Greedy Algorithms

Credits: Many of these slides were originally authored by Jeff Edmonds, York University. Thanks Jeff!

Optimization Problems

- Shortest path is an example of an optimization problem: we wish to find the path with lowest weight.
- What is the general character of an optimization problem?

Optimization Problems

Ingredients:

- **Instances:** The possible inputs to the problem.
- **Solutions for Instance:** Each instance has an exponentially large set of valid solutions.
- **Cost of Solution:** Each solution has an easy-to-compute cost or value.

Specification

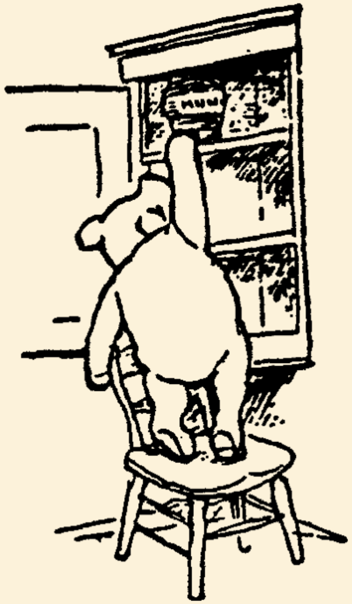
- **Preconditions:** The input is one instance.
- **Postconditions:** A valid solution with optimal cost. (minimum or maximum)

Greedy Solutions to Optimization Problems

Every two-year-old knows the *greedy algorithm*.

In order to get what you want,
just start grabbing what looks best.

Surprisingly, many important and practical optimization problems can be solved this way.



Example 1: Making Change

Problem: Find the minimum # of quarters, dimes, nickels, and pennies that total to a given amount.

The Greedy Choice

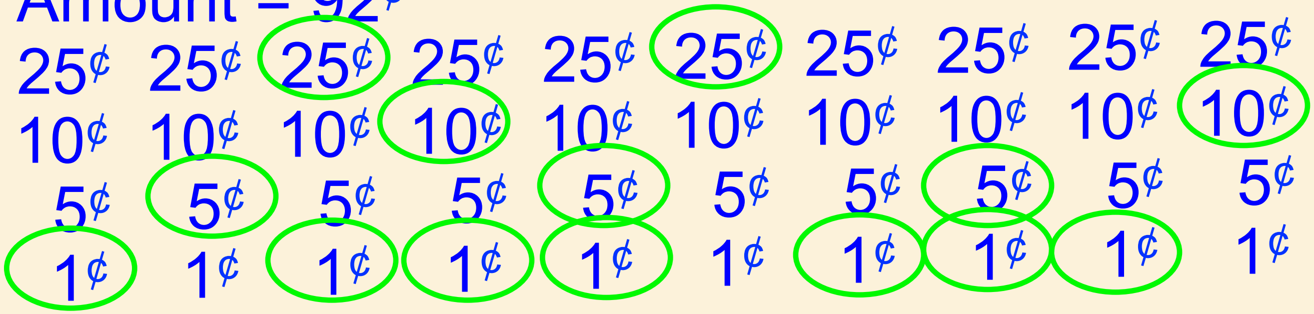
Commit to the object that looks the ``best''

Must prove that this locally greedy choice does not have negative global consequences.

Making Change Example

Instance: A drawer full of coins and an **amount** of change to return

Amount = 92¢



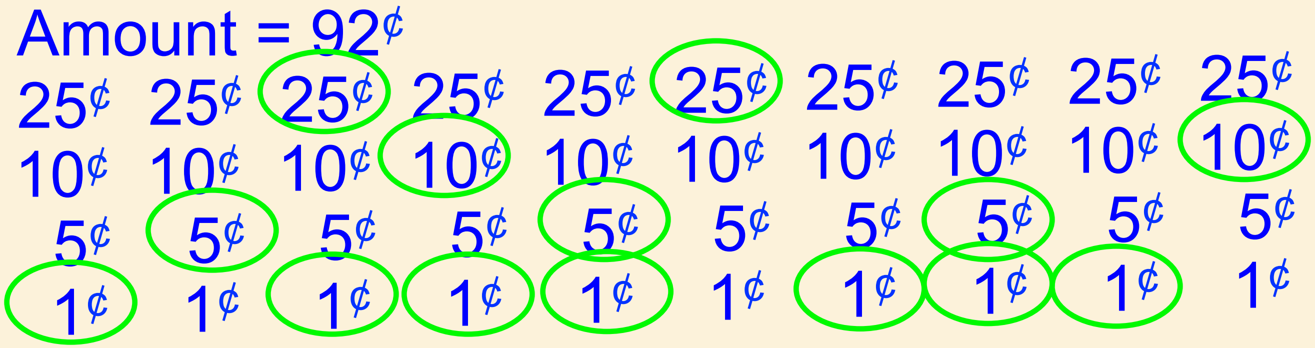
25¢	25¢	25¢	25¢	25¢	25¢	25¢	25¢	25¢	25¢
10¢	10¢	10¢	10¢	10¢	10¢	10¢	10¢	10¢	10¢
5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢
1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢

Solutions for Instance:

A subset of the coins in the drawer that total the amount

Making Change Example

Instance: A drawer full of coins and an amount of change to return



Amount = 92¢

25¢ 25¢ 25¢ 25¢ 25¢ 25¢ 25¢ 25¢ 25¢ 25¢
10¢ 10¢ 10¢ 10¢ 10¢ 10¢ 10¢ 10¢ 10¢ 10¢
5¢ 5¢ 5¢ 5¢ 5¢ 5¢ 5¢ 5¢ 5¢ 5¢
1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢

Solutions for Instance: A subset of the coins that total the amount.

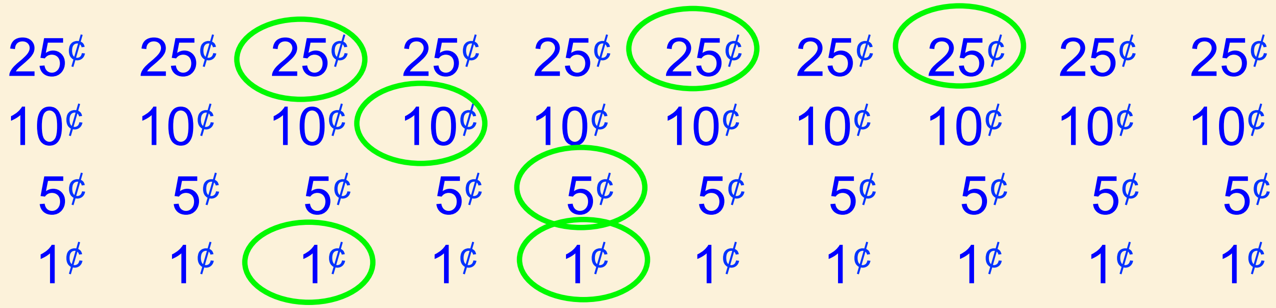
Cost of Solution: The number of coins in the solution **= 14**

Goal: Find an optimal valid solution.

Making Change Example

Instance: A drawer full of coins and an amount of change to return

Amount = 92¢



25¢	25¢	25¢	25¢	25¢	25¢	25¢	25¢	25¢	25¢
10¢	10¢	10¢	10¢	10¢	10¢	10¢	10¢	10¢	10¢
5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢
1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢

Greedy Choice:

Start by grabbing quarters until exceeds amount, then dimes, then nickels, then pennies.

Does this lead to an optimal # of coins?

Cost of Solution: **7**

Hard Making Change Example

Problem: Find the minimum # of 4, 3, and 1 cent coins to make up 6 cents.

Greedy Choice: Start by grabbing a 4-cent coin.

Consequences:

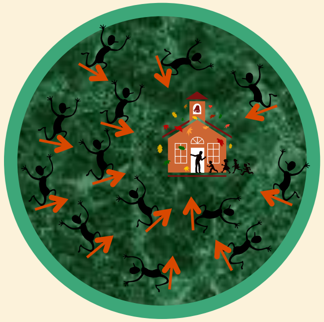
4+1+1 = 6 mistake
3+3=6 better

Greedy Algorithm does not work!

When Does It Work?

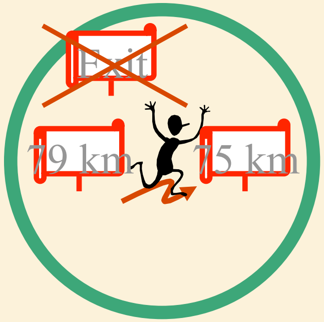
- Greedy Algorithms: Easy to understand and to code, but do they work?
- For most optimization problems, all greedy algorithms tried do **not** work (i.e. yield sub-optimal solutions)
- But **some** problems **can** be solved optimally by a greedy algorithm.
- The proof that they work, however, is subtle.
- As with all iterative algorithms, we use loop invariants.

Define Step



The algorithm chooses the “best” object from amongst those not considered so far and either commits to it or rejects it.

Make Progress



Another object considered

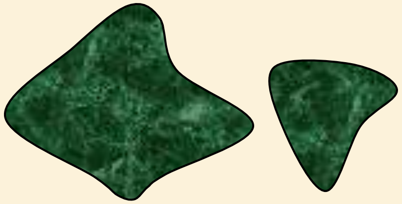
Exit Condition

All objects have been considered

Designing a Greedy Algorithm

```
< pre-condition >
CodeA
loop
< loop-invariant>
while ¬ exit condition
    CodeB
end loop
CodeC
< post-condition >
```

Loop Invariant

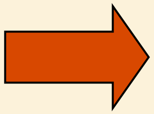


We have not gone wrong.

There is at least one optimal solution consistent with the choices made so far.

Establishing the Loop Invariant

Establishing Loop Invariant



<preCond>
codeA

<loop-invariant>

Initially no choices have been made and hence all optimal solutions are consistent with these choices.

Maintaining Loop Invariant

Must show that < loop-invariant > + CodeB → < loop-invariant >

< LI >: ∃ optimal solution $\mathrm{OptS}_{\mathrm{LI}}$ consistent with choices so far

CodeB : Commit to or reject next object

< LI >: ∃ optimal soln $\mathrm{OptS}_{\mathrm{Ours}}$ consistent with prev objects + new object

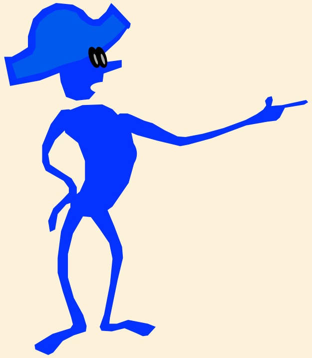
Note : $\mathrm{OptS}_{\mathrm{Ours}}$ may or may not be the same as $\mathrm{OptS}_{\mathrm{LI}}$!

Proof must massage $\mathrm{optS}_{\mathrm{LI}}$ into $\mathrm{optS}_{\mathrm{ours}}$ and prove that $\mathrm{optS}_{\mathrm{ours}}$:

- is a valid solution
- is consistent both with previous and new choices.
- is optimal

Three Players

Algorithm:
commits to or rejects next best object



Prover:
Proves LI is maintained.

His actions are not part of the algorithm

$optS_{LI}$

Fairy God Mother:
Holds the hypothetical optimal sol $optS_{LI}$.

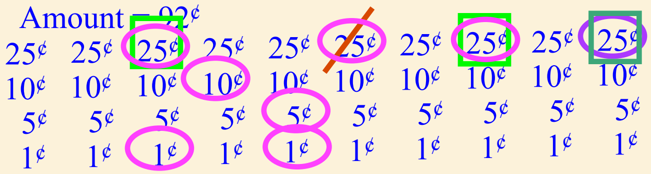
The algorithm and prover do not know $optS_{LI}$.

Proving the Loop Invariant is Maintained

- We need to show that the action taken by the algorithm maintains the loop invariant.
- There are 2 possible actions:
  - Case 1. Commit to current object
  - Case 2. Reject current object

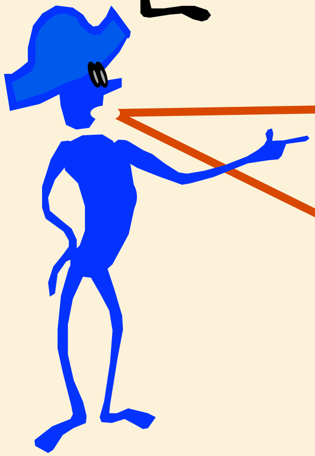
Case 1. Committing to Current Object

Massaging optS_{LI} into $\text{optS}_{\text{ours}}$



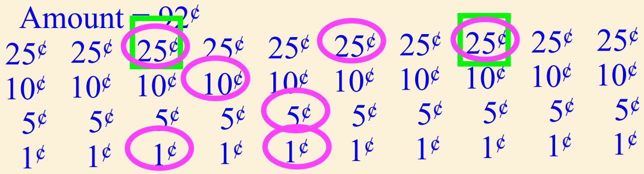
I hold $\text{optS}_{\text{ours}}$ witnessing that there is an opt sol consistent with previous & new choices.

I commit to keeping another 25¢



I instruct how to massage optS_{LI} into $\text{optS}_{\text{ours}}$ so that it is consistent with previous & new choice.

As Time Goes On


Amount = 92¢

25¢	25¢	25¢	25¢	25¢	25¢	25¢	25¢	25¢	25¢
10¢	10¢	10¢	10¢	10¢	10¢	10¢	10¢	10¢	10¢
5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢
1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢

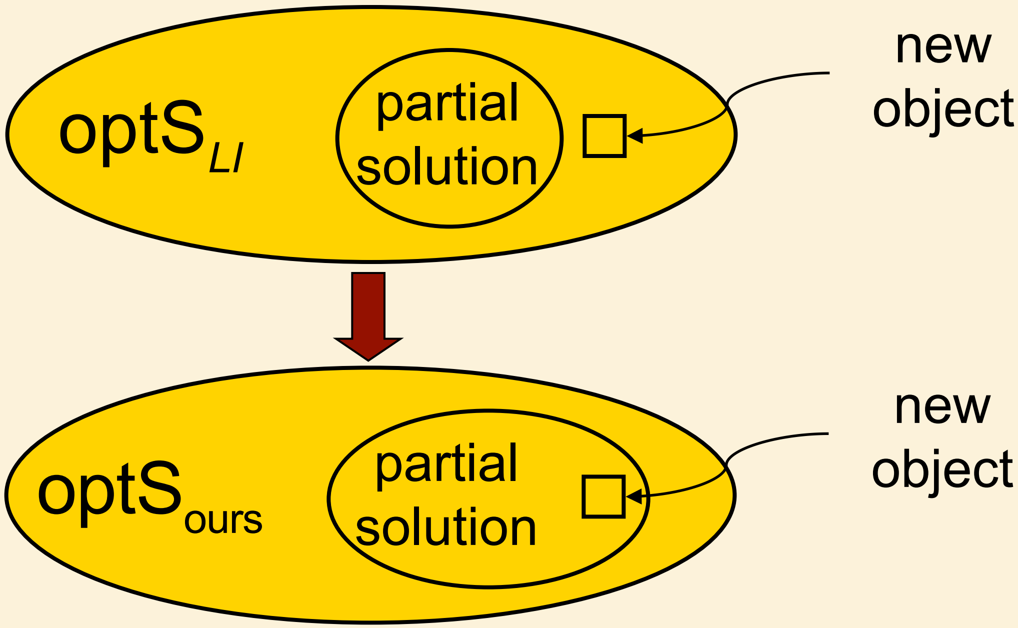

$\mathrm{optS}_{\mathrm{LI}}$

I always hold an opt sol $\mathrm{optS}_{\mathrm{LI}}$
but one that keeps changing.

I keep making more
choices.

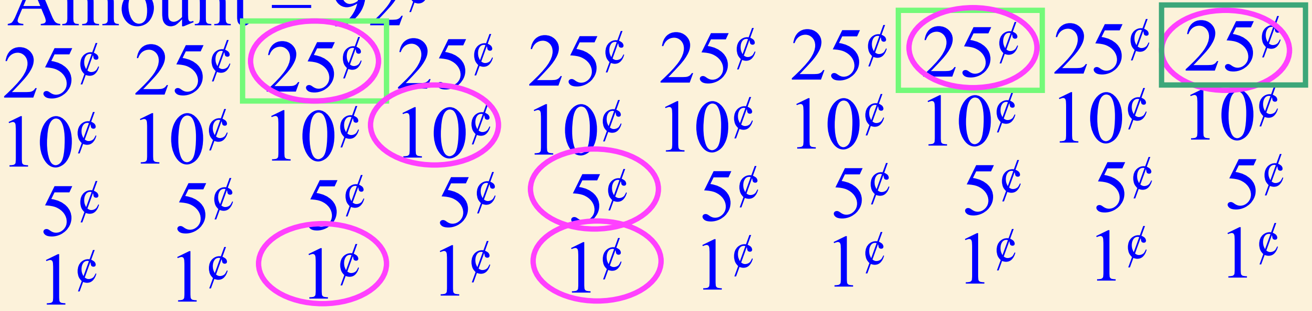
I know that her $\mathrm{optS}_{\mathrm{LI}}$
is consistent with these choices.
Hence, I know more and more of $\mathrm{optS}_{\mathrm{LI}}$
In the end, I know it all.

Case 1A.
The object we commit to is already part of optS_{LI}



Massaging optS_{LI} into $\text{optS}_{\text{ours}}$

Amount = 92¢

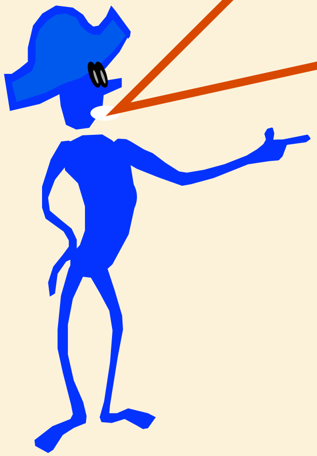

25¢ 25¢ 25¢ 25¢ 25¢ 25¢ 25¢ 25¢ 25¢ 25¢

10¢ 10¢ 10¢ 10¢ 10¢ 10¢ 10¢ 10¢ 10¢ 10¢

5¢ 5¢ 5¢ 5¢ 5¢ 5¢ 5¢ 5¢ 5¢ 5¢

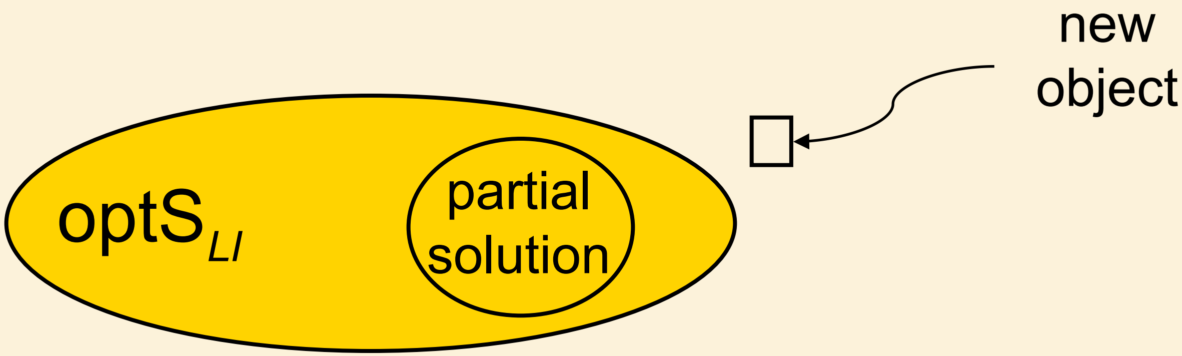
1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢

optS_{LI}



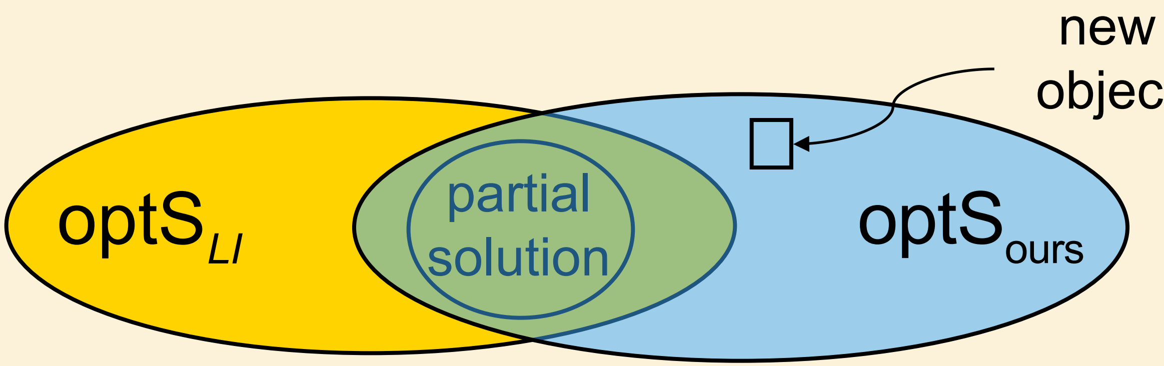
If it happens to be the case that the new object selected is consistent with the solution held by the fairy godmother, then we are done.

Case 1B.
The object we commit to is **not** part of optS$_{LI}$



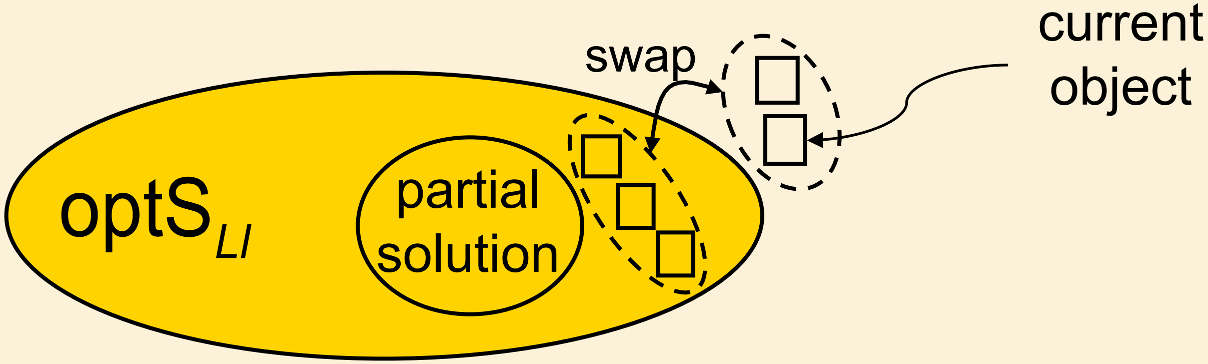
Case 1B. The object we commit to is **not** part of $optS_{LI}$

- This means that our partial solution is not consistent with $optS_{LI}$.
- The Prover must show that there is a new optimal solution $optS_{ours}$ that is consistent with our partial solution.
- This has two parts
  - All objects previously committed to must be part of $optS_{ours}$.
  - The new object must be part of $optS_{ours}$.



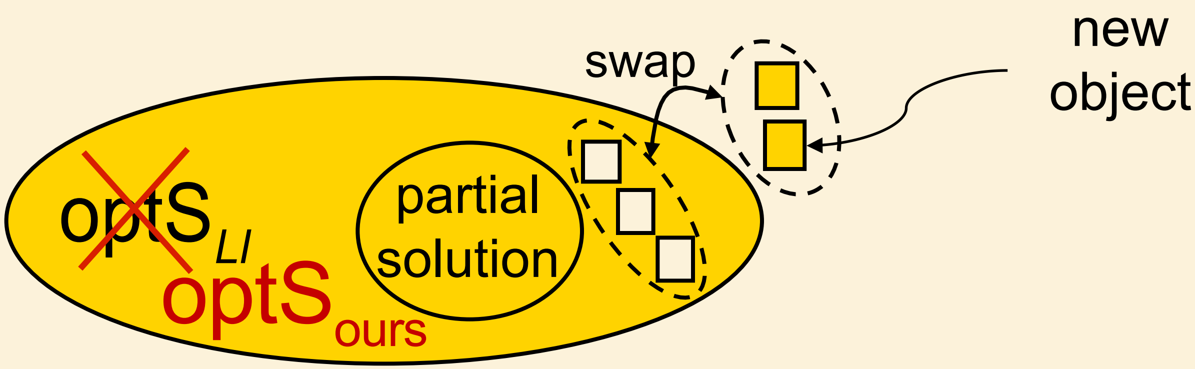
Case 1B. The object we commit to is **not** part of $optS_{LI}$

- **Strategy of proof:** construct a consistent $optS_{ours}$ by replacing one or more objects in $optS_{LI}$ (but not in the partial solution) with another set of objects that includes the current object.
- We must show that the resulting $optS_{ours}$ is still
  - Valid
  - Consistent
  - Optimal



Case 1B. The object we commit to is **not** part of $optS_{LI}$

- **Strategy of proof:** construct a consistent $optS_{ours}$ by replacing one or more objects in $optS_{LI}$ (but not in the partial solution) with another set of objects that includes the current object.
- We must show that the resulting $optS_{ours}$ is still
  - Valid
  - Consistent
  - Optimal

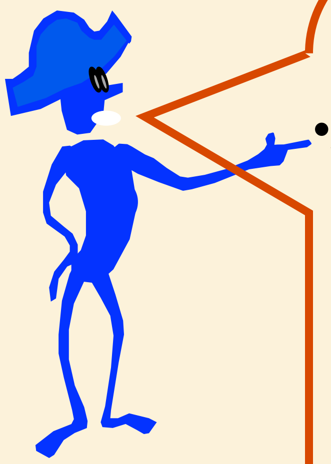


Massaging optS_{LI} into $\text{optS}_{\text{ours}}$

Amount = 92¢

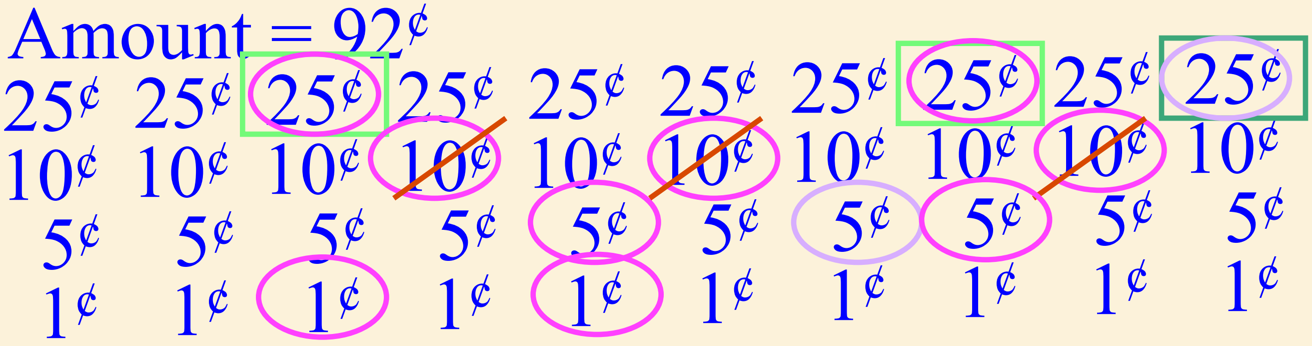
25¢	25¢	25¢	25¢	25¢	~~25¢~~	25¢	25¢	25¢	25¢
10¢	10¢	10¢	10¢	10¢	10¢	10¢	10¢	10¢	10¢
5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢
1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢

optS_{LI}



Replace	With
•A different 25¢	•Alg's 25¢

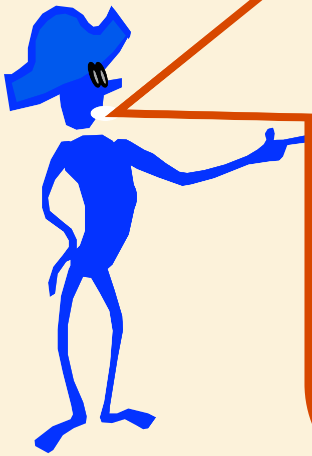
Massaging optS$_{LI}$ into optS$_{ours}$



Amount = 92¢

25¢ 25¢ 25¢ 25¢ 25¢ 25¢ 25¢ 25¢ 25¢ 25¢
10¢ 10¢ 10¢ ~~10¢~~ 10¢ ~~10¢~~ 10¢ 10¢ ~~10¢~~ 10¢
5¢ 5¢ 5¢ 5¢ 5¢ 5¢ 5¢ 5¢ 5¢ 5¢
1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢

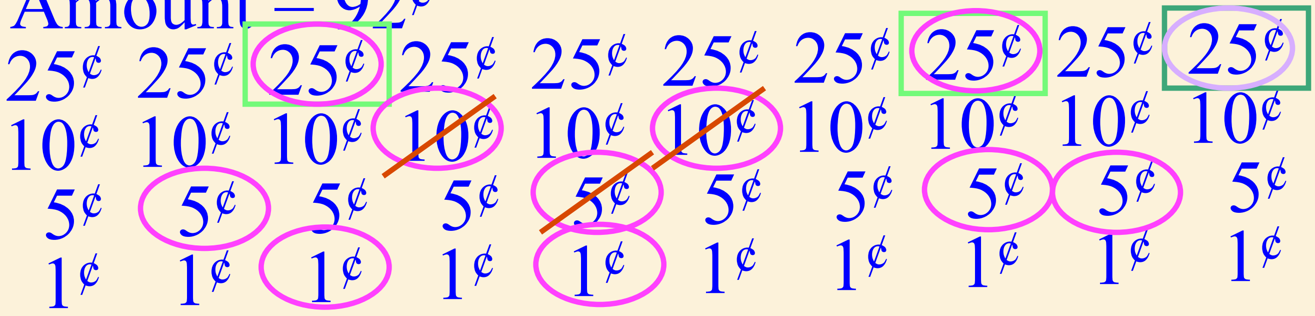
optS$_{LI}$



Replace	With
•A different 25¢	•Alg's 25¢
•3×10¢	•Alg's 25¢ + 5¢

Massaging $optS_{LI}$ into $optS_{ours}$

Amount = 92¢



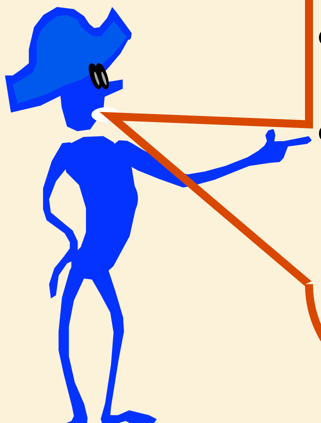
25¢ 25¢ 25¢ 25¢ 25¢ 25¢ 25¢ 25¢ 25¢ 25¢

10¢ 10¢ 10¢ ~~10¢~~ 10¢ ~~10¢~~ 10¢ 10¢ 10¢ 10¢

5¢ 5¢ 5¢ 5¢ ~~5¢~~ 5¢ 5¢ 5¢ 5¢ 5¢

1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢

$optS_{LI}$



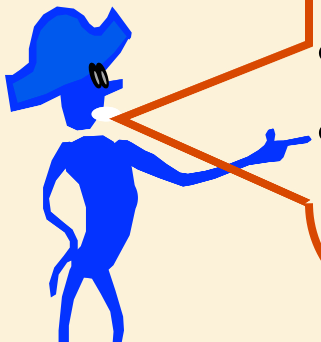
Replace	With
•A different 25¢	•Alg's 25¢
•3×10¢	•Alg's 25¢ + 5¢
•2×10¢ + 1×5¢	•Alg's 25¢

Massaging $optS_{LI}$ into $optS_{ours}$

Amount = 92¢

25¢	25¢	25¢	25¢	25¢	25¢	25¢	25¢	25¢	25¢
10¢	10¢	10¢	~~10¢~~	10¢	10¢	10¢	10¢	10¢	10¢
5¢	5¢	5¢	5¢	5¢	~~5¢~~	5¢	~~5¢~~	~~5¢~~	5¢
1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢

$optS_{LI}$



Replace	With
•A different 25¢	•Alg's 25¢
•3×10¢	•Alg's 25¢ + 5¢
•2×10¢ + 1×5¢	•Alg's 25¢
•1×10¢ + 3×5¢	•Alg's 25¢

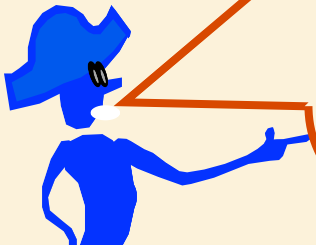
Massaging optS$_{LI}$ into optS$_{ours}$

Amount = 92¢

25¢	25¢	25¢	25¢	25¢	25¢	25¢	25¢	25¢	25¢
10¢	10¢	10¢	10¢	10¢	~~10¢~~	10¢	10¢	~~10¢~~	10¢
5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢
1¢	1¢	1¢	1¢	1¢	~~1¢~~	~~1¢~~	~~1¢~~	~~1¢~~	~~1¢~~

optS$_{LI}$

Replace	With
•A different 25¢	•Alg's 25¢
•3×10¢	•Alg's 25¢ + 5¢
•2×10¢ + 1×5¢	•Alg's 25¢
•1×10¢ + 3×5¢	•Alg's 25¢
•?? + 5×1¢	•Alg's 25¢



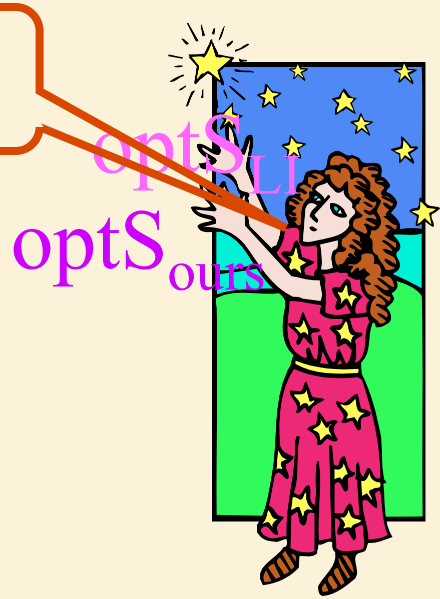
Must Consider All Cases

optS_{LI}	#Coins	$\text{optS}_{\text{Ours}}$	#Coins
1Q	1	1Q	1
3D	3	1Q 1N	2
2D 1N	3	1Q	1
2D 5P	7	1Q	1
1D 3N	4	1Q	1
1D 2N 5P	8	1Q	1
1D 1N 10P	12	1Q	1
1D 15P	16	1Q	1
5N	5	1Q	1
4N 5P	9	1Q	1
3N 10P	13	1Q	1
2N 15P	17	1Q	1
1N 20P	21	1Q	1
25P	25	1Q	1

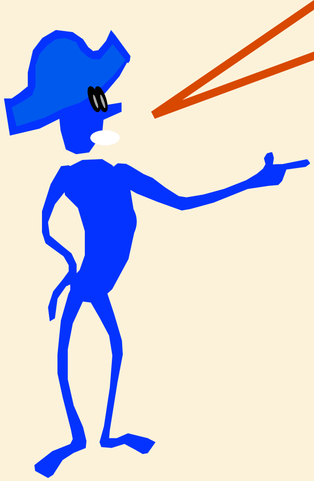
- Note that in all cases our new solution $\text{optS}_{\text{ours}}$ is:
  - **Valid:** the sum is still correct
  - **Consistent** with our previous choices (we do not alter these).
  - **Optimal:** we never add more coins to the solution than we delete

Massaging $optS_{LI}$ into $optS_{ours}$

Done



She now has something. We must prove that it is what we want.



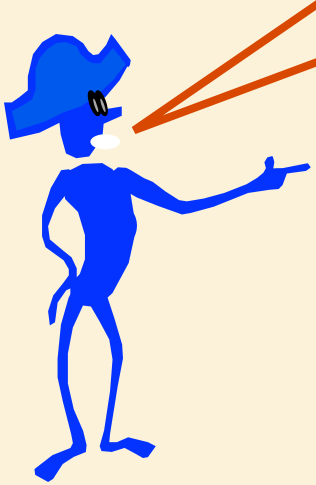
Massaging $optS_{LI}$ into $optS_{ours}$

$optS_{ours}$ is valid

$optS_{LI}$ was valid and we introduced no new conflicts.

Total remains unchanged.

$optS_{ours}$



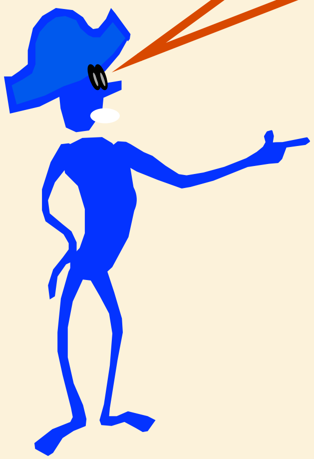
Replace	With
• A different 25¢	•Alg's 25¢
•3×10¢	•Alg's 25¢ + 5¢
•2×10¢ + 1×5¢	•Alg's 25¢
•1×10¢ + 3×5¢	•Alg's 25¢
•?? + 5×1¢	

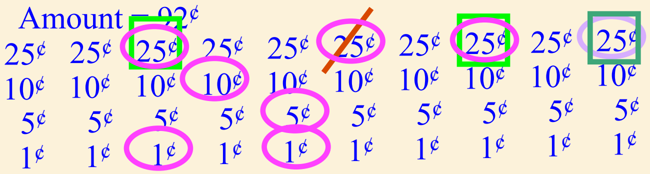
Massaging $optS_{LI}$ into $optS_{ours}$

$optS_{ours}$ is consistent

$optS_{LI}$ was consistent with previous choices and we made it consistent with new.

$optS_{ours}$





Amount = 92¢

25¢	25¢	25¢	25¢	25¢	25¢	25¢	25¢	25¢	25¢
10¢	10¢	10¢	10¢	10¢	10¢	10¢	10¢	10¢	10¢
5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢	5¢
1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢	1¢

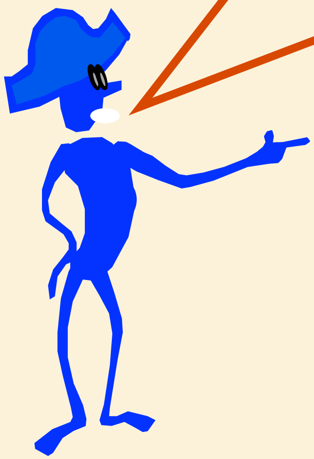
Massaging $optS_{LI}$ into $optS_{ours}$

$optS_{ours}$ is optimal

We do not even know the cost of an optimal solution.

$optS_{LI}$ was optimal and $optS_{ours}$ cost (# of coins) is not bigger.


$optS_{ours}$



Replace	With
• A different 25¢	•Alg's 25¢
•3×10¢	•Alg's 25¢ + 5¢
•2×10¢ + 1×5¢	•Alg's 25¢
•1×10¢ + 3×5¢	•Alg's 25¢
•?? + 5×1¢	

Committing to Other Coins

- Similarly, we must show that when the algorithm selects a dime, nickel or penny, there is still an optimal solution consistent with this choice.

$$optS_{LI} \xrightarrow{+\text{dime}} optS_{Ours}$$

$$optS_{LI} \xrightarrow{+\text{nickel}} optS_{Ours}$$

$$optS_{LI} \xrightarrow{+\text{penny}} optS_{Ours}$$

Example: Dimes

- We only commit to a dime when less than 25¢ is unaccounted for.
- Therefore the coins in optS_{LI} that this dime replaces have to be dimes, nickels or pennies.

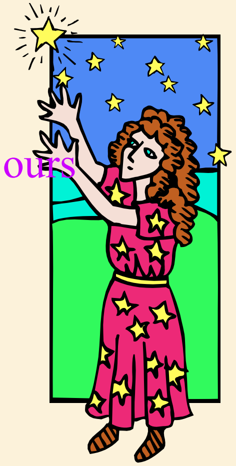
optS_{LI}	#Coins		$\text{optS}_{\text{Ours}}$	#Coins
1D	1		1D	1
2N	2		1D	1
1N 5P	6		1D	1
10P	10		1D	1

Committing to Other Coins

- We must consider all possible coins we might select:
  - **Quarter:** Swap for another quarter, 3 dimes (with a nickel) etc.
  - **Dime:** Swap for another dime, 2 nickels, 1 nickel + 5 pennies etc.
  - **Nickel:** Swap for another nickel or 5 pennies.
  - **Penny:** Swap for another penny.

Massaging $optS_{LI}$ into $optS_{ours}$

Case 1



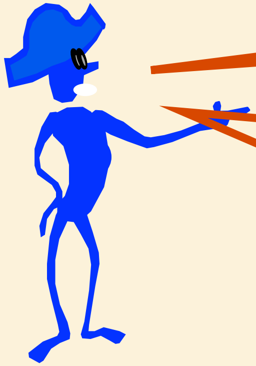
$optS_{ours}$

$optS_{ours}$ is valid

$optS_{ours}$ is consistent

$optS_{ours}$ is optimal

$optS_{ours} \Rightarrow$ <LI>



Maintaining Loop Invariant

<LI>
¬<exit Cond> $\Rightarrow$ <LI>
codeB

Case 2. Rejecting the Current Object

Rejecting the Current Object

Strategy of Proof:

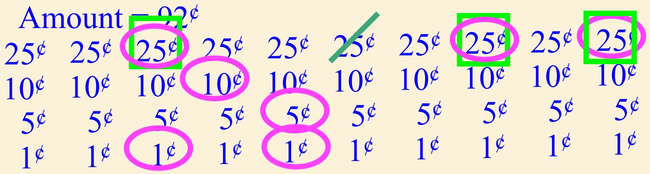
1. There is at least one optimal solution $optS_{LI}$ consistent with previous choices.
2. Any optimal solution consistent with previous choices cannot include current object.
3. Therefore $optS_{LI}$ cannot include current object.

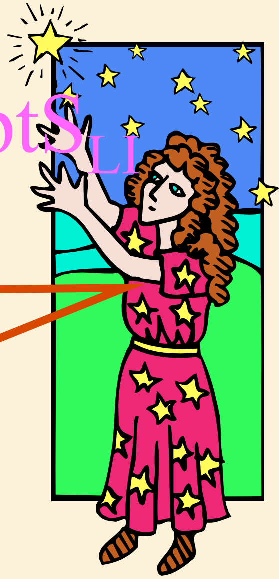
Rejecting an Object

- Making Change Example:
  - We only reject an object when including it would make us exceed the total.
  - Thus $optS_{LI}$ cannot include the object either.

Massaging $optS_{LI}$ into $optS_{ours}$

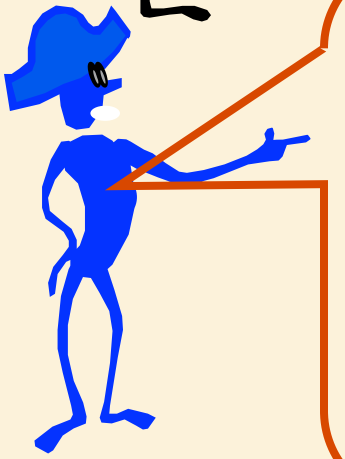
Case 2





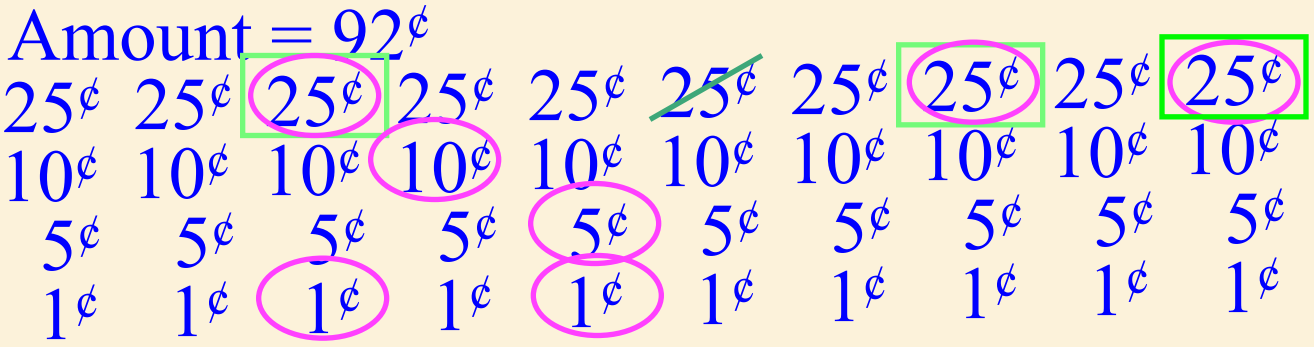
I hold $optS_{LI}$ witnessing that there is an opt sol consistent with previous choices.

I reject the next 25¢



I must make sure that what the Fairy God Mother has is consistent with this new choice.

Massaging optS_{LI} into $\text{optS}_{\text{ours}}$



Amount = 92¢

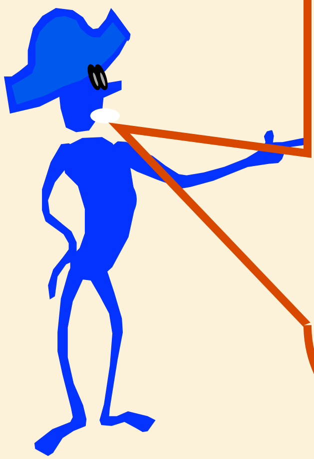
25¢ 25¢ 25¢ 25¢ 25¢ ~~25¢~~ 25¢ 25¢ 25¢ 25¢

10¢ 10¢ 10¢ 10¢ 10¢ 10¢ 10¢ 10¢ 10¢ 10¢

5¢ 5¢ 5¢ 5¢ 5¢ 5¢ 5¢ 5¢ 5¢ 5¢

1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢

optS_{LI}



The Algorithm has
92¢-75¢ = 17¢ < 25¢ unchoosen.

Fairy God Mother must
have < 25¢ that I don't know about.

optS_{LI} does not contain the 25¢ either.

Clean up loose ends

<loop-invariant>
<exit Cond>
codeC

→ <postCond>

<exit Cond> → Alg has committed to or rejected each object.

Has yielded a solution S.

<LI> → ∃ opt sol consistent with these choices.

S must be optimal.

codeC → Alg returns S .

→ <postCond>

Making Change Example

Problem: Find the minimum # of quarters, dimes, nickels, and pennies that total to a given amount.

Greedy Choice: Start by grabbing quarters until exceeds amount, then dimes, then nickels, then pennies.

Does this lead to an optimal # of coins?

Yes

Hard Making Change Example

Problem: Find the minimum # of
4, 3, and 1 cent coins to make up 6 cents.

Greedy Choice: Start by grabbing a 4 coin.

Massaging $optS_{LI}$ into $optS_{ours}$

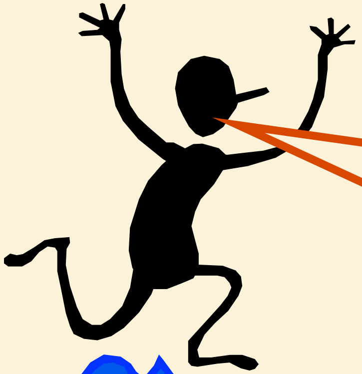
Amount = 6¢

4¢ 4¢ 4¢ 4¢ 4¢ 4¢ 4¢ 4¢ 4¢ 4¢

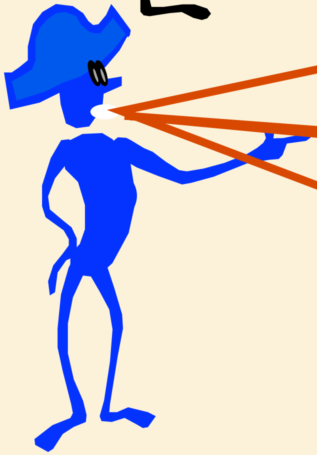
3¢ 3¢ 3¢ 3¢ 3¢ 3¢ 3¢ 3¢ 3¢ 3¢

1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢

I hold $optS_{LI}$.



I commit to keeping a 4¢



I will now instruct how to massage $optS_{LI}$ into $optS_{ours}$ so that it is consistent with previous & new choice.

Oops!

Hard Making Change Example

Problem: Find the minimum # of 4, 3, and 1 cent coins to make up 6 cents.

Greedy Choice: Start by grabbing a 4 coin.

Consequences:

4+1+1 = 6 mistake
3+3=6 better

Greedy Algorithm does not work!

Analysing Arbitrary Systems of Denominations

- Suppose we are given a system of coin denominations. How do we decide whether the greedy algorithm is optimal?
- It turns out that this problem can be solved in $O(D^3)$ time, where D = number of denominations (e.g., $D=6$ in Canada) (Pearson 1994).

Designing Optimal Systems of Denominations

In Canada, we use a 6 coin system:
1 cent, 5 cents, 10 cents, 25 cents, 100 cents and 200 cents.

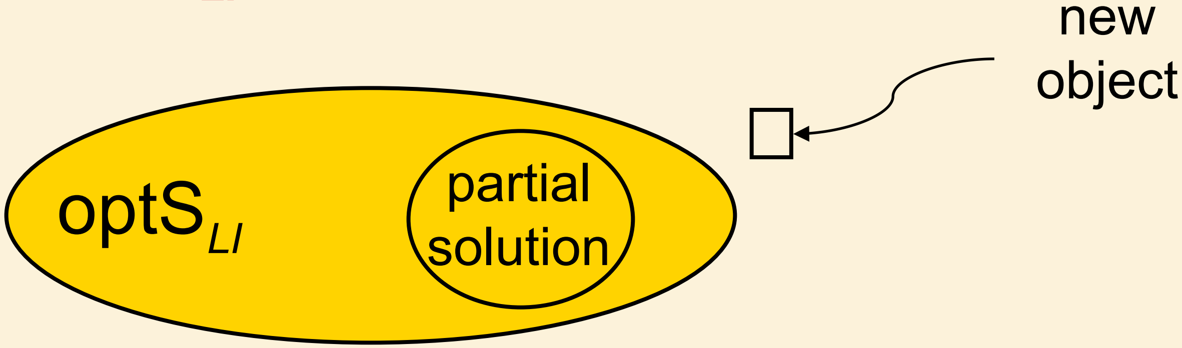
Assuming that N, the change to be made, is uniformly distributed over $\{1,...,499\}$, the expected number of coins per transaction is 5.9.

The optimal (but non-greedy) 6-coin systems are $(1,6,14,62,99,140)$ and $(1,8,13,69,110,160)$, each of which give an expected 4.67 coins per transaction.

The optimal *greedy* 6-coin systems are $(1,3,8,26,64,\{202 \text{ or } 203 \text{ or } 204\})$ and $(1,3,10,25,79,\{195 \text{ or } 196 \text{ or } 197\})$ with an expected cost of 5.036 coins per transaction.

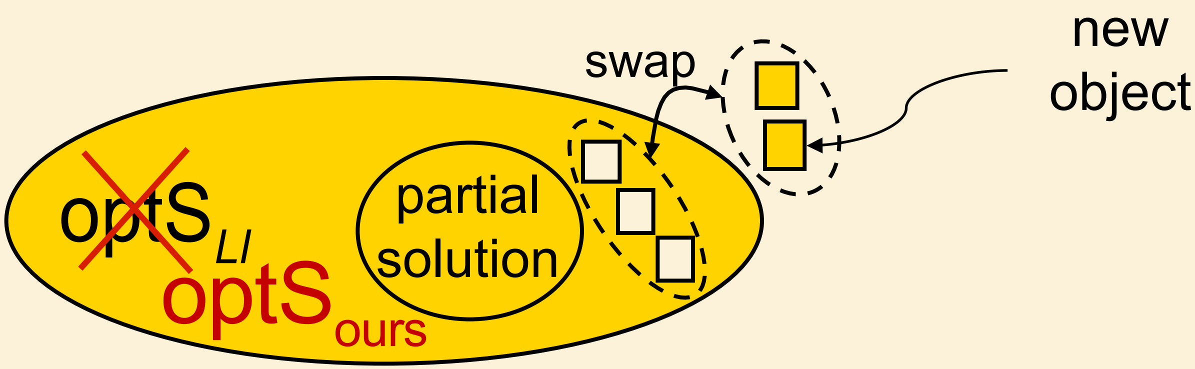
Summary

- We must prove that every coin chosen or rejected in greedy fashion still leaves us with a solution that is
  - Valid
  - Consistent
  - Optimal
- We prove this using an inductive ‘cut and paste’ method.
- We know from the previous iteration we have a partial solution $\boldsymbol{S_{part}}$ that is part of some complete optimal solution $\boldsymbol{optS_{LI}}$.



Summary

- **Selecting a coin:** we show that we can replace a subset of the coins in $optS_{LI} \setminus S_{part}$ with the selected coin (+ perhaps some additional coins).
  - **Valid** because we ensure that the trade is fair (sums are equal)
  - **Consistent** because we have not touched S_{part}
  - **Optimal** because the number of the new coin(s) is no greater than the number of coins they replace.
- **Rejecting a coin:** we show that we only reject a coin when it could not be part of $optS_{LI}$.



Example 2: Job/Event Scheduling

The Job/Event Scheduling Problem

Ingredients:

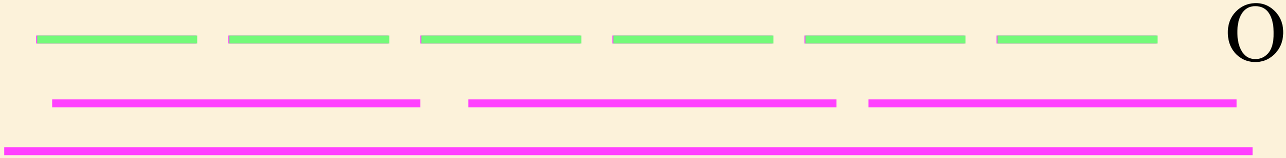
- Instances: Events with starting and finishing times

  $<<s_1,f_1>,<s_2,f_2>,\ldots ,<s_n,f_n>>$.

- Solutions: A set of events that do not overlap.
- Value of Solution: The number of events scheduled.

- Goal: Given a set of events, schedule as many as possible.
- Example: Scheduling lectures in a lecture hall.

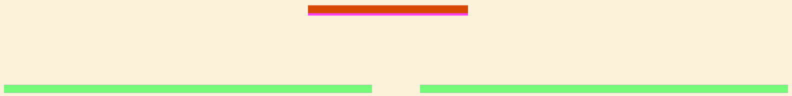
Possible Criteria for Defining "Best"



Optimal

Greedy Criterion:The Shortest Event

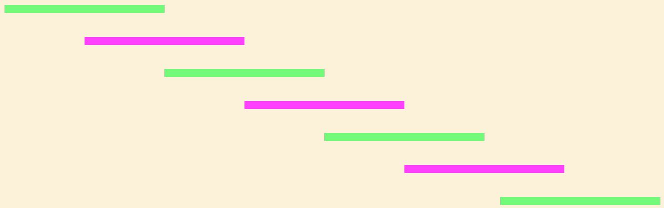
Motivation: Does not book the room
for a long period of time.



Schedule first
Optimal

Counter Example

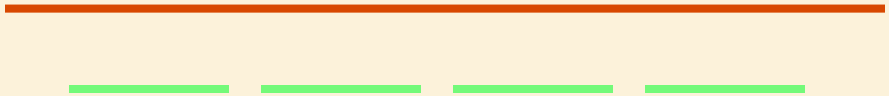
Possible Criteria for Defining "Best"



Optimal

Greedy Criterion: The Earliest Starting Time

Motivation: Gets room in use as early as possible

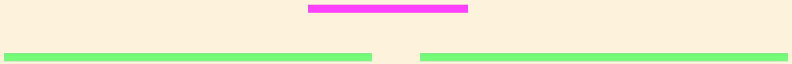


Schedule first

Optimal

Counter Example

Possible Criteria for Defining "Best"

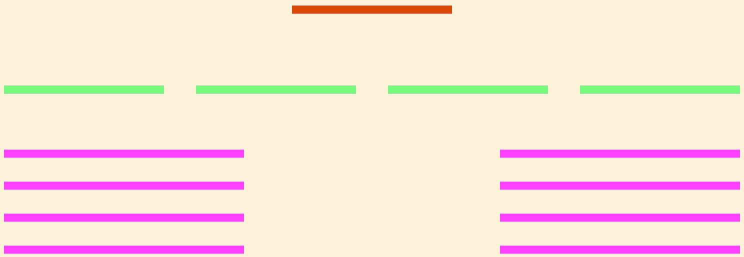


Optimal

Greedy Criterion:

Conflicting with the Fewest Other Events

Motivation: Leaves many that can still be scheduled.



Schedule first
Optimal

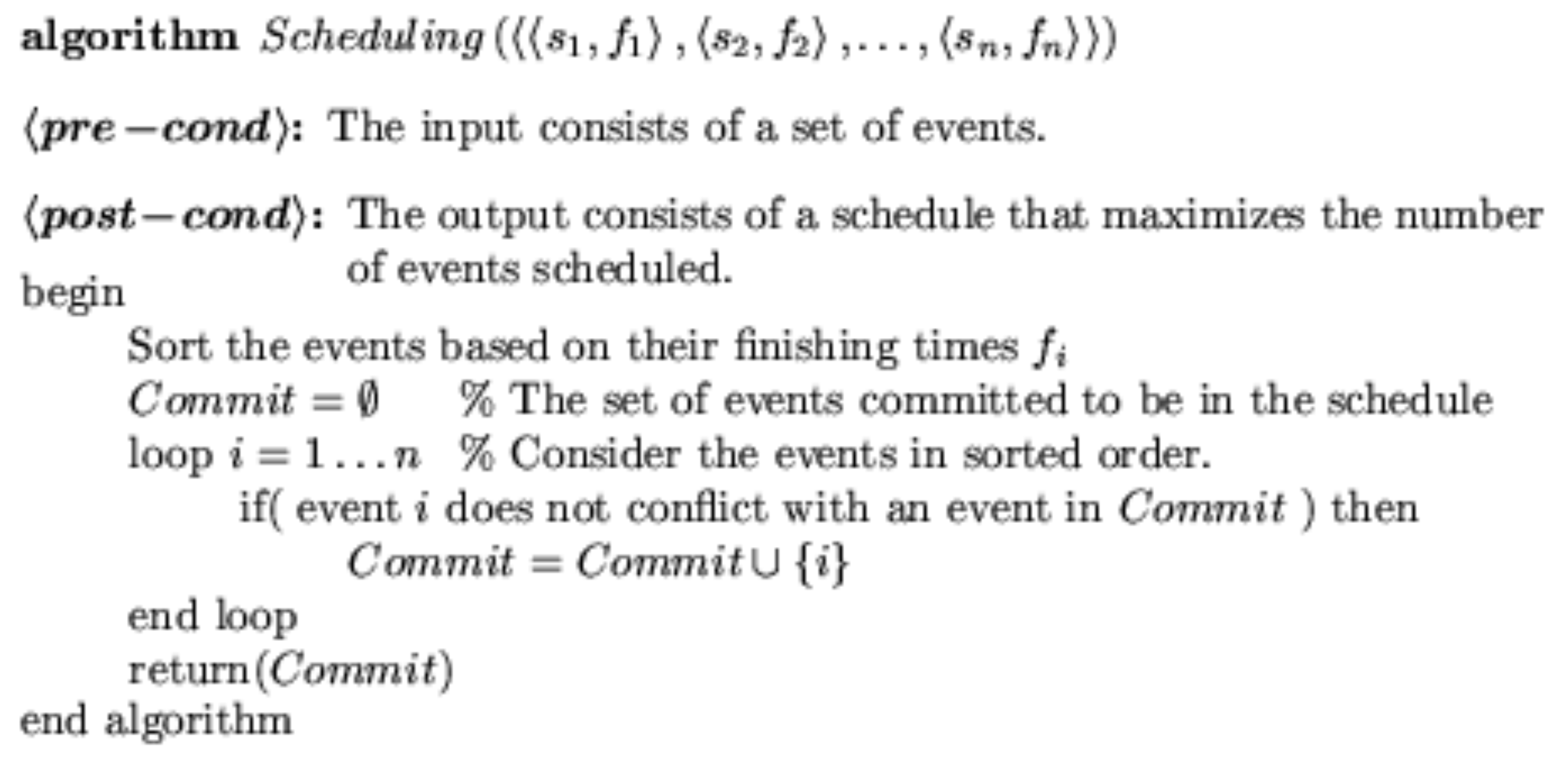
Counter Example

Possible Criteria for Defining "Best"

Greedy Criterion: Earliest Finishing Time

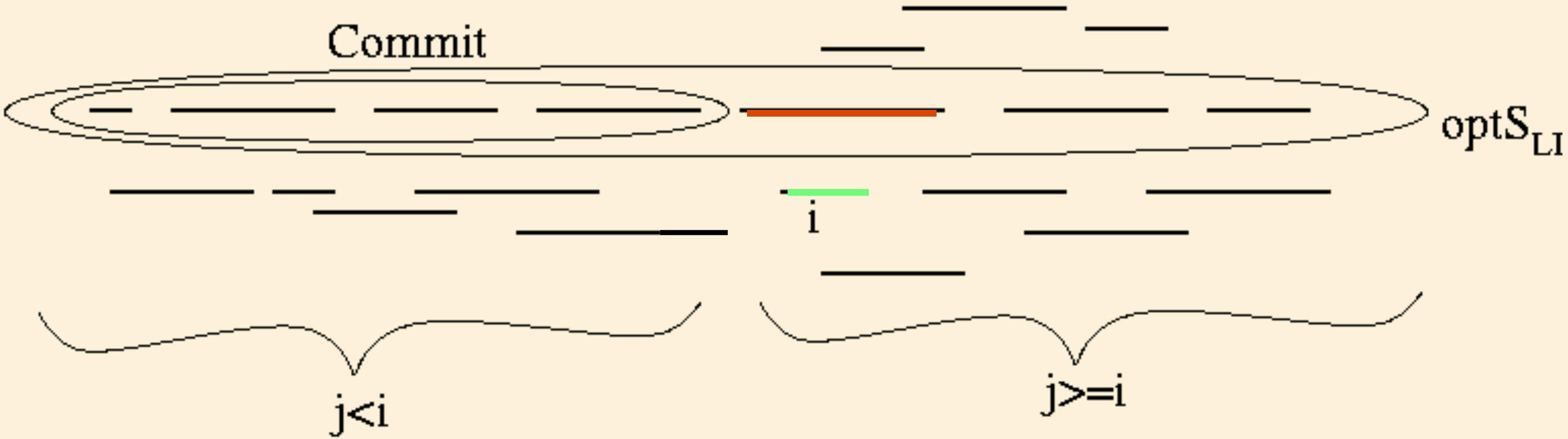
Motivation: Schedule the event that will free up your room for someone else as soon as possible.

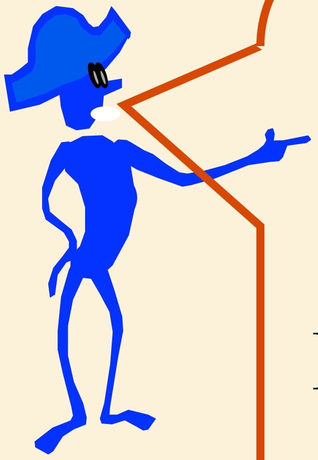
The Greedy Algorithm



```
algorithm $Scheduling(\langle\langle s_1, f_1\rangle, \langle s_2, f_2\rangle, \ldots, \langle s_n, f_n\rangle\rangle)$

$\langle pre-cond \rangle$: The input consists of a set of events.

$\langle post-cond \rangle$: The output consists of a schedule that maximizes the number
                  of events scheduled.
begin
    Sort the events based on their finishing times $f_i$
    $Commit = \emptyset$      % The set of events committed to be in the schedule
    loop $i = 1 \ldots n$   % Consider the events in sorted order.
        if( event $i$ does not conflict with an event in $Commit$ ) then
            $Commit = Commit \cup \{i\}$
    end loop
    return($Commit$)
end algorithm
```

Massaging **optS**$_{LI}$ into **optS**$_{ours}$

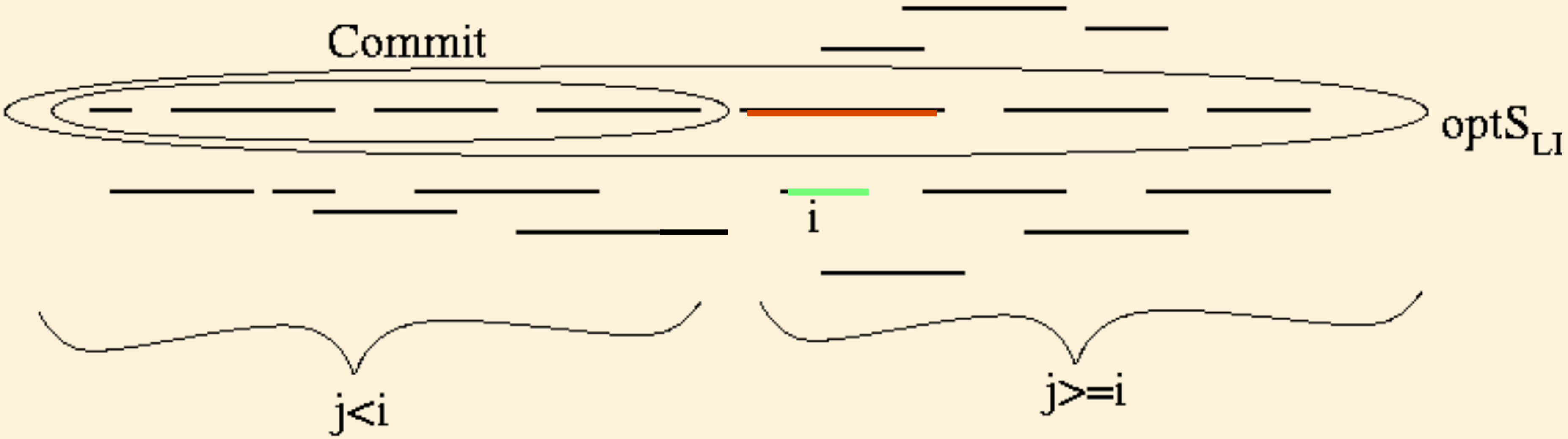


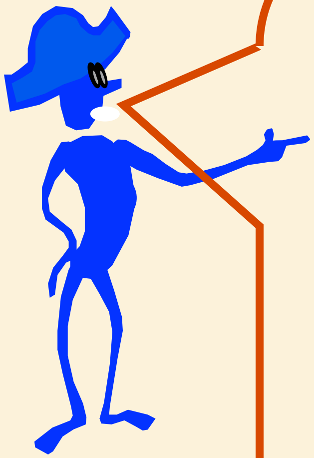


Start by adding new event i.

Delete events conflicting with job i.

Massaging **optS$_{LI}$** into **optS$_{ours}$**

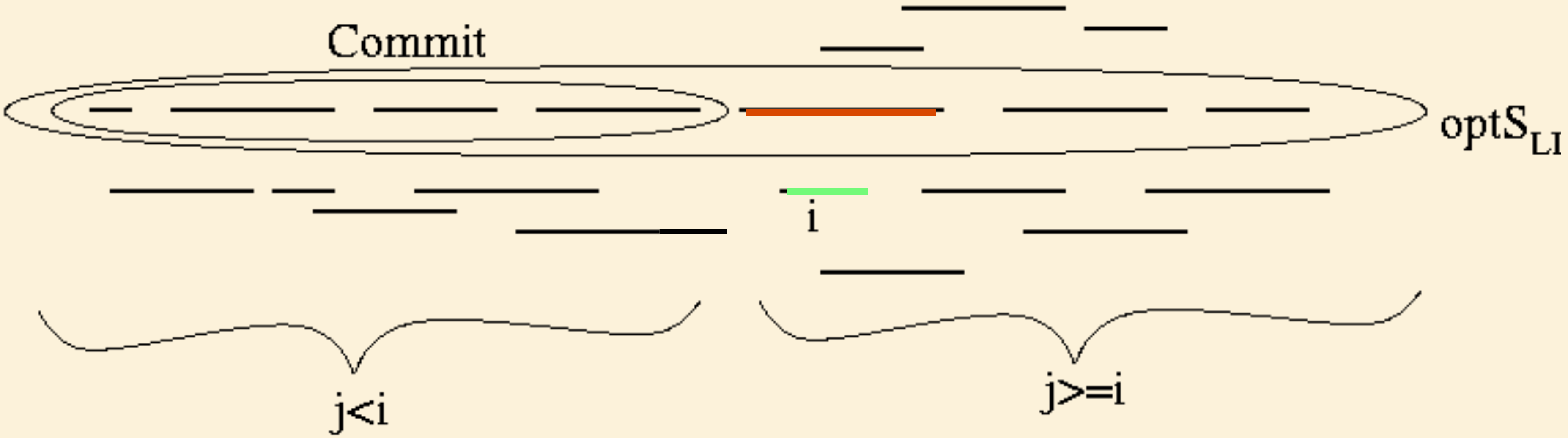


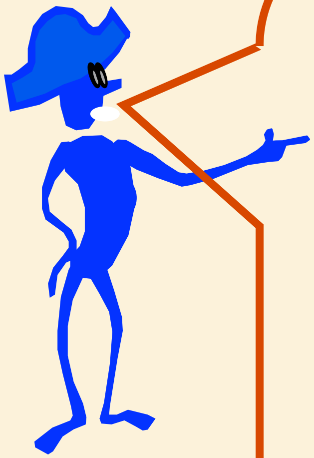


optS$_{ours}$ is valid

optS$_{LI}$ was valid and we removed any new conflicts.

Massaging optS_{LI} into $\text{optS}_{\text{ours}}$

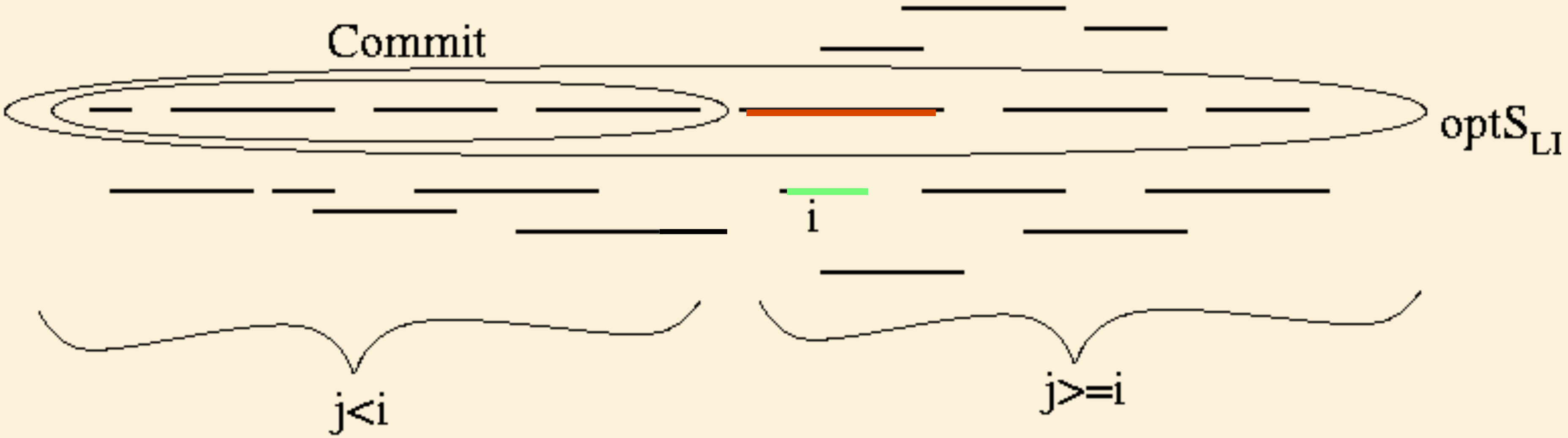


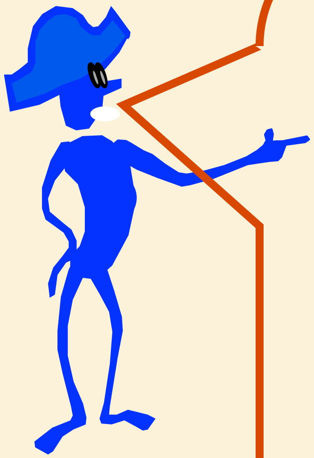


$\text{optS}_{\text{ours}}$ is consistent with our choices.

optS_{LI} was consistent with our prior choices.
We added event i.
Events in Commit don't conflict with event i
and hence were not deleted.

Massaging optS_{LI} into $\text{optS}_{\text{ours}}$



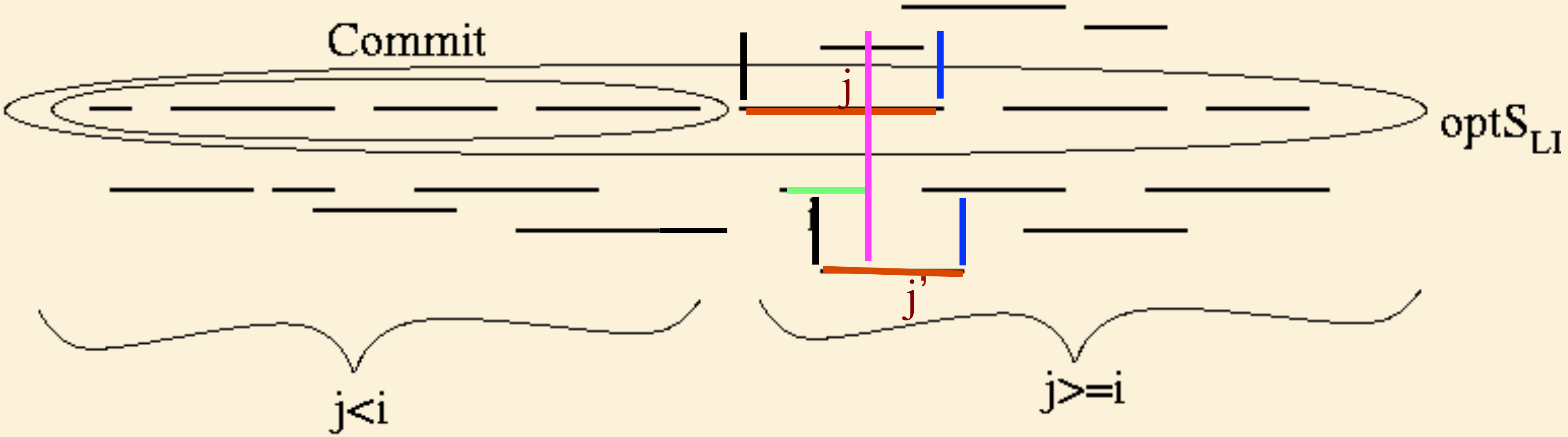


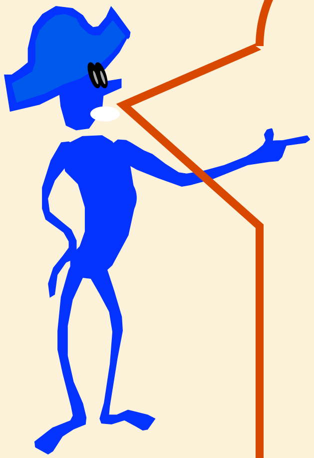
$\text{optS}_{\text{ours}}$ is optimal

optS_{LI} was optimal.

If we delete at most one event then $\text{optS}_{\text{ours}}$ is optimal too.

Massaging $optS_{LI}$ into $optS_{ours}$





Deleted at most one event j

$i<j \Rightarrow f_i \leq f_j$

[j conflicts with i] $\Rightarrow s_j \leq f_i$

$\Rightarrow$ j runs at time f_i.

Two such j conflict with each other.

Only one in $optS_{LI}$.

Massaging $optS_{LI}$ into $optS_{ours}$

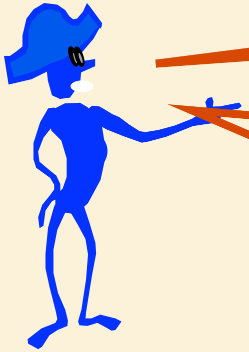
Case 1

$optS_{ours}$ is valid

$optS_{ours}$ is consistent

$optS_{ours}$ is optimal

$optS_{ours}$ → <LI>

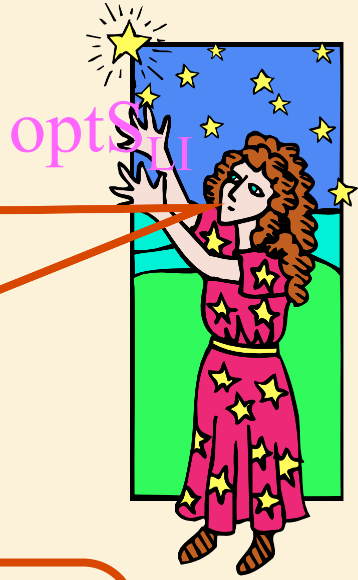


Maintaining Loop Invariant

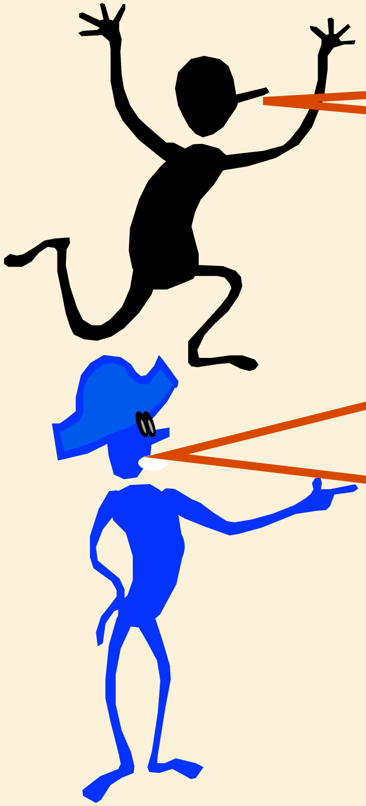
<LI>
¬<exit Cond> → <LI>
codeB

Massaging **optS**$_{LI}$ into **optS**$_{ours}$

Case 2



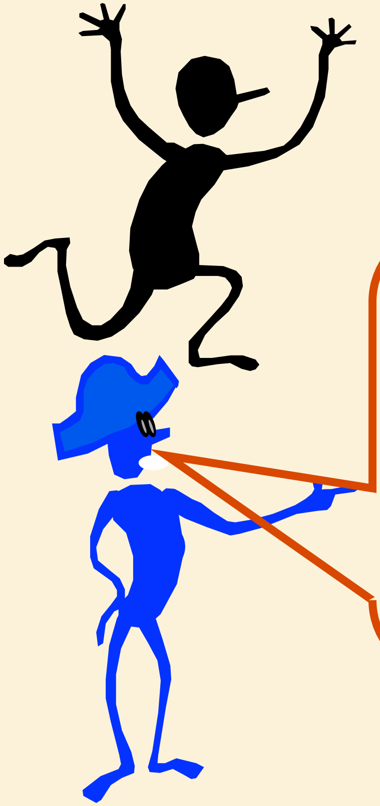
I hold **optS**$_{LI}$ witnessing that there is an opt sol consistent with previous choices.



I reject next event i.

Event i conflicts with events committed to so it can't be in **optS**$_{LI}$ either.

Massaging $\mathbf{optS}_{\mathbf{LI}}$ into $\mathbf{optS}_{\mathbf{ours}}$



Maintaining Loop Invariant

<LI>

¬<exit Cond> → <LI>

codeB

Clean up loose ends

<loop-invariant>
<exit Cond> → <postCond>
codeC

<exit Cond> → Alg commits to or reject each event.
Has a solution S.

<LI> → ∃ opt sol consistent with these choices

S must be optimal.

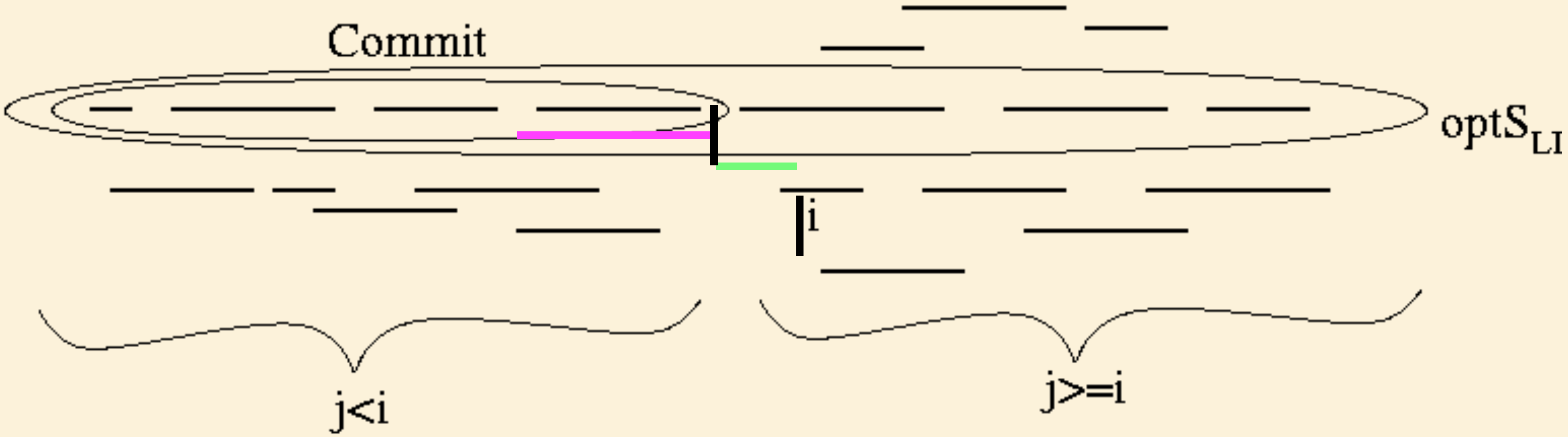
codeC → Alg returns **optS** .

→ <postCond>

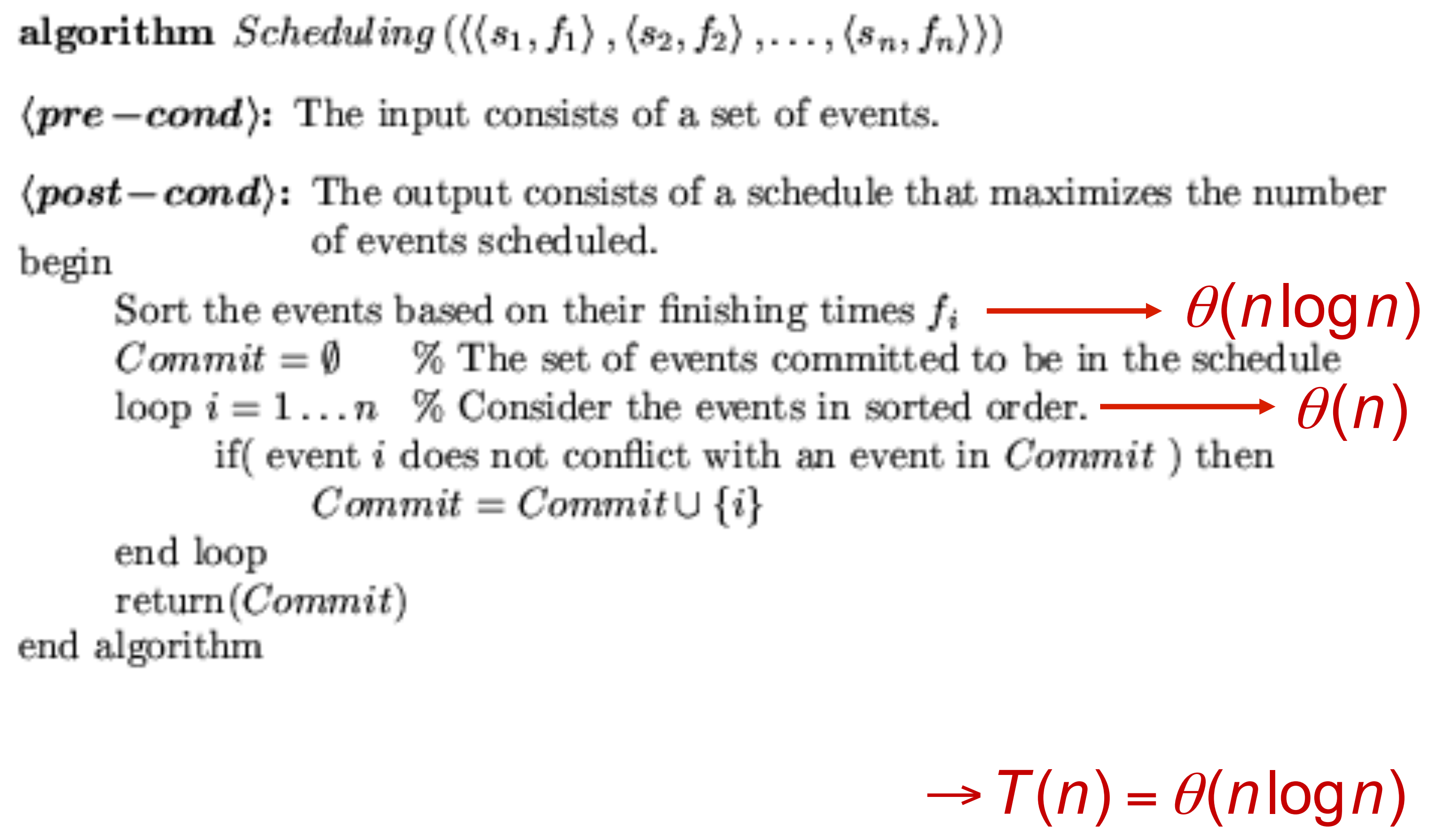
Running Time

Greedy algorithms are very fast because they only consider each object once.

Checking whether next event i conflicts with previously committed events requires only comparing it with the last such event.



Running Time



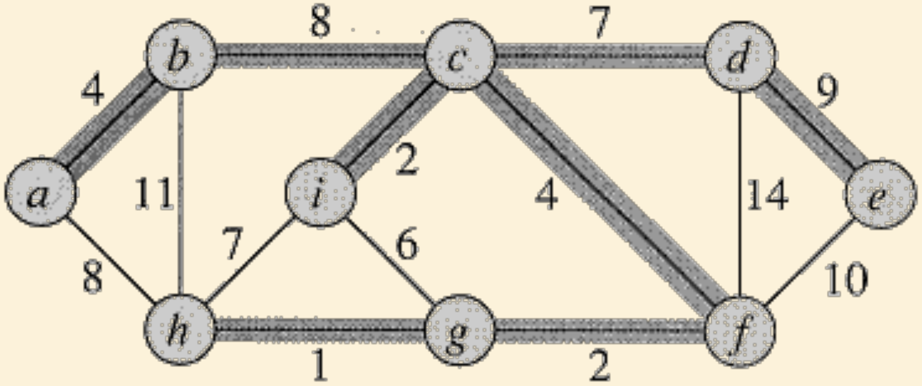
```
algorithm $Scheduling(\langle\langle s_1, f_1\rangle, \langle s_2, f_2\rangle, \ldots, \langle s_n, f_n\rangle\rangle)$

$\langle pre-cond\rangle$: The input consists of a set of events.

$\langle post-cond\rangle$: The output consists of a schedule that maximizes the number
                of events scheduled.
begin
    Sort the events based on their finishing times $f_i$ → $\theta(n\log n)$
    $Commit = \emptyset$    % The set of events committed to be in the schedule
    loop $i = 1 \ldots n$   % Consider the events in sorted order. → $\theta(n)$
        if( event $i$ does not conflict with an event in $Commit$ ) then
            $Commit = Commit \cup \{i\}$
    end loop
    return($Commit$)
end algorithm
```

$\rightarrow T(n) = \theta(n\log n)$

Example 3: Minimum Spanning Trees

Minimum Spanning Trees

- Example Problem
  - You are planning a new terrestrial telecommunications network to connect a number of remote mountain villages in a developing country.
  - The cost of building a link between pairs of neighbouring villages *(u,v)* has been estimated: *w(u,v).*
  - You seek the minimum cost design that ensures each village is connected to the network.
  - The solution is called a *minimum spanning tree (MST)*.



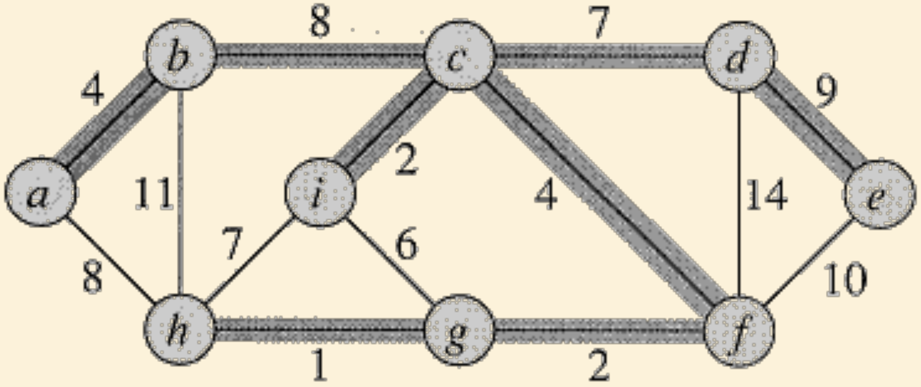
Minimum Spanning Trees

The problem is defined for any undirected, connected, weighted graph.

The weight of a subset T of a weighted graph is defined as:

$$w(T) = \sum_{(u,v) \in T} w(u,v)$$

Thus the MST is the spanning tree T that minimizes $w(T)$

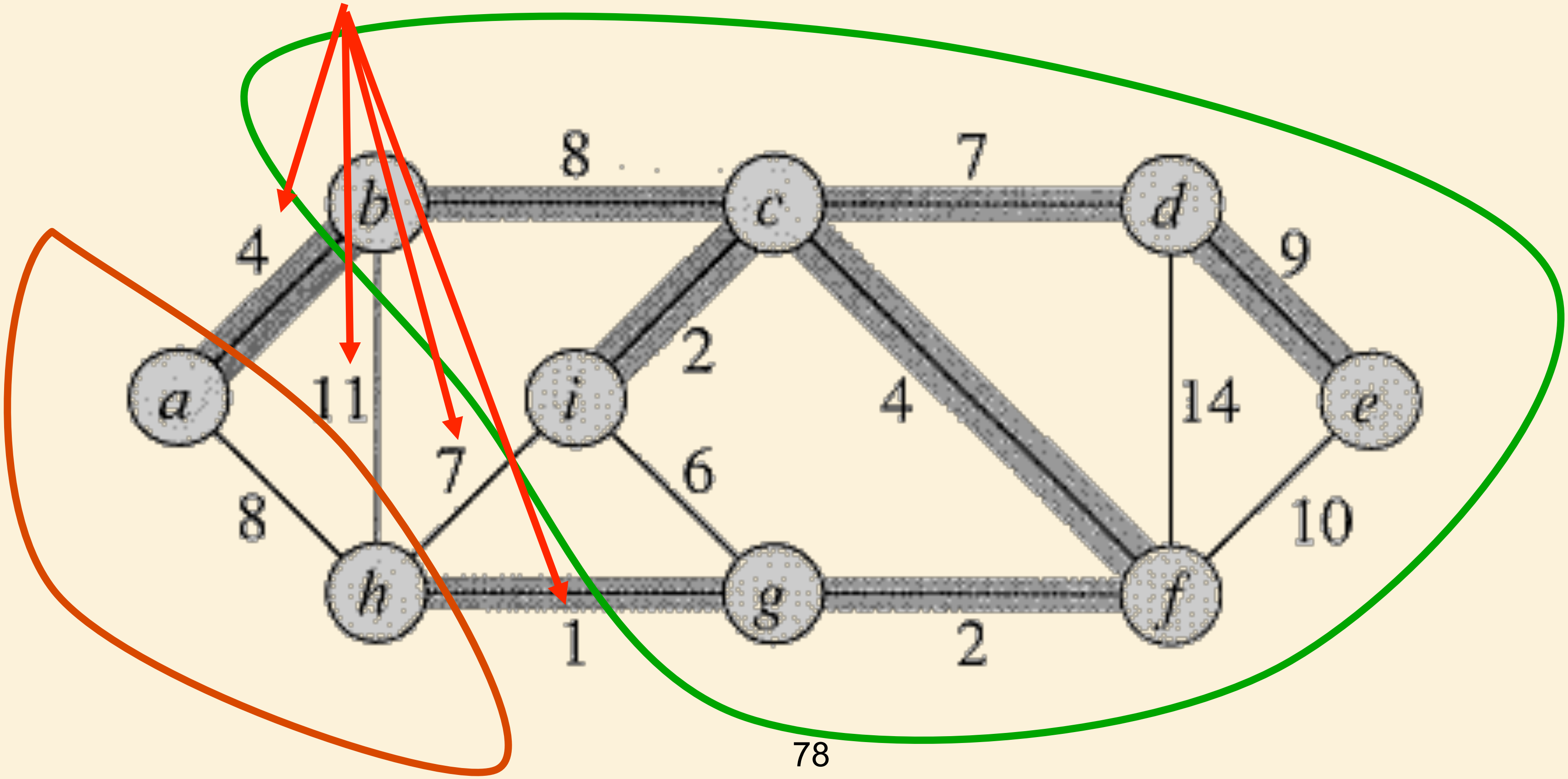


Building the Minimum Spanning Tree

- Iteratively construct the set of edges *A* in the MST.
- Initialize *A* to {}
- As we add edges to *A*, maintain a Loop Invariant:
  - *A* is a subset of some MST
- Maintain loop invariant and make progress by only adding **safe** edges.
- An edge *(u,v)* is called ***safe*** for *A* iff $A \cup (\{u,v\})$ is also a subset of some MST.

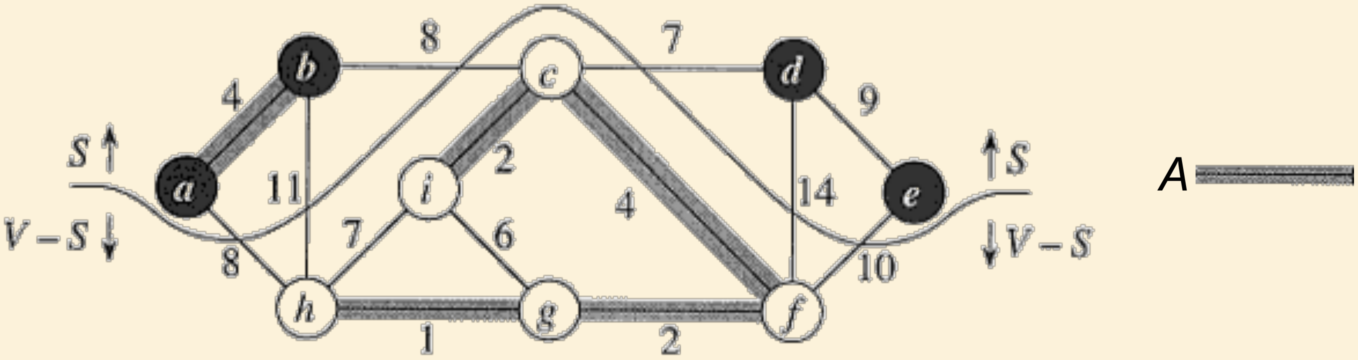
Finding a safe edge

- Idea: Every 2 disjoint subsets of vertices must be connected by at least one edge.
- Which one should we choose?



Some definitions

- A *cut* $(S, V-S)$ is a partition of vertices into disjoint sets S and $V-S$.
- Edge $(u,v) \in E$ *crosses* cut $(S, V-S)$ if one endpoint is in S and the other is in $V-S$.
- A cut *respects* a set of edges A iff no edge in A crosses the cut.
- An edge is a *light* edge crossing a cut iff its weight is minimum over all edges crossing the cut.

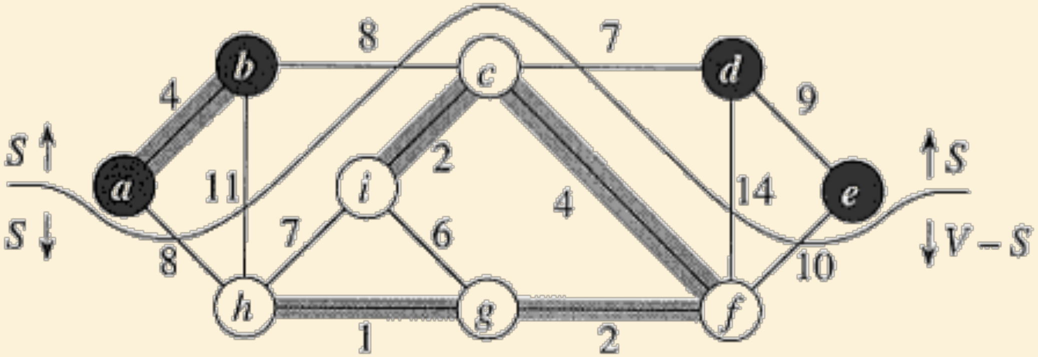


Minimum Spanning Tree Theorem

- Let
  - *A* be a subset of some MST
  - *(S,V-S)* be a cut that respects *A*
  - *(u,v)* be a **light** edge crossing *(S,V-S)*

- Then
  - *(u,v)* is safe for *A*.

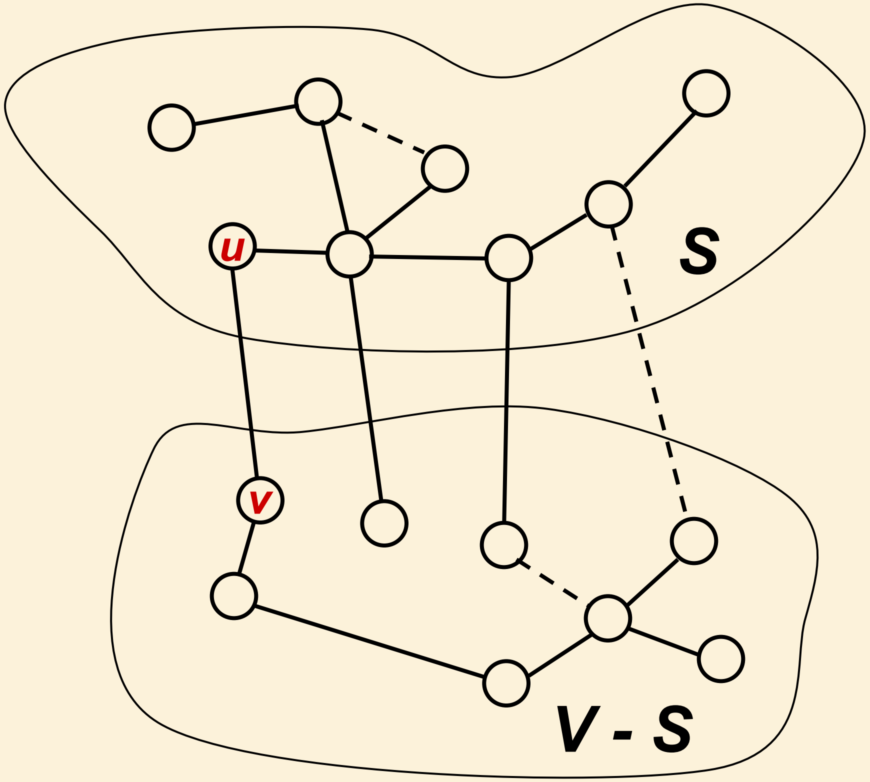
A ▬



Basis for a greedy algorithm

Proof

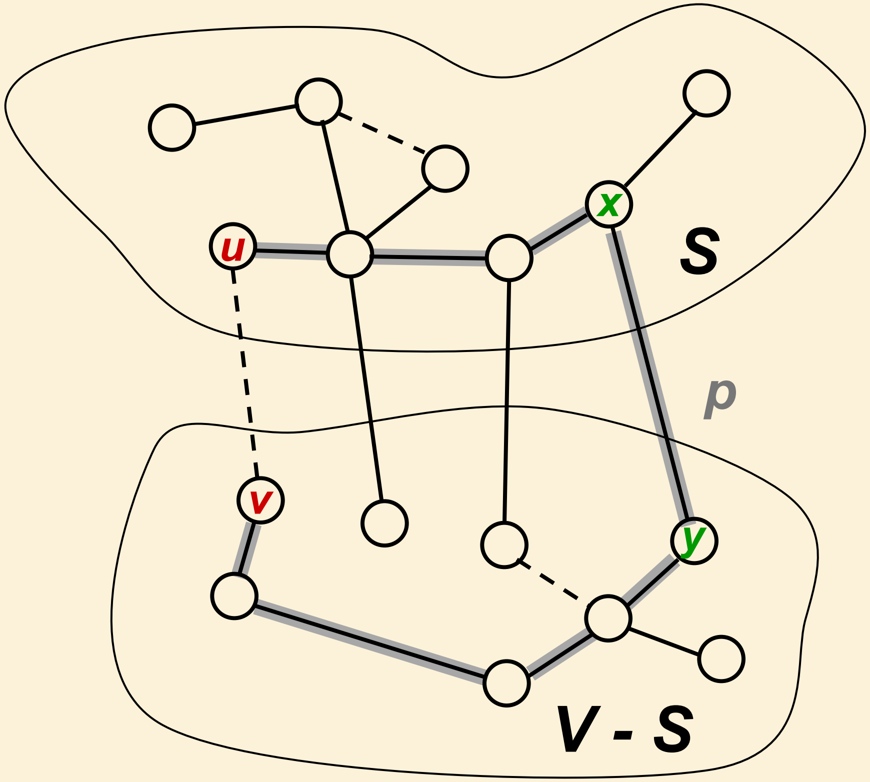
- Let *G* be a connected, undirected, weighted graph.
- Let *T* be an MST that includes ***A***.
- Let (*S*,*V*-*S*) be a cut that respects ***A***.
- Let ***(u,v)*** be a light edge between *S* and *V*-*S*.
- If *T* contains ***(u,v)*** then we're done.



—— Edge $\in T$

- - - - - Edge $\notin T$

- Suppose *T* does not contain ***(u,v)***
  - Can construct different MST *T′* that includes ***A***∪***(u,v)***
  - The edge ***(u,v)*** forms a **cycle** with the edges on the path ***p*** from *u* to *v* in *T*.
  - There is at least one edge in ***p*** that crosses the cut: let that edge be ***(x,y)***
  - ***(x,y)*** is not in ***A***, since the cut *(S,V-S)* respects ***A***.
  - Form new spanning tree *T′* by deleting ***(x,y)*** from *T* and adding ***(u,v)***.
  - *w(T′)* ≤ *w(T)*, since *w(**u,v**)* ≤ *w(**x,y**)* → *T′* is an MST.
  - ***A*** ⊆ *T′*, since ***A*** ⊆ *T* and *(**x,y**)*∉***A*** → ***A***∪***(u,v)*** ⊆ *T′*
  - Thus ***(u,v)*** is safe for ***A***.



Kruskal's Algorithm for computing MST

- Starts with each vertex being its own component.
- Repeatedly merges two components into one by choosing the light edge that crosses the cut between them.
- Scans the set of edges in monotonically increasing order by weight (greedy).

Kruskal's Algorithm: Loop Invariant

Let A = solution under construction.

Let E_i = the subset of i lowest-weight edges thus far considered

< loop-invariant >:

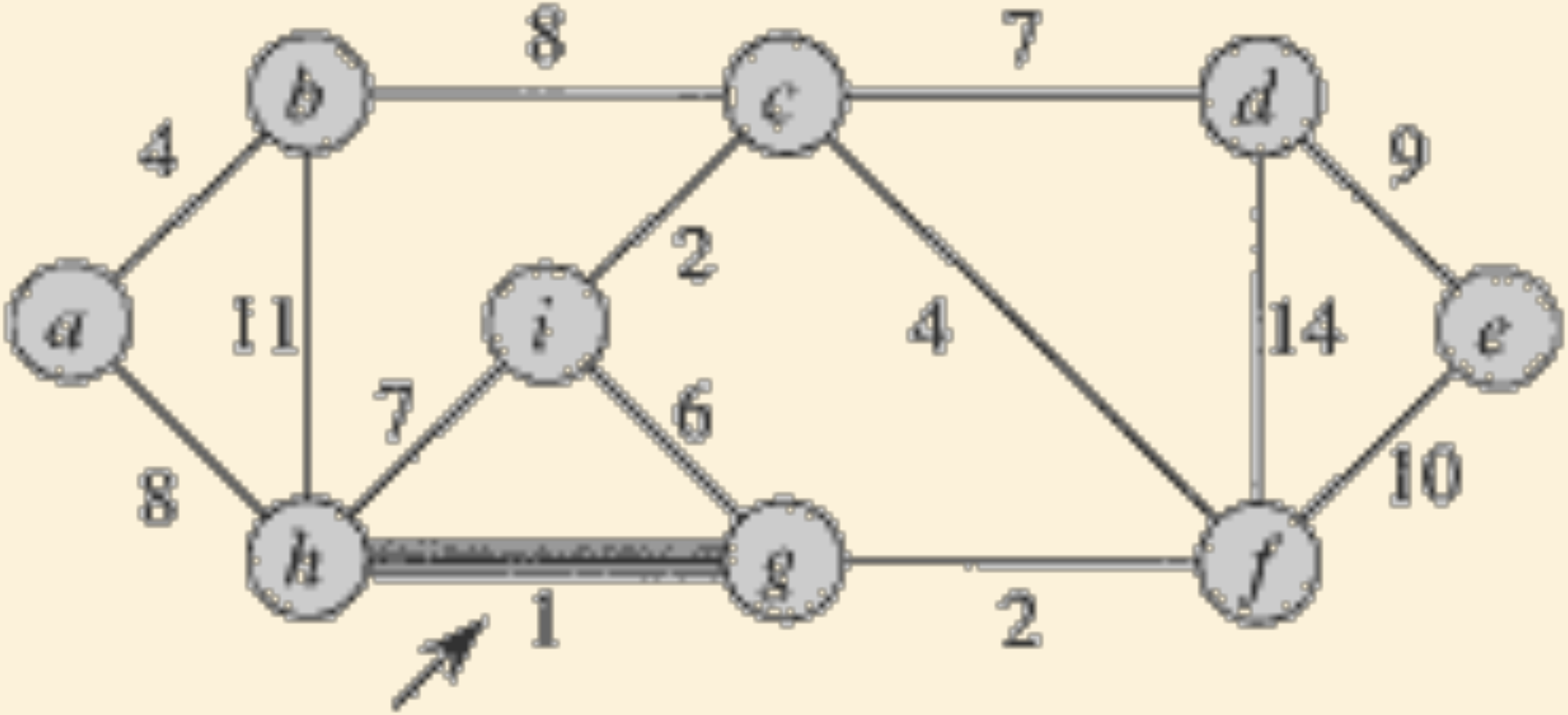
$\exists$ MST T :

1) $A \in T$,

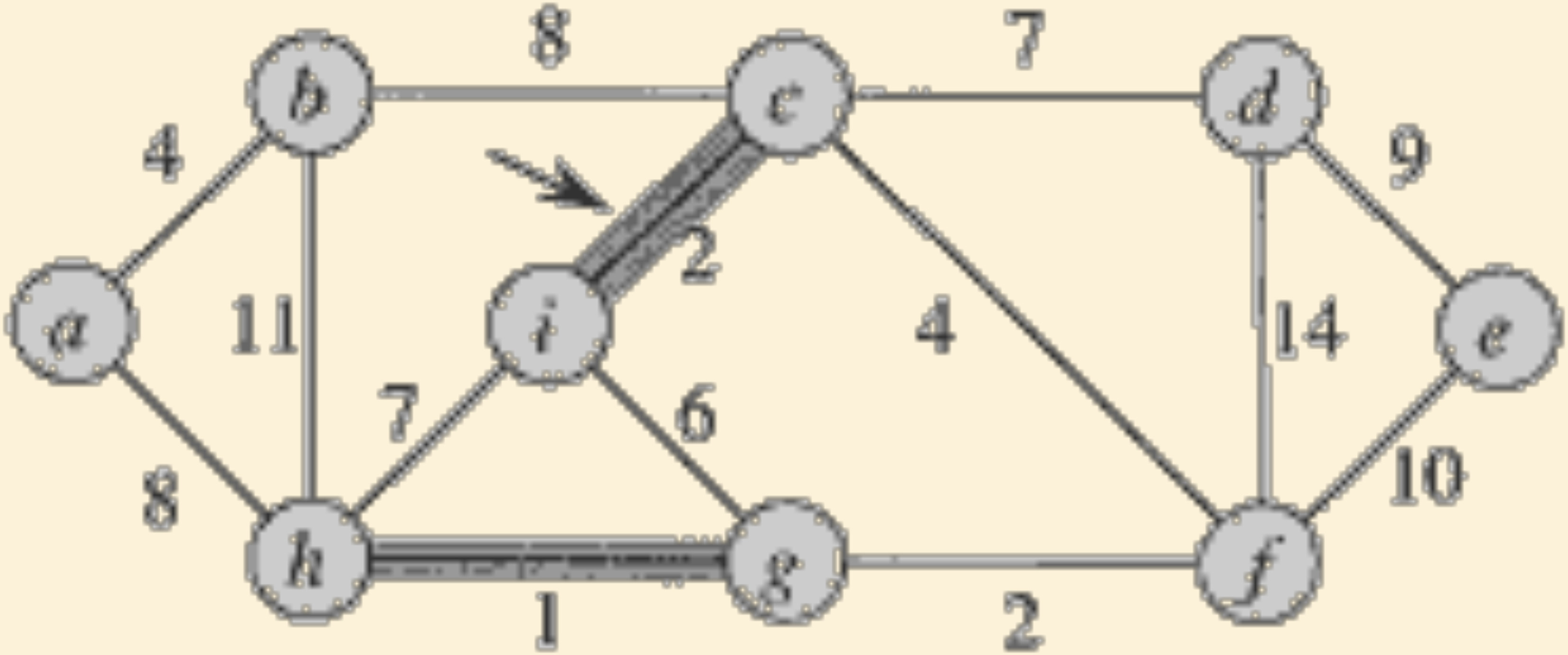
2) $\forall (u,v) \in E_i$:

$(u,v) \in A$ *or* $(u,v) \notin T$

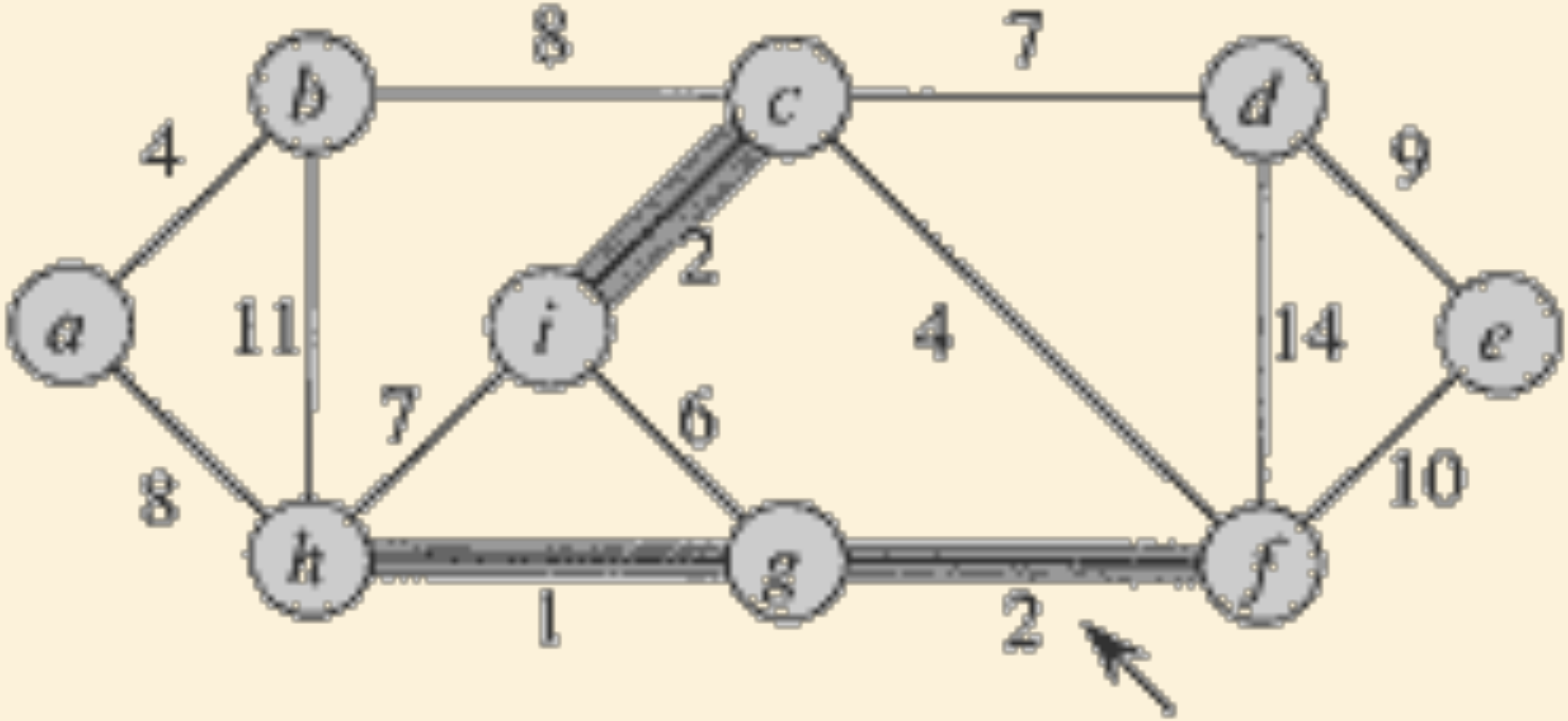
Kruskal's Algorithm: Example



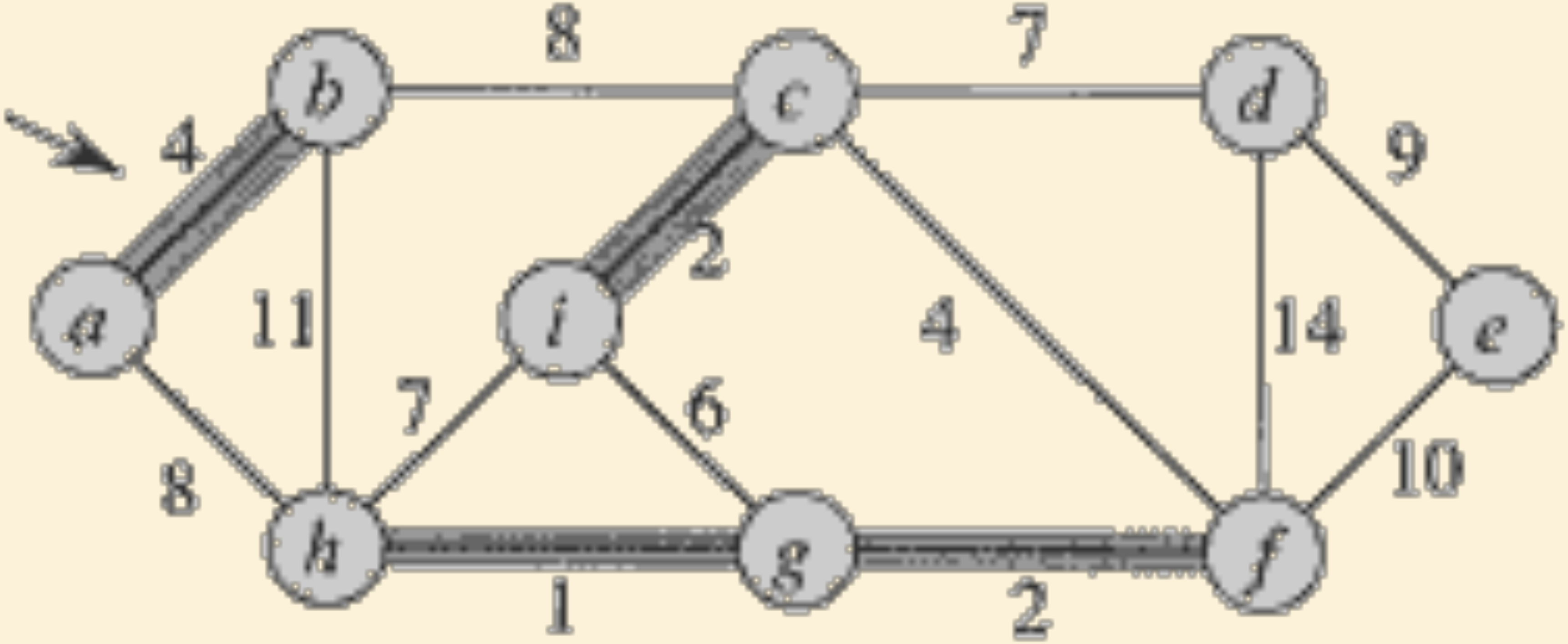
Kruskal’s Algorithm: Example



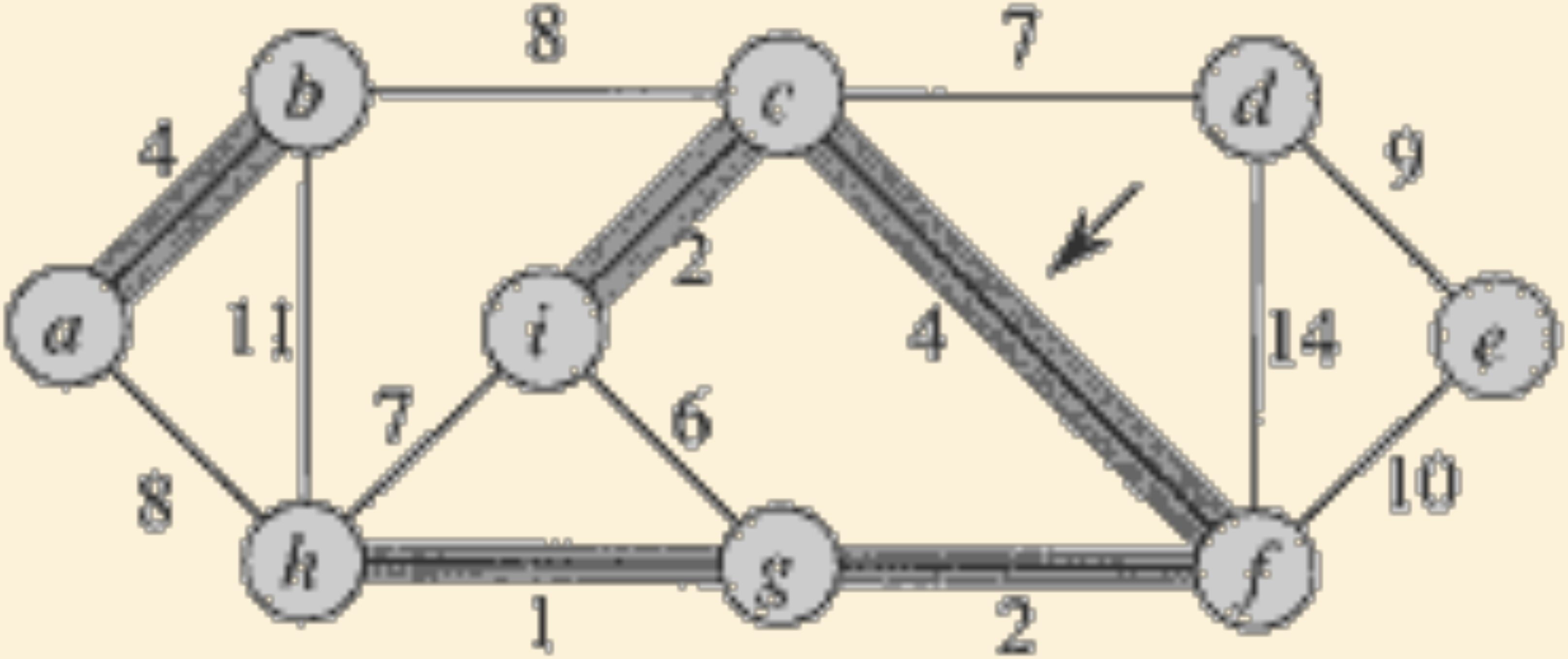
Kruskal’s Algorithm: Example



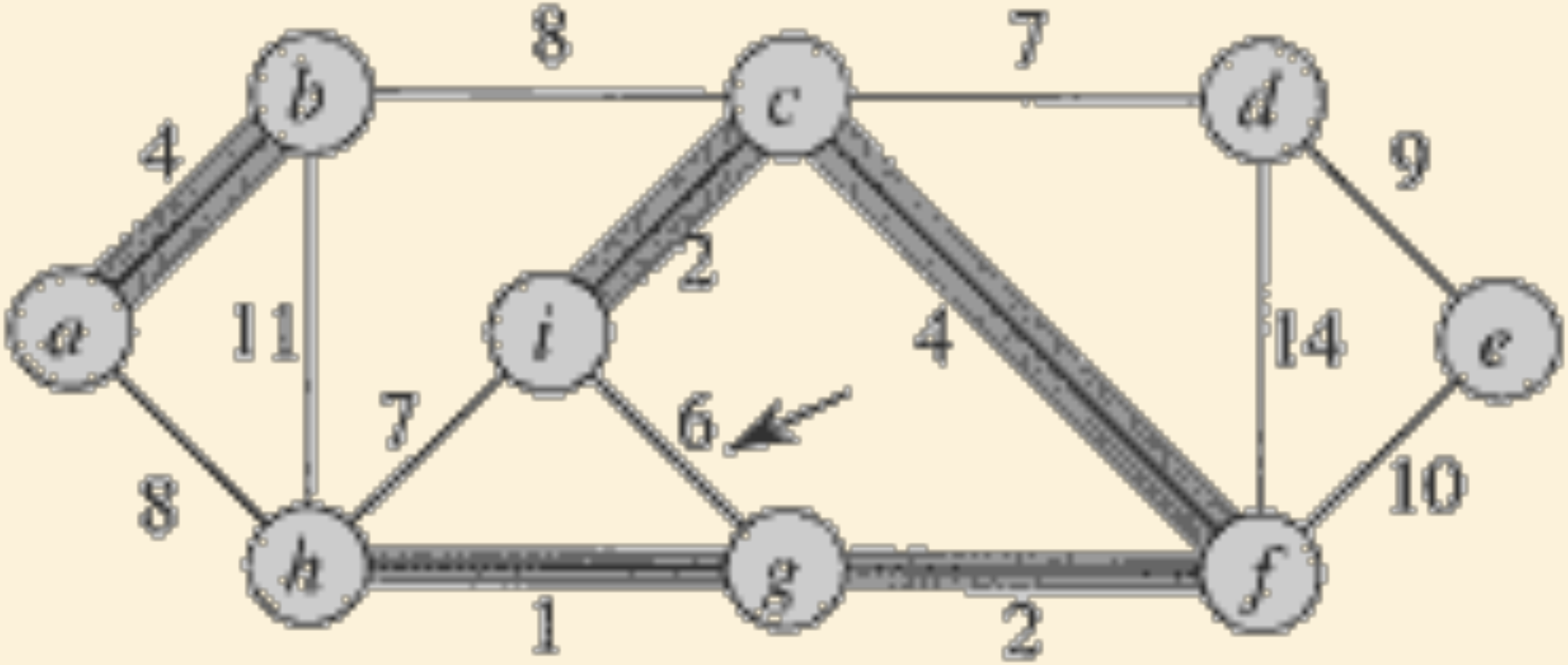
Kruskal’s Algorithm: Example



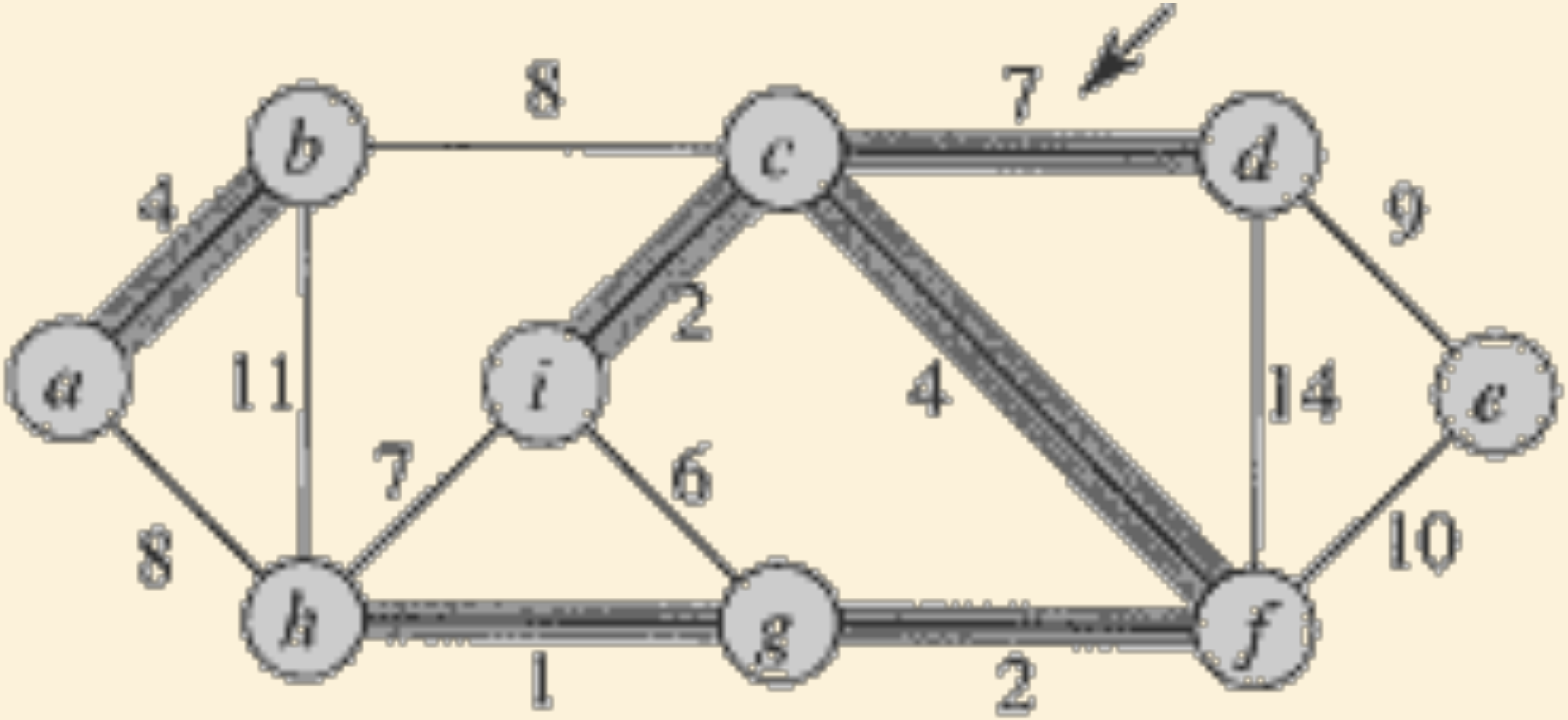
Kruskal’s Algorithm: Example



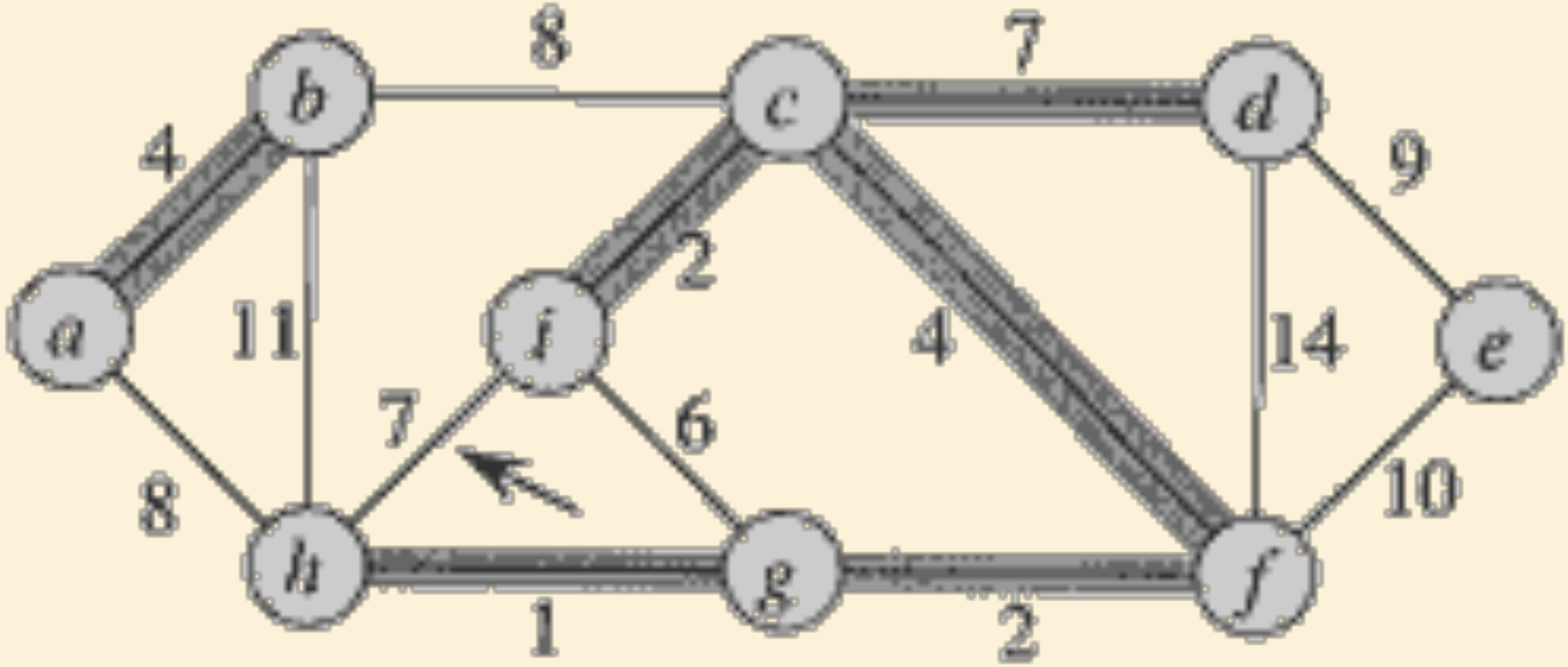
Kruskal’s Algorithm: Example



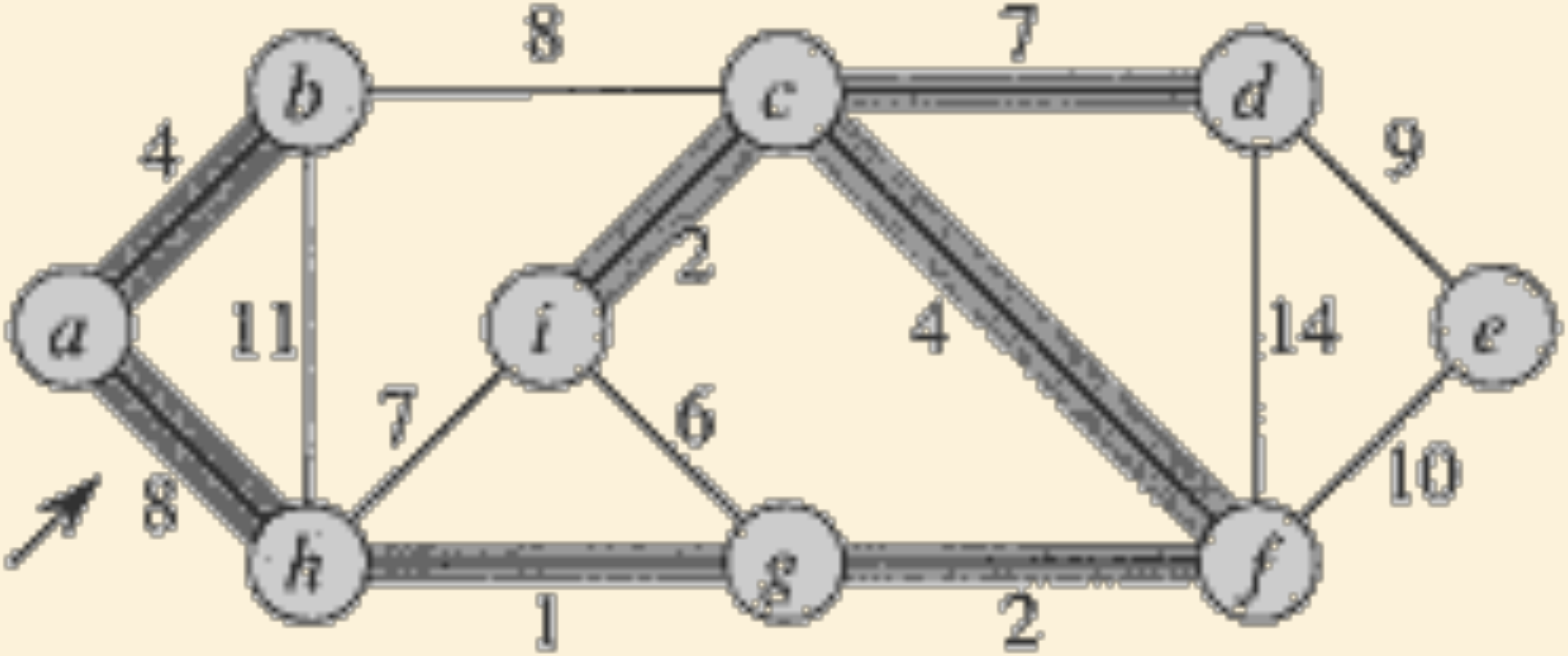
Kruskal's Algorithm: Example



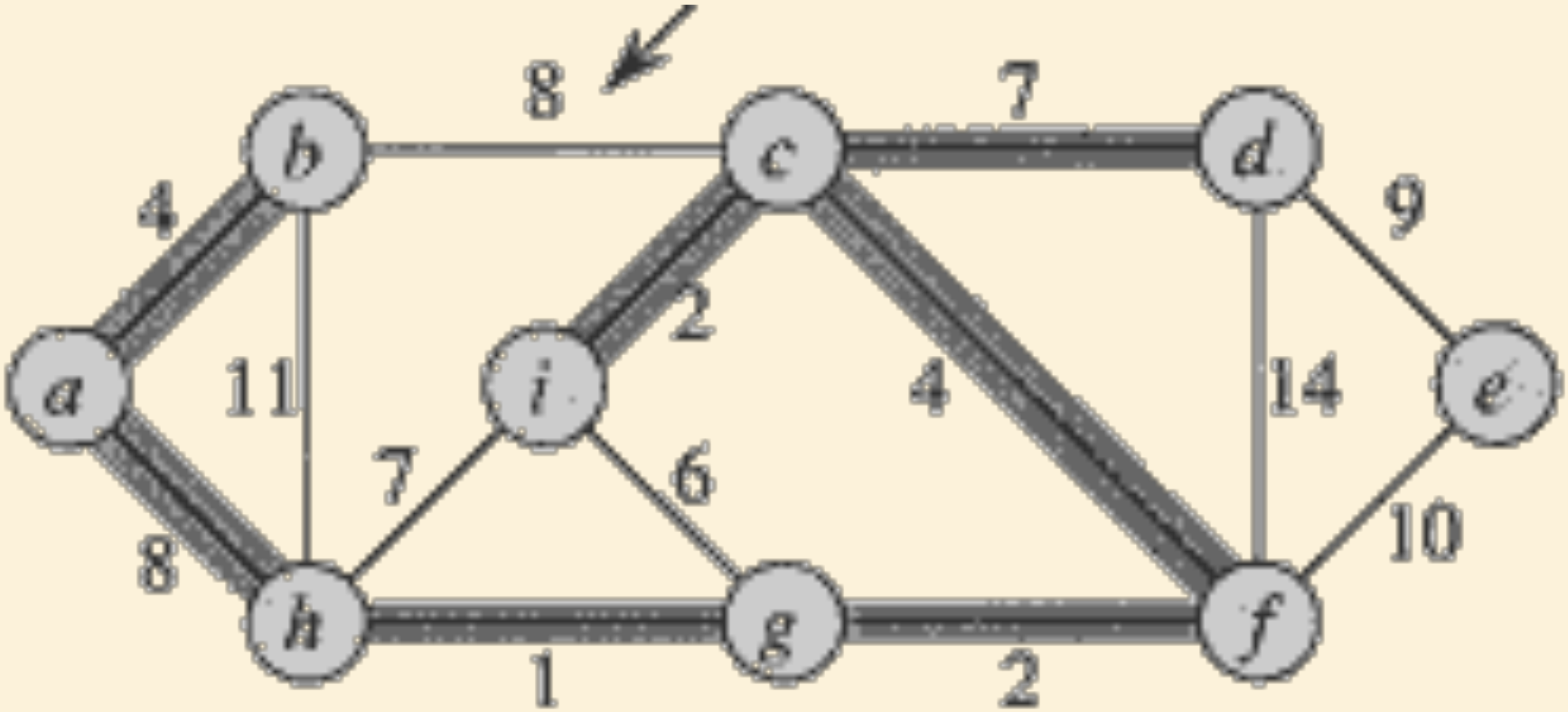
Kruskal's Algorithm: Example



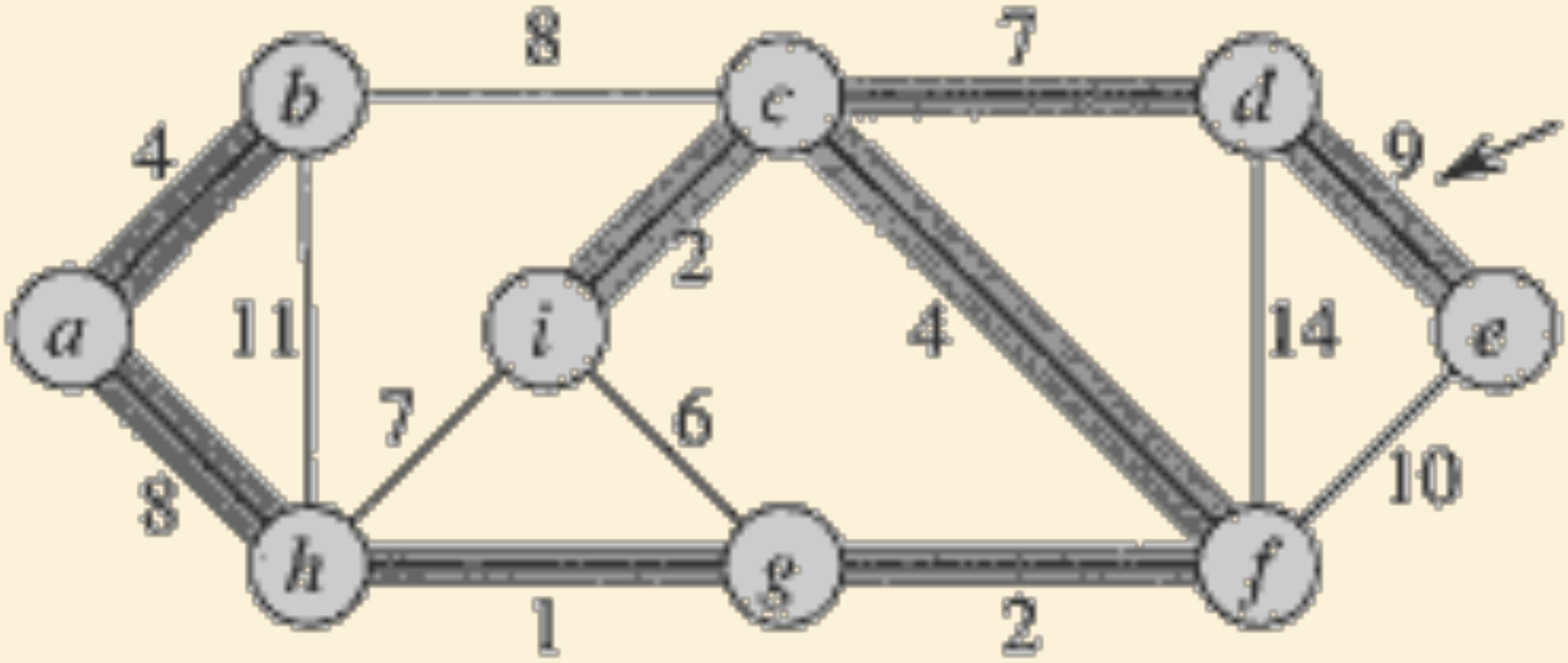
Kruskal's Algorithm: Example



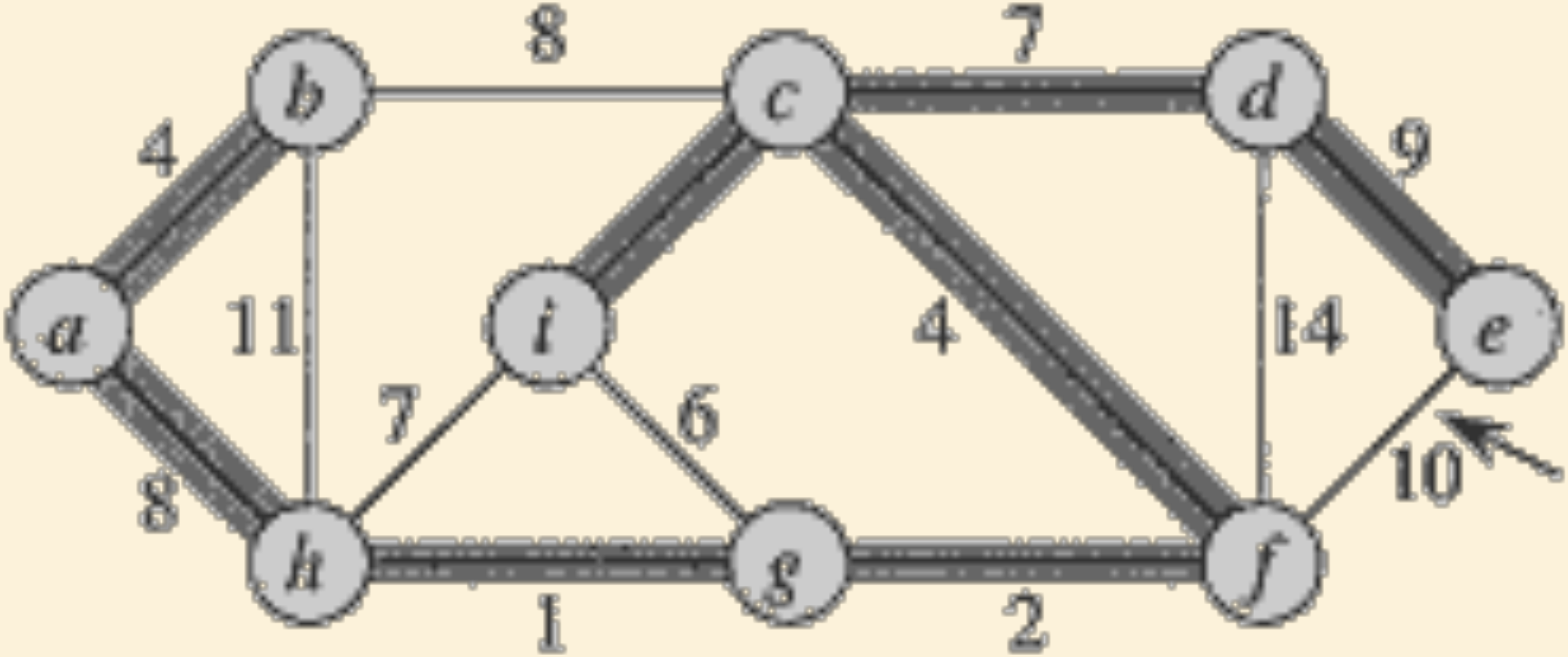
Kruskal’s Algorithm: Example



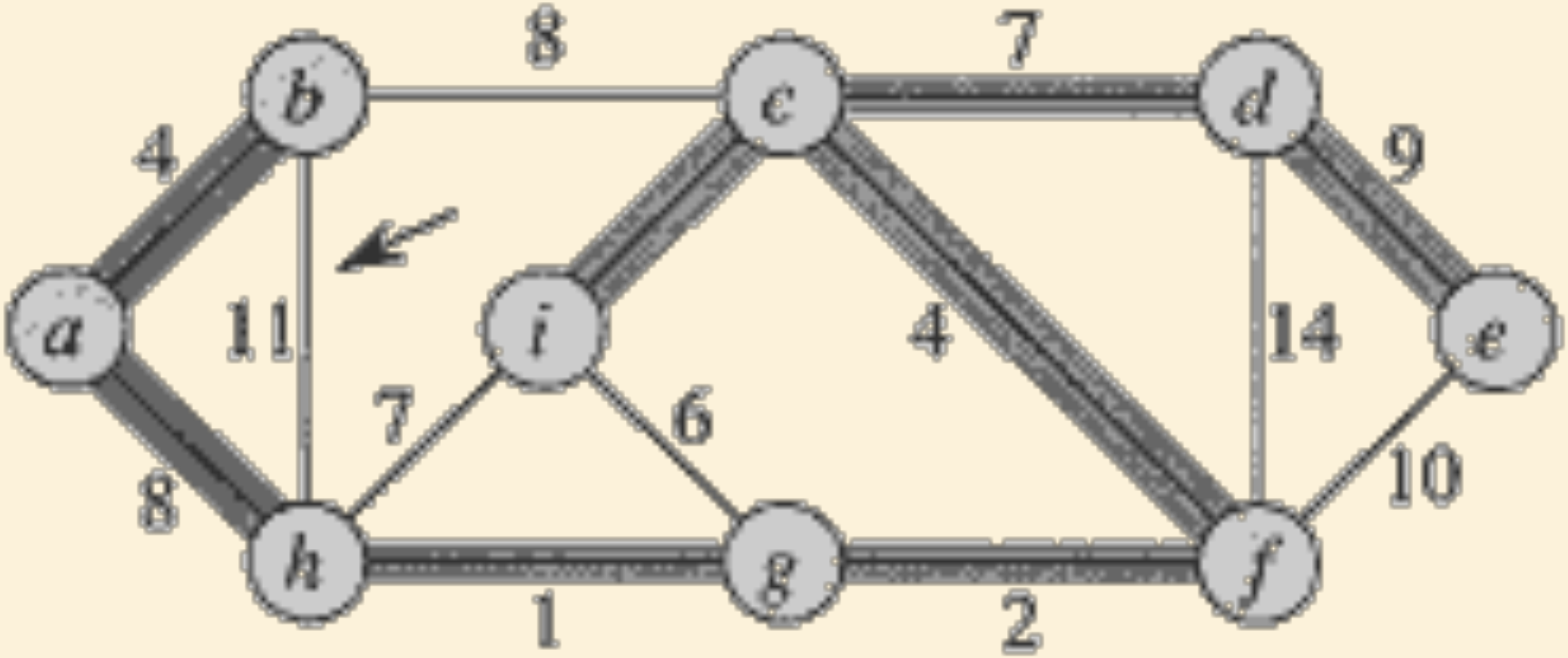
Kruskal’s Algorithm: Example



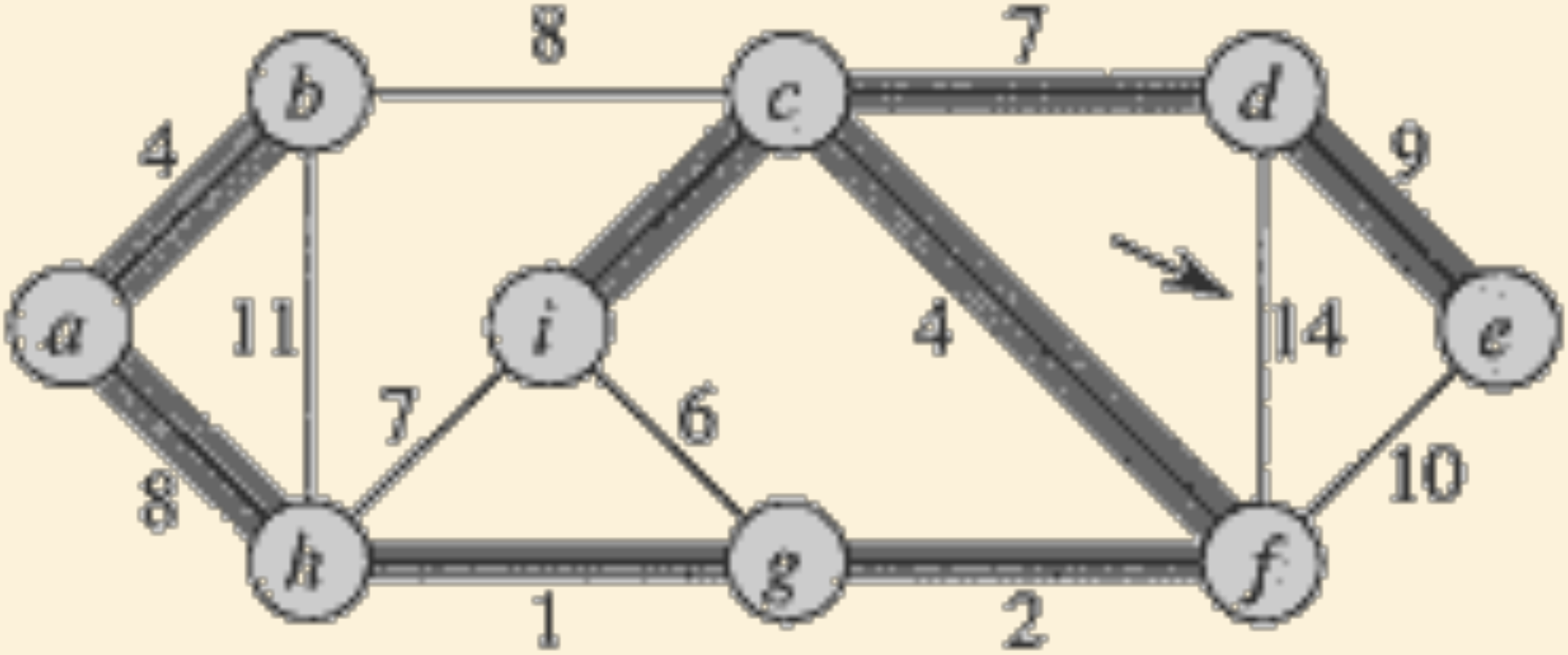
Kruskal's Algorithm: Example



Kruskal’s Algorithm: Example



Kruskal's Algorithm: Example



Finished!

Disjoint Set Data Structures

- Disjoint set data structures can be used to represent the disjoint connected components of a graph.
- Make-Set(*x*) makes a new disjoint component containing only vertex *x*.
- Union(*x*,*y*) merges the disjoint component containing vertex x with the disjoint component containing vertex *y*.
- Find-Set(*x*) returns a vertex that represents the disjoint component containing *x*.

Disjoint Set Data Structures

- Most efficient representation represents each disjoint set (component) as a tree.
- Time complexity of a sequence of *m* operations, *n* of which are Make-Set operations, is:

  $O(m \times \alpha(n))$

  where $\alpha(n)$ is Ackerman's function, which grows extremely slowly.

n	$\alpha(n)$
3	1
7	2
2047	3
10^{80}	4

Kruskal's Algorithm for computing MST

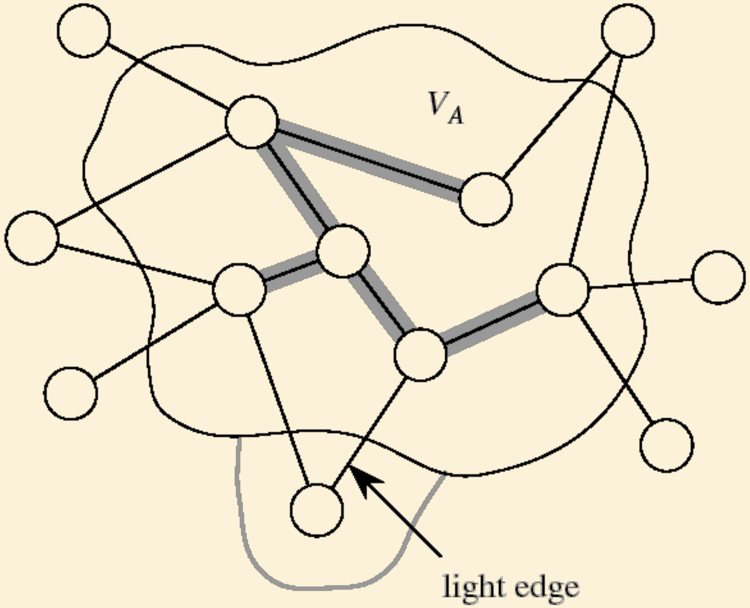
```
Kruskal($G$,$w$)
$A = \emptyset$
for each vertex $v \in V[G]$
    Make-Set(v)
sort E[G] into nondecreasing order: $E[1...n]$
for $i = 1 : n$
< loop-invariant >:
$\exists$ MST $T : 1) A \in T,$
            $2)\ \forall (u,v) \in E[1...i-1]: (u,v) \in A$ or $(u,v) \notin T$
    $(u,v) = E[i]$
    if Find-Set($u$) $\neq$ Find-Set($v$)
            $A = A \cup \{(u,v)\}$
            Union($u$,$v$)
return $A$
```

Running Time = $O(E \log E)$

$= O(E \log V)$

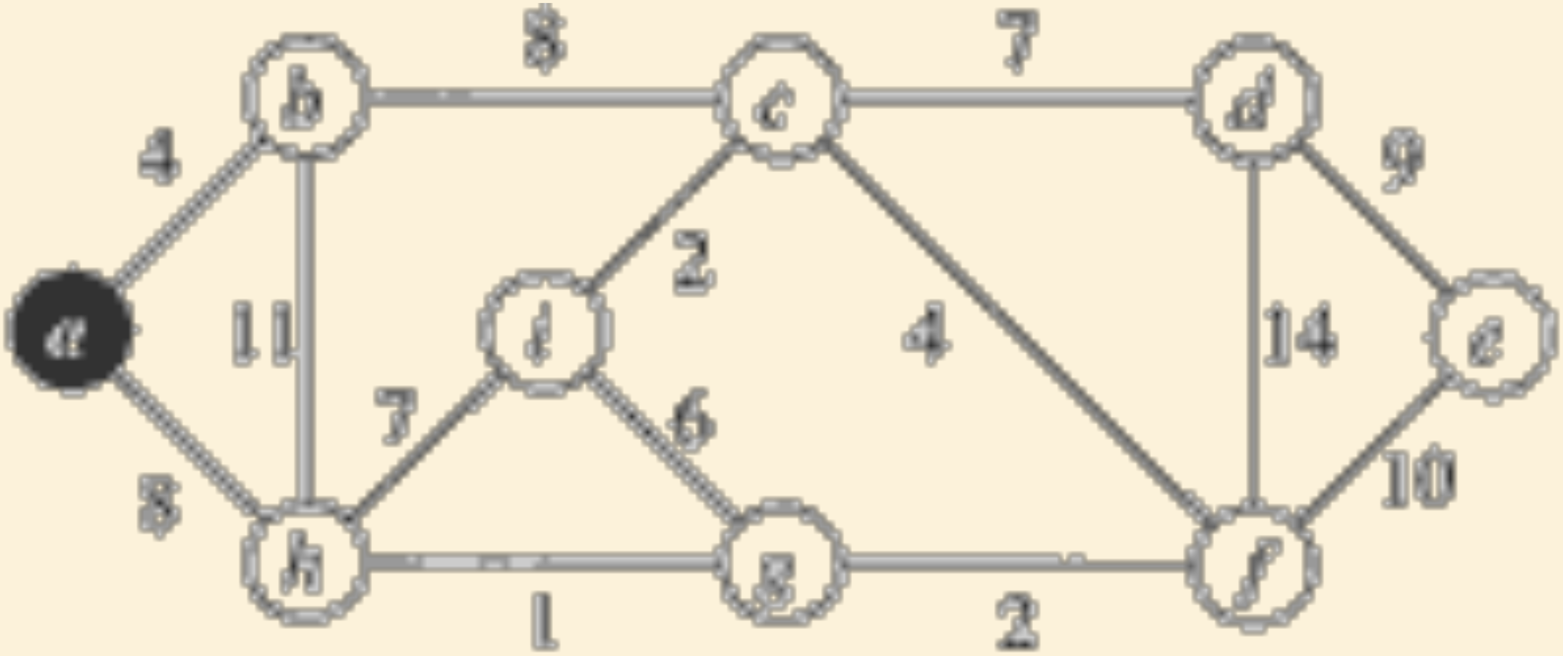
Prim's Algorithm for Computing MST

- Build one tree *A*
- Start from arbitrary root *r*
- At each step, add light edge connecting V_A to $V - V_A$ (greedy)

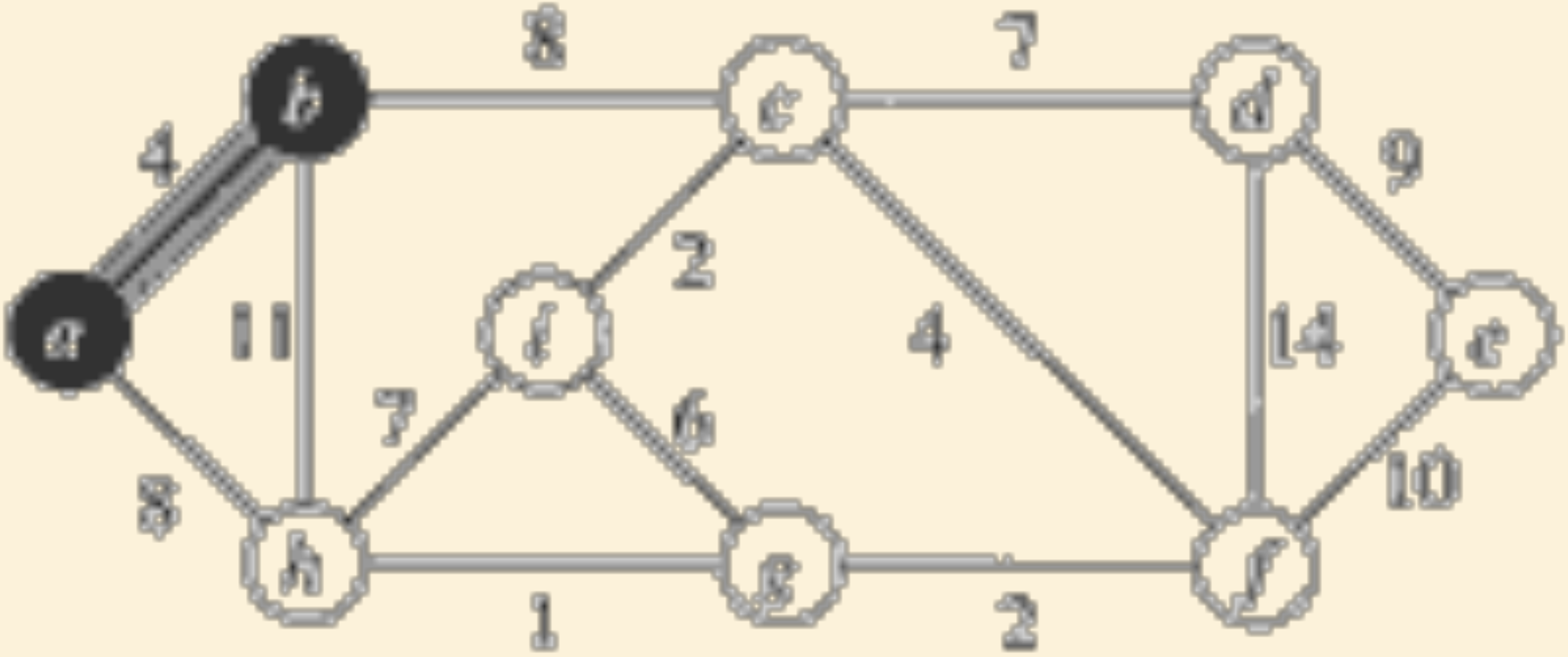


[Edges of A are shaded.]

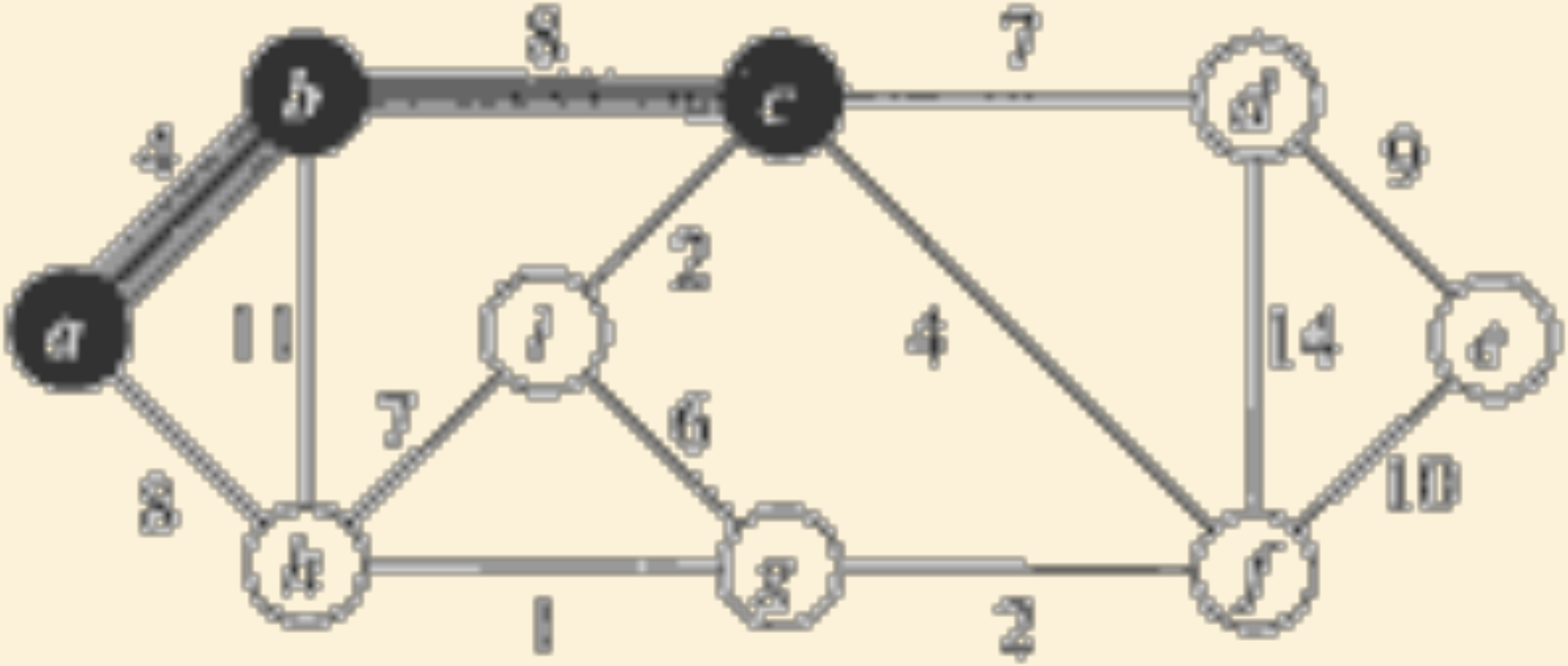
Prim’s Algorithm: Example



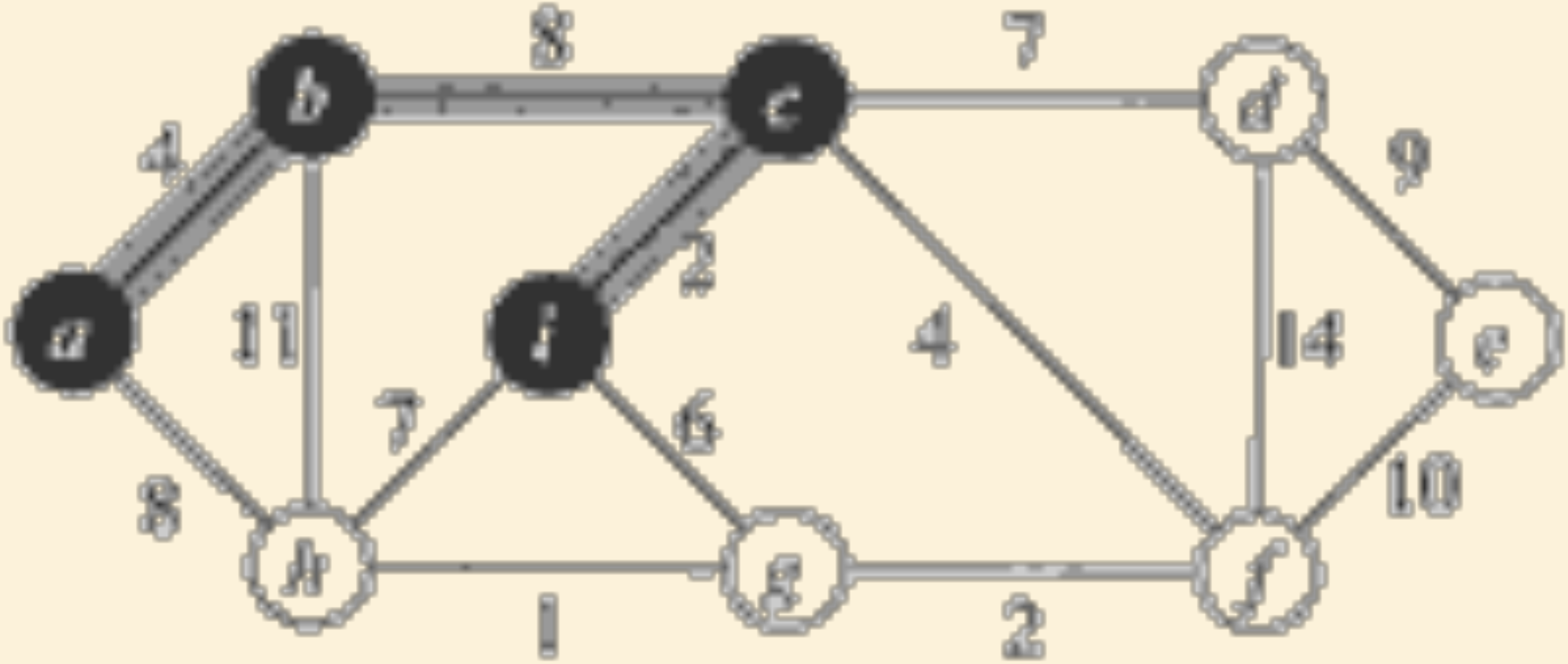
Prim’s Algorithm: Example



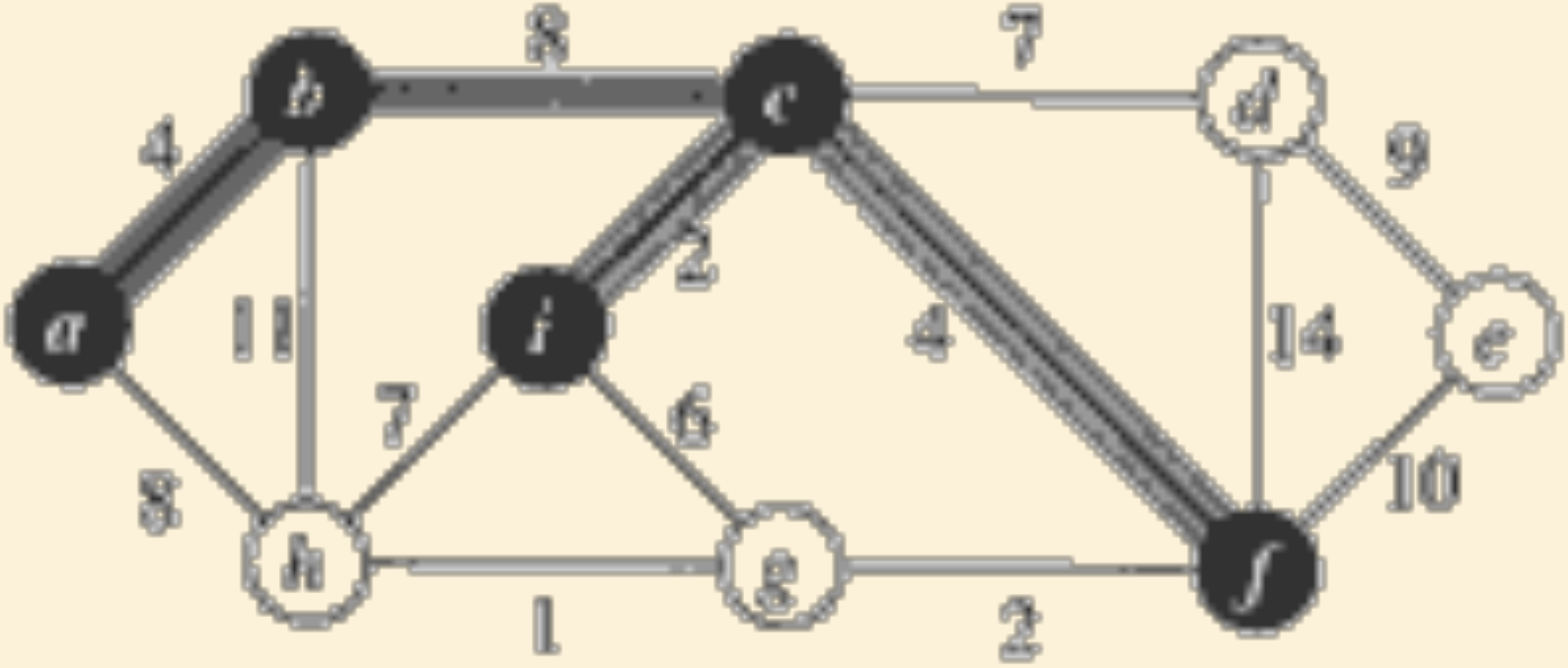
Prim’s Algorithm: Example



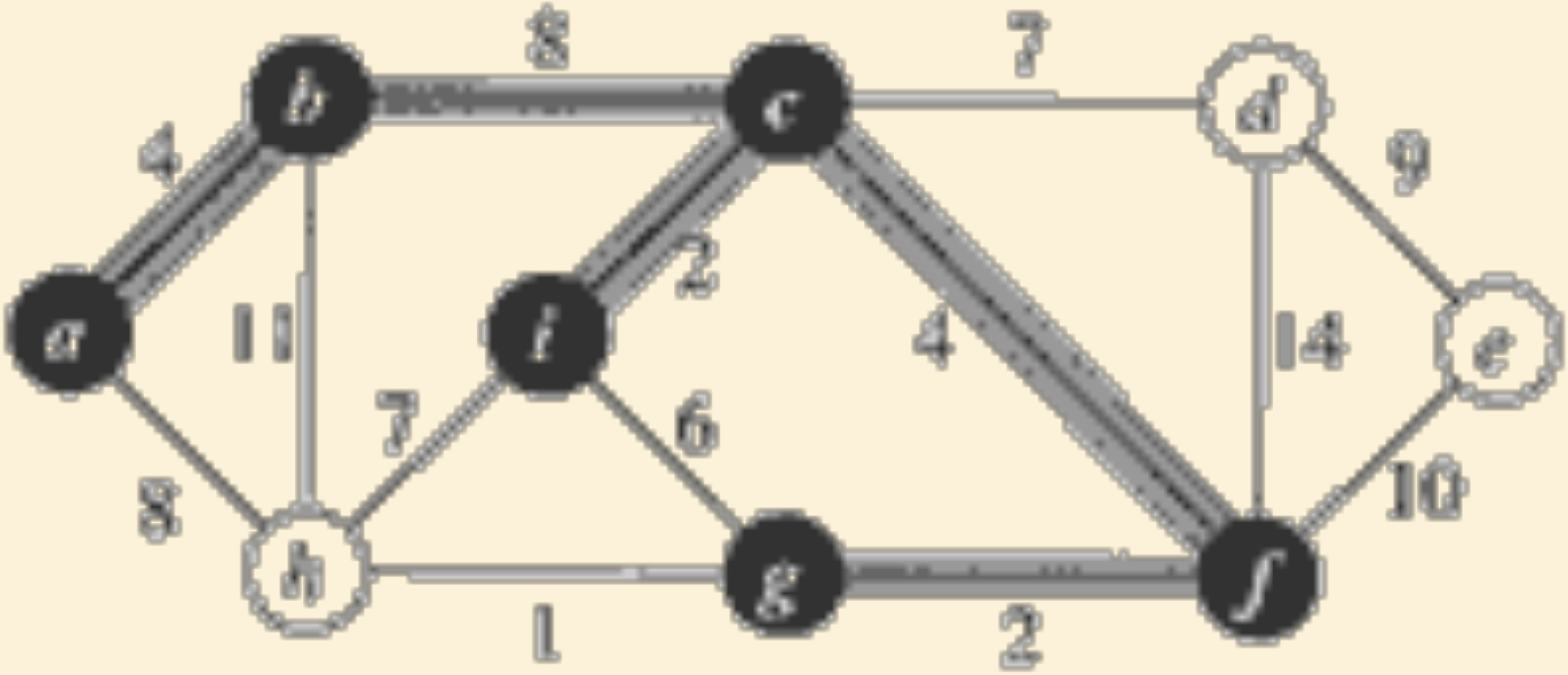
Prim’s Algorithm: Example



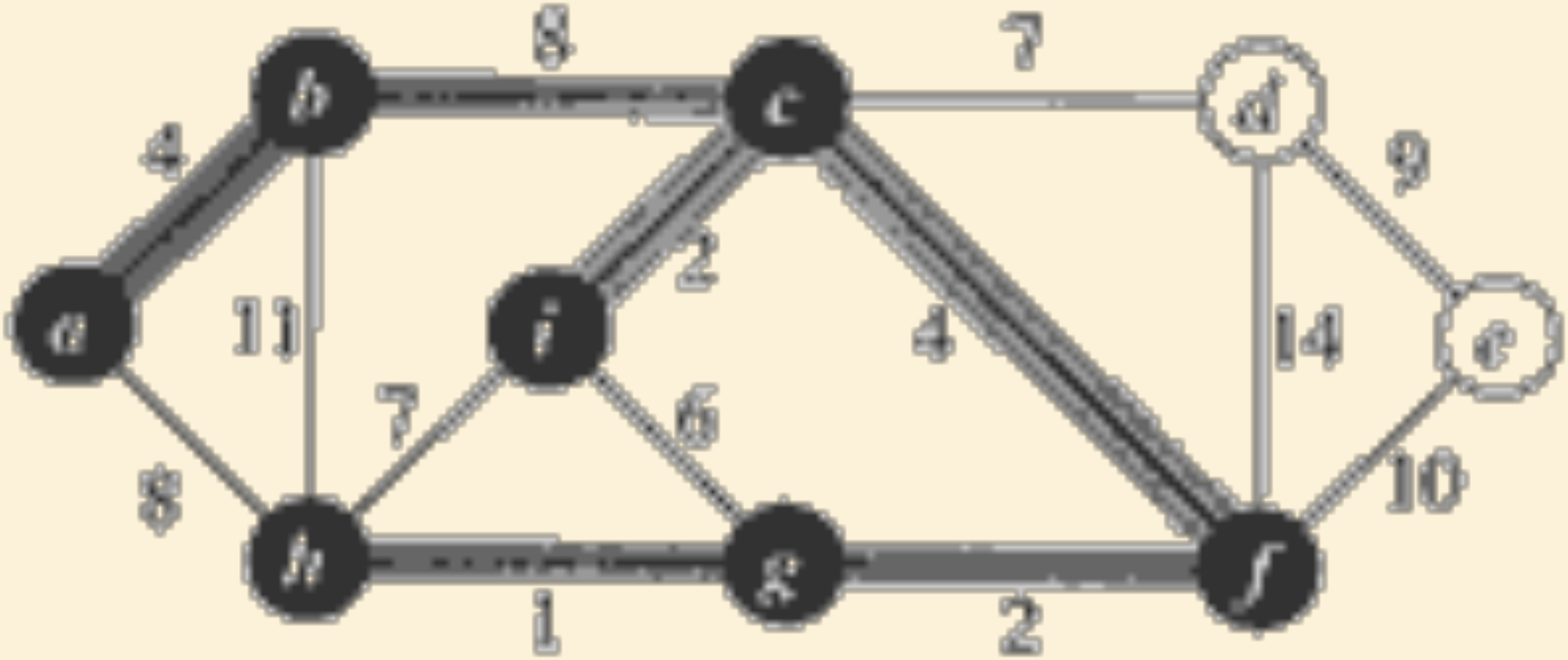
Prim’s Algorithm: Example



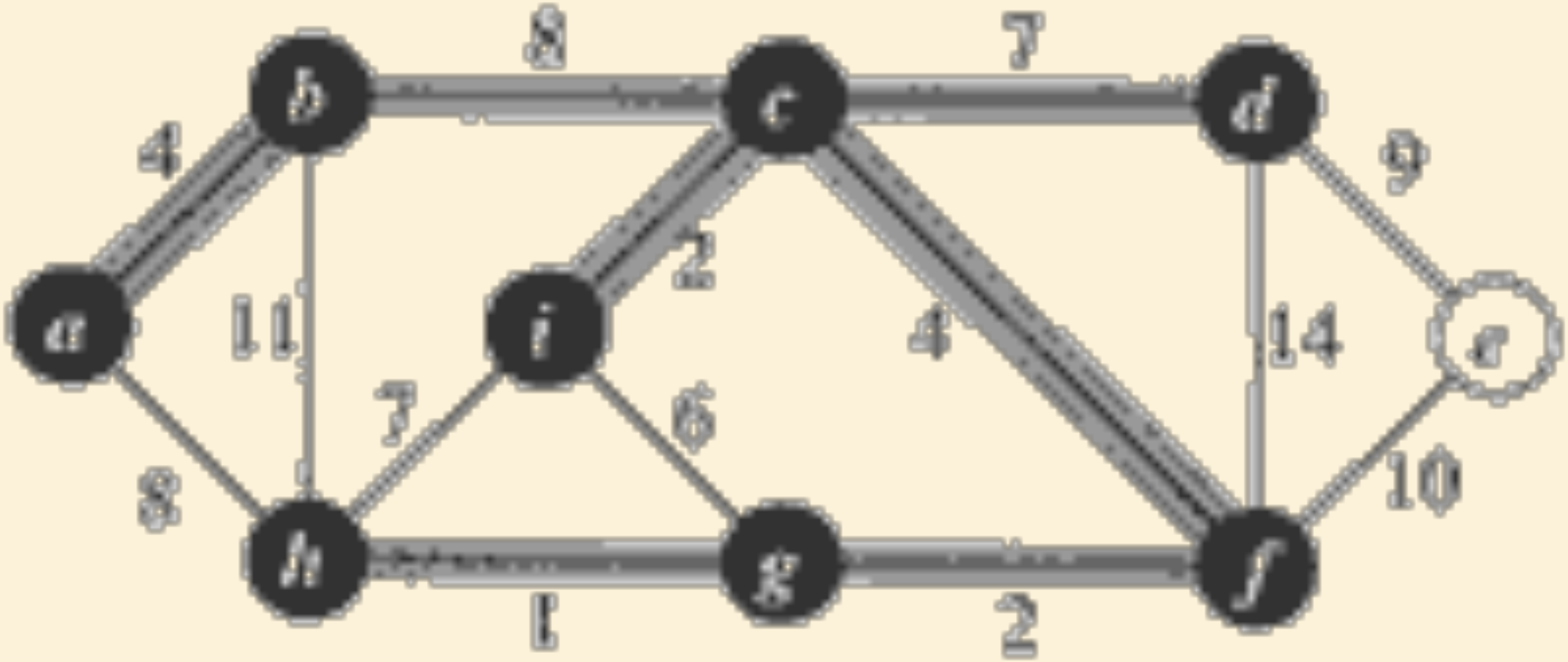
Prim’s Algorithm: Example



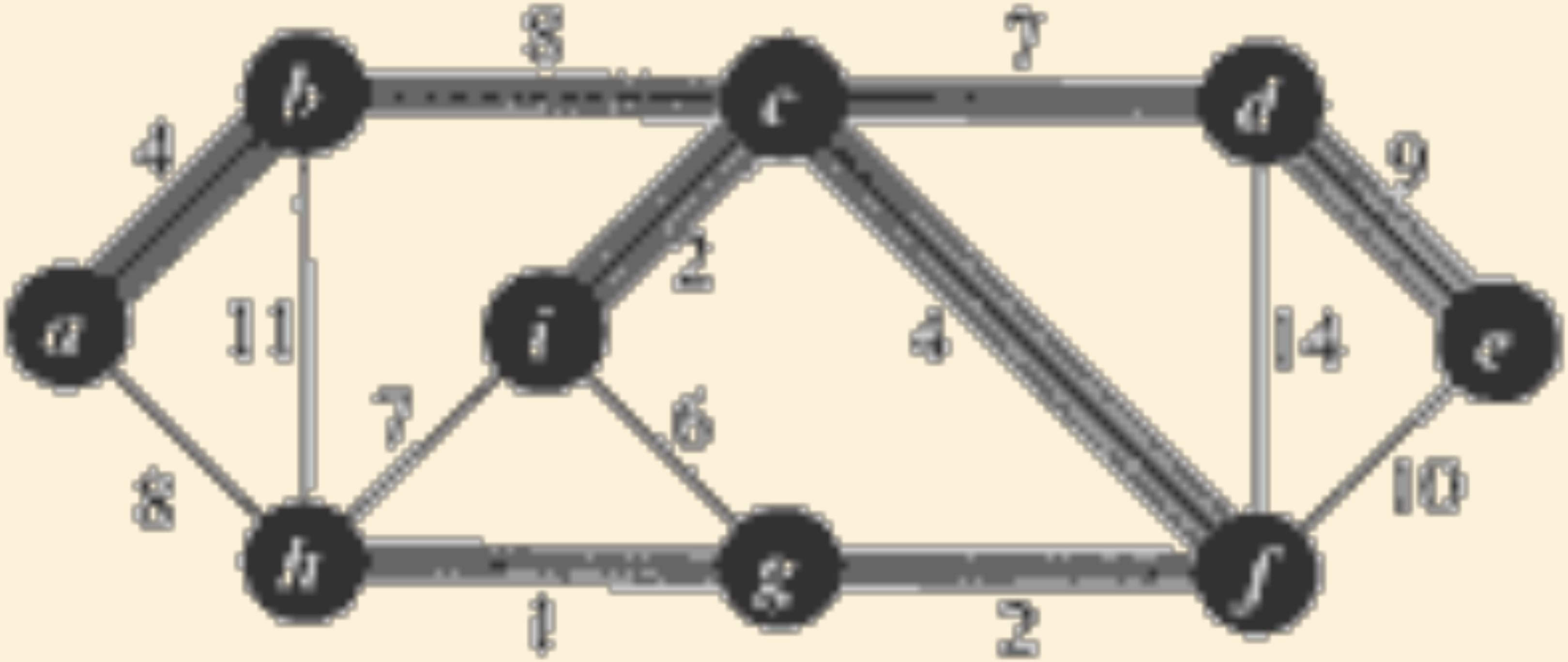
Prim’s Algorithm: Example



Prim's Algorithm: Example



Prim’s Algorithm: Example



Finished!

Finding light edges quickly

- All vertices not in the partial MST formed by *A* reside in a min-priority queue.
- Key(*v*) is minimum weight of any edge (*u,v*), $u \in V_{A.}$
- Priority queue can be implemented as a min heap on key(*v*).
- Each vertex in queue knows its potential parent in partial MST by π [v].

Prim's Algorithm

Let $A = \{(v, \pi[v]) : v \in V - \{r\} - Q\}$

Let $V_A = V - Q$

<loop-invariant>:

1. $\exists$ MST $T : A \in T$

2. $\forall v \in Q$, if $\pi[v] \neq$ NIL

then $key[v]$ = weight of light edge connecting v to V_A

```
PRIM($V, E, w, r$)
$Q \leftarrow \emptyset$
for each $u \in V$
    do $key[u] \leftarrow \infty$
       $\pi[u] \leftarrow$ NIL
       INSERT($Q, u$)
DECREASE-KEY($Q, r, 0$)    ▷ $key[r] \leftarrow 0$
while $Q \neq \emptyset$
    do $u \leftarrow$ EXTRACT-MIN($Q$)
       for each $v \in Adj[u]$
           do if $v \in Q$ and $w(u, v) < key[v]$
                 then $\pi[v] \leftarrow u$
                      DECREASE-KEY($Q, v, w(u, v)$)
```

Prim's Algorithm

```
PRIM($V, E, w, r$)
$Q \leftarrow \emptyset$
for each $u \in V$
    do $key[u] \leftarrow \infty$
       $\pi[u] \leftarrow$ NIL                                   $O(V)$
       INSERT($Q, u$)
DECREASE-KEY($Q, r, 0$)    $\triangleright$ $key[r] \leftarrow 0$
while $Q \neq \emptyset$                                         Executed |V| times
    do $u \leftarrow$ EXTRACT-MIN($Q$)                           $O(\log V)$
       for each $v \in Adj[u]$                                   Executed |E| times
           do if $v \in Q$ and $w(u, v) < key[v]$
                then $\pi[v] \leftarrow u$
                     DECREASE-KEY($Q, v, w(u, v)$)               $O(\log V)$
```

Running Time = $O(E \log V)$

Algorithm Comparison

- Both Kruskal's and Prim's algorithm are greedy.
  - Kruskal's: Queue is static (constructed before loop)
  - Prim's: Queue is dynamic (keys adjusted as edges are encountered)