CS 170
Efficient Algorithms and Intractable Problems

Lecture 24
Randomized Algorithms

Nika Haghtalab and John Wright

EECS, UC Berkeley

Announcements

End-of-semester course evaluations are open now
→ You can receive an additional homework drop if you fill it out (see “End-of-Semester Feedback Form” on Ed on how to receive HW drop)

Wrapping Up Intelligent Search

End-of-semester course evaluations are open now
→ You can receive an additional homework drop if you fill it out (see “End-of-Semester Feedback Form” on Ed on how to receive HW drop)

Branch-and-Bound

Rule out optimality for minimization problem:

→ We need a function $lowerbound(P_i)$ that looks at a partial solution P_i and quickly gives us a lower bound on the value of any possible completion of P_i.

→ If $lowerbound(P_i) >$ best-so-far, the entire branch under P_i can be eliminated.

Branch-and-bound for a minimization problem

```
Start with problem $P_0$ and let $S = \{P_0\}$, the set of active subproblems
best-so-far = ∞
Repeat while $S \neq \emptyset$:
    Choose a subproblem (partial solution) $P \in S$ and remove it from $S$
    Expand the problem into smaller subproblems $P_1, P_2, \dots, P_k$
    For each $P_i$:
        If $P_i$ is a complete solution, update best-so-far if it's the best value so far
        Else if $lowerbound(P_i) <$ best-so-far, add $P_i$ to $S$.
Return best-so-far
```

Branch-and-Bound for TSP

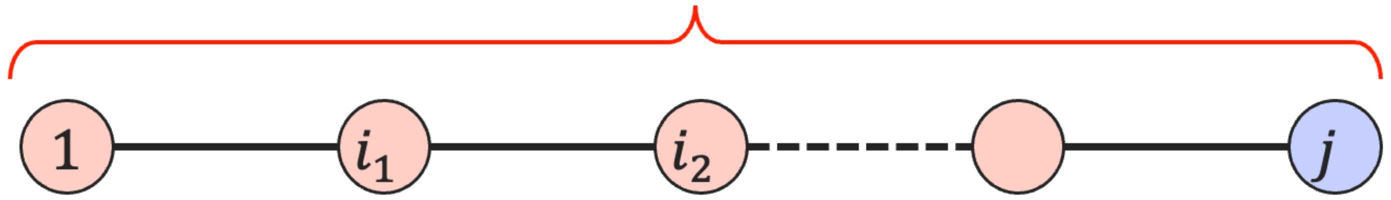
Recall: TSP(graph $G = ([n], E)$ and edge lengths $d_e > 0$ for all $e \in E$, returns a tour (a cycle passing through all nodes) of the smallest length.

Partial Solutions: Same subproblems as in our DP algorithm for TSP.

Lecture 13

Think of subproblems as partial tour!
→ It starts from city 1, ends in city j, and passing through all cities in a set S (which includes 1 and j).

Set S of cities (including 1 and j)



Subproblems: For all $j \leq n$ and $S \subseteq \{1, \ldots, n\}$, s.t. S includes 1 and j.

$T[S, j]$ = length of the shortest path visiting all cities in S exactly once, starting from 1 and ending at j.

Lower-Bounding Value of Partial TSP

Subproblems: For all $j \leq n$ and $S \subseteq \{1, \ldots, n\}$, s.t. S includes 1 and j.

$T[S, j] =$ the shortest path visiting all cities in S exactly once, starting from 1 and ending at j.

$lowerbound(T[S, j])$ needs to lower bound the completion of this tour.

Lower-Bounding Value of Partial TSP (cont.)

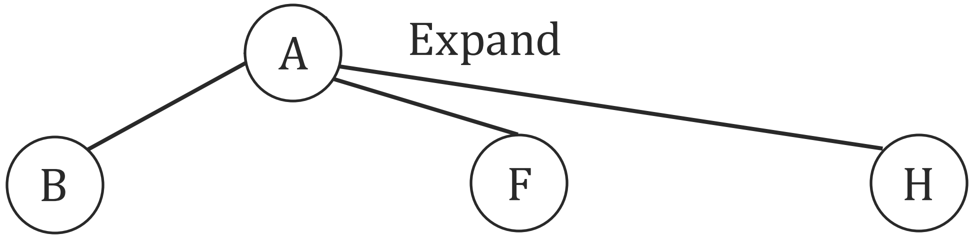
Lemma: Let $lowerbound(T[S,j]) = \mathrm{MST}(V \setminus S) + \min_{x \in V \setminus S} d_{1x} + \min_{x \in V \setminus S} d_{jx} + T[S,j]$.

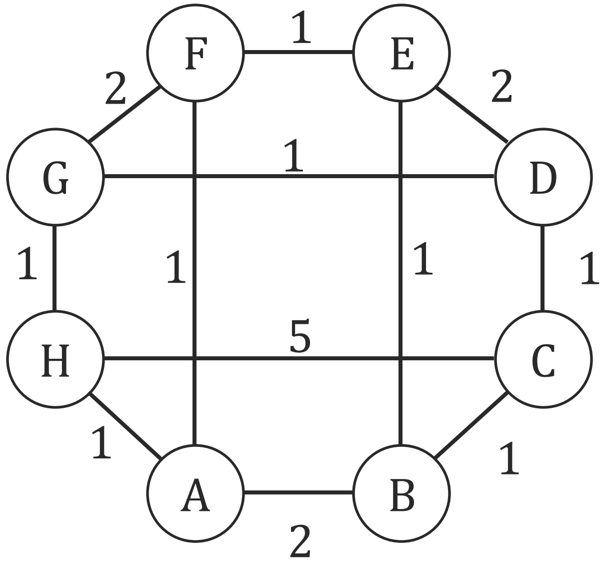
This is a valid lower bound, i.e., any tour that uses $T[S,j]$ as a partial tour, has a length that is at least $lowerbound(T[S,j])$

Proof:

Example of Branch-and-Bound TSP

Best-so-far = ∞

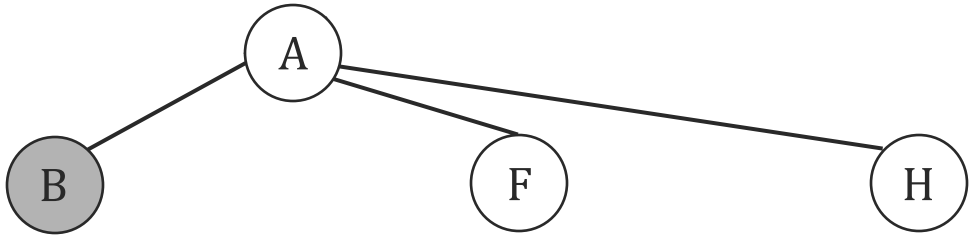




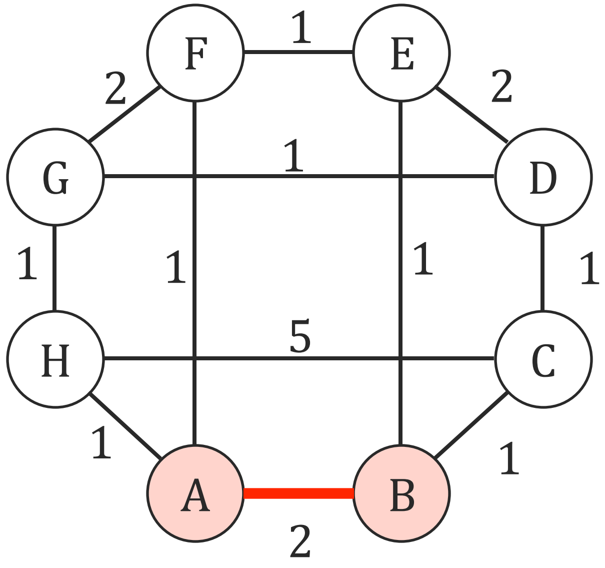
Example from Sec 9 of the textbook

Example of Branch-and-Bound TSP

Best-so-far = ∞



Discard?

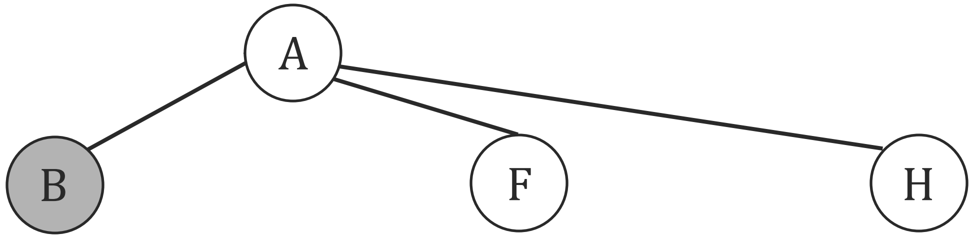


Current partial solution shown in red.

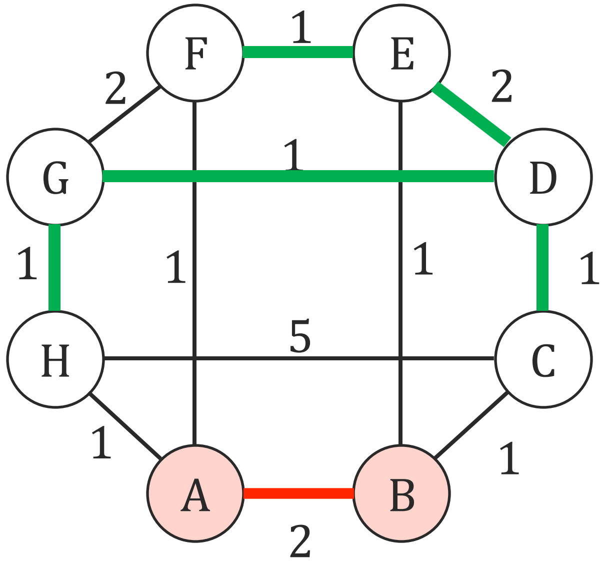
Example from Sec 9 of the textbook

Example of Branch-and-Bound TSP

Best-so-far = ∞



Discard?

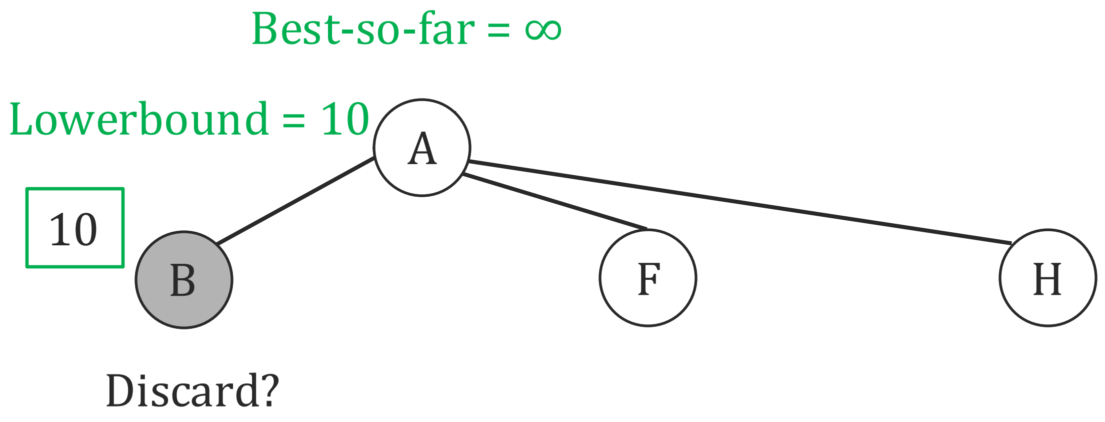


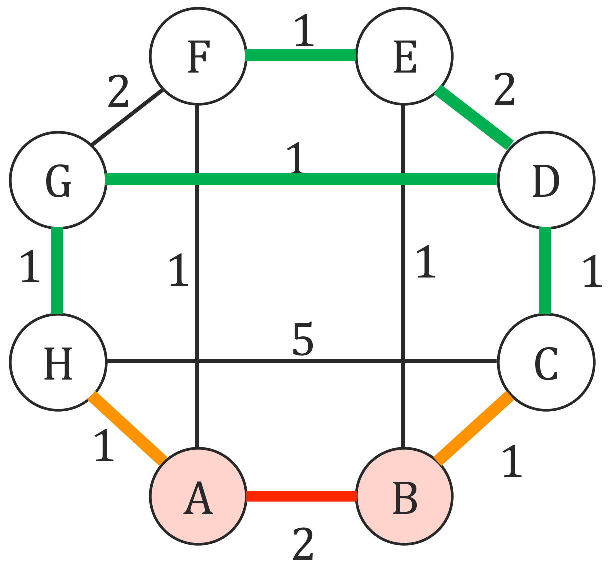
Current partial solution shown in red.

MST of the complement set shown in green.

Example from Sec 9 of the textbook

Example of Branch-and-Bound TSP





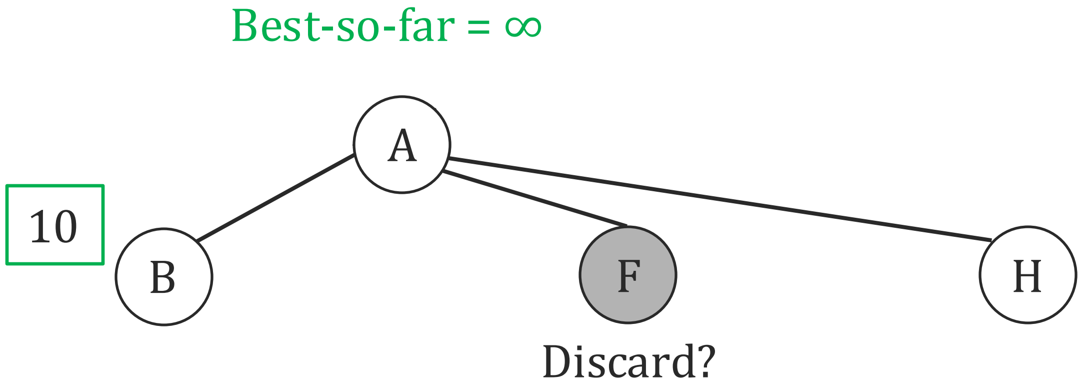
Current partial solution shown in red.

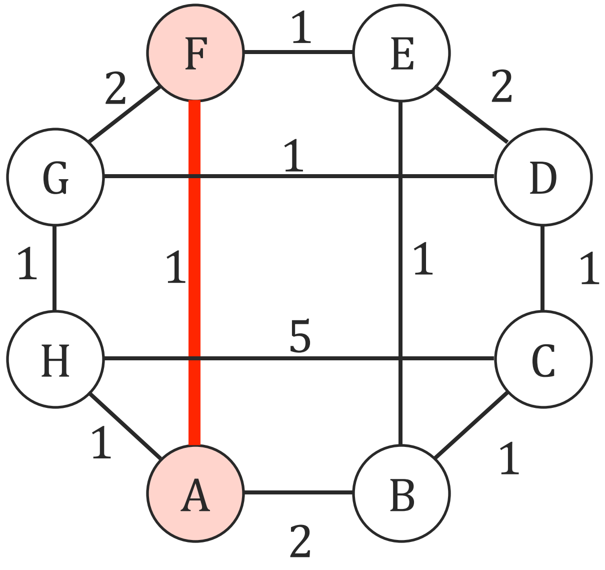
MST of the complement set shown in green.

Lightest edges connecting the blue tour to the complement are shown in orange.

Example from Sec 9 of the textbook

Example of Branch-and-Bound TSP





Current partial solution shown in red.

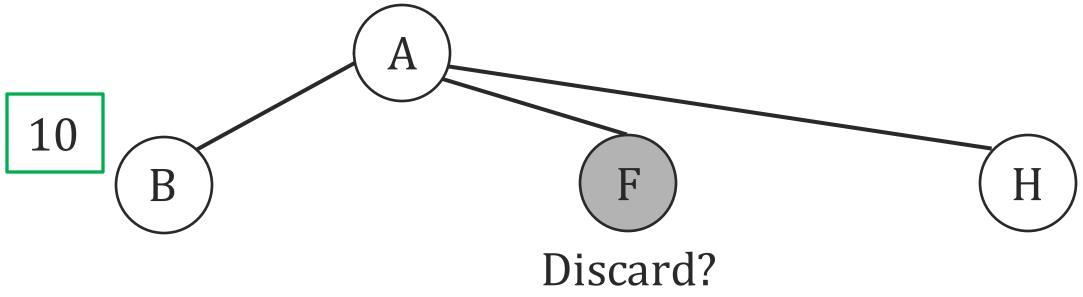
MST of the complement set shown in green.

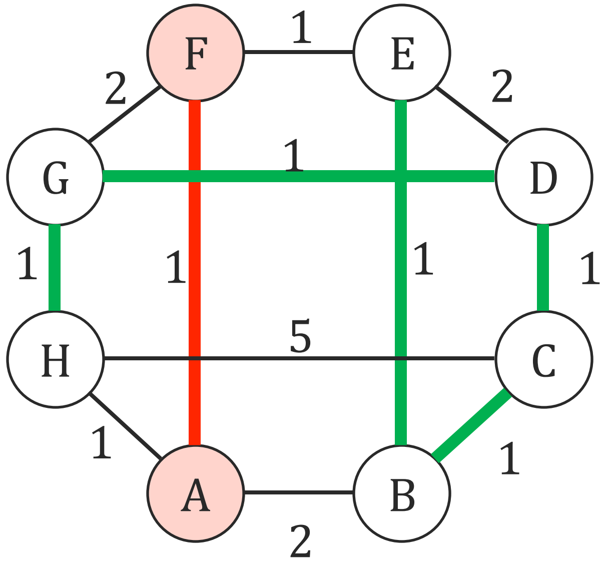
Lightest edges connecting the blue tour to the complement are shown in orange.

Example from Sec 9 of the textbook

Example of Branch-and-Bound TSP

Best-so-far = ∞





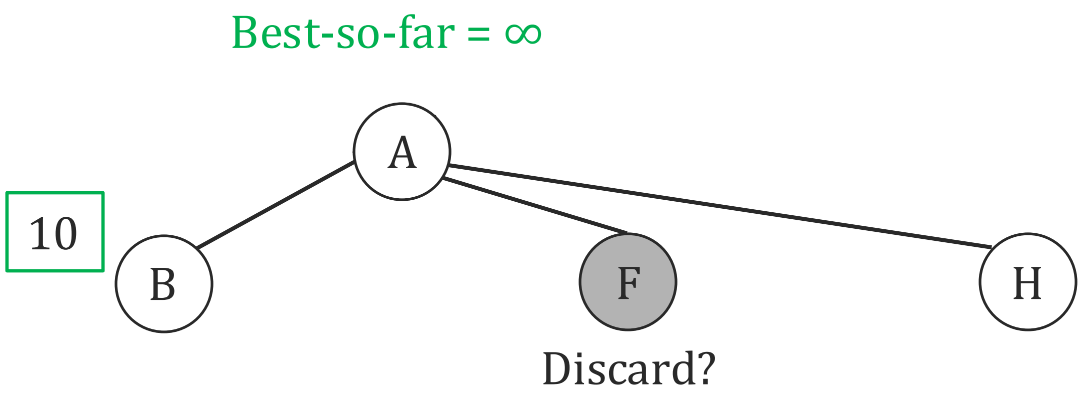
Example from Sec 9 of the textbook

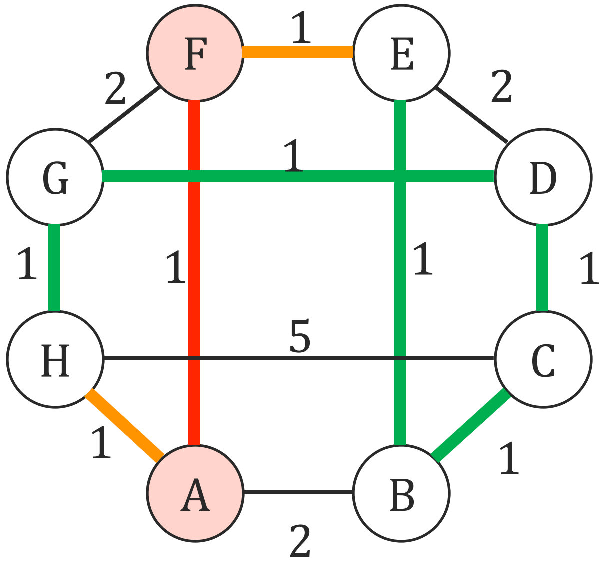
Current partial solution shown in red.

MST of the complement set shown in green.

Lightest edges connecting the blue tour to the complement are shown in orange.

Example of Branch-and-Bound TSP





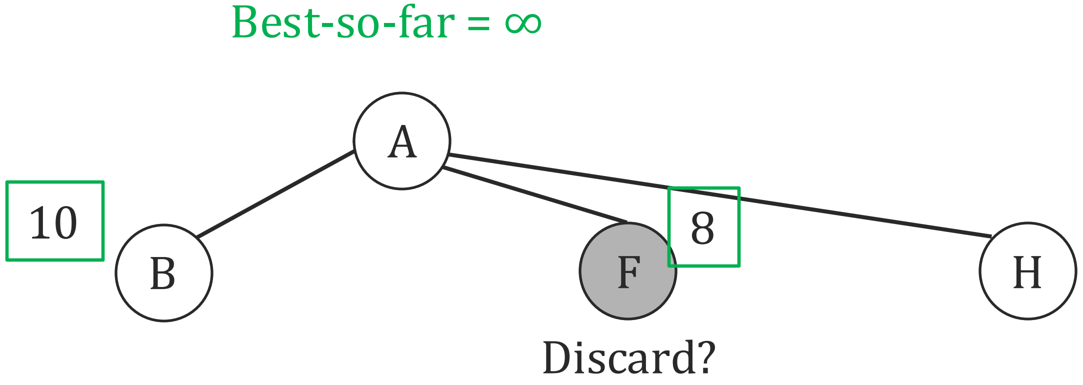
Current partial solution shown in red.

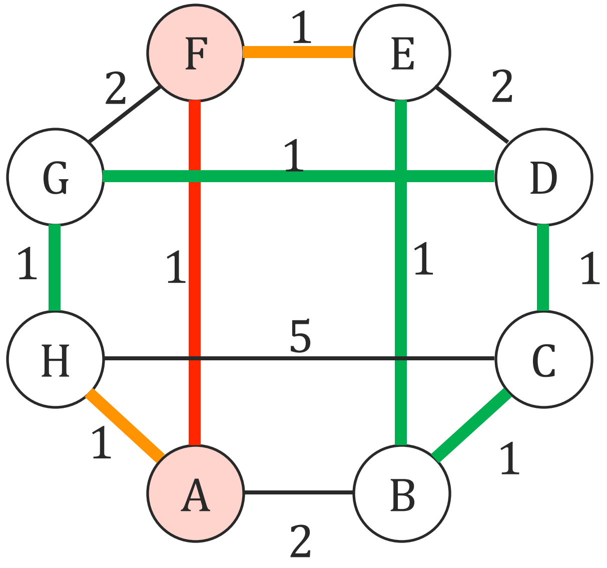
MST of the complement set shown in green.

Lightest edges connecting the blue tour to the complement are shown in orange.

Example from Sec 9 of the textbook

Example of Branch-and-Bound TSP





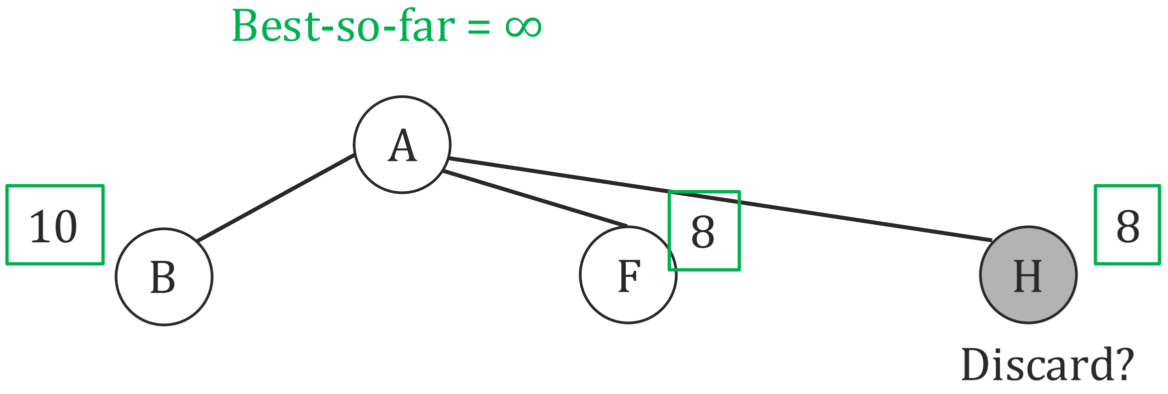
Current partial solution shown in red.

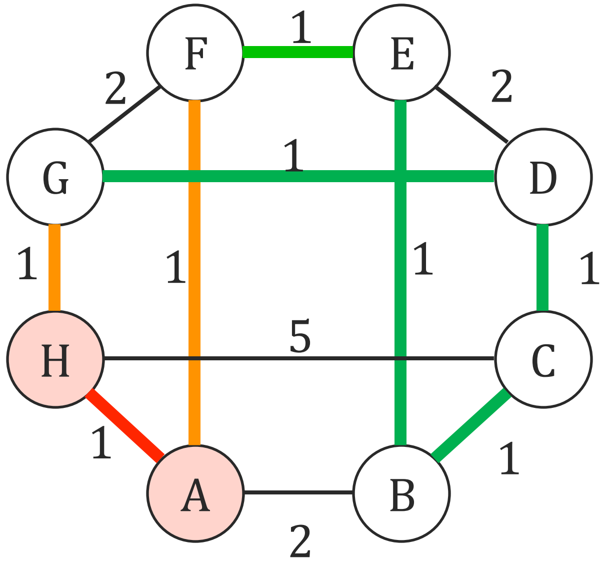
MST of the complement set shown in green.

Lightest edges connecting the blue tour to the complement are shown in orange.

Example from Sec 9 of the textbook

Example of Branch-and-Bound TSP





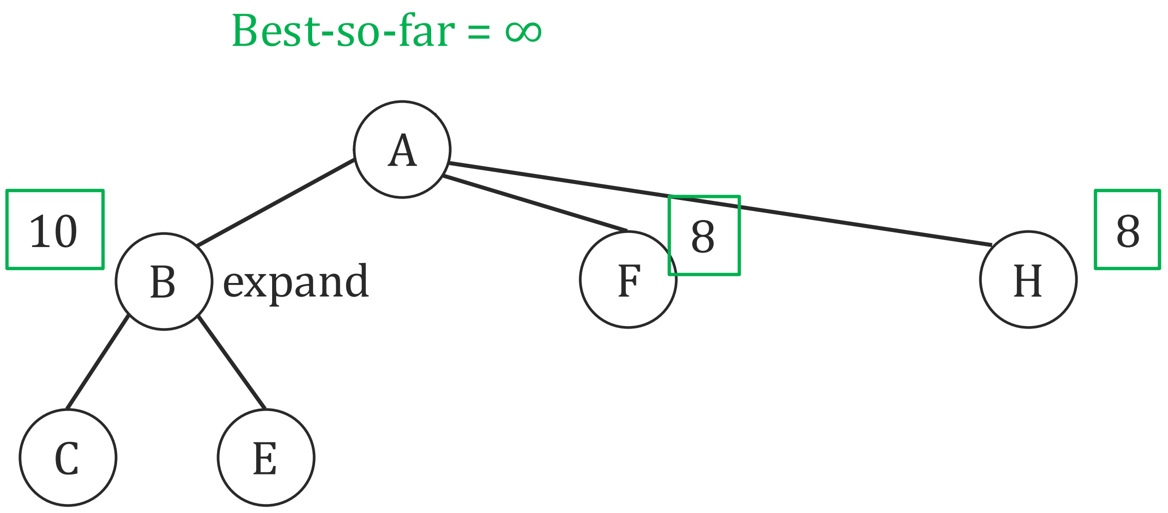
Current partial solution shown in red.

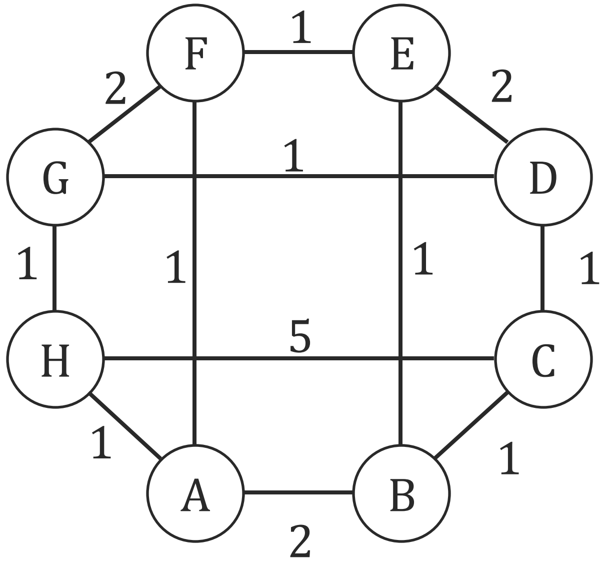
MST of the complement set shown in green.

Lightest edges connecting the blue tour to the complement are shown in orange.

Example from Sec 9 of the textbook

Example of Branch-and-Bound TSP





Current partial solution shown in red.

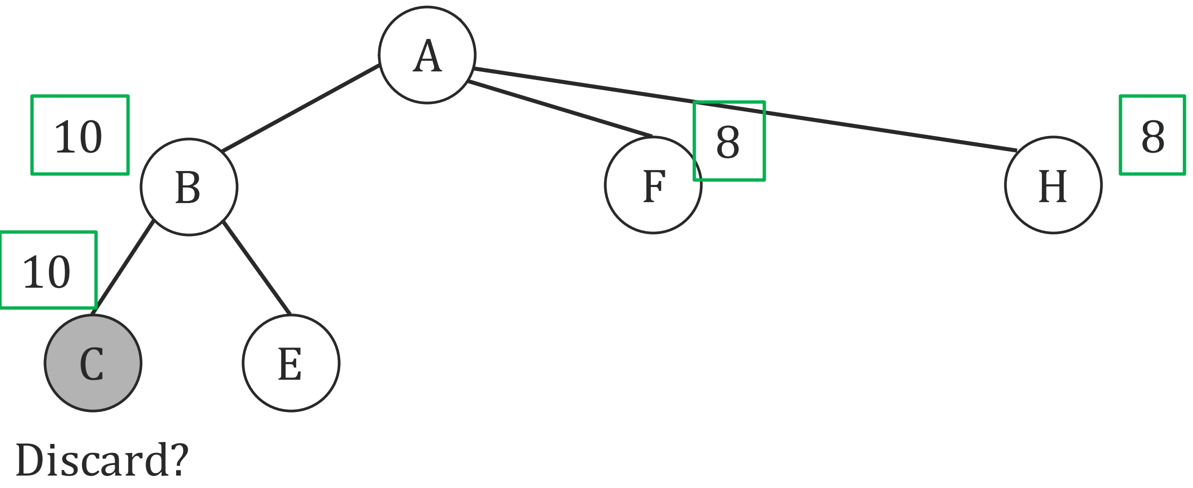
MST of the complement set shown in green.

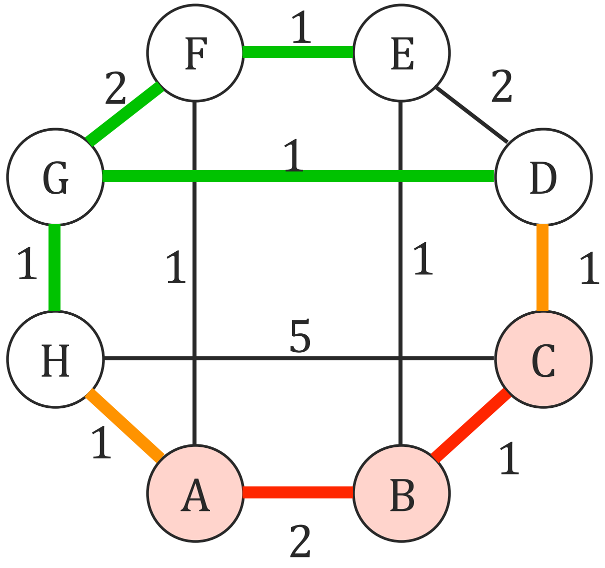
Lightest edges connecting the blue tour to the complement are shown in orange.

Example from Sec 9 of the textbook

Example of Branch-and-Bound TSP

Best-so-far = ∞





Current partial solution shown in red.

MST of the complement set shown in green.

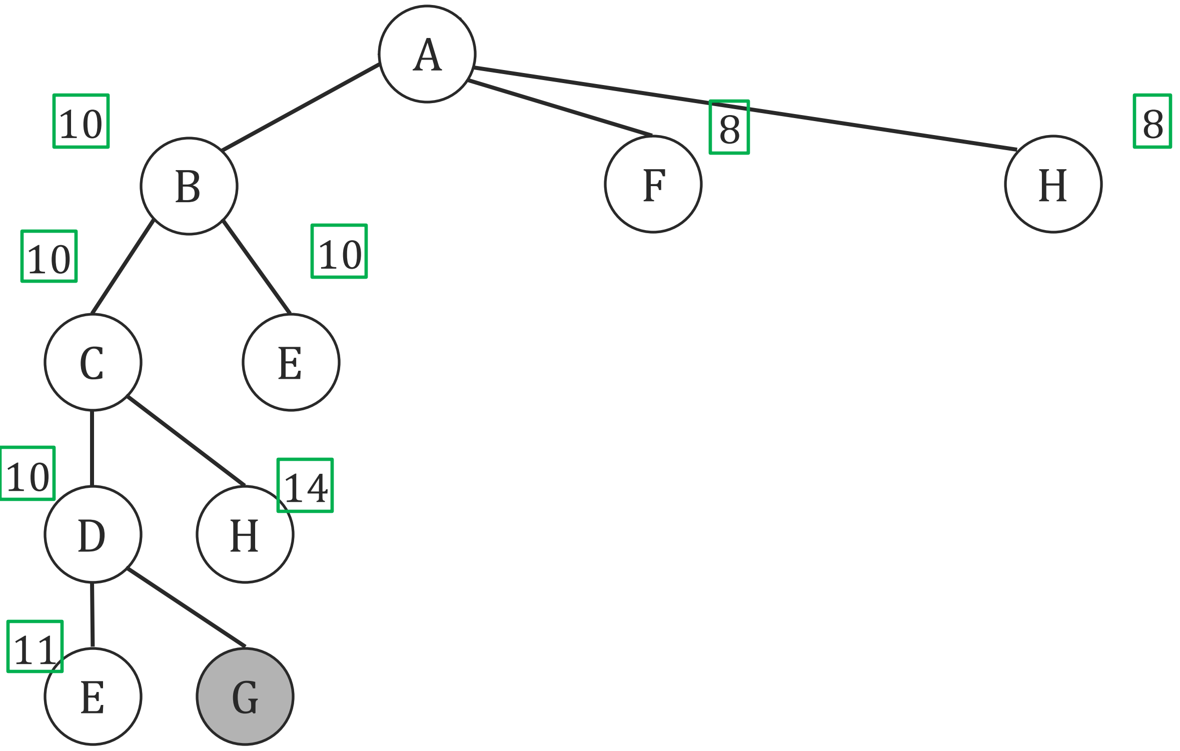
Lightest edges connecting the blue tour to the complement are shown in orange.

Example from Sec 9 of the textbook

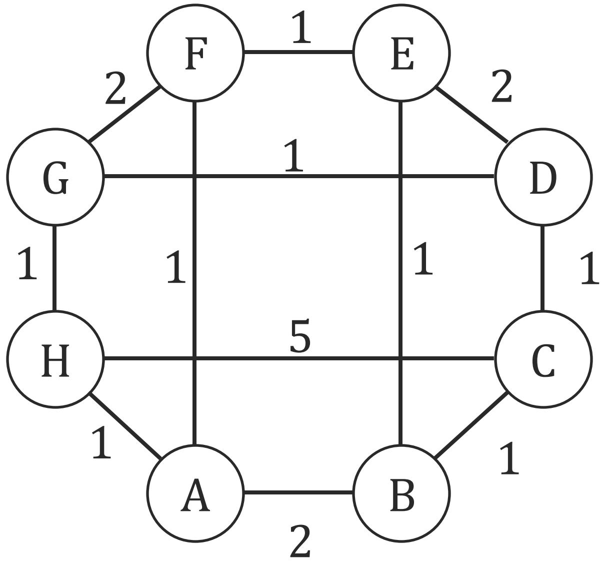
Skipping forward a few steps

Example of Branch-and-Bound TSP

Best-so-far = ∞



Discard?



Current partial solution shown in red.

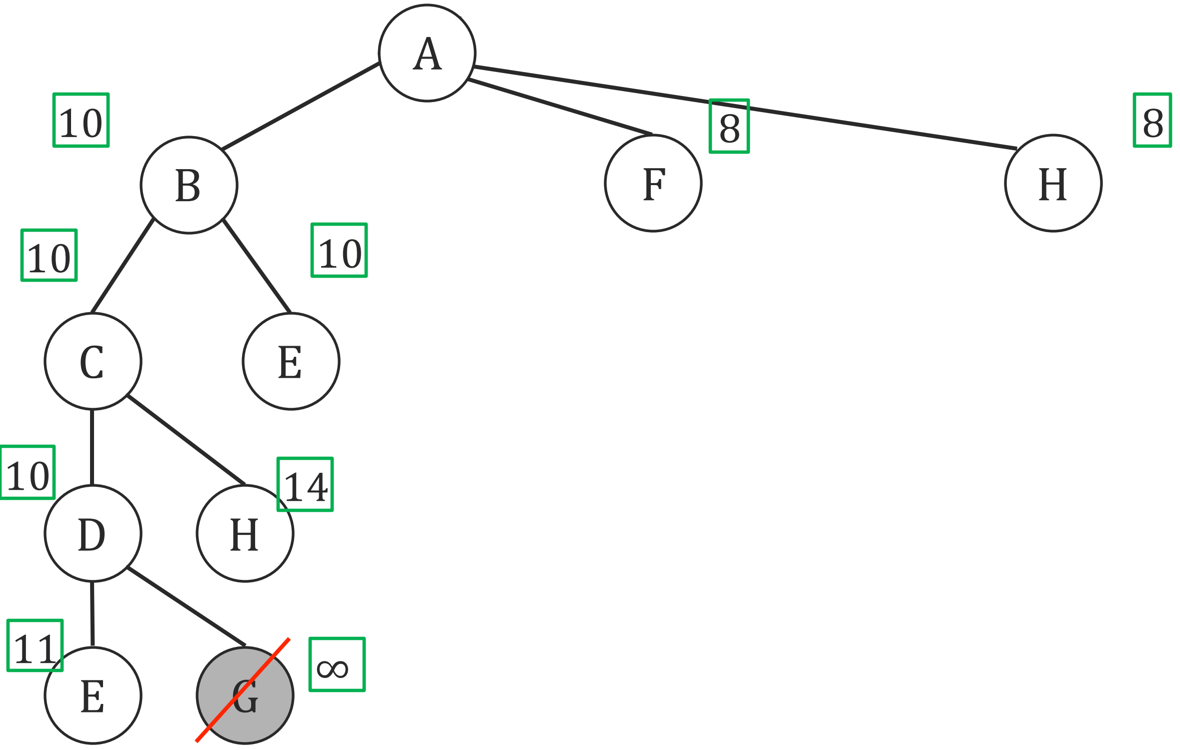
MST of the complement set shown in green.

Lightest edges connecting the blue tour to the complement are shown in orange.

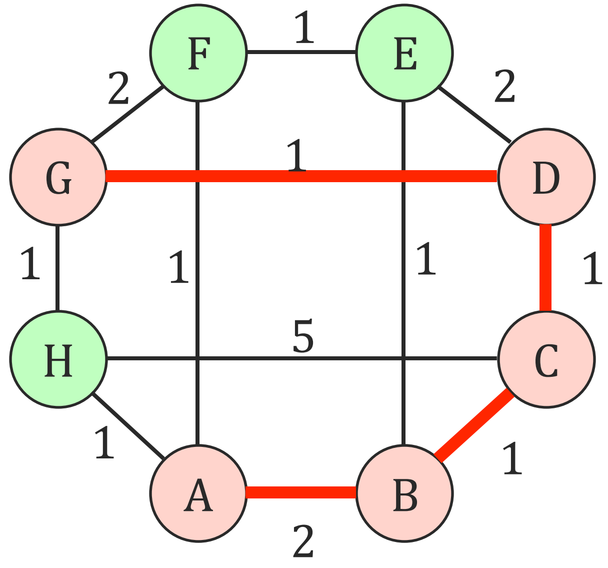
Example from Sec 9 of the textbook

Example of Branch-and-Bound TSP

Best-so-far = ∞



Discard! Never expand



The complement set is not connected!
MST has ∞ weight.

Current partial solution shown in red.

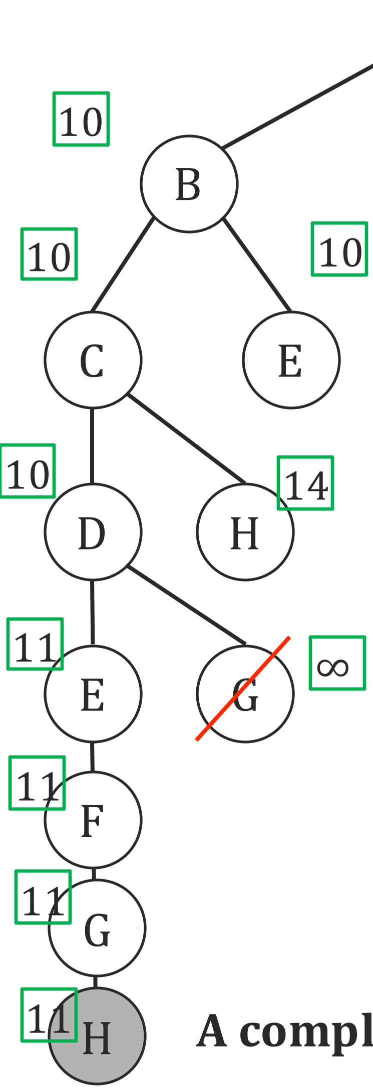
MST of the complement set shown in green.
Lightest edges connecting the blue tour to the complement are shown in orange.

Example from Sec 9 of the textbook

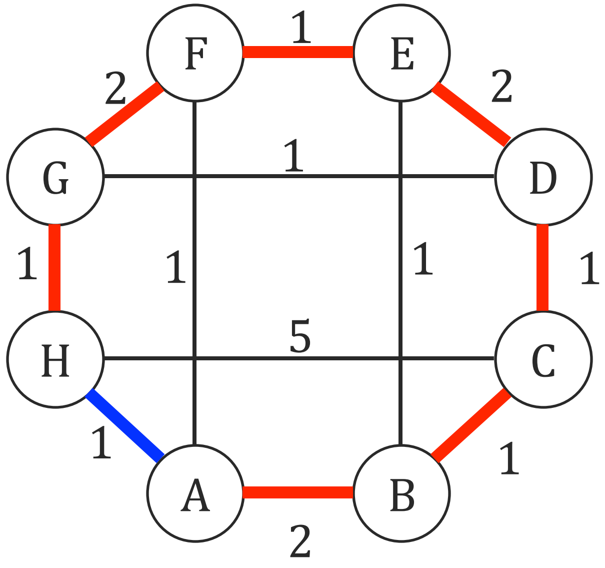
Skipping forward a few steps

Example of Branch-and-Bound TSP

Best-so-far = ~~∞~~ , **11**



A complete solution! With cost 11.



Example from Sec 9 of the textbook

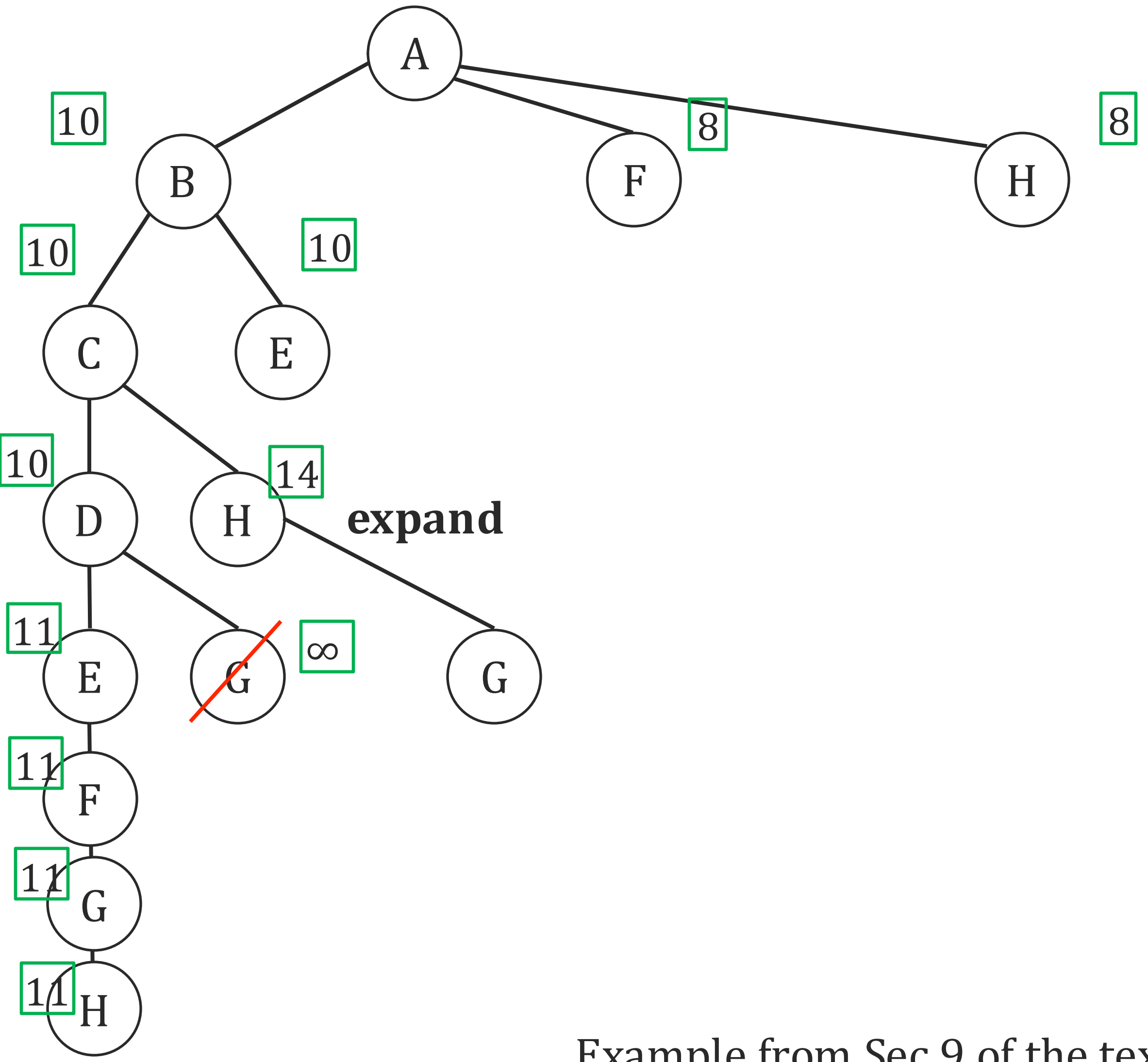
Current partial solution shown in red.

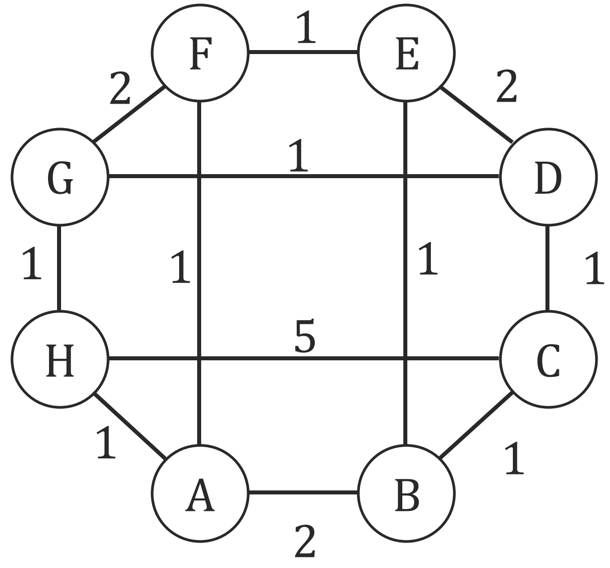
MST of the complement set shown in green.

Lightest edges connecting the blue tour to the complement are shown in orange.

Example of Branch-and-Bound TSP

Best-so-far = ~~∞~~ **, 11**





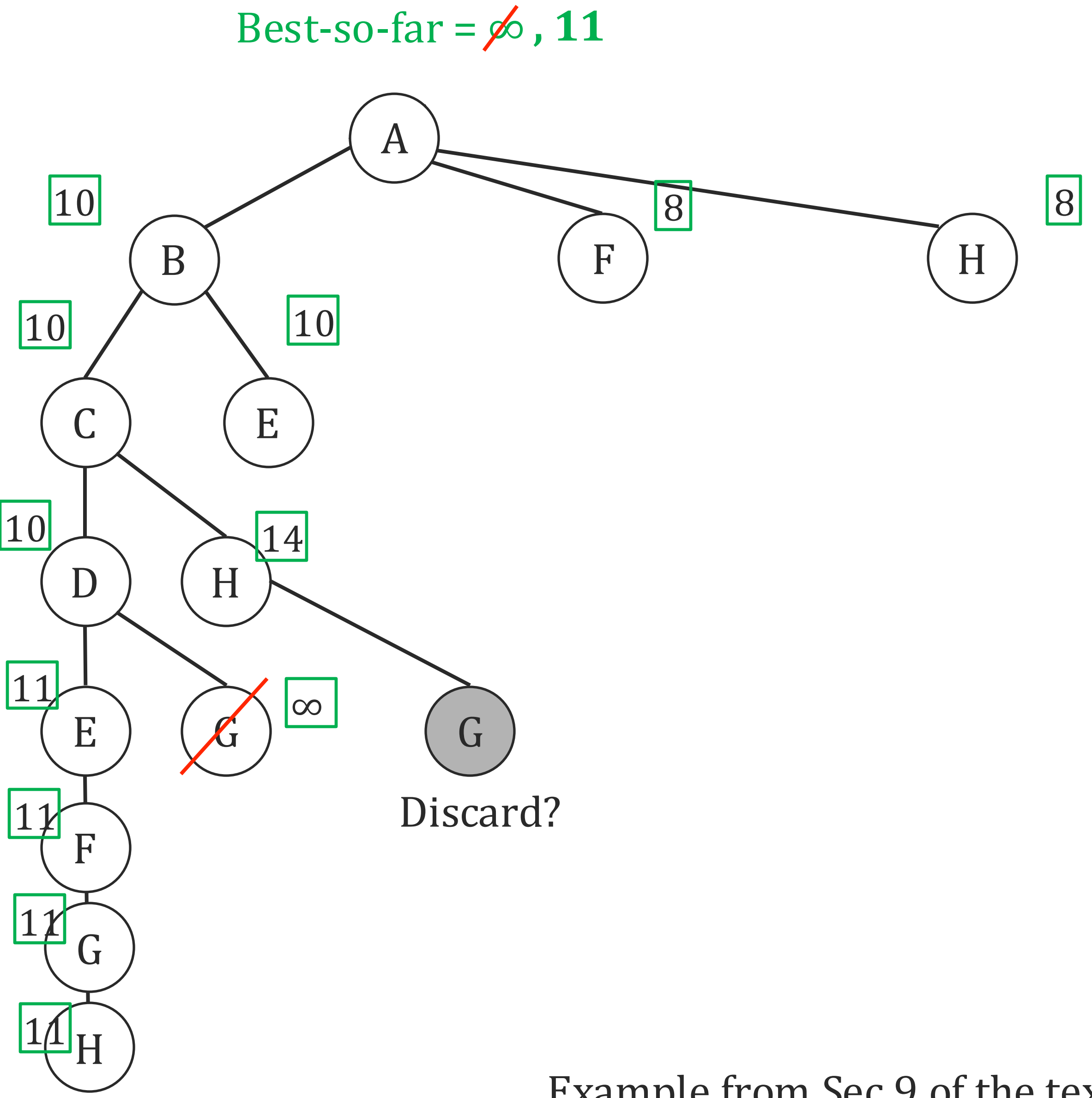
Example from Sec 9 of the textbook

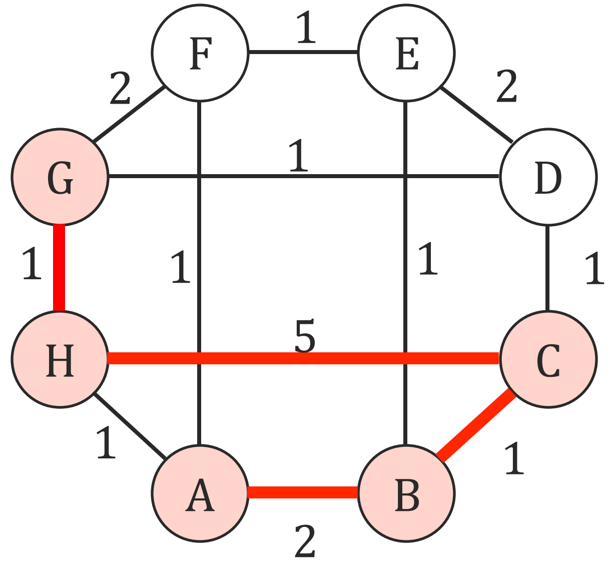
Current partial solution shown in red.

MST of the complement set shown in green.

Lightest edges connecting the blue tour to the complement are shown in orange.

Example of Branch-and-Bound TSP



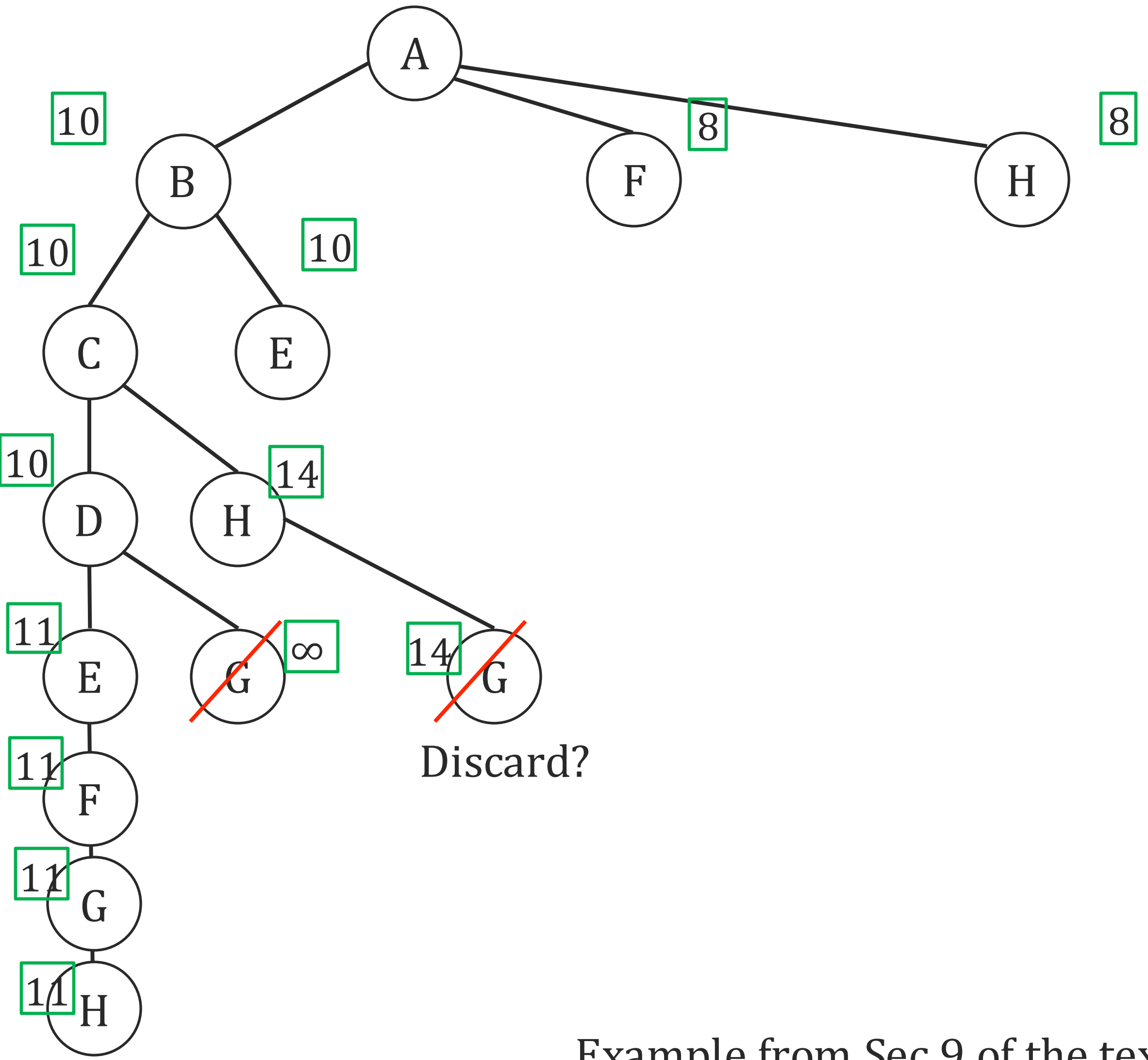


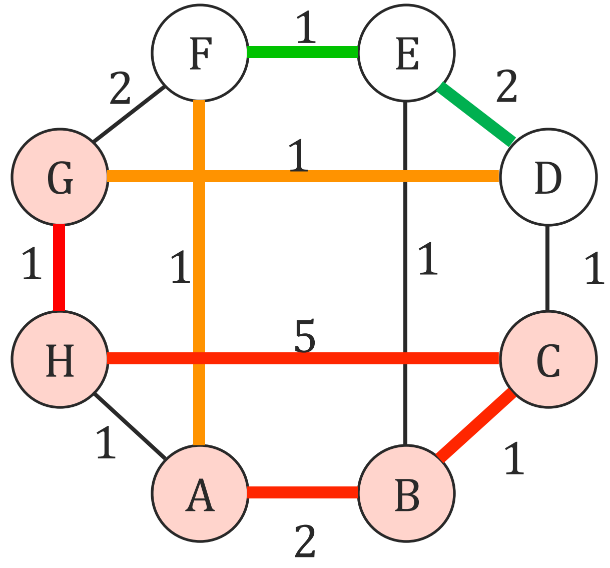
Example from Sec 9 of the textbook

Current partial solution shown in red.
MST of the complement set shown in green.
Lightest edges connecting the blue tour to the complement are shown in orange.

Example of Branch-and-Bound TSP

Best-so-far = ~~∞~~ , **11**





Lowerbound =14 > best-so-far

Example from Sec 9 of the textbook

Current partial solution shown in red.
MST of the complement set shown in green.
Lightest edges connecting the blue tour to the complement are shown in orange.

See textbook for the complete run of the algorithm

Randomized Algorithms

Deterministic Versus Randomized Algorithms

So far, almost all algorithms we've discussed in this class have been deterministic algorithms.

Deterministic algorithms:
→ Take input
→ Do read/write computation to memory
→ Write the output

Randomized Algorithms:
→ Everything a deterministic algorithm does
→ And an infinite sequence of random coin flips

Talking about Randomized Algorithms

$$ALG(\underbrace{x}_{\text{Deterministic input } x},\ \underbrace{r_1,\ r_2}_{\substack{\text{Random coin flips } r_1, r_2 \text{ ....} \\ \text{Or uniformly random number in } [a,b]}} \ldots)$$

The <u>output</u> and <u>computation path</u> of a randomized algorithm are **random variables**

Statements we'd like to make about randomized algorithms

→ **Accuracy/correctness:** for all inputs x, there is a reasonable $c > 0$

$$\Pr[ALG(x, r_1, r_2 \ldots) \text{is correct}] \geq c$$

→ **Runtime:** for all inputs x, there is a reasonable C

$$\mathrm{E}[\text{runtime of } ALG(x, r_1, r_2 \ldots)] \leq C \quad \text{or} \quad \mathrm{Var}[\text{runtime of } ALG(x, r_1, r_2 \ldots)] \leq C$$

c and C could be a function of the input size.

Two Types of Randomized Algorithms

Las-Vegas Algorithms:

- They always output the correct answer (output is deterministic).
- Their runtime is random variable. We usually talk about E[$runtime$].
- E.g. **QuickSort, QuickSelect**.

Lecture 4

Monte Carlo Algorithms:

- They could be wrong (output is randomized) and we talk about Pr[correctness].
- Their runtime is bounded deterministically.
- E.g. Randomized **Min Cut** algorithm, randomized **Primality testing.**

This lecture!

Probability of Correctness

We said that the Monte Carlo Algorithm can be incorrect (or suboptimal) occasionally. There are two types of error tolerance that are acceptable for Monte Carlo algs.

One-sided error:

- If the answer is "Yes", then the ALG says "Yes" with probability 1.
- If the answer is "No", then ALG says "No" with probability $p > 0$.

Two-sided error:

- ALG is correct with probability $\frac{1}{2} + \epsilon$.

Both can be boosted to give correctness with probability 0.99!

Boosting Correctness via Repeated Trials

One-sided error:

- If the answer is “Yes”, then the ALG says “Yes” with probability 1.
- If the answer is “No”, then ALG says “No” with probability $p > 0$

```
For t = $1, \dots, \frac{10}{p}$
    If ALG=”No”, return No.    // Using fresh randomness
return “Yes”
```

What’s the probability of error?

Boosting Correctness via Repeated Trials

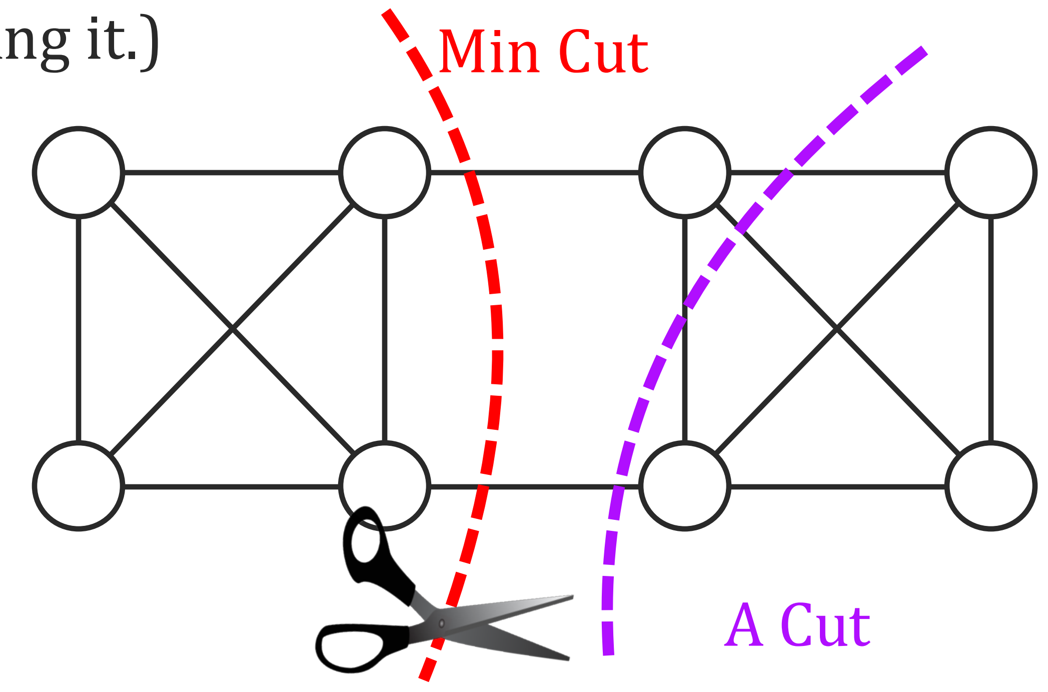
Two-sided error: ALG is correct with probability $\frac{1}{2} + \epsilon$.

```
For t = 1, ..., $\Theta\left(\frac{1}{\epsilon^2}\right)$
    Run ALG        // Using fresh randomness
return Majority vote of the runs.
```

The probability of correctness is also 0.999.

Minimum Cut Problem (Recall)

Input: Given an undirected graph $G = (V, E)$
Output: Return the minimum cut (i..e, a partition of vertices to two sets, with minimum number of edges crossing it.)



Deterministic Algorithm: We saw Min-cut / Max flow as an LP

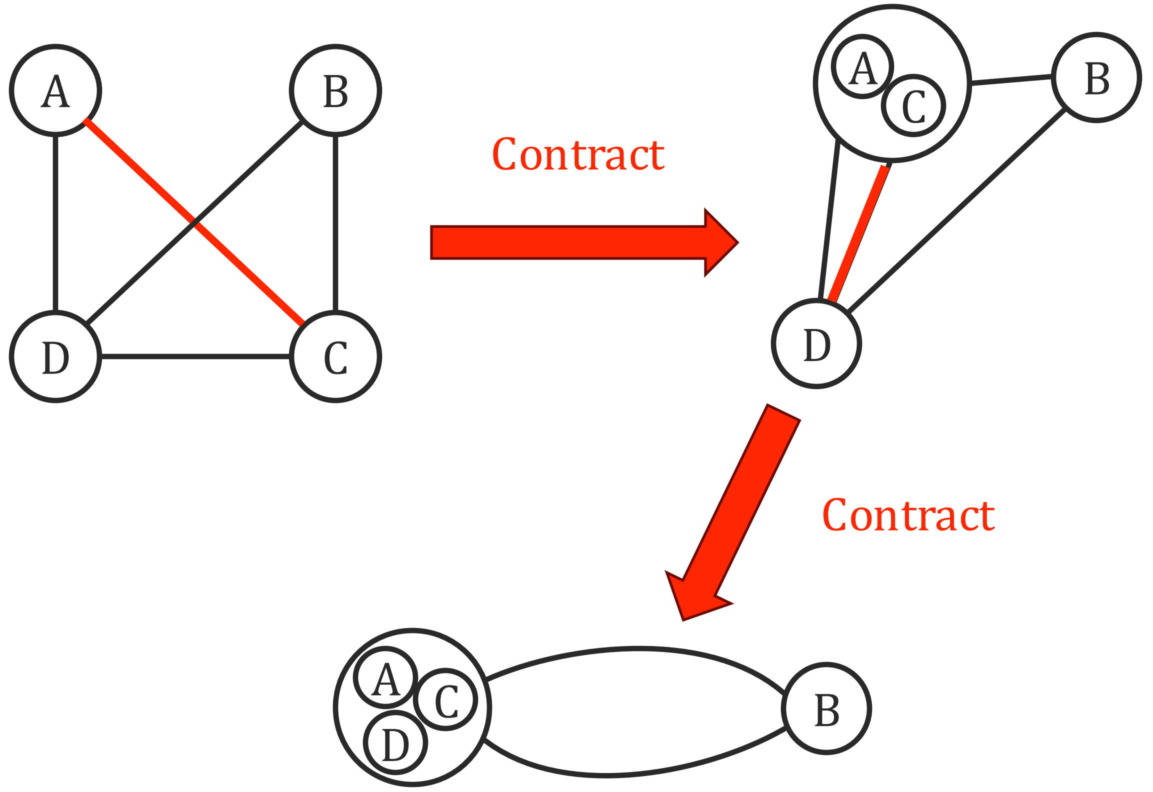
Today: We will see a beautiful randomized Alg for it! We assume unweighted graphs, though it works for weighted ones too.

Karger's Algorithm (randomized contraction)

```
Rand-contraction($G = (V, E)$)
    **Repeat** until 2 vertices are left
        Take a uniformly random $e$
        Contract $e$
    **Return** the cut that corresponds to the 2 vertices
```

Runtime of this alg: $O(m)$

Contraction of edge (u, v): Merge u and v into one giant node. All other edges adjacent to u and v come out the giant node
(keep the parallel edges but delete self loops)



Correctness of Karger’s Algorithm

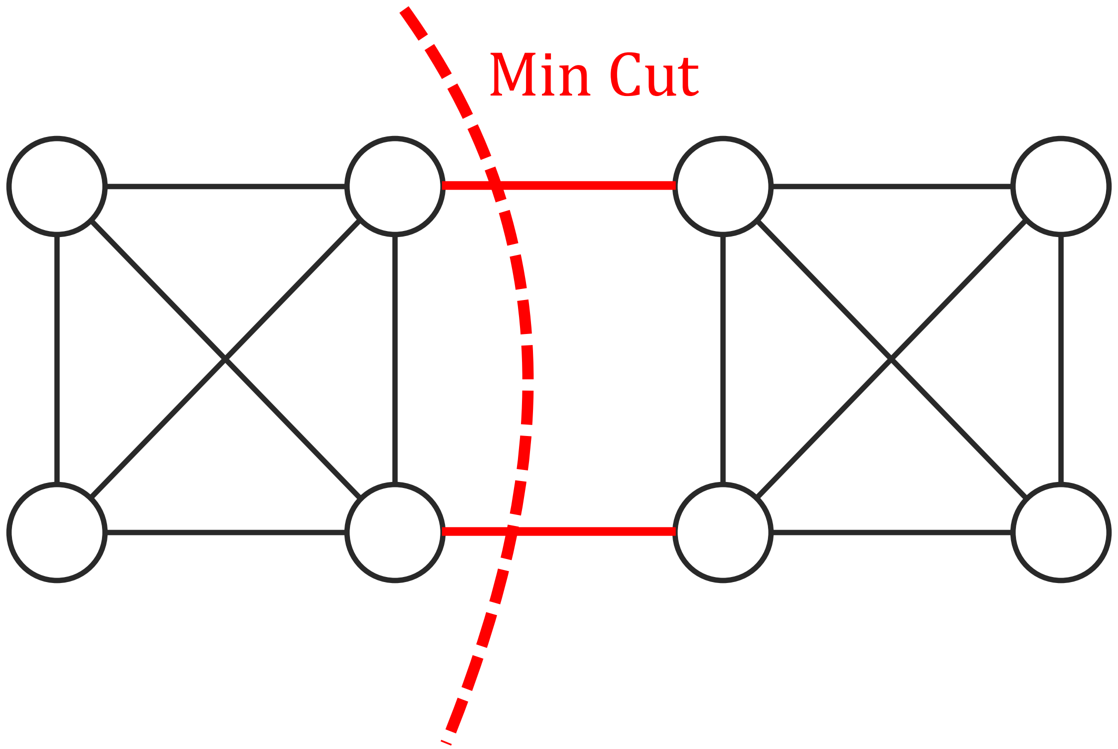
Theorem: The probability that Karger’s algorithm returns a minimum cut in a graph with n vertices is $2/n(n-1)$.

This is great actually!
→ There are $\approx 2^n$ cuts
→ So, this algorithm does significantly better than picking a random cut.

This is like a 1-sided error. Boost the prob of success by repeat this ALG $\Theta(n^2)$ times and returning the smallest cut you see. The success prob becomes 0.999!

High-level Intuition

When does Karger's Algorithm return the wrong cut?
→ It is wrong if and only if it **contracts an edge that crosses the min cut**.



Luckily, there aren't many edges in the minimum cut! So, it is not very likely that we'd pick one of them.

Analysis of Karger's Algorithm

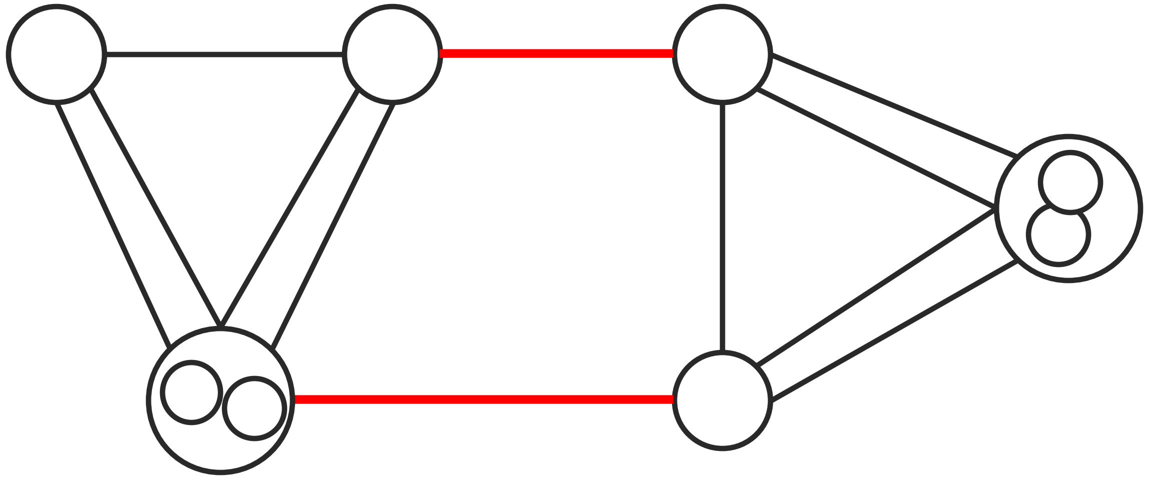
Theorem: The probability that Karger's algorithm returns a minimum cut in a graph with n vertices is $2/n(n-1)$.
Proof: Let C be a minimum cut, and assume that Karger's algorithm contracts edges $e_1, e_2, \ldots, e_{n-2}$.

Let G_i be the "good" event, where the selected e_i doesn't cross the cut.

$$\begin{aligned}\Pr[\text{ALG is correct}] &= \Pr[G_1 \wedge G_2 \wedge \cdots \wedge G_{n-2}]\,. \\ &= \Pr[G_1] \cdot \Pr[G_2 | G_1] \ldots \Pr[G_{n-2} | G_1, G_2, \ldots, G_{n-3}\,]\end{aligned}$$

Analysis of a single step of Karger's Algorithm

We will show that $\Pr[G_i | G_1, G_2, \ldots, G_{i-1}] \geq \frac{n-i-1}{n-i+i}$



2 non-cut edges have been contracted.

Analysis of Karger's Algorithm

Theorem: The probability that Karger's algorithm returns a minimum cut in a graph with n vertices is $2/n(n-1)$.

Proof: Let C be a minimum cut, and assume that Karger's algorithm contracts edges $e_1, e_2, \dots, e_{n-2}$.

Let G_i be the "good" event, where the selected e_i doesn't cross the cut.

$$\begin{aligned} \Pr[\text{ALG is correct}] &= \Pr[G_1 \wedge G_2 \wedge \cdots \wedge G_{n-2}]\,. \\ &= \Pr[G_1] \cdot \Pr[G_2 | G_1] \dots \Pr[G_{n-2} | G_1, G_2, \dots, G_{n-3}\,] \end{aligned}$$

From last slide

$$\Pr[G_i | G_1, G_2, \dots, G_{i-1}\,] \geq \frac{n-i-1}{n-i+i}$$

Wrap up Karger's Algorithm

Runtime:

- One round of Karger's Alg can be done in $O(m)$ runtime
- It has success probability of $\Omega(1/n^2)$, so we need to repeating it $O(n^2)$ rounds to boost the correctness probability to 0.999
- Total runtime: $O(m\, n^2)$
  → Actually, this can be improved to $\approx O(n^2)$ since not all computation needs to be repeated. (not in scope for this class)
- The linear programming solution, while deterministic, can be slower.

Prime Numbers

Prime numbers: 2, 3, 5, 7, 11, 13, …

Prime numbers are super useful!
→e.g., In cryptography you want to produce large (128bits, 256bit, ….) primes

There are lots of prime numbers!
→ If you pick 100 random 128-bit numbers, very likely that at least 1 of them is prime.

To generate primes effectively, it's enough to be able to test whether a number is prime.

Primality Testing: given a number, determine if it is a prime number.

Primality Testing

Primality Testing: Given a number N, is it a prime number?

A straight-forward algorithms:

→ For all $z = 1, \dots, \sqrt{N}$, see if z divides N?

→ Runtime is poly(N) ….

→ But, this is not pseudo-polynomial time algorithm, not polynomial time!

→ For it to be polynomial time, it needs to be poly(#bits $of\ N$) or polylog(N).

Fermat's Little Theorem

All prime numbers satisfy a neat little test!

Fermat's Little Theorem

If p is a prime, then for all $x = 1, \dots, p-1$ we have that $x^{p-1} \equiv 1 \pmod{p}$

This suggests that we might be able to deduce whether N is a prime by looking at whether $x^{p-1} \not\equiv 1 \ (\mathrm{mod\ N})$ for some choice of x. Let's choose x at random!

Fermat's Primality Test

Choose x uniformly at random from all $x = 1, \dots, N-1$.
Return "prime" if $x^{N-1} \equiv 1 \pmod{N}$, otherwise return "composite"

What if N is composite?

Let's say input was composite number $N = 9$. All arithmetic here is mod 9.

$$\left.\begin{array}{l}
1^8 \equiv 1 \\
2^8 \equiv 4 \not\equiv 1 \\
3^8 \equiv 0 \not\equiv 1 \\
4^8 \equiv 7 \not\equiv 1 \\
5^8 \equiv 7 \not\equiv 1 \\
6^8 \equiv 0 \not\equiv 1 \\
7^8 \equiv 4 \not\equiv 1 \\
8^8 \equiv 1
\end{array}\right\}$$

Out of 8 choices for a random $x \in \{1, \ldots, 8\}$, only 2 of them would lead Fermat's test to erroneously state that 9 is a prime!

Fermat's test would have been correct with prob 0.75!

Can we say that Fermat's test succeeds with a reasonable probability, for all N?

The Exception: Carmichael Numbers

Unfortunately, that it not the case.

There are composite numbers N for which $x^{N-1} \equiv 1 \pmod{N}$ for many xs.
→ For these inputs, the probability of success is too small.

Carmichael numbers:
Composite number N for which $x^{N-1} \equiv 1 \pmod{N}$ for all x that's coprime with N.

There are infinitely many of these! But they are very rare and spread apart. Smallest Carmichael number is $561 = 3 \times 11 \times 17$.

Limited Primality-Testing non-Carmichael

In this lecture, we show that Fermat's test is a good randomized primality, as long as the input is not a Carmichael number.

Theorem: Assume that N is not a Carmichael number. Then the Fermat's test satisfies the following requirements.
1. If N is prime, it states "prime" with probability 1.
2. If N is composite (but not Carmichael), it states "composite" with prob $> 1/2$.

Remark 1: Can boost the prob. to 0.99 by repeating the a few times (e.g. >6 times).

Remark 2: There is an algorithm based on the same idea as Fermat's test that work also for all integers! We won't cover it in class though.

Correctness of the Primality Test

Theorem: Assume that N is a composite, but not Carmichael number. Then with prob $> 1/2$ Fermat's outputs "composite". i.e.

$$x^{N-1} \not\equiv 1\ (mod\ N) \text{ for at least half of } x = 1, \dots, N-1$$

---

1. N is Not Carmichael => there is co-prime a such that $a^{N-1} \not\equiv 1\ (mod\ N)$

What's nice about co-primes? They have a unique inverse a^{-1}, such that $a \times a^{-1} \equiv 1\ (mod\ N)$

2. Take any **bad** b_i (for which $b_i^{N-1} \equiv 1\ (mod\ N)$), then b_i maps to a **good** g_i=$b_i a$:

$$g_i^N = (b_i a)^{N-1} = b_i^{N-1} a^{N-1} \not\equiv 1\ (mod\ N)$$

3. Also, the mapping is one-to-one: If $b_i \neq b_j$, we must have $g_i \neq g_j$:

$$g_i = g_j \Rightarrow g_i a^{-1} \equiv g_j a^{-1} \Rightarrow b_i \equiv b_j$$

Correctness of the Primality Test (cont.)

We proved that for every **bad** b_i (for which $b_i^{N-1} \equiv 1\ (mod\ N)$) there is a distinct **good** g_i=$b_i a$ (for which $g_i^{N-1} \not\equiv 1\ (mod\ N)$)

Primality Testing through the ages

200 BC: Eratosthenes (Greek polymath) described the *prime number sieve* for finding all the prime numbers up to a certain value.

1976: Miller and and Rabin came up with a randomized algorithm (similar to what we discussed but one more idea to deal with Carmichael numbers)

1977 …. 2002: Other randomized algorithms

2002: Agrawal, Kayal, and Saxena gave a polynomial time *deterministic* algorithm for primality testing (de-randomizing one of their earlier algorithms from 1999)

Complexity Classes and Wrapup

There are problems for which we know polynomial time randomized algorithms, but no deterministic polynomial time algorithms!
→ E.g., Polynomial testing

Are randomized algorithms actually more powerful than deterministic algorithms?
→ Major complexity theory open problem. We don't know yet!

Next time

Online algorithms: natural place where you want randomness.