* $\rightarrow$ *abb* $\rightarrow$ *babb* has a cost of 2, and is best possible. Why?

An aside

- The idea of using edit steps to convert one string into another has applications to *file comparisons* and *revisions*

An aside

- The idea of using edit steps to convert one string into another has applications to *file comparisons* and *revisions*
- such as the *diff* file, which is based on today's (upcoming) algorithm.

How to find an optimal set of edit steps

- Let's first set things up with the following notational help.

How to find an optimal set of edit steps

- Let's first set things up with the following notational help.
- Recall that $A = a_1 a_2 \ldots a_n$ and $B = b_1 b_2 \ldots b_m$.

How to find an optimal set of edit steps

- Let's first set things up with the following notational help.

- Recall that $A = a_1 a_2 \ldots a_n$ and $B = b_1 b_2 \ldots b_m$.

- Let $A(i) = a_1 a_2 \ldots a_i$ (i.e. $A(i)$ is the first i characters of string A), and

How to find an optimal set of edit steps

- Let's first set things up with the following notational help.
- Recall that $A = a_1 a_2 \ldots a_n$ and $B = b_1 b_2 \ldots b_m$.
- Let $A(i) = a_1 a_2 \ldots a_i$ (i.e. $A(i)$ is the first i characters of string A), and
- $B(i) = b_1 b_2 \ldots b_i$ (i.e. $B(i)$ is the first i characters of string B).

How to find an optimal set of edit steps

- Let's first set things up with the following notational help.
- Recall that $A = a_1 a_2 \ldots a_n$ and $B = b_1 b_2 \ldots b_m$.
- Let $A(i) = a_1 a_2 \ldots a_i$ (i.e. $A(i)$ is the first i characters of string A), and
- $B(i) = b_1 b_2 \ldots b_i$ (i.e. $B(i)$ is the first i characters of string B).
- **Restatement of the Problem.** Change $A(n)$ to $B(m)$ with the minimum number of edit steps.

Analysis of the Optimization Problem at Hand

- Put your induction cap on.

Analysis of the Optimization Problem at Hand

- Put your induction cap on.
- Suppose we know the best way to change $A(n-1)$ to $B(m)$ by induction.

Analysis of the Optimization Problem at Hand

- Put your induction cap on.
- Suppose we know the best way to change $A(n-1)$ to $B(m)$ by induction.
- Assume we only know one possible solution.

Analysis of the Optimization Problem at Hand

- Put your induction cap on.
- Suppose we know the best way to change $A(n-1)$ to $B(m)$ by induction.
- Assume we only know one possible solution.
- Then with one deletion, namely a_n from $A(n)$, we know how to convert $A(n)$ to $B(m)$.

Analysis of the Optimization Problem at Hand

- Put your induction cap on.
- Suppose we know the best way to change $A(n-1)$ to $B(m)$ by induction.
- Assume we only know one possible solution.
- Then with one deletion, namely a_n from $A(n)$, we know how to convert $A(n)$ to $B(m)$.
- But the above idea may not be the best way! What else can happen?

Optimization ideas, continued

- It might be better to replace a_n with b_m.

Optimization ideas, continued

- It might be better to replace a_n with b_m.

- Or it might be the case that character a_n is the same as character b_m.

Optimization ideas, continued

- It might be better to replace a_n with b_m.

- Or it might be the case that character a_n is the same as character b_m.

- We need a systematic way of enumerating the choices. For now, concentrate only on the cost (and not the actual edit steps.

Optimization ideas, continued

- It might be better to replace a_n with b_m.
- Or it might be the case that character a_n is the same as character b_m.
- We need a systematic way of enumerating the choices. For now, concentrate only on the cost (and not the actual edit steps.
- Let $C(i, j)$ be the minimum cost of changing $A(i)$ to $B(j)$.

Optimization ideas, continued

- It might be better to replace a_n with b_m.
- Or it might be the case that character a_n is the same as character b_m.
- We need a systematic way of enumerating the choices. For now, concentrate only on the cost (and not the actual edit steps.
- Let $C(i,j)$ be the minimum cost of changing $A(i)$ to $B(j)$.
- **Goal.** Find a relation between $C(n,m)$ and some of the $C(i,j)$'s for some smaller values of i and j.

Cost of individual edit steps

- **Case 1: Delete.** If a_n is deleted in a minimum change from A to B then $C(n, m) = C(n-1, m) + 1$.

Cost of individual edit steps

- **Case 1: Delete.** If a_n is deleted in a minimum change from A to B then $C(n, m) = C(n-1, m) + 1$.
- **Case 2: Insert.** If the minimum change from A to B requires an insertion of a charcter to match b_m then $C(n, m) = C(n, m-1) + 1$.

Cost of individual edit steps

- **Case 1: Delete.** If a_n is deleted in a minimum change from A to B then $C(n, m) = C(n-1, m) + 1$.
- **Case 2: Insert.** If the minimum change from A to B requires an insertion of a charcter to match b_m then $C(n, m) = C(n, m-1) + 1$.
- That is, we find by induction the minimum change from $A(n)$ to $B(m-1)$ and then insert b_m at the end of the string.

Cost of individual edit steps

- **Case 1: Delete.** If a_n is deleted in a minimum change from A to B then $C(n, m) = C(n-1, m) + 1$.
- **Case 2: Insert.** If the minimum change from A to B requires an insertion of a charcter to match b_m then $C(n, m) = C(n, m-1) + 1$.
- That is, we find by induction the minimum change from $A(n)$ to $B(m-1)$ and then insert b_m at the end of the string.
- **Case 3: Replace.** If a_n is is replacing b_m then we first find the minimum cost of changing $A(n-1)$ to $B(m-1)$ and add 1 to the cost as long as $a_n \neq b_m$.

Cost of individual edit steps

- **Case 1: Delete.** If a_n is deleted in a minimum change from A to B then $C(n,m) = C(n-1,m) + 1$.

- **Case 2: Insert.** If the minimum change from A to B requires an insertion of a charcter to match b_m then $C(n,m) = C(n,m-1) + 1$.

- That is, we find by induction the minimum change from $A(n)$ to $B(m-1)$ and then insert b_m at the end of the string.

- **Case 3: Replace.** If a_n is is replacing b_m then we first find the minimum cost of changing $A(n-1)$ to $B(m-1)$ and add 1 to the cost as long as $a_n \neq b_m$.

- **Case 4: Match.** If $a_n = b_m$ then $C(n,m) = C(n-1,m-1)$.

Tallying the cost

- Let

$$c(i,j) = \begin{cases} 0 & \text{if } a_i = b_j \\ 1 & \text{if } a_i \neq b_j \end{cases}$$

Tallying the cost

- Let

$$c(i,j) = \begin{cases} 0 & \text{if } a_i = b_j \\ 1 & \text{if } a_i \neq b_j \end{cases}$$

- In summary,

$$C(n,m) = min \begin{cases} C(n-1,m)+1 & \text{delete } a_n \\ C(n,m-1)+1 & \text{insert } b_m \\ C(n-1,m-1)+c(n,m) & \text{possible match,} \end{cases}$$

Tallying the cost

- Let

$$c(i,j) = \begin{cases} 0 & \text{if } a_i = b_j \\ 1 & \text{if } a_i \neq b_j \end{cases}$$

- In summary,

$$C(n,m) = min \begin{cases} C(n-1,m)+1 & \text{delete } a_n \\ C(n,m-1)+1 & \text{insert } b_m \\ C(n-1,m-1)+c(n,m) & \text{possible match,} \end{cases}$$

- where $C(i,0) = i$ for $i = 0, 1, \ldots, n$, and

Tallying the cost

- Let

$$c(i,j) = \begin{cases} 0 & \text{if } a_i = b_j \\ 1 & \text{if } a_i \neq b_j \end{cases}$$

- In summary,

$$C(n,m) = min \begin{cases} C(n-1,m)+1 & \text{delete } a_n \\ C(n,m-1)+1 & \text{insert } b_m \\ C(n-1,m-1)+c(n,m) & \text{possible match,} \end{cases}$$

- where $C(i,0) = i$ for $i = 0,1,\ldots,n$, and
- $C(0,j) = j$ for $j = 0,1,\ldots,m$.

Tallying the cost

- Let
$$c(i,j) = \begin{cases} 0 & \text{if } a_i = b_j \\ 1 & \text{if } a_i \neq b_j \end{cases}$$
- In summary,
$$C(n,m) = min \begin{cases} C(n-1,m)+1 & \text{delete } a_n \\ C(n,m-1)+1 & \text{insert } b_m \\ C(n-1,m-1)+c(n,m) & \text{possible match,} \end{cases}$$
- where $C(i,0) = i$ for $i = 0, 1, \ldots, n$, and
- $C(0,j) = j$ for $j = 0, 1, \ldots, m$.
- Have we captured all possibilities for the edit steps?

All scenarios accounted for, continued

- The character a_n must be handled.

All scenarios accounted for, continued

- The character a_n must be handled.
- If a_n is deleted, then this is Case 1.

All scenarios accounted for, continued

- The character a_n must be handled.
- If a_n is deleted, then this is Case 1.
- If a_n becomes b_m then this is either Case 3 or Case 4.

All scenarios accounted for, continued

- The character a_n must be handled.
- If a_n is deleted, then this is Case 1.
- If a_n becomes b_m then this is either Case 3 or Case 4.
- If a_n is mapped to an earlier character, then this is Case 2.

Moral of the Story, so far

- The idea is sound, BUT

Moral of the Story, so far

- The idea is sound, BUT

- we have reduced one problem of size n to three problems of similar size,

Moral of the Story, so far

- The idea is sound, BUT
- we have reduced one problem of size n to three problems of similar size,
- so the faux algorithm is exponential.

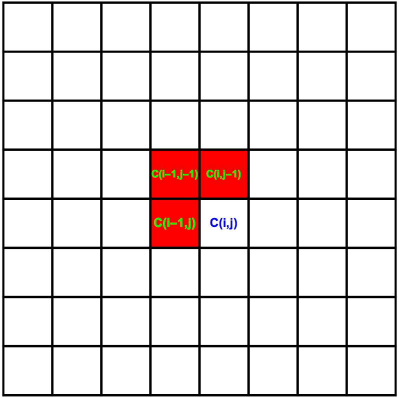
Moral of the Story, so far

- The idea is sound, BUT
- we have reduced one problem of size n to three problems of similar size,
- so the faux algorithm is exponential.
- **Reality.** There are only $(n+1) \times (m+1)$ subproblems so we should keep track of information as we acquire it.

Moral of the Story, so far

- The idea is sound, BUT
- we have reduced one problem of size n to three problems of similar size,
- so the faux algorithm is exponential.
- **Reality.** There are only $(n+1) \times (m+1)$ subproblems so we should keep track of information as we acquire it.
- To do so, we'll use an $n+1$ by $m+1$ array such that the (i,j)th entry contains information about $C(i,j)$.

Artist's interpretation of the array



$C(i,j)$ is computed from information in the red area and in an implementation you maintain an array C such that $C[i,j]$ contains information about the problem $C(i,j)$.

Algorithm MinEdDis(A, n, B, m)

- **Input** A (string of length n) and B (string of length m)
- **Output** C, the $n+1 \times m+1$ cost matrix.
- **begin**
- **for** $i = 0$ **to** n **do** $C[i, 0] = i$
- **for** $j = 0$ **to** m **do** $C[0, j] = j$
- **for** $i = 1$ **to** n **do**
- **for** $j = 1$ **to** m **do**
- $x = C[i-1, j] + 1$
- $y = C[i, j-1] + 1$
- **if** $a_i = b_i$ **then** $z = C[i-1, j-1]$
- **else** $z = C[i-1, j-1] + 1$
- $C[i, j] = min(x, y, z)$
- **end**

Complexity of Algorithm MinEdDis

- The run time is $O(mn)$ (as is storage).

Complexity of Algorithm MinEdDis

- The run time is $O(mn)$ (as is storage).

- Another moral of the story is that Dynamic Programming is speedy but uses up memory.

Complexity of Algorithm MinEdDis

- The run time is $O(mn)$ (as is storage).
- Another moral of the story is that Dynamic Programming is speedy but uses up memory.
- **Exercise.** Think about how you would keep track of each edit step.

Next: Greedy Algorithms

- When faced with several choices, make one that is

Next: Greedy Algorithms

- When faced with several choices, make one that is
- best possible at this point, even though it may lead to a non-optimal solution.

Next: Greedy Algorithms

- When faced with several choices, make one that is
- best possible at this point, even though it may lead to a non-optimal solution.
- In other words, be greedy at each step.

Next: Greedy Algorithms

- When faced with several choices, make one that is
- best possible at this point, even though it may lead to a non-optimal solution.
- In other words, be greedy at each step.
- **Greedy Algorithms** are typically designed for optimization problems, where

Greed Algorithms

- **Greedy Algorithms** are typically designed for optimization problems, where

Greed Algorithms

- **Greedy Algorithms** are typically designed for optimization problems, where
- there are several possible legal solutions,

Greed Algorithms

- **Greedy Algorithms** are typically designed for optimization problems, where
- there are several possible legal solutions,
- there is a value (usually numerical) associated with each solution, and

Greed Algorithms

- **Greedy Algorithms** are typically designed for optimization problems, where
- there are several possible legal solutions,
- there is a value (usually numerical) associated with each solution, and
- we want to find a legal solution with the minimum or maximum possible value, depending on the context of the problem.

Greedy Algorithm: Big Example 1

- **Scheduling**

Greedy Algorithm: Big Example 1

- **Scheduling**

- **Set-up.** We are given n jobs to execute;

Greedy Algorithm: Big Example 1

- **Scheduling**
- **Set-up.** We are given n jobs to execute;
- The ith job has start time s_i and finish time f_i.

Greedy Algorithm: Big Example 1

- **Scheduling**
- **Set-up.** We are given n jobs to execute;
- The ith job has start time s_i and finish time f_i.
- **Goal.** Find a maximum set of nonoverlapping jobs,

Greedy Algorithm: Big Example 1

- **Scheduling**
- **Set-up.** We are given n jobs to execute;
- The ith job has start time s_i and finish time f_i.
- **Goal.** Find a maximum set of nonoverlapping jobs,
- where job i and job j are considered to be nonoverlapping iff $[s_i, f_i) \cap [s_j, f_j) = \varnothing$.

Greedy Algorithm: Big Example 1

- **Scheduling**
- **Set-up.** We are given n jobs to execute;
- The ith job has start time s_i and finish time f_i.
- **Goal.** Find a maximum set of nonoverlapping jobs,
- where job i and job j are considered to be nonoverlapping iff $[s_i, f_i) \cap [s_j, f_j) = \varnothing$.
- Alternatively, jobs i and j are nonoverlapping as long as either $s_i \geq f_j$ or $s_j \geq f_i$.

Artist's Interpretation of the Scheduling Problem

- Here are three jobs with start and finish times, respectively, s_1, f_1, s_2, f_2, and s_3, f_3.

Artist's Interpretation of the Scheduling Problem

- Here are three jobs with start and finish times, respectively, s_1, f_1, s_2, f_2, and s_3, f_3.

- 

Artist's Interpretation of the Scheduling Problem

- Here are three jobs with start and finish times, respectively, s_1, f_1, s_2, f_2, and s_3, f_3.
- 

- Impossible to see.

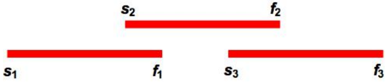
Artist's Interpretation of the Scheduling Problem

- Here are three jobs with start and finish times, respectively, s_1, f_1, s_2, f_2, and s_3, f_3.

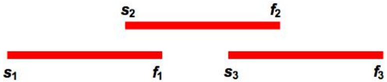
- 

- Impossible to see.

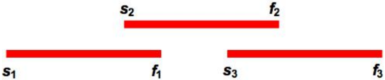
- Here's a slight tweak so you can see all three jobs.

- 

Scheduling, continued

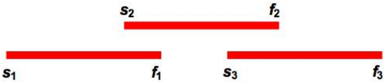
- 

Scheduling, continued

- 

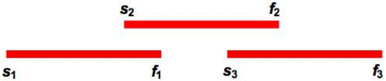
- We can see that jobs 1 and 2 overlap as do jobs 2 and 3.

Scheduling, continued

- 

- We can see that jobs 1 and 2 overlap as do jobs 2 and 3.
- But jobs 1 and 3 do not overlap.

Scheduling, continued

- 

- We can see that jobs 1 and 2 overlap as do jobs 2 and 3.

- But jobs 1 and 3 do not overlap.

- Next, an algorithm to maximize the number of non-overlapping jobs...

Algorithm GreedySched(S)

- **Input:** $S = \{(s_1, f_1), (s_2, f_2), \ldots, (s_n, f_n)\}$.
- **Output:** A largest set of non-overlapping jobs.
- Sort jobs by finish time: $f_1 \leq f_2 \leq \cdots \leq f_n$
- $A = \varnothing$
- **for** $i = 1$ **to** n **do**
- **if** job i does not overlap any job in A
- **then** $A = A \cup \{f_i\}$
- **return** A

Proof that Algorithm GreedySched is correct

- **Notation.** Let $G(S)$ be the set of jobs produced by algorithm GreedySched with input S.

Proof that Algorithm GreedySched is correct

- **Notation.** Let $G(S)$ be the set of jobs produced by algorithm GreedySched with input S.
- And let $OPT(S)$ be any set of optimal jobs given input S.

Proof that Algorithm GreedySched is correct

- **Notation.** Let $G(S)$ be the set of jobs produced by algorithm GreedySched with input S.
- And let $OPT(S)$ be any set of optimal jobs given input S.
- That is, $OPT(S)$ is a largest set of non-overlapping jobs from S. Then

Proof that Algorithm GreedySched is correct

- **Notation.** Let $G(S)$ be the set of jobs produced by algorithm GreedySched with input S.
- And let $OPT(S)$ be any set of optimal jobs given input S.
- That is, $OPT(S)$ is a largest set of non-overlapping jobs from S. Then

- Theorem

  The algorithm GreedySched produces an optimal set of nonoverlapping jobs. In particular, $|G(S)| = |OPT(S)|$ *where* $OPT(S)$ *is some largest set of nonoverlapping jobs in* S.

Proof that Algorithm GreedySched is correct

- **Proof.**

Proof that Algorithm GreedySched is correct

- **Proof.**

- It suffices to show that both $|G(S)| \leq |OPT(S)|$ and $|OPT(S)| \leq |G(S)|$ are true.

Proof that Algorithm GreedySched is correct

- **Proof.**
- It suffices to show that both $|G(S)| \leq |OPT(S)|$ and $|OPT(S)| \leq |G(S)|$ are true.
- Note that trivially we have $|G(S)| \leq |OPT(S)|$. Why?

Proof that Algorithm GreedySched is correct

- **Proof.**
- It suffices to show that both $|G(S)| \leq |OPT(S)|$ and $|OPT(S)| \leq |G(S)|$ are true.
- Note that trivially we have $|G(S)| \leq |OPT(S)|$. Why?
- So it remains to show that $|OPT(S)| \leq |G(S)|$.

Proof that Algorithm GreedySched is correct

- **Proof.**
- It suffices to show that both $|G(S)| \leq |OPT(S)|$ and $|OPT(S)| \leq |G(S)|$ are true.
- Note that trivially we have $|G(S)| \leq |OPT(S)|$. Why?
- So it remains to show that $|OPT(S)| \leq |G(S)|$.
- To that end, suppose BWOC that $|OPT(S)| > |G(S)|$.

Proof that Algorithm GreedySched is correct, continued

- Suppose BWOC that $|OPT(S)| > |G(S)|$.

Proof that Algorithm GreedySched is correct, continued

- Suppose BWOC that $|OPT(S)| > |G(S)|$.
- **Set-up.** Choose $OPT(S)$ to be an optimal set of jobs that agrees with $G(S)$ on the largest possible number of jobs.

Proof that Algorithm GreedySched is correct, continued

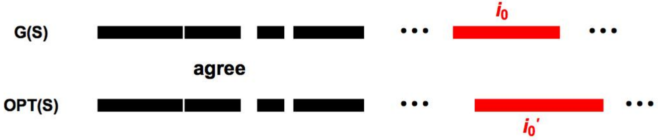
- Suppose BWOC that $|OPT(S)| > |G(S)|$.
- **Set-up.** Choose $OPT(S)$ to be an optimal set of jobs that agrees with $G(S)$ on the largest possible number of jobs.
- Since $G(S)$ and $OPT(S)$ are different sets, there exists a job i_0 that is not in both of the sets $G(S)$ and $OPT(S)$.

Proof of correctness, continued

- **Case 1.** $i_0 \in G(S)$ (and thus $i_0 \notin OPT(S)$).

Proof of correctness, continued

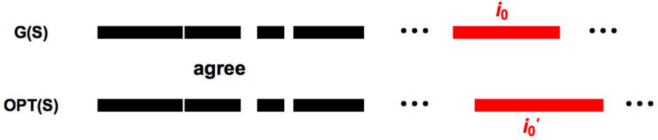
- **Case 1.** $i_0 \in G(S)$ (and thus $i_0 \notin OPT(S)$).



-

Proof of correctness, continued

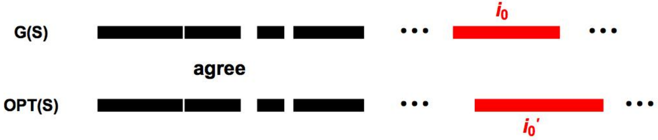
- **Case 1.** $i_0 \in G(S)$ (and thus $i_0 \notin OPT(S)$).

- 

- Then the finish time of i_0 is less than or equal to the finish time of i'_0 because

Proof of correctness, continued

- **Case 1.** $i_0 \in G(S)$ (and thus $i_0 \notin OPT(S)$).



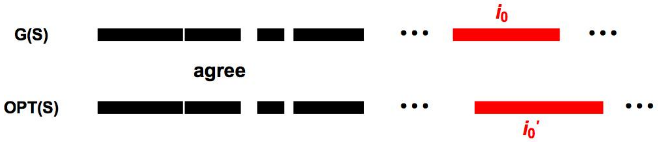
-

- Then the finish time of i_0 is less than or equal to the finish time of i'_0 because

- otherwise algorithm GreedySched would have chosen i'_0 over i_0.

Proof of correctness, continued

- **Case 1.** $i_0 \in G(S)$ (and thus $i_0 \notin OPT(S)$).



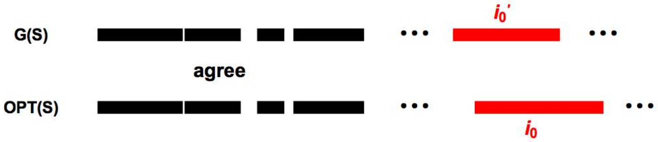
- Then the finish time of i_0 is less than or equal to the finish time of i_0' because
- otherwise algorithm GreedySched would have chosen i_0' over i_0.
- In that case, we simply replace i_0' with i_0 in $OPT(S)$ thus producing a new $OPT(S)$ that agrees with one more job in $G(S)$, a contradiction.

Proof of correctness, Case 2.

- **Case 2.** $i_0 \in OPT(S)$ (and thus $i_0 \notin G(S)$).

Proof of correctness, Case 2.

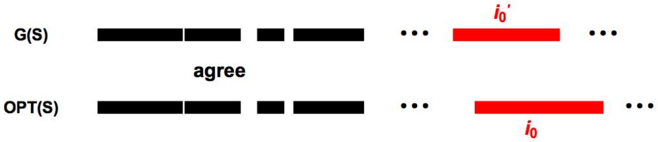
- **Case 2.** $i_0 \in OPT(S)$ (and thus $i_0 \notin G(S)$).



-

Proof of correctness, Case 2.

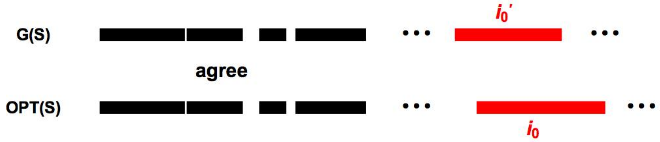
- **Case 2.** $i_0 \in OPT(S)$ (and thus $i_0 \notin G(S)$).



- Then the finish time of i_0' is less than or equal to the finish time of i_0 because

Proof of correctness, Case 2.

- **Case 2.** $i_0 \in OPT(S)$ (and thus $i_0 \notin G(S)$).

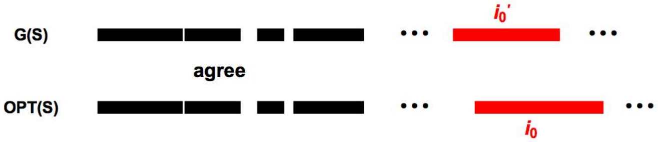


- Then the finish time of i_0' is less than or equal to the finish time of i_0 because

- otherwise algorithm GreedySched would have chosen i_0 over i_0'.

Proof of correctness, Case 2.

- **Case 2.** $i_0 \in OPT(S)$ (and thus $i_0 \notin G(S)$).



- Then the finish time of i_0' is less than or equal to the finish time of i_0 because
- otherwise algorithm GreedySched would have chosen i_0 over i_0'.
- In that case, we simply replace i_0 with i_0' in $OPT(S)$ thus producing a new $OPT(S)$ that agrees with one more job in $G(S)$, a contradiction.

Proof of Correctness of GreedySched concluded

- We have shown that $|G(S)| = |OPT(S)|$ and thus the algorithm GreedySched(S) produces a set of nonoverlapping jobs that is equal in cardinality to a largest set of nonoverlapping jobs.

Proof of Correctness of GreedySched concluded

- We have shown that $|G(S)| = |OPT(S)|$ and thus the algorithm GreedySched(S) produces a set of nonoverlapping jobs that is equal in cardinality to a largest set of nonoverlapping jobs.

- Thus algorithm GreedySched is correct.

Proof of Correctness of GreedySched concluded

- We have shown that $|G(S)| = |OPT(S)|$ and thus the algorithm GreedySched(S) produces a set of nonoverlapping jobs that is equal in cardinality to a largest set of nonoverlapping jobs.
- Thus algorithm GreedySched is correct.
- **QED**

Proof of Correctness of GreedySched concluded

- We have shown that $|G(S)| = |OPT(S)|$ and thus the algorithm GreedySched(S) produces a set of nonoverlapping jobs that is equal in cardinality to a largest set of nonoverlapping jobs.
- Thus algorithm GreedySched is correct.
- **QED**
- **Exercise:** what is the run time of GreedySched?

Proof of Correctness of GreedySched concluded

- We have shown that $|G(S)| = |OPT(S)|$ and thus the algorithm GreedySched(S) produces a set of nonoverlapping jobs that is equal in cardinality to a largest set of nonoverlapping jobs.
- Thus algorithm GreedySched is correct.
- **QED**
- **Exercise:** what is the run time of GreedySched?
- **Exercise:** what if you want to maximize the sum of the lengths of the nonoverlapping jobs? Can you think of a dynamic programming solution?

Next

Data Compression