Lecture 23 - NP Complete Problems

Eric A. Autry

Last time: NP Complete Problems

This time: Solving NP Complete Problems

Next time: Reading Week

Homework 7 due today.

Project 4 due Thursday, December 6th.

For the upcoming weeks...

I will hold regular office hours on Monday and Wednesday.

I will send an announcement tomorrow about potential TA office hours.

Over the weekend (or early next week) I will post a set of practice problems for the final along with their solutions. I will include a couple common interview questions.

I will post all homework solutions Tuesday night.

I will hold regular class times next week:

- Tuesday will be a ‘free’ lecture on some material that we did not have time to cover (this will not be on the exam).
- Thursday will be a review session for the final.

I will also hold office hours during finals week. I will send an announcement about when and where those will be.

NP Complete Problems

We have now seen a list of NP complete problems:

- SAT
- Traveling Salesman Problem
- Independent Set
- Vertex Cover
- Clique
- Hamiltonian Path
- Hamiltonian Cycle
- Longest Path
- Knapsack
- Subset Sum
- Bin Packing
- and more...

Reductions

We started looking at reductions, which are used to prove that problems are NP complete.

Already Saw:

- Independent Set to Vertex Cover
- Independent Set to Clique

Today:

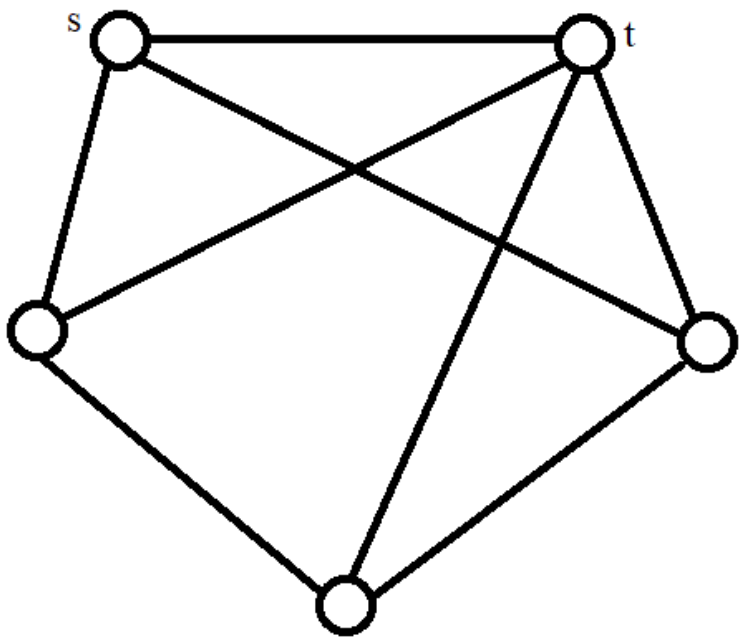
- Hamiltonian Path to Hamiltonian Cycle
- 3SAT to Independent Set

Hamiltonian Path to Hamiltonian Cycle

Here we start with an instance of the Hamiltonian Path problem:

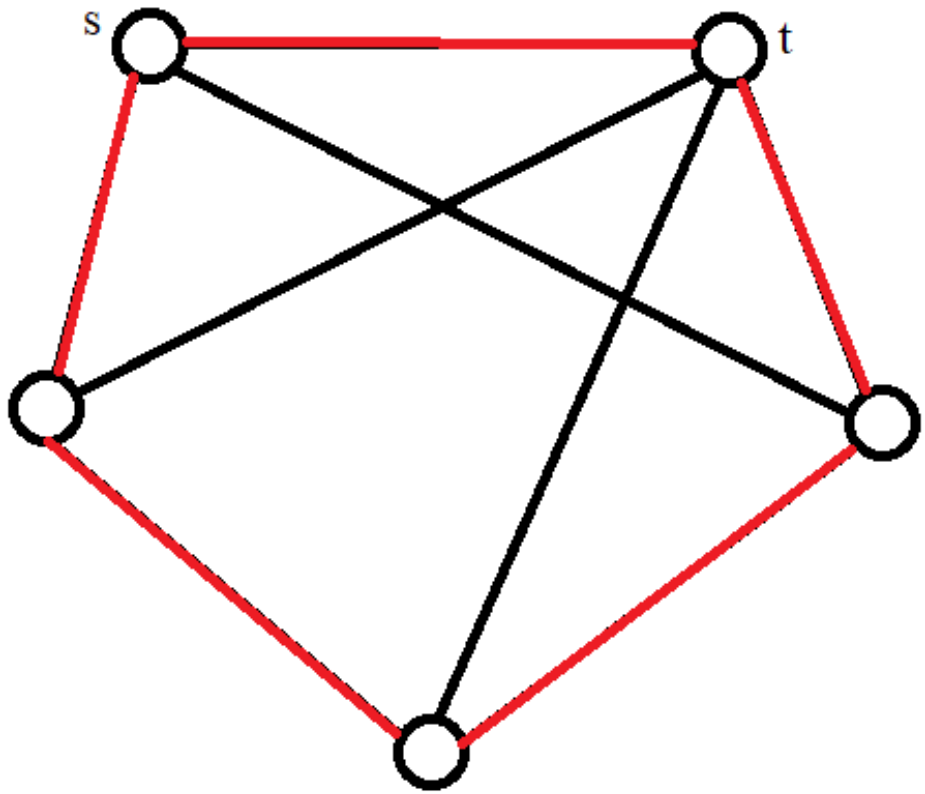
- Given a graph, find a path that visits each vertex only once, or report that such a path is not possible.

Specifically, we will assume that we have some start vertex s and some destination vertex t, and ask whether a Hamiltonian Path connecting s to t is possible.



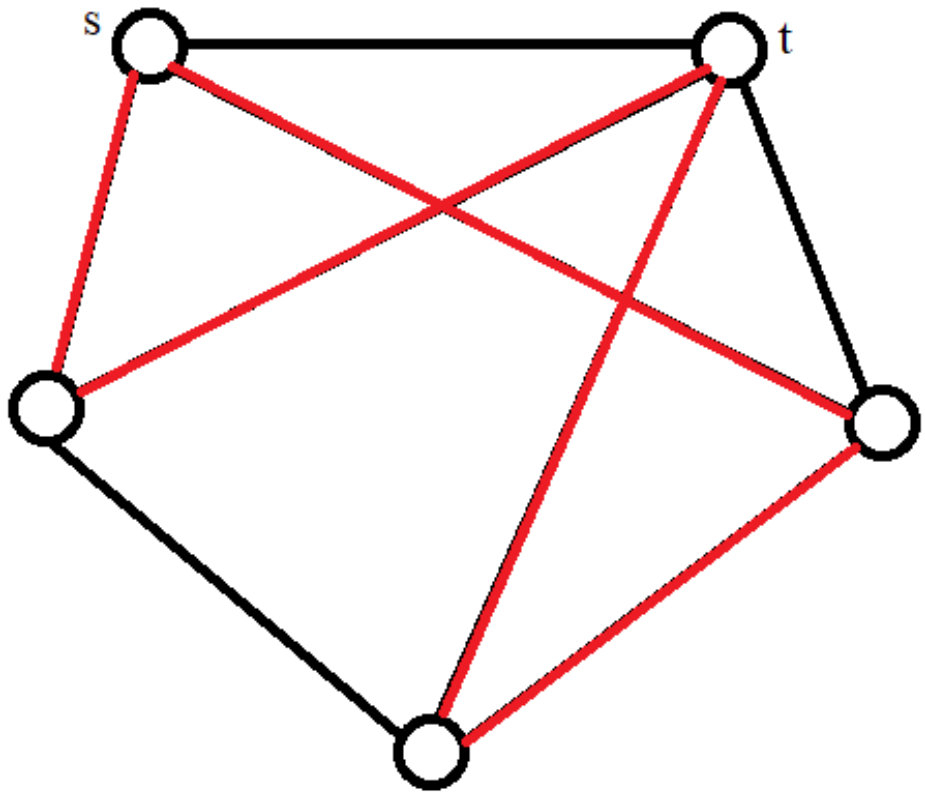
Hamiltonian Path to Hamiltonian Cycle

Note: if we have a Hamiltonian Cycle in our graph that uses the edge between s and t, then removing this edge would give us a Hamiltonian Path!



Hamiltonian Path to Hamiltonian Cycle

However, it is possible to have a cycle that doesn't use the edge between s and t. There is no single edge we can remove to obtain our Hamiltonian Path from this cycle:



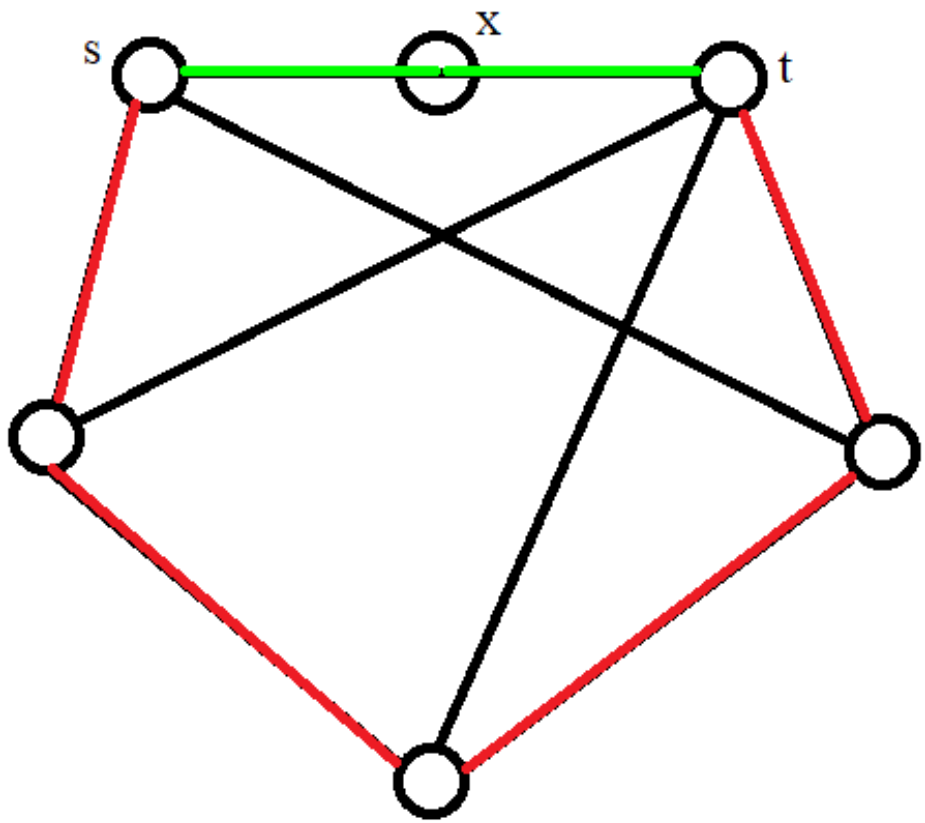
Hamiltonian Path to Hamiltonian Cycle

We can convert this problem into a Hamiltonian Cycle problem by adding a single vertex x to the graph that is connected only to vertices s and t.

- If this new graph has a Hamiltonian Cycle, then clearly the edges from s to x and from x to t must have been used consecutively.
- And all other vertices must have been visited.
- So removing x from the cycle results in a Hamiltonian Path between s and t.

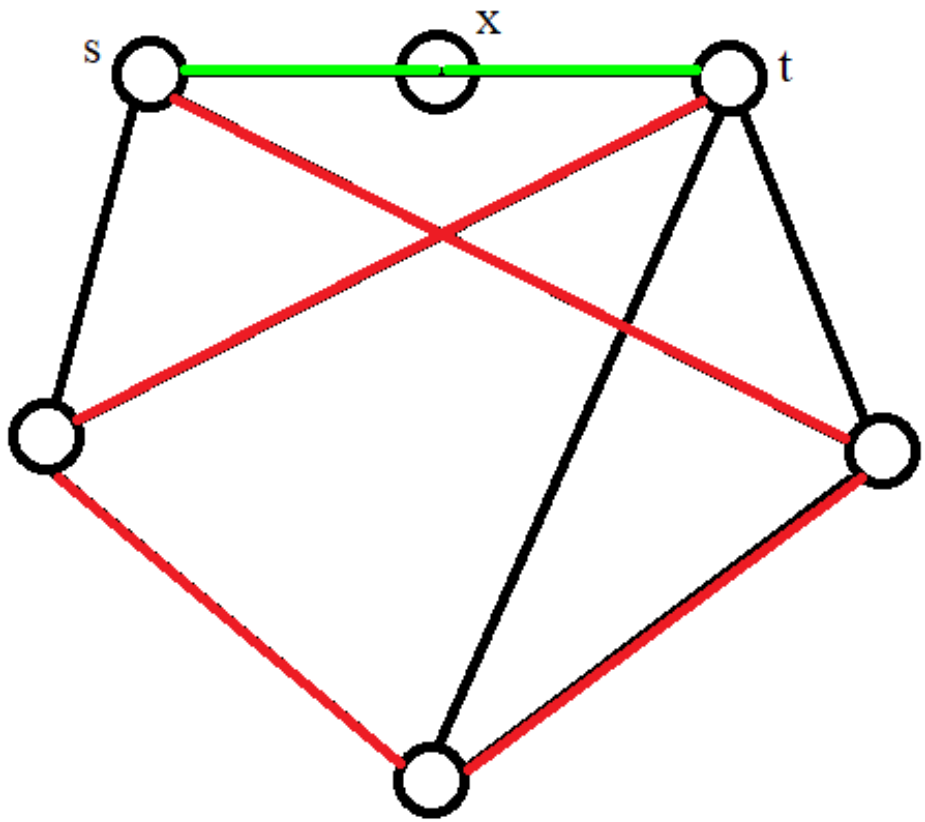
Hamiltonian Path to Hamiltonian Cycle

Note: we are forcing the Hamiltonian Cycle to use an 'edge' between s and t by forcing it to go through the dummy vertex x. This allows us to remove vertex x and obtain a Hamiltonian Path from s to t.



Hamiltonian Path to Hamiltonian Cycle

There may be multiple possible Hamiltonian Cycles, but since they all have to go through vertex x, we can always reconstruct a Hamiltonian Path from s to t by removing vertex x.



Hamiltonian Path to Hamiltonian Cycle

We can convert this problem into a Hamiltonian Cycle problem by adding a single vertex x to the graph that is connected only to vertices s and t.

- What if there is no Hamiltonian Cycle in the new graph?
- This means that there was no Hamiltonian Path in the original graph.

Note that the following statements are equivalent (contrapositives):

- If the new graph has no Hamiltonian Cycle, then the original graph had no Hamiltonian Path.
- If the original graph has a Hamiltonian Path, then the new graph has a Hamiltonian Cycle.

But the second statement is clearly true, because we could just add the edges (s, x) and (x, t) to the path to get the cycle.

Hamiltonian Path to Hamiltonian Cycle

So our reduction is:

- Given a graph G and vertices s and t for the Hamiltonian Path problem.
- Create a new graph G' that has an extra vertex x and edges (s, x) and (s, t).
- Solve the Hamiltonian Cycle problem on this new graph G'.
- If there is a cycle, then the Hamiltonian Path comes from removing vertex x from the cycle.
- If there is no cycle, then there is no path.
- As a final check: it takes only polynomial time to convert from G to G' and to convert the cycle to the path.

Note: we can also reduce Hamiltonian Cycle to Hamiltonian Path, i.e., the two are equivalent.

3SAT to Independent Set

3SAT is a special version of the SAT problem where each clause has only 3 literals in it:

$$(\neg x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z) \wedge (x \vee y \vee z) \wedge (\neg x \vee \neg y)$$

This problem is the canonical NP complete problem, and has been studied for many years.

We would like to reduce this problem to the Independent Set problem.

Note: this would prove that the Independent Set problem is also NP Complete.

- Because if the Independent Set problem can be solved efficiently, we could use this reduction to solve 3SAT efficiently.

Proving NP Completeness

Say we have a problem X and a problem Y. If we reduce X to Y in polynomial time, we write

$$X \leq_p Y.$$

In a sense, we are saying that Y is at least as difficult to solve as X.

This is because of our reduction: we took a problem of type X and showed that if we had a way to solve Y, we could use that method to also solve X.

Now, it is possible that there exists a more efficient algorithm to solve X, and so it could be an easier problem.

But it cannot be harder to solve than problem Y. If there is an easier way to solve Y, then we can use that, along with our reduction, to solve X.

Proving NP Completeness

A quick note on proving NP Completeness:

- If we have a problem X that we would like to prove is NP Complete, we reduce an NP Complete problem (call it NPC) to X:

$$NPC \leq_p X.$$

- This shows us that X is at least as difficult as the NP Complete problem, if not more difficult.

- Since 3SAT is a known NP Complete problem, then reducing 3SAT to Independent Set proves that Independent Set is also NP Complete.

Proving NP Completeness

Note that the reverse is not true. If

$$X \leq_p NPC,$$

we can say nothing about the difficulty of X aside from the fact that it is not harder than NP Complete.

- However, if we already know (or at least suspect) that X is NP hard, then this reduction to an NP Complete problem can still be useful: we have converted the new problem into a well-studied problem.
- It does not prove that X is NP Complete.

3SAT to Independent Set

3SAT is a special version of the SAT problem where each clause has only 3 literals in it:

$$(\neg x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z) \wedge (x \vee y \vee z) \wedge (\neg x \vee \neg y)$$

To reduce this problem to an Independent Set problem, we need to create some sort of graph.

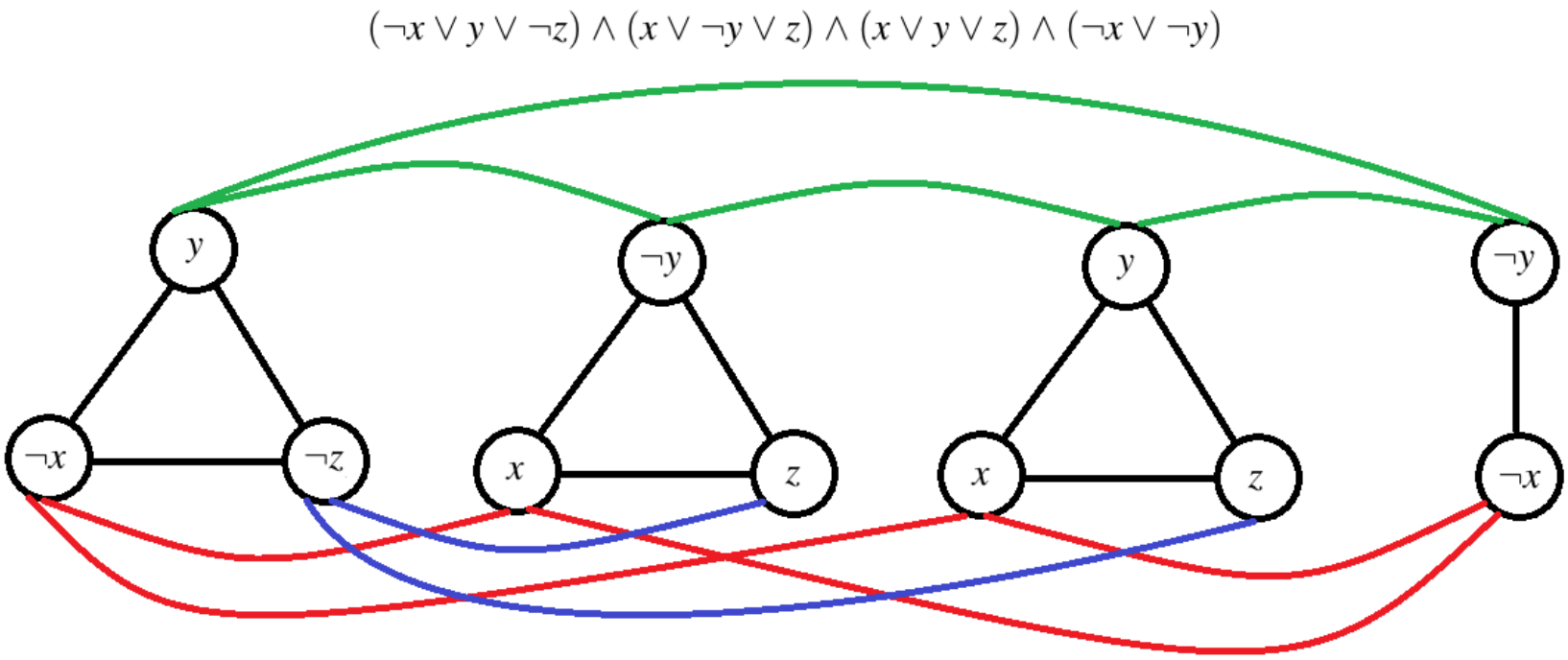
Idea: we will create gadgets where each literal becomes a vertex while each clause becomes a triangle.

Why a triangle?

The three vertices are connected, which forces us to choose only one vertex for each clause in the independent set problem.

We then connect any instance of x with every instance of $\neg x$.

3SAT to Independent Set



3SAT to Independent Set

Important observation: the Independent Set solution S cannot contain both x and $\neg x$.

- Because they always have edges between them.

So our truth assignments will be to say x is True if $x \in S$ and x is False if $\neg x \in S$.

Now, the Independent Set problem needs one last input: the goal size of the independent set.

- There are k clauses and we would like to find one satisfying assignment per clause.
- So we will look for an independent set of size k.

3SAT to Independent Set

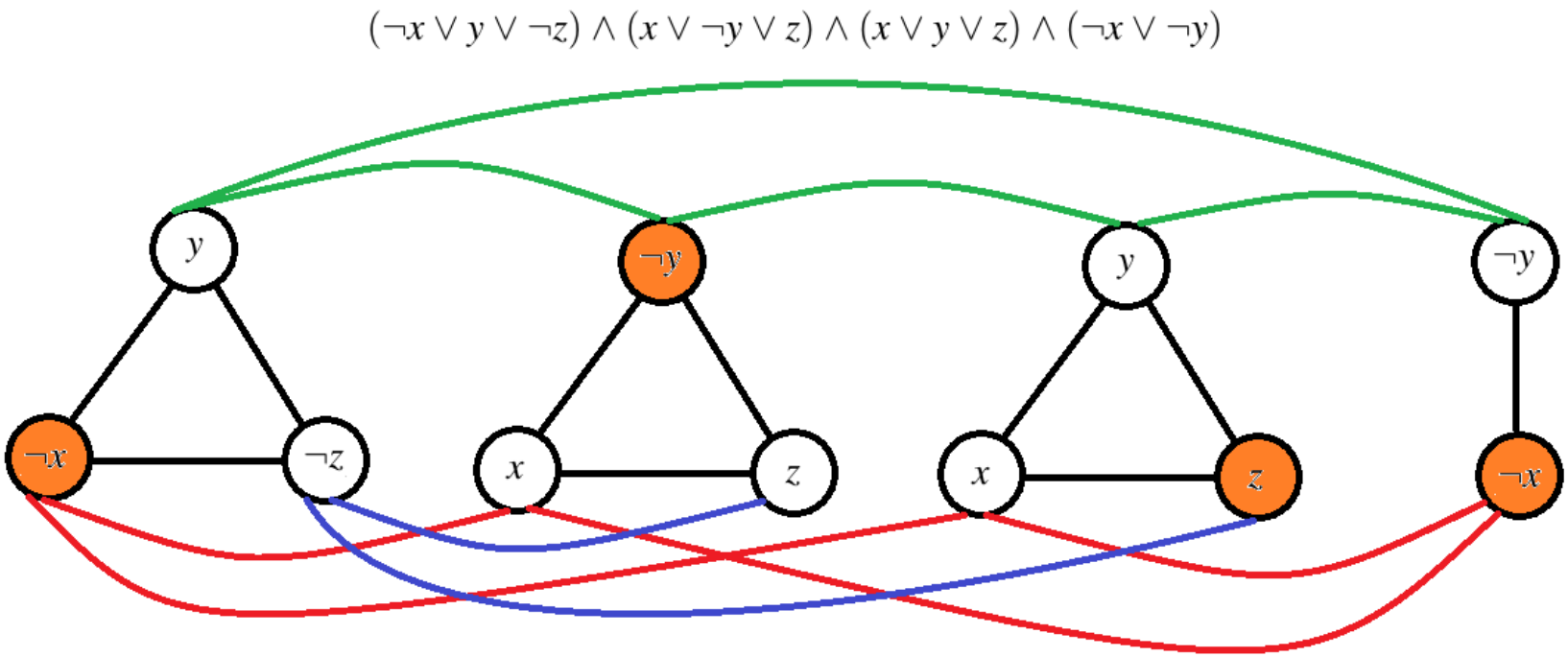
So what happens if we successfully find an independent set S of size k?

- Because the clauses were triangles, we know there must have been exactly one vertex selected from each clause.
- But this means that every clause had a variable assigned that satisfied it.
  - If the selected variable was x, then setting x to True satisfied the clause.
  - If the selected variable was $\neg x$, then setting x to False satisfied the clause.
- Because there were edges between every variable and its negation, this truth assignment is valid.

3SAT to Independent Set

One possible assignment for this problem:

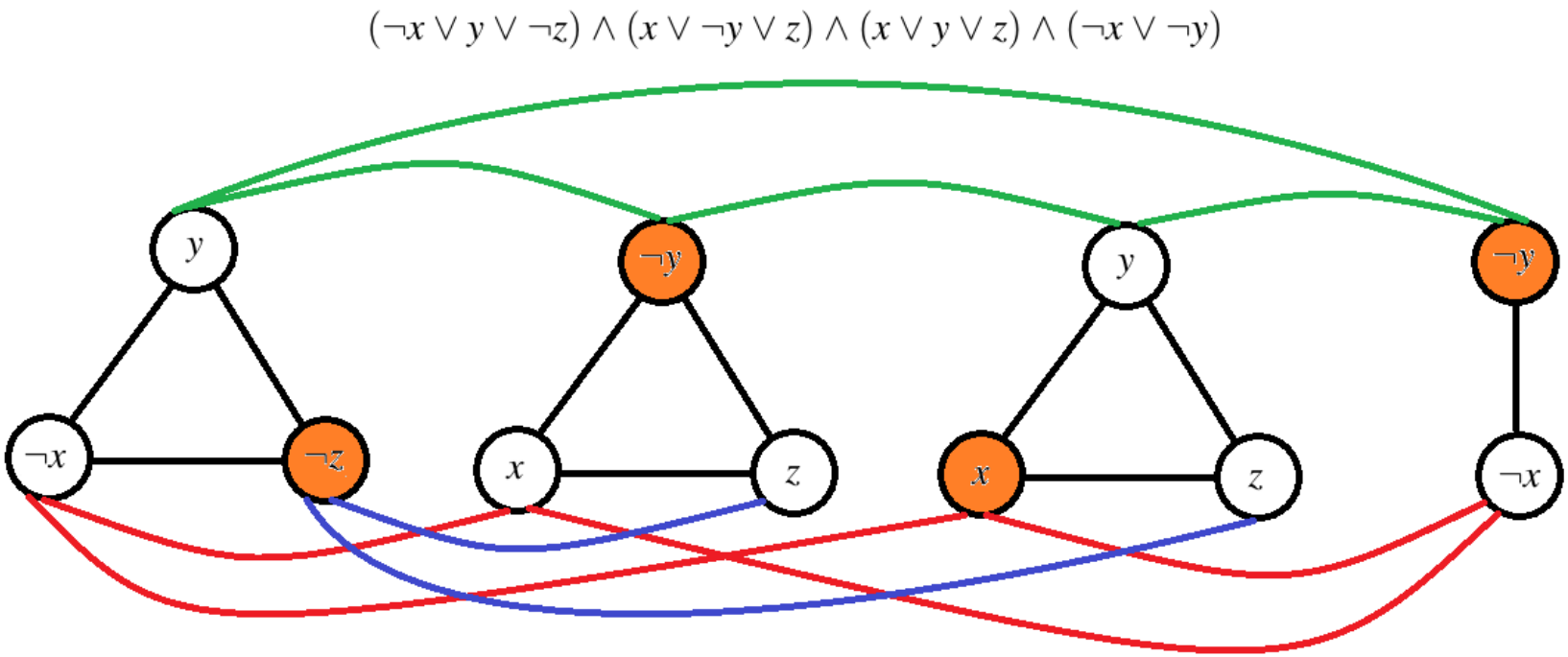
$$x = F, \quad y = F, \quad z = T$$



3SAT to Independent Set

There can be multiple solutions. Another possible assignment for this problem:

$$x = T, \quad y = F, \quad z = F$$



3SAT to Independent Set

What if there was no independent set of size k?

- Then we have to show that this implies there was no satisfying truth assignment.

Consider the contrapositive: if there was a satisfying truth assignment, then there must have been an independent set of size k.

- Well, if such an assignment exists, then each clause can be satisfied, i.e., has on literal that is True.
- So look at the gadget for each clause and select 1 vertex per gadget that corresponds to a True literal.
- This creates an independent set of size k.

3SAT to Independent Set

So the reduction:

- First, we consider the decision problem for 3SAT (can all k clauses of n variables be satisfied?) and the decision problem for Independent Set (can an independent set of size g be found?).
- We are reducing from 3SAT, so we start with k clauses of n variables.
- We create a graph G using our gadget formulation where every clause becomes a clique and all vertices are labeled with their literals, finally adding edges between every literal and all of its negations.
- We set $g = k$.
- We then solve the Independent Set problem on graph G, looking for a set of size $g = k$.
- If there is a solution, those selected vertices correspond to true literals.
- If there is no independent set of size $g = k$, then there is no satisfying assignment.

Dynamic Programming and the Knapsack Problem

Recall the knapsack problem:

- We have n items that we want to place into our knapsack.
- Each item has an integer weight w_i and value v_i.
- The knapsack has a *capacity* weight W.

The Knapsack **optimization** problem is to determine the set of items we can place into our knapsack (without exceeding the weight limit) that have the largest total value.

Basis Idea: for each item, we can either use put the item into our knapsack (use-it) or not (lose-it). This gives us the following recursive algorithm.

- Remember that the recursive algorithm for use-it or lose-it returns only the size of the optimal solution.
- The DP table we later construct will give us the ability to retrace the solution.

Dynamic Programming and the Knapsack Problem

```
def knapsack(w, v, W):
    # w is list of weights,
    # v is list of values,
    # W is capacity.

    # Base Case 1: no items left.
    if len(w) == 0:
        return 0

    # Base Case 2: Knapsack full.
    if W == 0:
        return 0

    # Base Case 3: Next item wont fit.
    if weights[0] > W:
        return knapsack(w[1:], v[1:], W)

    # Use-it or Lose-it.
    useIt = v[0] + knapsack(w[1:], v[1:], W-w[0])
    loseIt = knapsack(w[1:], v[1:], W)
    return max(useIt, loseIt)
```

Dynamic Programming and the Knapsack Problem

Now we can convert this into a 2D DP table:

- Items across the top.
- Possible capacity values down the side.
- Very similar to the coins problem...
  - In fact, that problem was a variation of the knapsack problem!
  - Each coin's weight is its monetary worth.
  - The knapsack's capacity is the amount of change to return.
  - Each coin's value is -1, which means max value is fewest number of coins.
- Side note: the ferry problem from Homework 7 is a version of the multiple knapsack problem. problem.

Runtime:

- DP table is n items by W possible capacities.
- So runtime is $O(nW)$.

Dynamic Programming and the Knapsack Problem

Runtime:

- DP table is n items by W possible capacities.
- So runtime is $O(nW)$.

Wait a second, I thought this was NP hard. That looks polynomial...

- It is polynomial in the value of the input, but not the **size** of the input.
- W is the capacity, but we really need the **size** of W.
- Say that W is stored using m bits.
- Then $m \sim \log_2 W$, and so $W \sim 2^m$.
- So the runtime is really $O(n2^m)$.

Dynamic Programming and the Knapsack Problem

Note: this is a subtle point that relates to the definition of NP Complete problems: they are problems that can be solved in polynomial time by nondeterministic Turing Machines.

- So when we talk about runtime, we technically should write the runtime as a function of the space used to store the input on the tape.
- In practical terms though, the runtime for the knapsack DP solution is $O(nW)$.

Approximation Algorithms

For NP hard problems, even very clever solution techniques cannot promise less than exponential time.

- Though some solvers, like SAT solvers or the DP approach for the knapsack problem, can run efficiently most of the time.

Many NP hard problems can be efficiently approximated.

- For these approximations, we are typically concerned with how well they can do.
- i.e., how close they can get to the optimal solution.
- We will look at approximations for TSP and Bin Packing.

MST and TSP (Project 4)

We are looking for a cheap TSP tour that visits all the cities.

- Note that MST *reaches* all the cities at a minimum cost M.
- Important observation: the weight of the MST is less than the weight of all tours.
  - Consider a tour that visits all cities only once.
  - When you remove an edge from this tour, you have a spanning tree.
  - But the *minimum* spanning tree has less weight than all spanning trees.
- But, the TSP solution is a tour.
  - Say the optimal TSP tour has weight σ.
  - Then we know that $M < \sigma$.

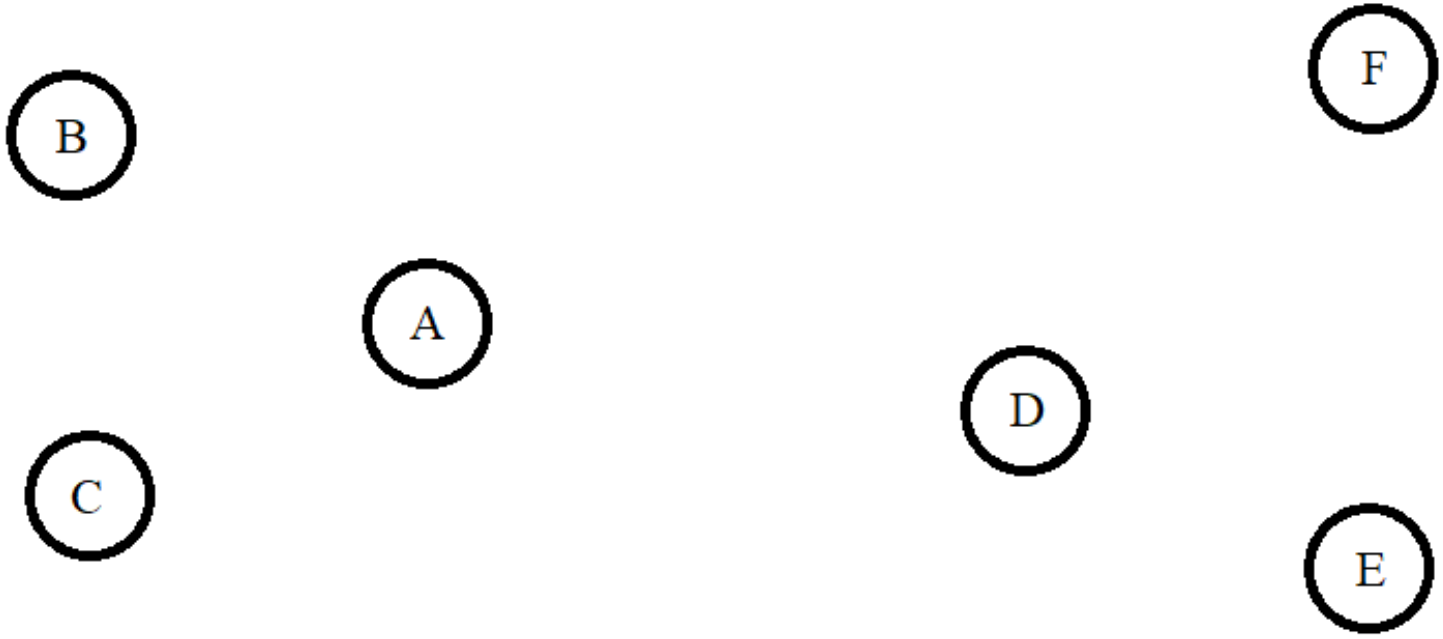
MST and TSP (Project 4)

Idea: what if we use DFS to explore the MST?

- For this approach, we will force DFS to revisit cities as it passes them. (i.e., we force our salesman to stay on the edges of the MST.)
  - We will change this later.
- We would have a tour of the cities, except that we might visit some cities twice.
- The cost for this 'tour' would be $2M$.
- But remember that $M < \sigma$, so the cost for this 'tour' would be $2M < 2\sigma$.
- We will have visited the cities for a cost that is within a factor of 2 of optimal!

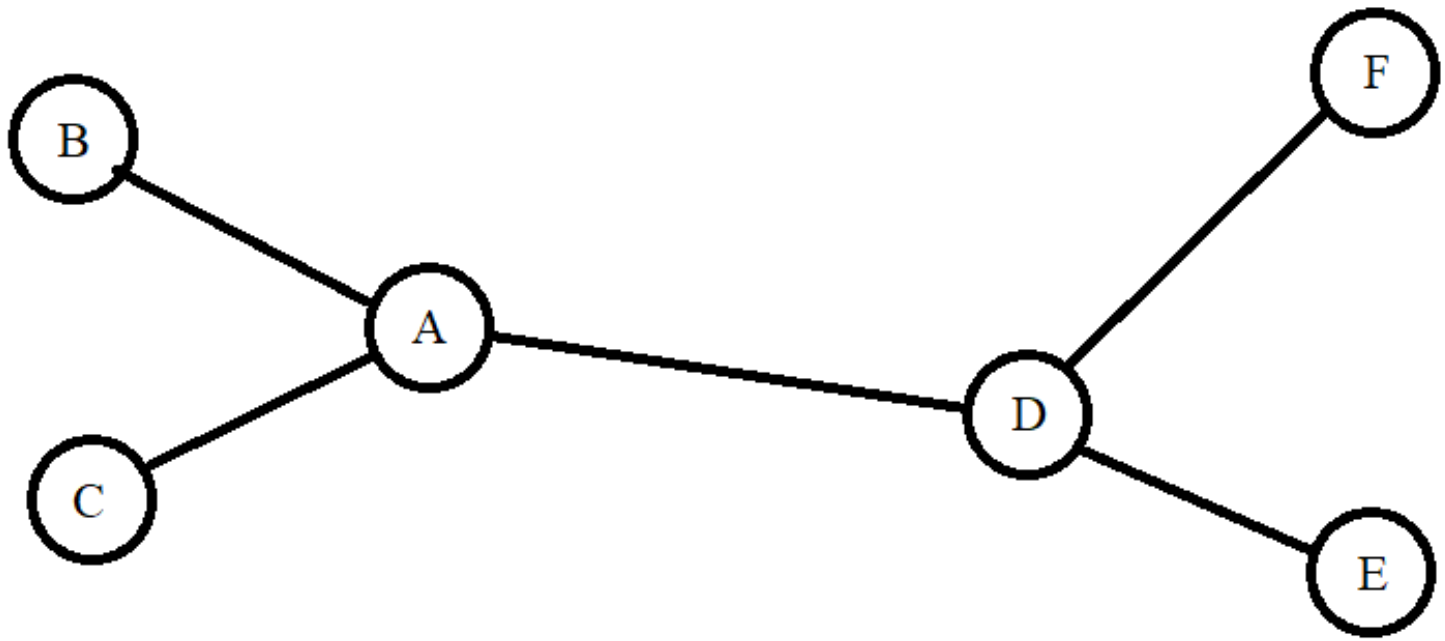
MST and TSP (Project 4)

Consider the following cities. Remember that the TSP problem gives us a complete graph, so while they are not drawn in, there are edges between every pair of cities.



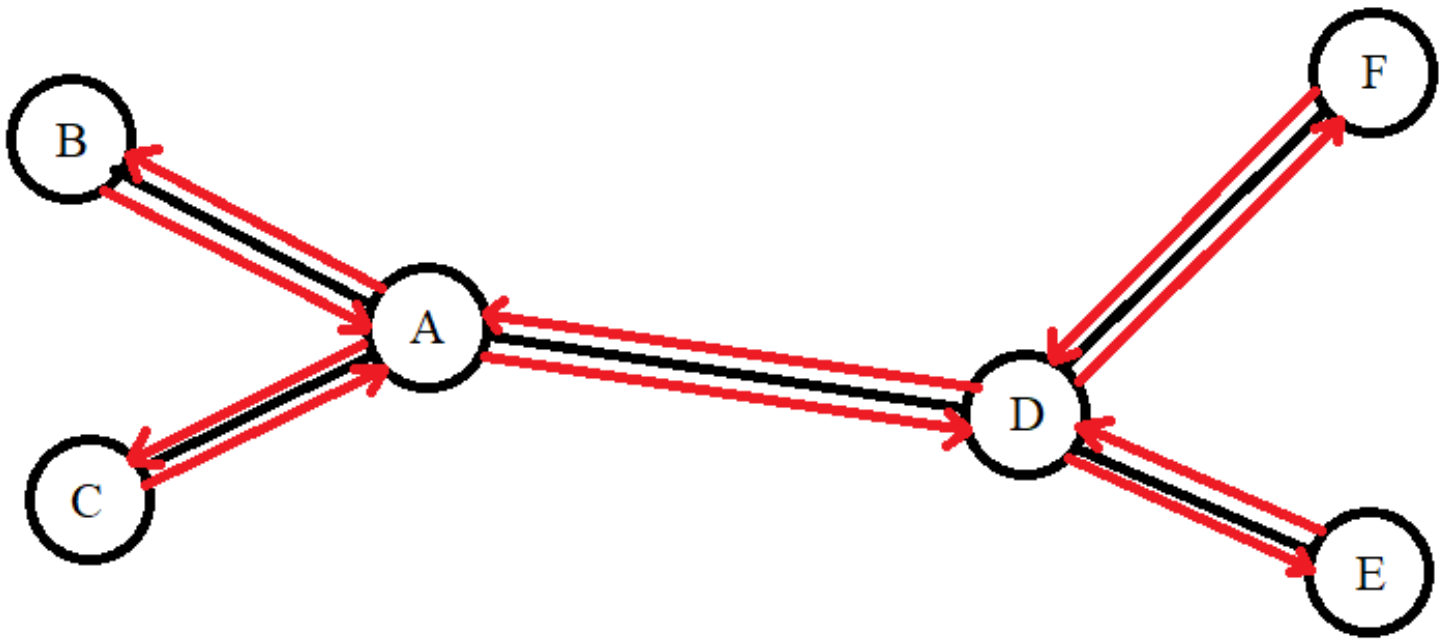
MST and TSP (Project 4)

We now draw the MST in black, which has a weight M.



MST and TSP (Project 4)

Starting at city A, we visit every city using DFS, remaining only on the MST edges. Note that this 'tour' uses each edge twice, for a cost of $2M$.



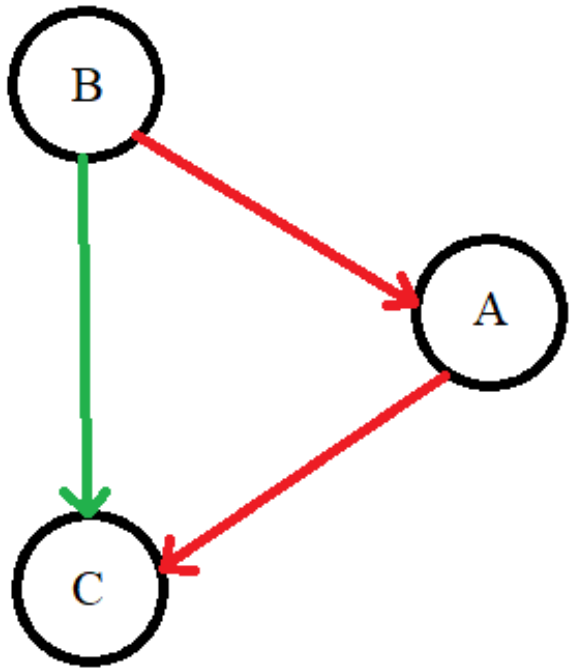
MST and TSP (Project 4)

Problem: our DFS-MST approach is not a proper tour.

- We will now need an important restriction on the TSP problem: that the distances obey the triangle inequality:
  - The sum of the lengths of two sides of a triangle is always greater than or equal to the length of the third side.
- Real life cities obey this property!

MST and TSP (Project 4)

The triangle inequality will help us because we can now make a significant observation: it is always better to simply go directly from B to C instead of revisiting city A in between.



MST and TSP (Project 4)

We can now modify our DFS-MST approach:

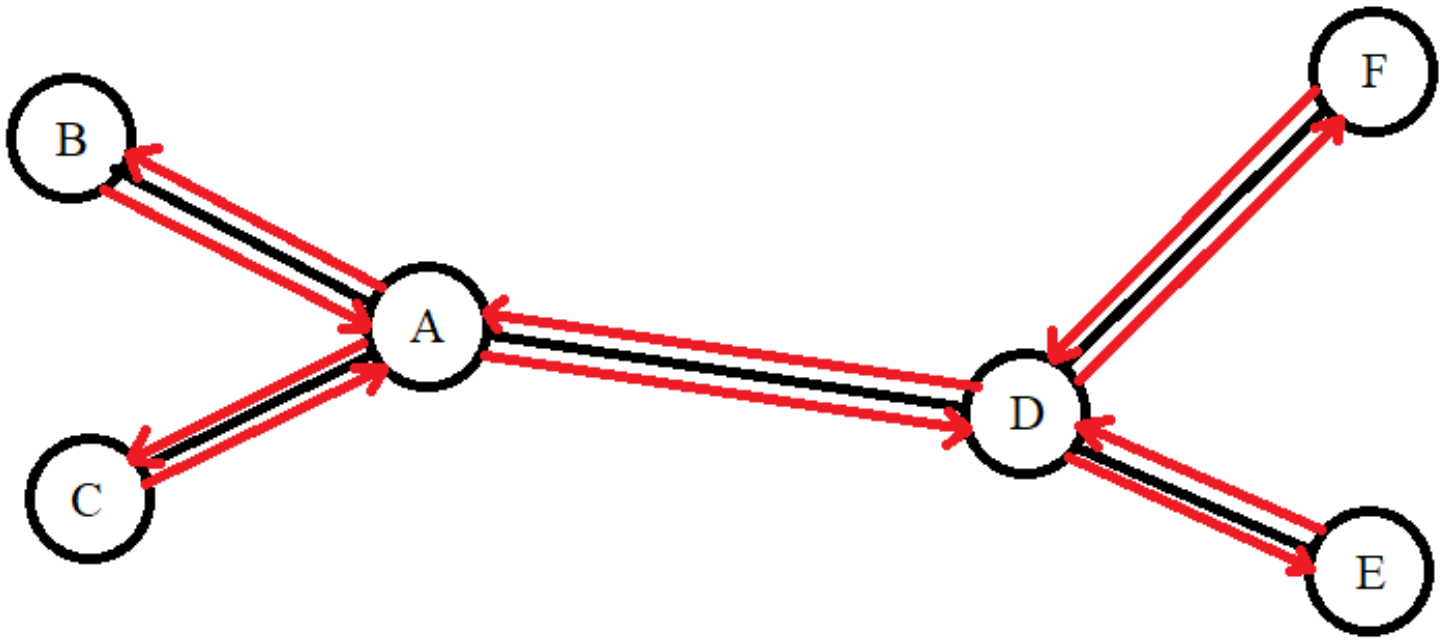
- We still use DFS to determine the order in which we will visit the cities.
- But now we will not allow it to return to cities as it explores.
- As an example: say the DFS goes through cities $A \to B \to A \to C$.
  - i.e., city B was a leaf of the MST and we had to return to A before moving on to C.
  - Recall: the TSP graph is a complete graph, so we can go directly from B to C.
  - So we will say that the tour visits $A \to B \to C$ directly.
  - Due to the triangle inequality, the distance of the path $B \to A \to C$ (two sides of the triangle) is greater than or equal to the distance of the path $B \to C$ (the third side).

So our approximation is less than $2M < 2\sigma$, and we will have an answer within a factor of 2 of the optimal tour!

MST and TSP (Project 4)

Originally, our MST ‘tour’ gave us (where the capital letters are used to denote when a new city is visited)

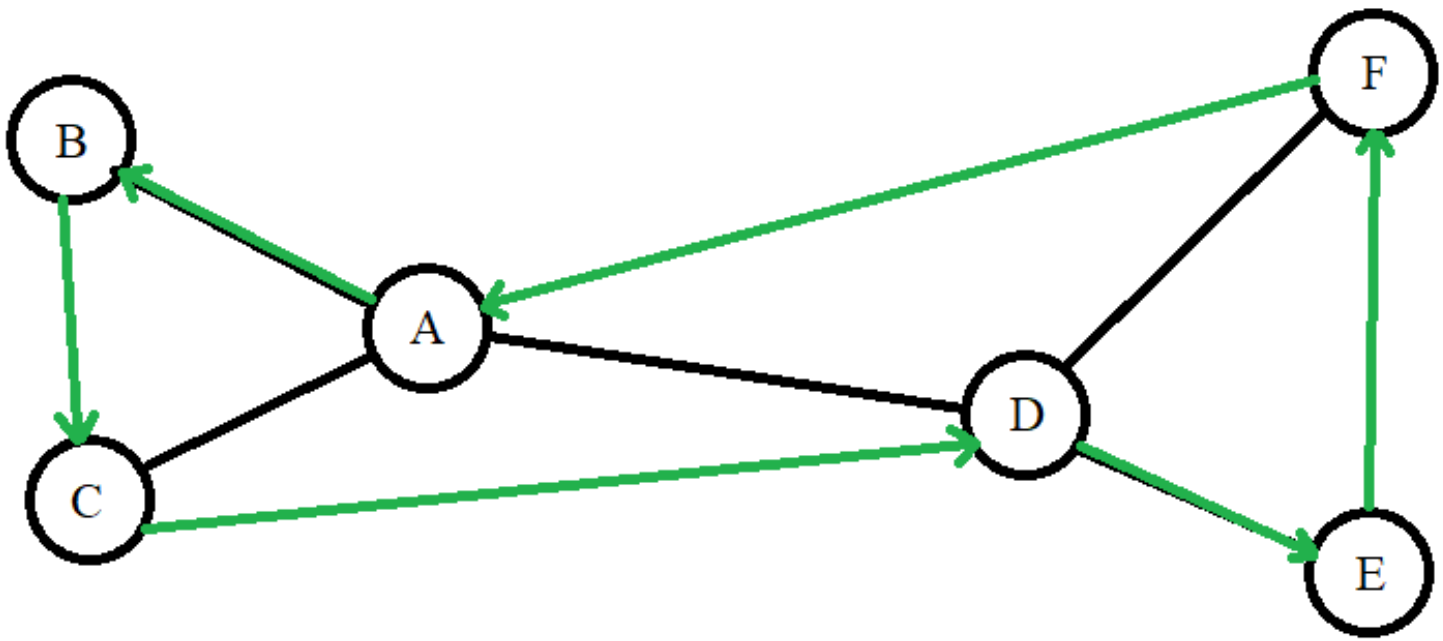
$$A \to B \to a \to C \to a \to D \to E \to d \to F \to d \to A$$



MST and TSP (Project 4)

Removing the cities that we revisited (keeping only the capital letters from before, but in the order they were encountered using DFS), we obtain the approximation tour:

$$A \to B \to C \to D \to E \to F \to A$$



First Fit Algorithm

Recall Bin Packing:

- We are given a set of items of varying volumes v_i.
- We are given bins that each have a capacity volume V.

The Bin Packing optimization problem is to determine the minimum number of bins required to fit all of the items.

We will use a greedy approach to approximate the solution.

First Fit Algorithm

Idea: when considering a new item, put it in the first bin that fits it (or use a new bin if necessary).

- Can we ever have 2 bins that are both less than half full?
- No! The items in the second bin could have fit in the first.
- So greed would have placed all of them in that first bin.

What is best possible ideal solution for the problem?

- All the bins are perfectly filled.
- But our greedy solution promises that each bin is at least half-full.
- This means our approximation uses at most twice the number of bins as the ‘perfect’ packing.
- The optimal packing will be between our greedy solution and the ‘perfect’ packing solution.
- So our approximation is within a factor of 2 of optimal!

The End

Thank you all so much for making this a great semester!

Please take the time to fill out the class survey on DukeHub.