COMP 285

Analysis of Algorithms

Welcome to COMP 285

Lecture 25: Approximation Algorithms

Lecturer: Chris Lucas (cflucas@ncat.edu)

HW8!

Due 12/01 @ 11:59PM ET

HW8!

Latest due date 12/04 @ 11:59PM ET

Final Exam

12/06 from 2:00pm-4:00pm

Final Exam Prep

Practice Final on course website/Blackboard

Final Exam Prep

11/29 and 12/01 Review Lectures

Final Exam

How would you recommend a student should prepare for the final exam?

How to prepare for the final exam?

- Reviewing written+coding homeworks
  - You will be asked to write code!
- Reviewing lectures slides/recordings, “more resources” on course website.
- Reviewing each quiz/walkthrough video
- Reviewing the practice midterm/real midterm
- Final week of lectures!
- Practice final!

Quiz!

[www.comp285-fall22.ml]() or Blackboard

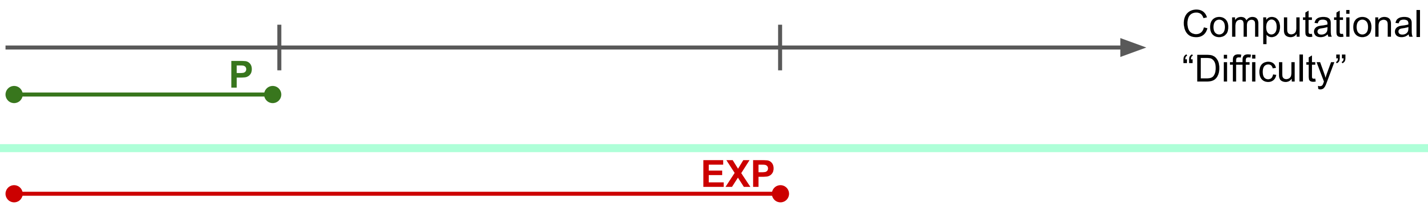
Recall where we ended last lecture…

Motivation

- P vs NP may be the most famous unsolved question in Computer Science $$$
- It gives us a way to reason about whether a problem is tractable or not.
  - Classifies problems based on how difficult they are to solve
  - If you're working on a new problem, don't waste your time trying to come up with a clever polynomial time solution if it's not possible!
- Similar to Big-Oh in that it's a theoretical framework, a tool for reasoning about algorithms, comparing algorithms, etc. P vs. NP is also a theoretical framework, a tool for reasoning about problems, comparing problems, etc.

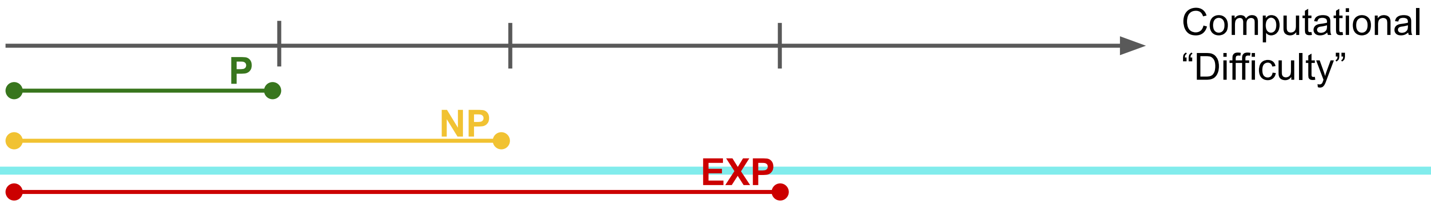
Polynomial Time (P) versus Exponential Time (EXP)

- P: set of decision problems that can be solved in polynomial time
  - $O(n^k)$, e.g. $n \log(n)$, n^2, n^{20}
- EXP: set of decision problems that can be solved in exponential time
  - $O(2^n)$, $O(10^n)$, $O(2^{n^c})$
- We work with **decision problems** (i.e. the answer to these problems is yes or no) but the implications are still often applicable to optimization problems.
- $P \subseteq EXP$: “P is a subset of EXP”
- There are lots of problems in EXP, because it is very slow, and includes problems where the only solution we know is “try everything”.



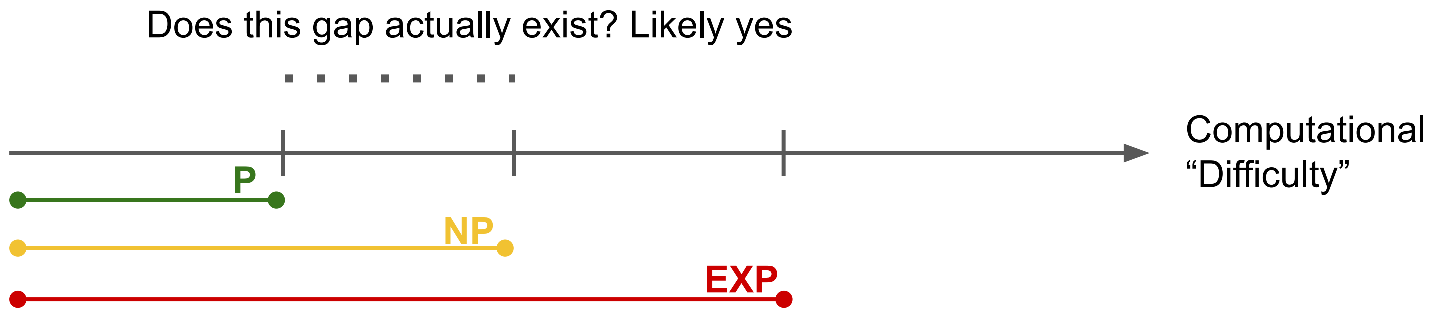
Nondeterministic Polynomial (NP)

- NP: set of all decision problems that can be **verified** in polynomial time.
- P is in NP, as you could just solve the problem in polynomial time and see if the answers are equal to verify.
- Examples
  - Is this array sorted?
  - Is string X a substring of string Y?
  - … all problems in P
  - Is there a subset of elements in this array that add up to k?
  - Given a graph, is there a path of at most length L that visits each node exactly once and returns where you started? (“Traveling Salesman Problem”)



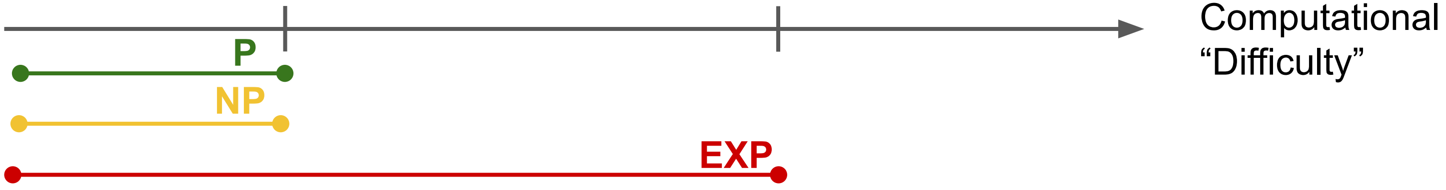
P vs NP

- We know that P ⊆ NP
- But, does P = NP? a $1 million-dollar question (actually)
  - Most likely P ≠ NP, it's just hasn't been proven yet.
- "Creating a nondeterministic computer is impossible"
- "Generating solutions can be harder than checking them"



What if P = NP though?

- Some implications:
  - We could cure a lot more diseases with efficient protein folding simulations.
  - But all passwords / encryption could be cracked.
- Scott Aaronson's philosophical argument: *If P=NP, then the world would be a profoundly different place than we usually assume it to be. There would be no special value in "creative leaps," no fundamental gap between solving a problem and recognizing the solution once it's found. Everyone who could appreciate a symphony would be Mozart; everyone who could follow a step-by-step argument would be Gauss.*



Big Questions!

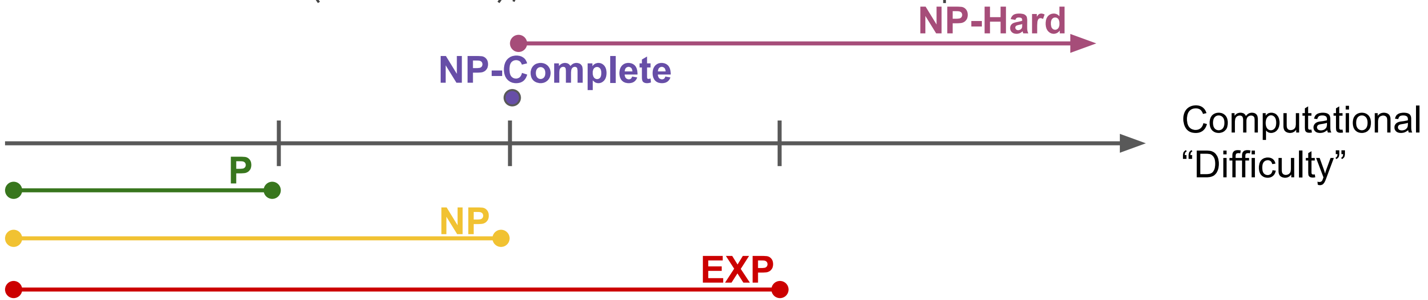
- What is NP complete, NP hard and what are reductions?
- What are approximation algorithms?

Big Questions!

- What is NP complete, NP hard and what are reductions?
- What are approximation algorithms?

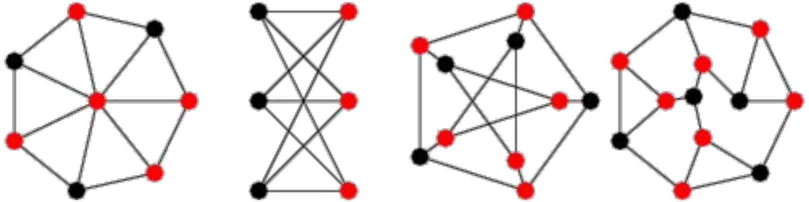
NP-Complete and NP-Hard

- **NP-Hard**: problems at least as hard as the hardest problems in NP.
- **NP-Complete**: problems that are NP-hard, but still in NP, i.e. "the hardest problems in NP".
- Why do we care about NP-Complete? Because if we find a way to solve one NP-Complete problem, we will solve them all.
- Why do we care about NP-Hard? Because if we can show a problem is both NP-Hard and NP (reducible), we know it is NP-Complete.



NP-Complete Problems

- Traveling Salesman Problem
- Generalized Sudoku
- Vertex cover: “Given a graph G, can you find a vertex cover of n nodes?”



- Boolean satisfiability: “Given a boolean expression like the following *(a or !b) and (c or d) or e* are there possible values for a, b, c, d, e that will make the statement true?”

Reductions

- Reductions are converting a problem into another problem.
- We do this all the time to solve problems, e.g. with graphs, we would transform them to be able to use an algorithm we know (like network flow).
- To prove a problem X is NP-Complete, you can:
  - Show it is NP-Hard (usually then reduce to a known NP-Complete problem to it)
  - Show it is NP (e.g. by showing its solution is verifiable in polynomial time)

Reduction Example

- Number Scrabble!
  - Imagine we are playing a game where the numbers are lined up 1 through 9 and we take turns selecting numbers. One of us wins when the numbers sum to 15.

1 2 3 4 5 6 7 8 9

Reduction Example

- Number Scrabble!
  - Imagine we are playing a game where the numbers are lined up 1 through 9 and we take turns selecting numbers. One of us wins when the numbers sum to 15.

2	**7**	**6**
9	**5**	**1**
4	**3**	**8**

- Can we rearrange?

Halting Problem

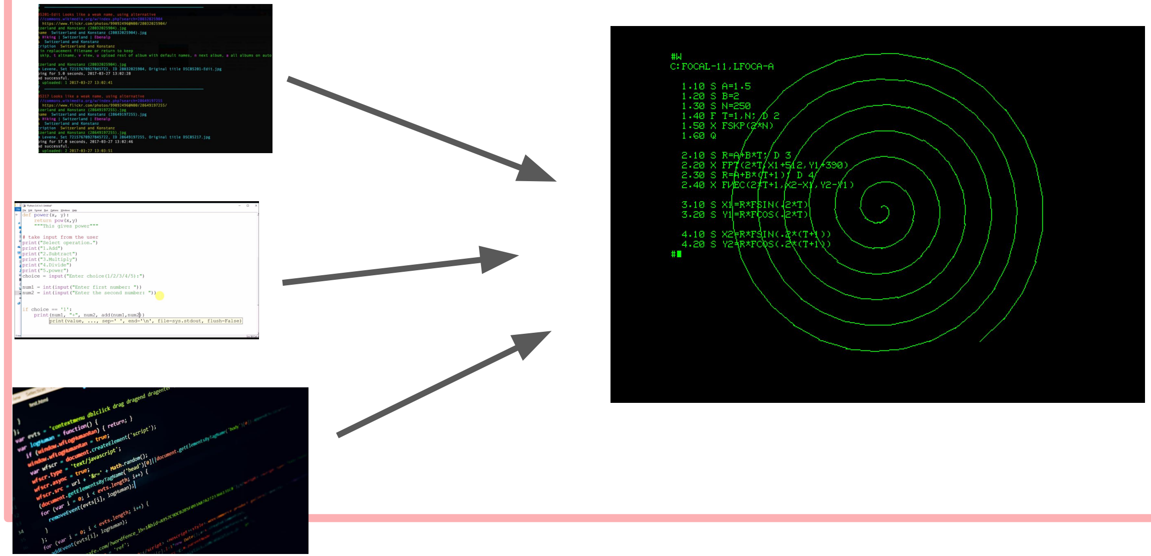
- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)

Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)

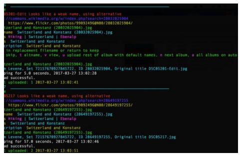
Halting Problem

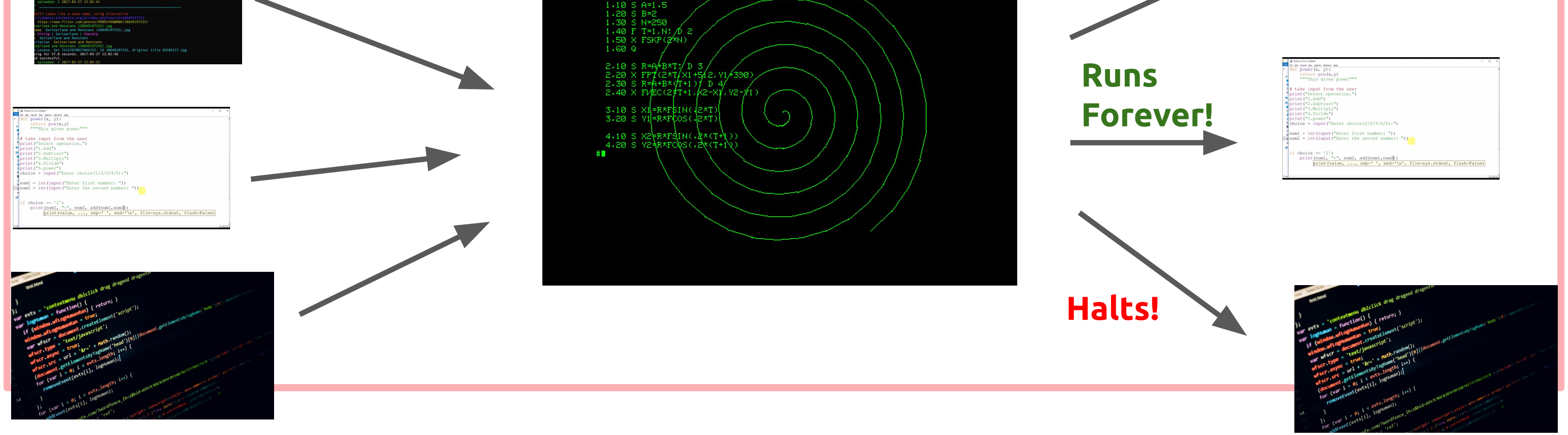
- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)



Halting Problem

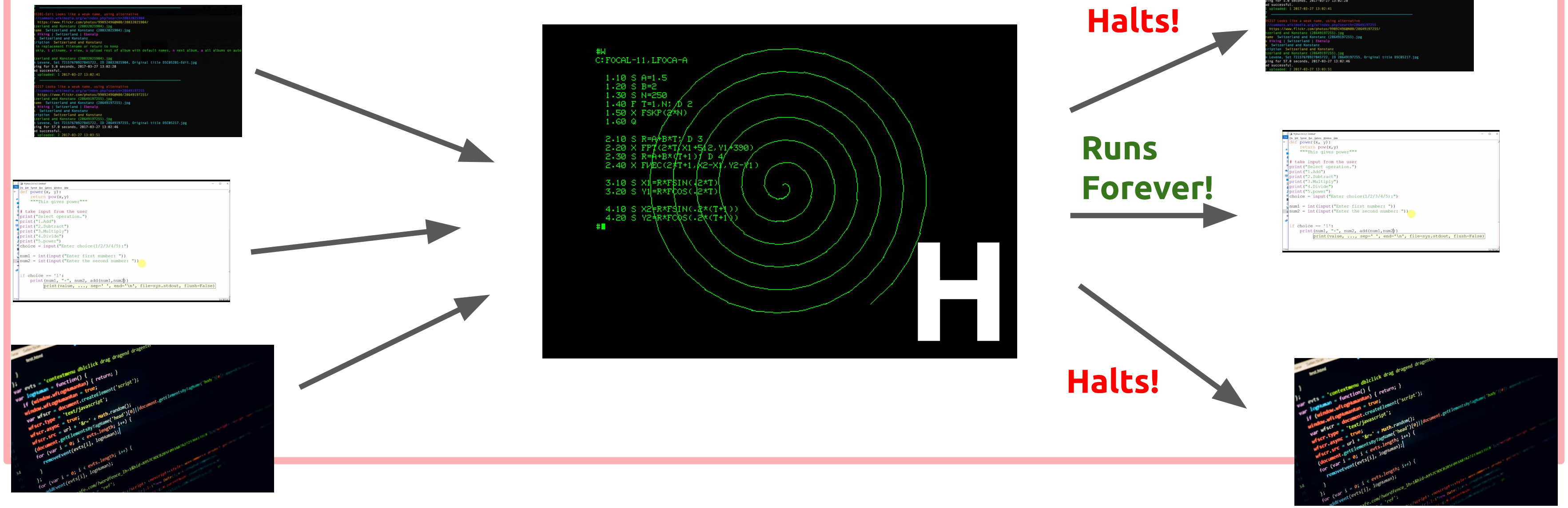
- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)





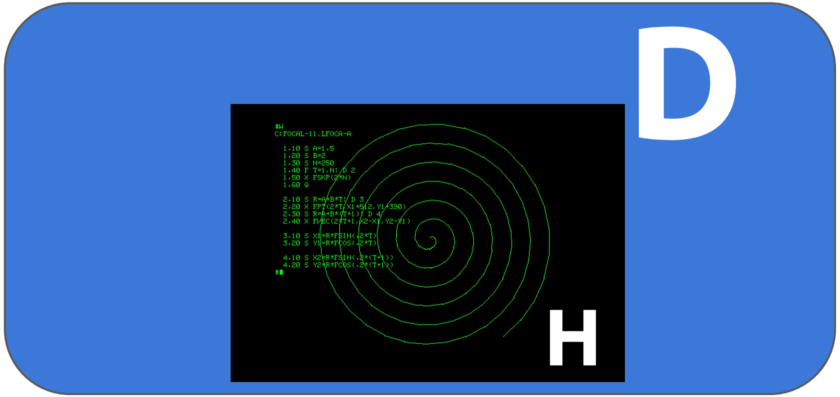
Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)



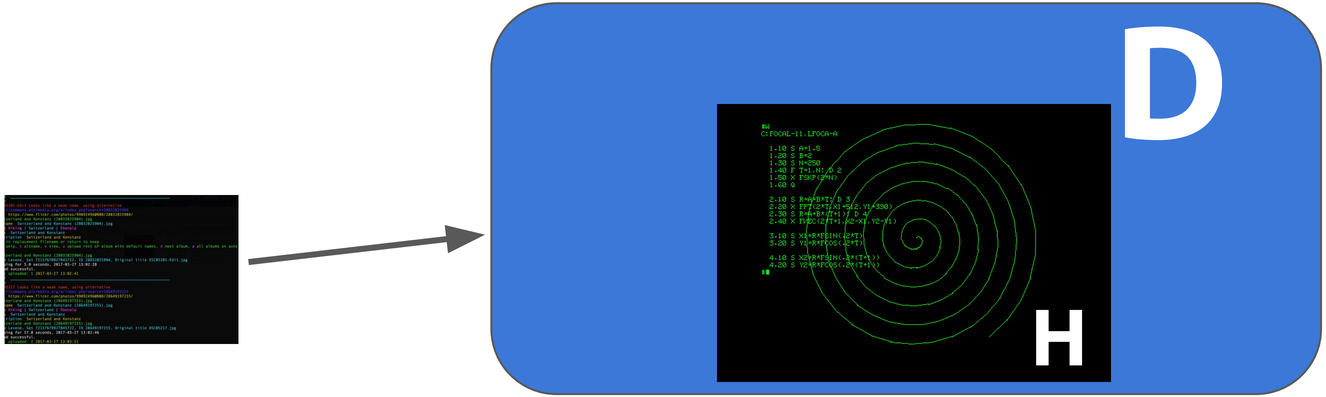
Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)



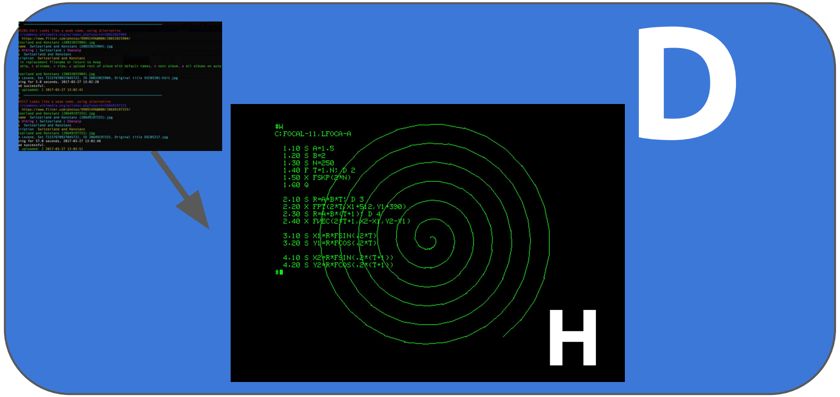
Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)



Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)



Halting Problem

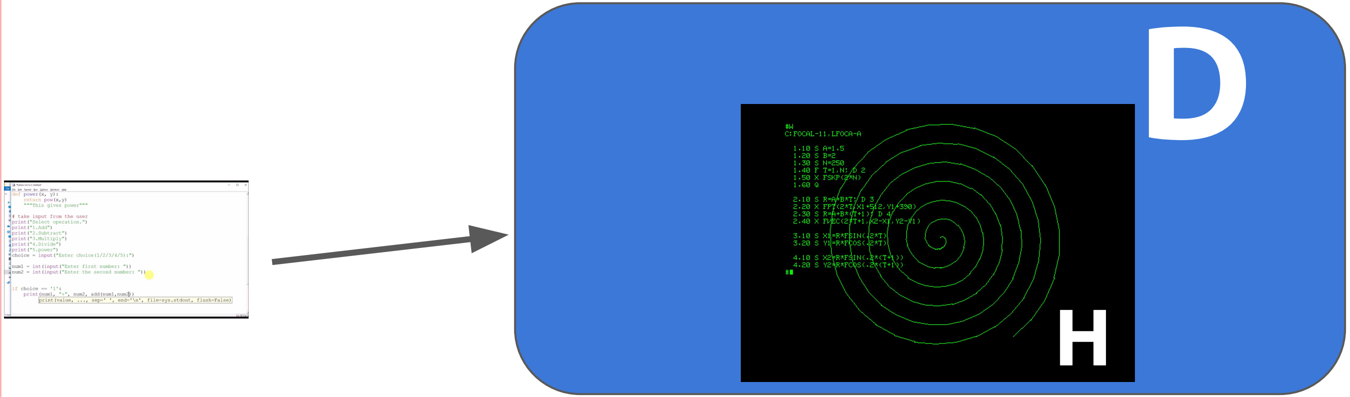
- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)

Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)

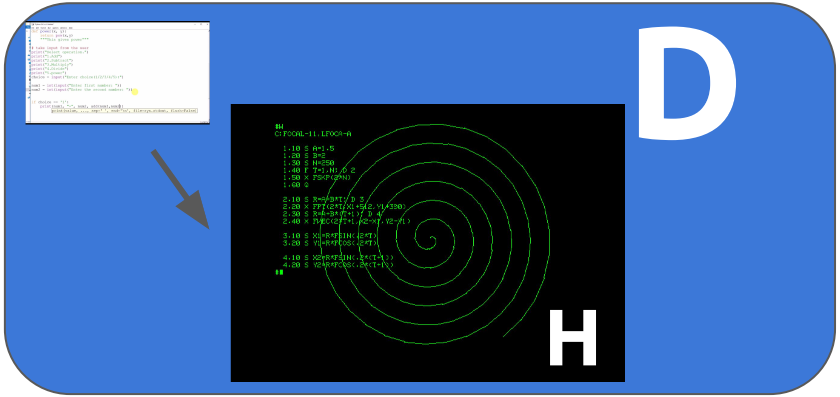
Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)



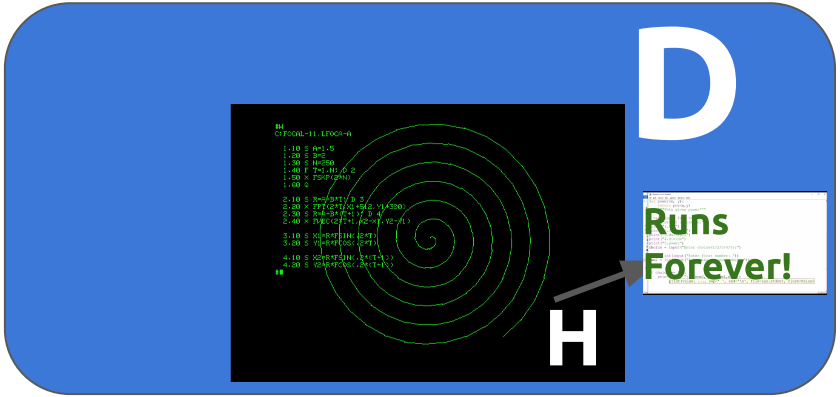
Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)



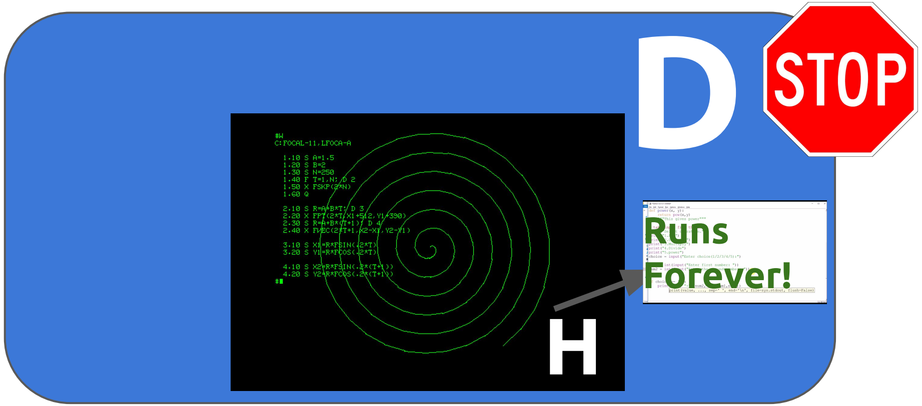
Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)



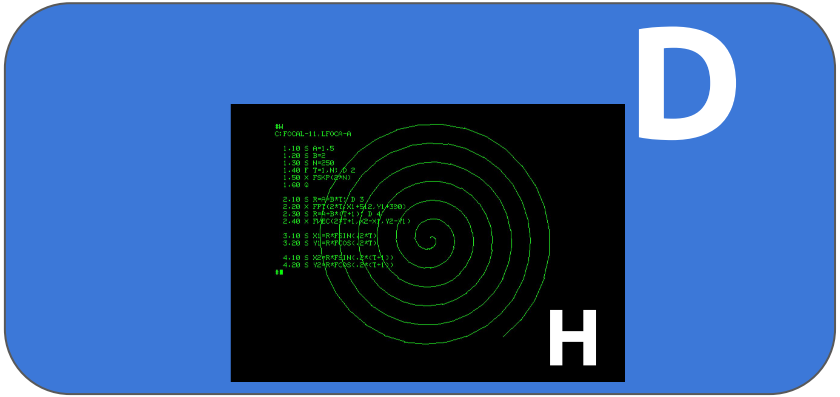
Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)



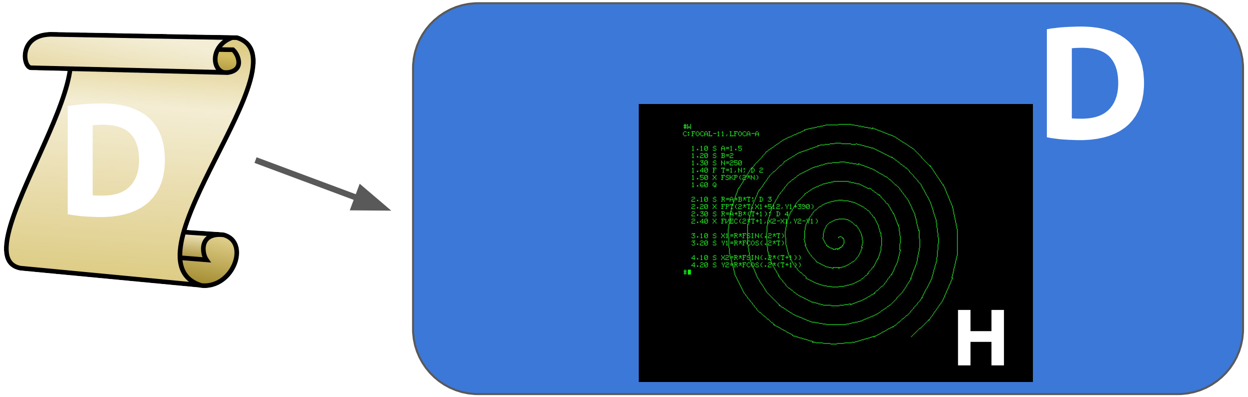
Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)



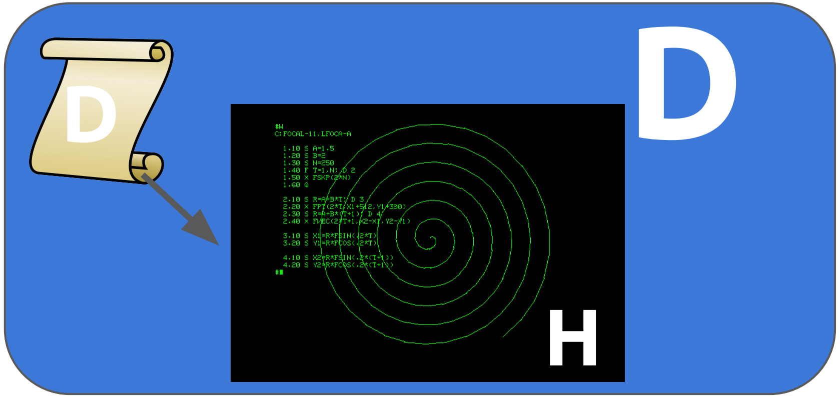
Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)



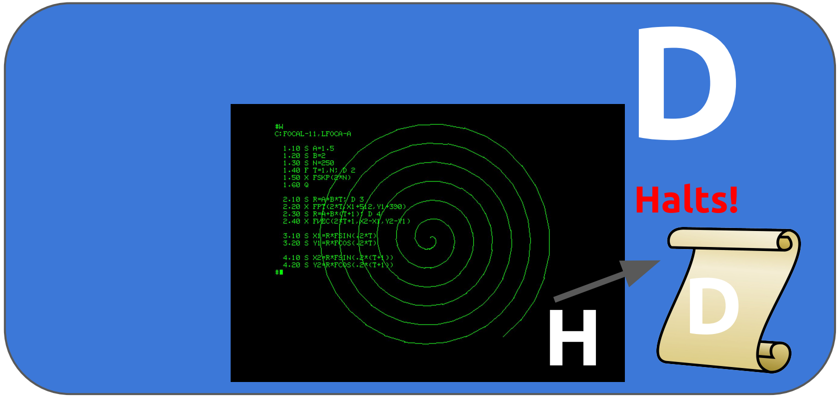
Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)



Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)

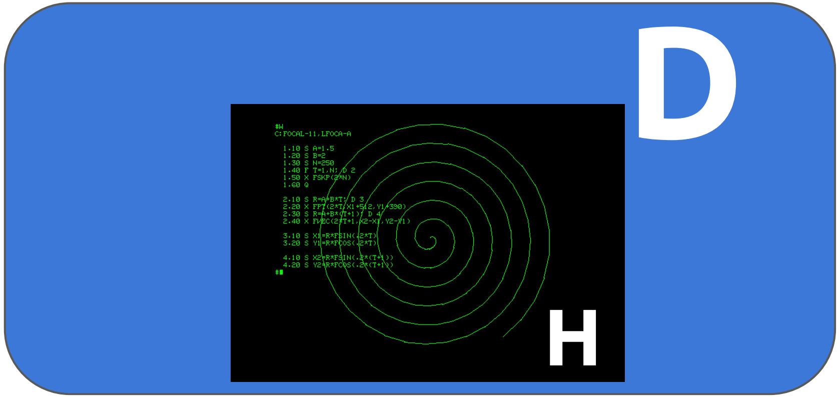


Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)

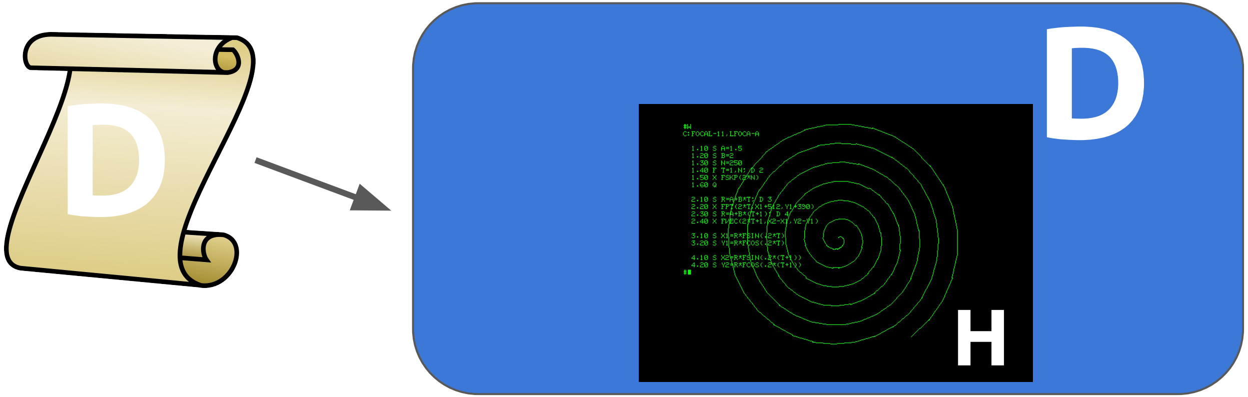
Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)



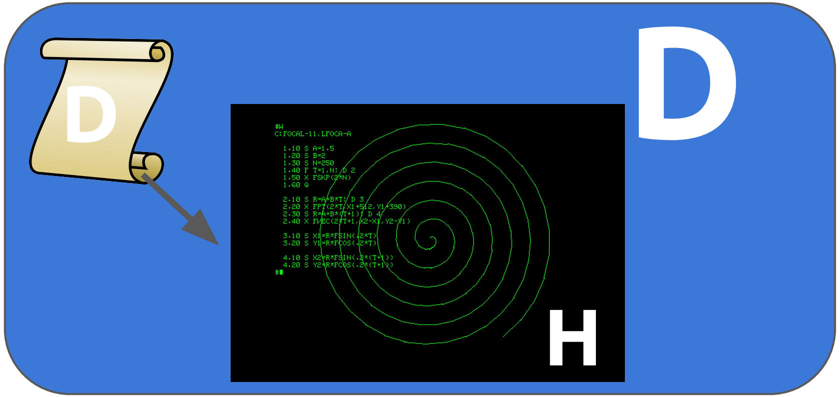
Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)



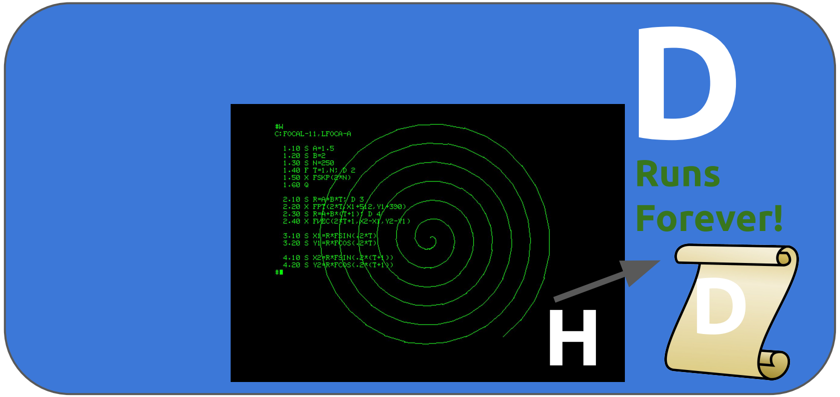
Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)



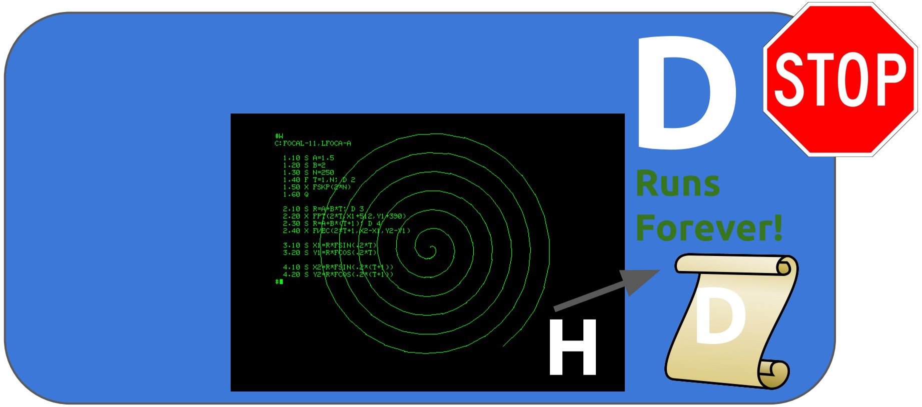
Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)



Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)



Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)

What can we do if our problem isn't in P?

- Pack up and go home. This is not worth pursuing…
- Accept it and move on! Our solution is going to be slow…
- We could constrain our problem further to make it P, to make it work in polynomial time…
- We could accept an *approximate but more time efficient* solution (must be within a reasonable margin)
    - Valuable in the real world!

Big Questions!

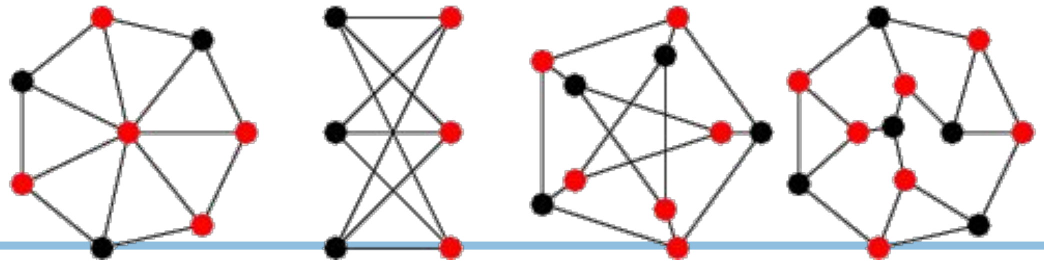
- What is NP complete, NP hard and what are reductions?
- What are approximation algorithms?

Approximation Algorithms

- Approximation Algorithms solve NP-complete optimization problems in polynomial time by producing answers that are sometimes not optimal.
- How "good" an approximation algorithm is measured by how far the approximate solution (which we call C) will be to the optimal solution (which we call C*). The factor is max(C/C*, C*/C). For example:
  - If working on a minimization problem and C* = 2, but the approximation will give no worse than C = 3, that means our approximation is a max(3/2, 2/3) = 3/2 = 1.5-approximation
  - If working on a maximization problem and C* = 2, but the approximation will give no worse than C = 1, that means our approximation is a max(1/2,2/1) = 2-approximation
- Approximation Algorithms are often accompanied by a proof on the bound.

Vertex Cover

- Given a graph G = (V, E), a vertex cover is a set of nodes in G that touches every single edge in G at at least one end.
- A Minimum Vertex Cover of a graph is the smallest set of nodes possible to provide a vertex cover for a graph.
- Solving this optimally requires exponential time. The best we can do is try every possible vertex subset with exhaustive search. Can we approximate it?
- https://visualgo.net/en/mvc



Vertex Cover Approximation Pseudocode

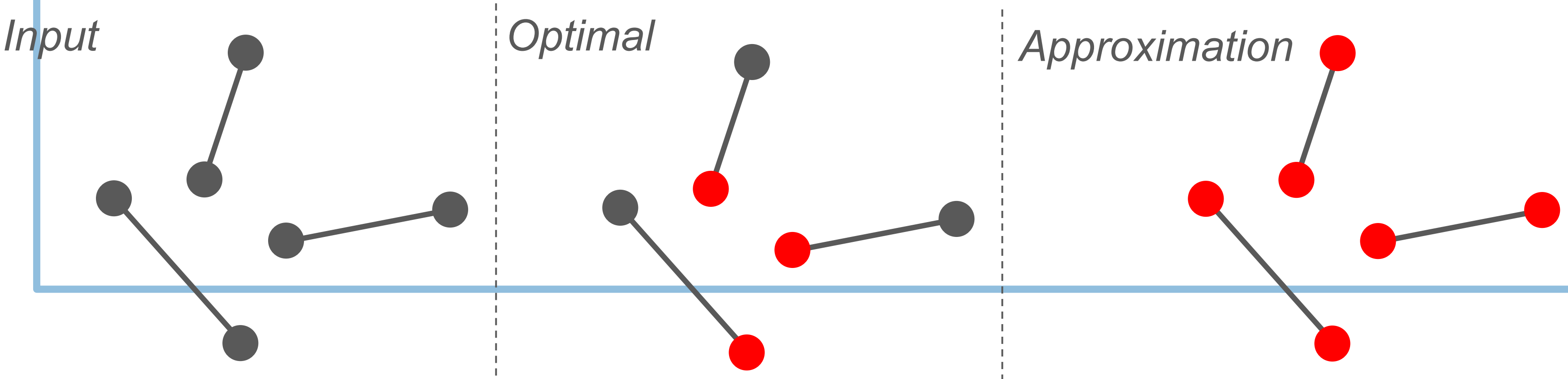
```
algorithm generateApproximateVertexCover
  input: a graph G
  output: a vertex cover of G
initialize C to an empty set
while there are still edges in G
    pick uncovered edge (u, v) arbitrarily
    add u and v to vertex cover C
    delete all edges incident on u and v
return C
```

Runtime?

How does this runtime compare to what is required to calculate the exact vertex cover?

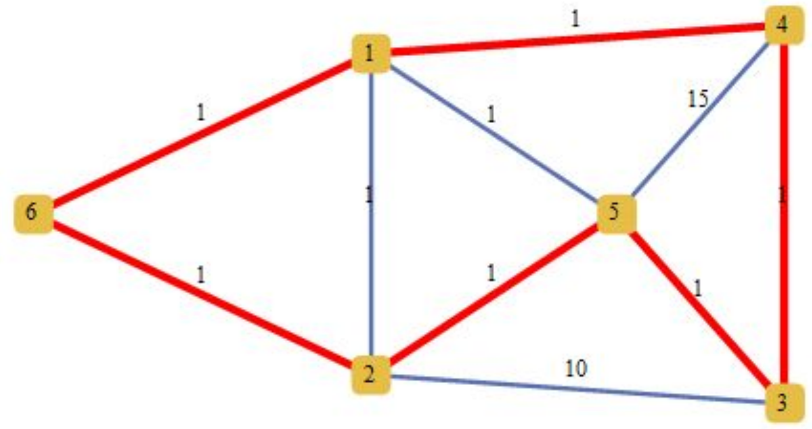
Vertex Cover Bound

- Notice that the N edges selected from our algorithm **share no vertices**.
- So 2N vertices are included in our approximate solution. <u>C = 2N</u>
- The optimal minimum vertex cover, by definition, has to cover at least those N vertex-disjoint edges with at least one vertex per edge, which means <u>C* >= N</u>.
- Combining the two underlined relations, we see that C <= 2C*
- So we have shown that this is a 2-approximate algorithm



Traveling Salesman Problem (TSP)

- Given a graph with cities as vertices and edges as roads with weights, find the best path that visits every city exactly once and winds up where we started (i.e. a "tour").
- The best algorithm we know how to find the best tour is currently exponential
- Motivation Reminder: Delivery or any round-trip routing problem.

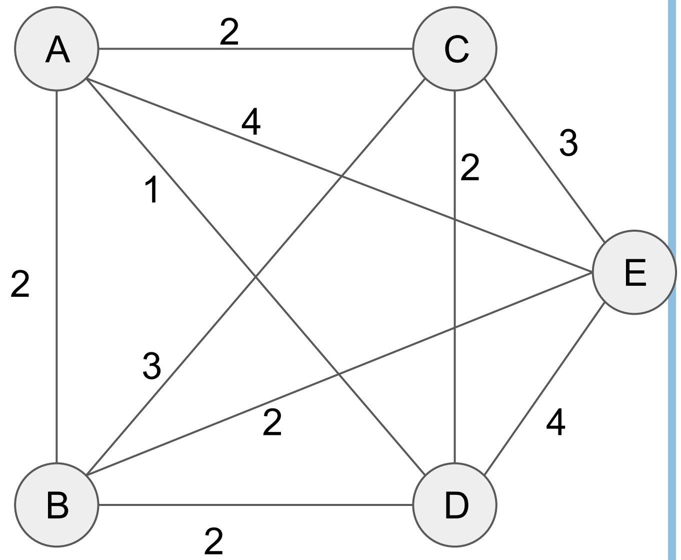


Traveling Salesman Problem (TSP)

What is the cost of the optimal tour?

10

Approximation Design: what other ways to do we know to connect all nodes?

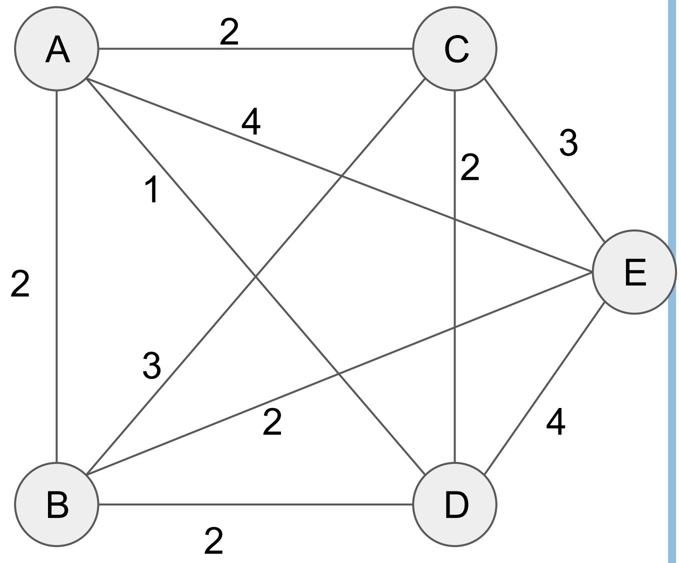


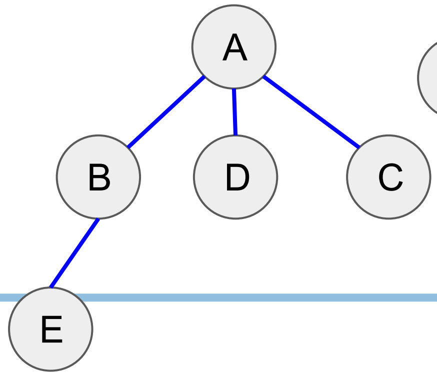
Traveling Salesman Problem (TSP)

What is the cost of the optimal tour?

10

Approximation Design: what other ways to do we know to connect all nodes? Minimum Spanning Trees!





Traveling Salesman Problem (TSP)

What is the cost of the optimal tour?

10

Approximation Design: what other ways to do we know to connect all nodes? Minimum Spanning Trees!

MST cost here is 7.

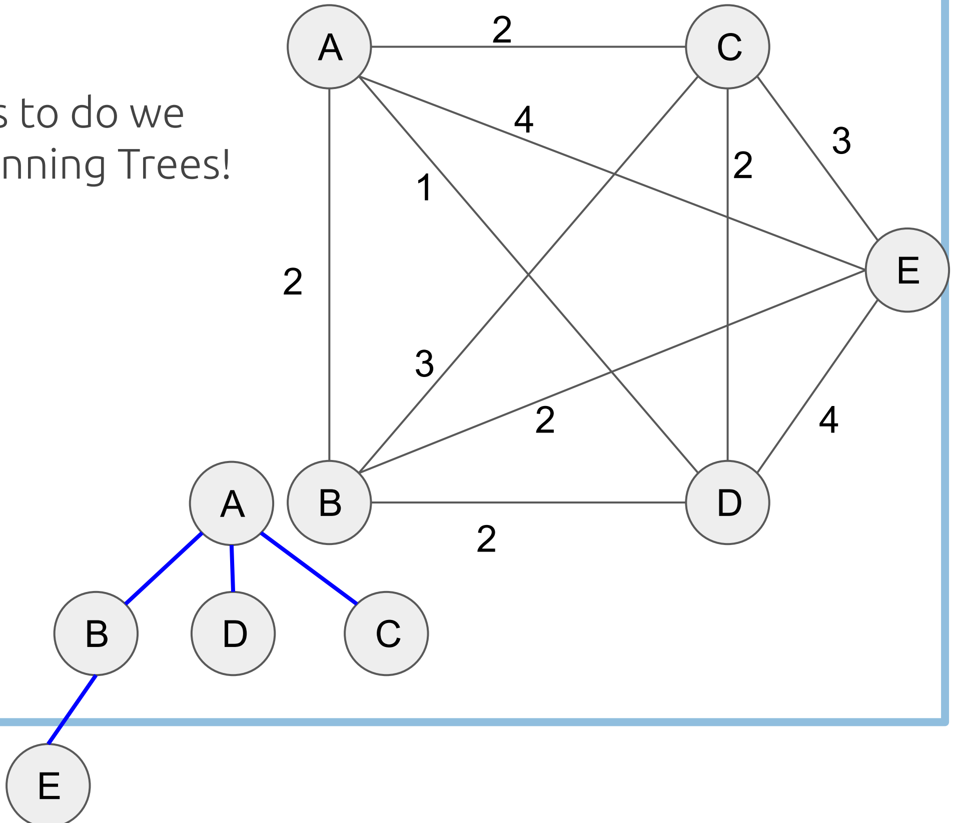
Cost of A-B-E-B-A-D-A-C-A?

14

Tour is not allowed repeat visits:

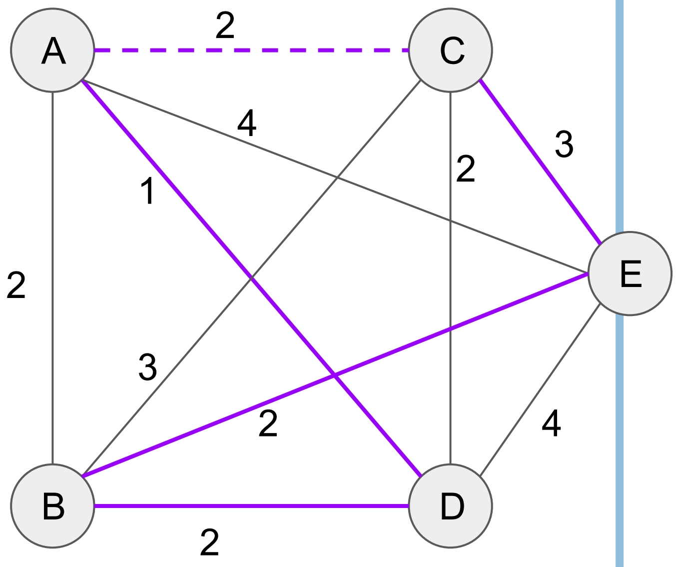
A-B-E-D-C-A

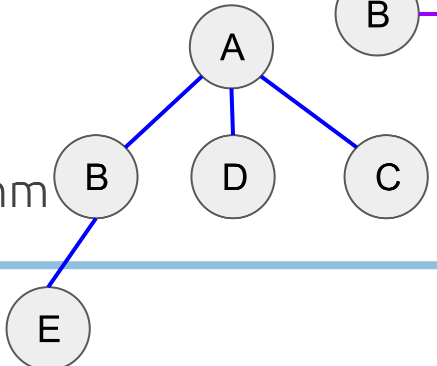
Cost is now 12



Traveling Salesman Problem Approximation Analysis

- Let's analyze the approximation algorithm
  - Find minimum spanning tree
  - Cut through repeated vertices
  - Return path as your cycle
- Consider taking one edge off the optimal tour to get a spanning tree (not necessarily the minimum)
  - C* >= MST
  - C <= 2 * MST (previous slide)
  - So C <= 2C*
- This is a 2-approximation algorithm





Takeaways (pt. 1)

- We want to avoid slow algorithms, so knowing if a problem is not in P is useful.
- P: decision problems that have polynomial time algorithms
- NP: decision problems that can be verified in polynomial time
- P ⊆ NP ⊆ EXP
- NP-Complete problems are both NP-Hard and NP, and lots of interesting problems are NP-Complete. They can often be “reduced” to each other.
- P ?= NP asks whether the above two complexity classes are the same. It is likely not true, but has not been proven.

Takeaways (pt. 2)

- The smallest change in a problem statement can make it P or NP, and it is not immediately obvious: MST versus Traveling Salesman versus Minimum Vertex Cover
- When we realize a problem will likely only have an exponential solution, we can come up with an algorithm that will give us an answer that is “good enough”
- Minimum Vertex Cover and Traveling Salesman are NP-Complete, but we can give out a 2-approximation to both those problems.
- For decades, the best we could do for TSP was exactly 1.5x the optimal tour.
  - After 44 years, we finally found an approximation that is something like 1.5 minus 0.2 billionth of a trillionth of a trillionth of a percent: https://www.quantamagazine.org/computer-scientists-break-traveling-salesperson-record-20201008/

COMP 285
Analysis of Algorithms

Welcome to COMP 285

Lecture 25: Approximation Algorithms

Lecturer: Chris Lucas (cflucas@ncat.edu)