Outline

Course Topics **1**

1 Aliens Problem **2**
1.1 Understanding the Problem 2
1.2 Trying to Solve the Problem 2
1.3 Working Out the Details 4
1.4 Reflect and Improve 5

2 Simple Path of Length 7 **6**
2.1 Understanding the Problem 6
2.2 Trying to Solve the Problem 6
2.3 Working Out the Details 8
2.4 Reflect and Improve 9

Course Topics

Over the course of the semester, we have seen a number of techniques that are useful for designing algorithms. In the first part of the semester, we learned algorithms that used greedy choice, dynamic programming, divide-and-conquer, and reductions to previously solved problems. In the second third of the semester, we learned about how to use randomization and amortization, as well as how to analyze online algorithms in comparison to the optimal solution. Any or all of these techniques may be useful in designing your own algorithms for Quiz 2.

In the case of reductions, it's good to have in mind another list: a list of problems that you know how to solve. To the right there is a list of problems and the runtimes of the algorithms that solve them. This list is, of course, not complete — it's missing problems such as scheduling, matrix multiplication testing, and so forth. However, each of the algorithms on the list has been (a) covered this semester in lecture, and (b) has at least one chapter devoted to it in CLRS. So these problems are in some sense the big ones.

Algorithm Design Tools.
- greedy choice
- dynamic programming
- divide and conquer
- reduction to solved problem
- randomization
- amortization
- competitive analysis

(Selected) Solved Problems.
- select: $\Theta(n)$
- sorting: $\Theta(n \lg n)$
- minimum spanning tree: $\Theta(E + V \lg V)$
- discrete Fourier transform: $\Theta(n \lg n)$
- all-pairs shortest paths: $\Theta(VE + V^2 \lg V)$
- hashing: $O(1)$ insert, delete, lookup
- network flow: $\Theta(f^* \cdot V)$ or $\Theta(V^3)$

1 Aliens Problem

Because the quiz is entirely focused on problem-solving, a good way to review for the quiz is to try to solve problems. To that end, let's consider the following problem:

Aliens Problem.
- building with n rooms, m doors
- known subset of rooms may contain aliens
- known subset of rooms contain survivors
- $k \ll n$ guard robots to place in rooms
- GOAL: protect survivors from aliens with guards

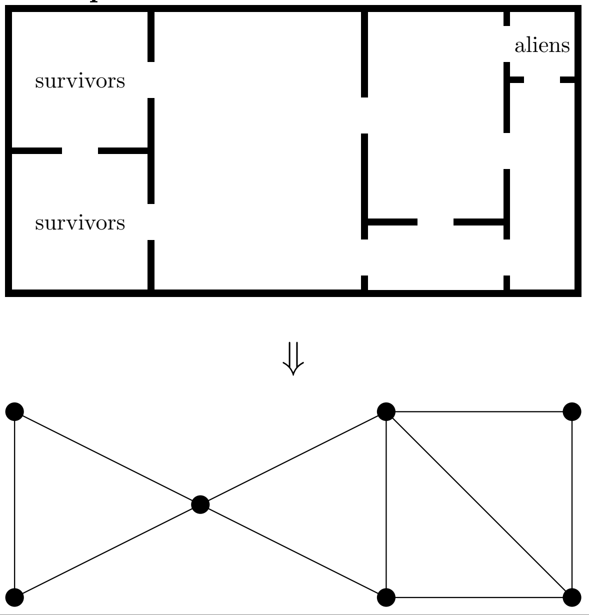
Problem 1. *Ripley is holed up inside a building trying to hide from aliens. The building consists of n rooms joined by m doors. Some rooms have a door to the outside, which could allow the aliens to enter. Other rooms are currently being occupied by a small band of survivors, whom Ripley is determined to protect.*

Luckily, Ripley has $k \ll n$ guard robots that she can place in various rooms around the building (including rooms that contain survivors or rooms that may contain aliens) to keep the aliens from getting to the survivors. If a room contains a guard robot, no aliens can pass through the room. Devise an algorithm to help Ripley and her band of survivors last the night (or report that survival is impossible).

1.1 Understanding the Problem

The very first step in problem-solving is understanding the problem. A good first step is coming up with an example, preferably one that is small enough to understand, but large enough to capture some of the complexity. The example shown on the right has seven rooms, with a large number of doors connecting them. In this example, if $k = 1$, then there are two possible locations in which to put the guard robot: one in the giant room at the center, or one in the room where the aliens might enter from.

Example.



But it's hard to model this as rooms and doors. Rooms have dimension, doors have locations, but the only thing we care about (according to the problem) is connectivity: whether there's a way for the aliens to enter through some allowable room, and then pass from room to room via doors until they reach a survivor without ever passing a guard robot. So, because each door connects two rooms, we model this as a graph problem, as depicted at right.

We can now restate the problem as follows: given an unweighted undirected graph $G = (V, E)$ and subsets $H \subseteq V$ and $A \subseteq V$, find a set $R \subseteq V$ of vertices with size k such that any path from an alien $a \in A$ to a survivor $h \in H$ contains some vertex in R.

1.2 Trying to Solve the Problem

With the problem rephrased this way, there is something about it that rings a bell. Our goal is to find a subset R of vertices such that any path from an alien to a human includes one of the vertices in R. This is somewhat reminiscent of the edges crossing an s-t cut: any path from s to t must contain some edge crossing the cut.

What algorithms do we know that involve s-t cuts? Well, we know that s-t cuts are closely linked to s-t flows: that the capacity of the minimum cut is equal to the maximum flow. In fact, there is a way to compute the minimum s-t cut from the maximum s-t flow. So if we could find a way to reduce the aliens problem to the minimum s-t cut problem, then we would be able to find an algorithm for the aliens problem.

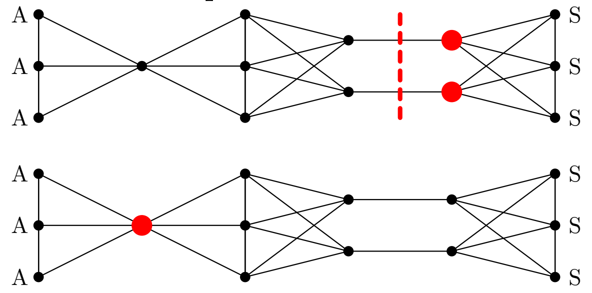
What might a reduction look like? Well, although we have discussed some of the similarities between the minimum s-t cut and the aliens problem, we must also keep in mind the differences. The minimum s-t cut problem involves finding a cut that minimizes the sum of the capacities of the edges crossing the cut in the direction from s to t. So we can think of the s-t cut problem as trying to find a set of edges of minimum total capacity that are necessary to get from s to t. But in the aliens problem, we want to find a set of vertices of size k that are necessary to get from A to H. There are several differences here.

Reduce to Minimum s-t Cut.

- transform vertices into edges
- give edges capacity 1
- use minimum to find a set of size $= k$
- s is a supernode connected to all aliens
- t is a supernode connected to all survivors

First, one problem involves finding a set of edges; the other involves finding a set of vertices. So if we want to take advantage of the known algorithm for min s-t cut, we need some way to convert between vertices and edges. We might consider doing the following: find a set of edges constituting a minimum cut, and then split the vertices incident to those edges into two groups based on which side of the cut that they fall into. If we pick the smaller of the two groups, we might get a good set of vertices. Unfortunately, the counterexample to the right shows that this technique doesn't work.

Counterexample.



Instead, we'll try to look for another way. It would be nice to have a one-to-one correspondence such that for each vertex, there is an edge in the transformed graph such that we choose a vertex to guard exactly when we choose to cut the edge. One way to do this would be to split every vertex in the original graph into two vertices in the transformed graph, connected by a single edge. We will figure out the details of this splitting process later.

Second, one problem involves minimizing the sum of the weights of items in the set; the other involves picking a set whose size is k. Luckily, we're trying to convert from the unweighted case to the weighted case, so it would seem like we could handle part of this difference by setting all weights / capacities equal to 1. Furthermore, note that if there is a set of vertices of size $< k$, then we can construct a set of vertices of size k that also separates the humans from the aliens by adding extra vertices to the set. So if we find the minimum-sized set R^* of vertices that separates the humans from the aliens, then we can solve the problem as follows: if $|R^*| > k$, then the survivors will die; if $|R^*| \leq k$, then we can add extra guard posts until we get R with $|R| = k$.

Third, one problem involves cutting off paths from a single source to a single sink; the other involves multiple humans and multiple aliens. However, we saw in lecture how to handle this kind of problem: create a new source that points to all of aliens, and create a new sink that is pointed to by all of the humans.

1.3 Working Out the Details

Now that we've worked out an idea for an algorithm, it's time to formalize it. We want a reduction that takes as input an instance of the aliens problem $\langle G = (V, E), A, H, k\rangle$ and outputs an instance $\langle G' = (V', E'), s, t\rangle$ of the minimum s-t cut problem. This reduction should have the property that we can use a minimum s-t cut of $\langle G', s, t\rangle$ to produce a way to protect the humans in H from the aliens in A.

Our goal is to set up G' so that there is a nearly one-to-one correspondence between paths in G and paths in G'. We have previously decided to take each vertex $v \in V$ and create two vertices in V' connected by an edge in E'. For reasons which will later become clear, call those two vertices v_{in} and v_{out}. Due to this known mapping between vertices in both graphs, it seems like it would be useful to make it so that each path in G' that passes through v_{in} or v_{out} corresponds to a path in G that passes through v. In fact, because the edge (v_{in}, v_{out}) was created to be the equivalent of the vertex v, it would be nice to make sure that any path in G that passes through v corresponds to a path G' that passes through the edge (v_{in}, v_{out}). In other words, we want to make it so that any path in G' that passes through v_{in} or v_{out} should pass through the edge (v_{in}, v_{out}).

Direct the edge (v_{in}, v_{out}) to point from v_{in} to v_{out}. To make sure that there's no way to construct a path that goes into v_{in} and leaves without going to v_{out}, it is sufficient to make v_{in} have no other outgoing edges. To make sure that there's no way to construct a path that goes through v_{out} without having just come from v_{in}, it is sufficient to make v_{out} have no other incoming edges. This gives us the constraint we want.

With this in mind, we can now write the reduction to the min s-t cut problem:

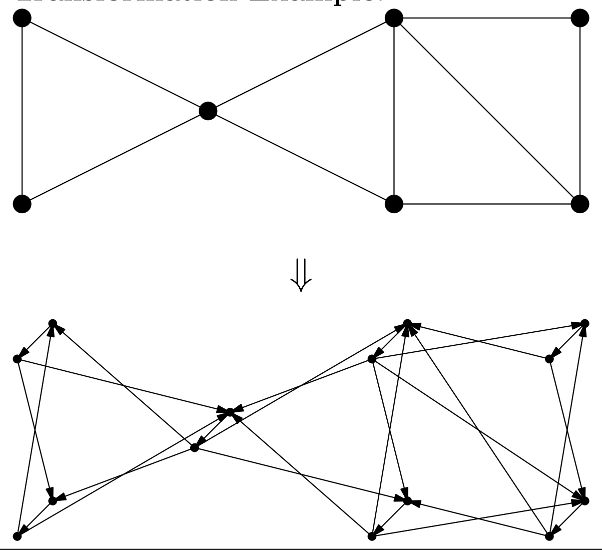
```
AlienReduction($G = (V, E), A, H$)
1   create an empty directed graph $G' = (V', E')$ with capacities
2   for each vertex $v \in V$
3         create vertices $v_{in}, v_{out} \in V'$
4         create directed edge $(v_{in}, v_{out}) \in E'$ with capacity 1
5   for each edge $(u, v) \in E$
6         create directed edge $(u_{out}, v_{in}) \in E'$ with capacity $\infty$
7         create directed edge $(v_{out}, u_{in}) \in E'$ with capacity $\infty$
8   create a new node $s \in V'$
9   for each vertex $v \in A$
10        create directed edge $(s, v_{in}) \in E'$ with capacity $\infty$
11  create a new node $t \in V'$
12  for each vertex $v \in H$
13        create directed edge $(v_{out}, t) \in E'$ with capacity $\infty$
14  return $\langle G', s, t\rangle$
```

And using this reduction as a subroutine, we can write the algo-

Transformation Details.

- turn vertex v into v_{in}, v_{out}
- create edge (v_{in}, v_{out}) with capacity 1
- for each edge (u, v), new edges:
  - edge (u_{out}, v_{in}) with capacity ∞
  - edge (v_{out}, u_{in}) with capacity ∞
- total: $2n$ vertices, $2m + n$ edges

Transformation Example.



rithm for solving our alien problem:

```
AlienProblem($G = (V, E), A, H, k$)
1  set $\langle G', s, t\rangle$ = AlienReduction($G, A, H$)
2  run Ford-Fulkerson($G', s, t$) to find the minimum cut
3  if the cut size is $> k$
4      return Nil
5  else
6      create an empty set $R$
7      for each edge $(v_{in}, v_{out})$ in the cut
8          add the vertex $v$ to $R$
9      while $|R| < k$
10         add an arbitrary new vertex to $R$
11     return $R$
```

The runtime of this algorithm is $\Theta(n+m)$ to perform the transformation, and $\Theta(f^*(n+m))$ to run Ford-Fulkerson. The maximum flow f^* is equal to the minimum cut, which is less than all other cuts. Because of the way that we have constructed the graph, the cut $S = \{v_{in} | v \in V\} \cup \{s\}$ ensures that the total capacity of all edges crossing from S to $V' - S$ is $|V| = n$, which gives us an upper bound on the flow. So our algorithm will get $\Theta(n(n+m))$, which is $\Theta(mn)$ for a connected graph.

1.4 Reflect and Improve

We have now constructed an algorithm. There is no randomization involved — the algorithm is always correct, so there isn't room for improvement there. But what about the runtime? Is there some way to improve it? Well, let's look back at the runtime analysis. We said that the algorithm takes time $\Theta(f^* \cdot m)$, and that $f^* \leq n$. But in many cases, we'll have even tighter bounds. If there is a way to position the k guards, then the value of the flow that we get is the capacity of the cut that we get, which is $\leq k$. So it's only when the algorithm returns Nil that it might have to spent $\Theta(mn)$ time — in all other cases, we can spend only $\Theta(km)$ time, which is better because $k \ll n$.

Runtime.
- Ford-Fulkerson: $O(f^* m)$
- runtime: $O(mn)$
- IMPROVEMENT: stop after k+1 iterations
- runtime: $O(km)$

Is there some way that we can improve the runtime in that one bad case? The answer is yes. Once we know that the value of the flow is $\geq k+1$, we don't need to keep running Ford-Fulkerson. So we can either modify Ford-Fulkerson to use $k+1$ iterations, or we can modify the graph that we create so that the flow from s to t is limited to $\leq k+1$ (by adding a replacement sink t' and a new edge (t, t') with capacity $k+1$). This yields an algorithm with runtime $\Theta(km)$ in all cases.

2 Simple Path of Length 7

Now that we've settled the alien problem, we can move on to a new problem.

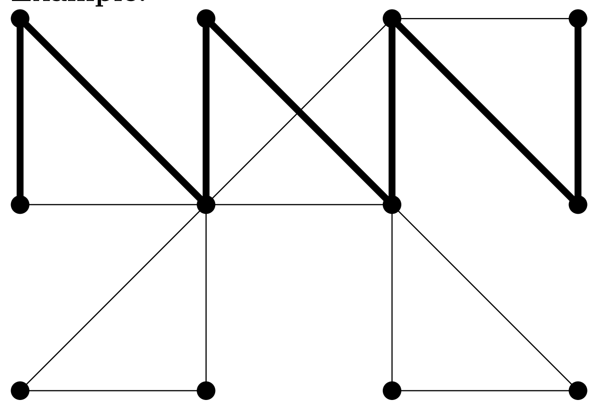
Simple Path of Length 7.
- given unweighted undirected graph
- return any simple path of length 7
- return NIL if no such path exists

Problem 2. *Given an unweighted undirected graph $G = (V, E)$, return any simple path of length* 7*, or return* NIL *if no such path exists.*

2.1 Understanding the Problem

We begin, of course, by constructing an example such as the one at the right. This example will help us understand what it is that we're searching for. The bold lines show a simple path of length 7. Note that the problem doesn't specify anything about the start or the end of the path — as long as it has length 7. So in the depicted graph, there are multiple solutions.

Example.



One crucial element of understanding this problem is understanding what it means for a path to be simple. A non-simple path in a graph is a path in which some vertex occurs more than once. In other words, a non-simple path contains a loop. A simple path contains no loops, and does not contain any vertex more than once. Therefore, every simple path of length 7 consists of a sequence of unique vertices of length 8 connected by edges.

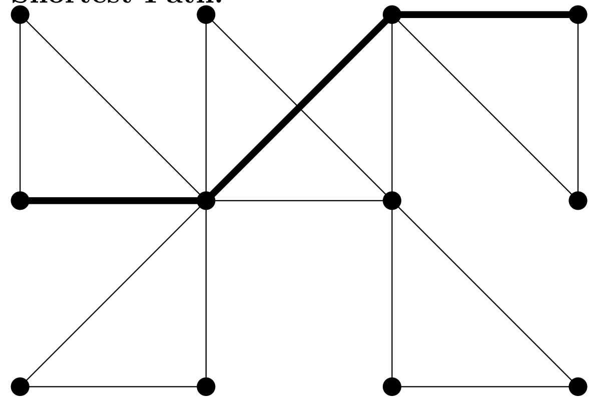
With this knowledge, we can come up with a very rough upper bound. If we examine all sequences of 8 unique vertices and check to see whether the necessary edges exist for that sequence, then we can find the correct answer. The runtime of this brute-force algorithm is $O(V^8)$, which gives us a very rough upper-bound for the problem. But at least we know it's polynomial!

Upper Bound.
- for each length-8 sequence of unique vertices
  - check whether all 7 edges exist
  - if so, return the path
- runtime: $O(V^8)$

2.2 Trying to Solve the Problem

Most of the path problems that we've seen so far involve shortest paths of some sort. Dijkstra's and Bellman-Ford can be used to find single-source shortest paths in weighted graphs. Floyd-Warshall and Johnson's algorithm can be used to solve the all-pairs shortest path problem. In an unweighted graph, performing breadth-first search is equivalent to computing single-source shortest paths. Depth-first search (which you probably saw in 6.006) is one of the only graph search algorithms that doesn't involve shortest paths.

Shortest Path.



So it's important to think about how the simple paths we want in this problem relate to shortest paths. For instance, can we use shortest paths to solve this problem?

At the right, we show the shortest path between the ends of the path of length 7 in the example that we picked. As you can see, the shortest path has length 3. In fact, careful examination reveals that all shortest paths in this graph have length ≤ 3. Unfortunately, this means that we're unlikely to be able to use shortest path algorithms to solve this problem.

In fact, after further examination of the problem, it seems that the problem might actually be more closely related to the longest simple path problem. Suppose that we have the longest simple path in the graph G. If that path has length $= 7$, then we're set — we've found the path we want. If that path has length > 7, then we can take a subpath of length 7. Otherwise, there is no simple path of length 7, and we can safely return Nil.

Relationship to Longest Path.
- let ℓ be the length of the longest path
- if $\ell = 7$, return path
- if $\ell > 7$, return subpath
- if $\ell < 7$, return Nil

Unfortunately, the longest simple path problem is quite difficult to solve. (For those of you who know complexity, you may recall that it's NP-hard to find the length of the longest simple path in general graphs.) So this line of reasoning (reducing to the longest path problem, which is actually even harder for us to solve) looks like something of a dead end.

However, the relationship between longest paths and shortest paths is an interesting one. We have a bunch of shortest-path algorithms that will find the shortest path. Some of these algorithms even handle negative-weight edges. One fairly well-known technique for finding longest paths in certain graphs is negating all of the edge weights and running a shortest-path algorithm.

What happens if we give all of our edges a weight of -1 and try to find longest paths? Unfortunately, that's not going to work — in undirected graphs, it's possible to zig-zag across any edge of weight -1 to make a path arbitrarilty short, so none of our shortest-path algorithms work when the graph is undirected. But even if we had a directed graph, this technique wouldn't work — any negative-weight loop will cause our shortest-path algorithms to fail. And because all of our edges have weight -1, every loop has negative weight. So in order to be able to compute longest paths in this fashion, we'd need a directed acyclic graph.

Let's simplify the problem by imagining that our input is not in fact an undirected graph, but is instead a DAG. Fortunately, computing path lengths in a DAG is easy because of the topological ordering associated with each DAG. This topological ordering makes it easy to maintain the invariant that when you are examining vertex v, all of the predecessors of v in the DAG have already been processed. As a result, the runtime of shortest or longest paths in a DAG is $O(V + E)$. (At right is a sketch of the algorithm.)

Longest Path on a DAG.
- GOAL: longest path in a DAG $G = (V, E)$
- for each vertex v in topological order:
  - set $d(v) = 0$
  - for each edge (u, v):
    - set $d(v) = \max\{d(v), 1 + d(u)\}$
- runtime: $O(V + E)$

Unfortunately, while this DAG result is nice, it doesn't solve the original problem. But it would be nice if we could use it in some way. For instance, is there a way to take our original graph $G = (V, E)$ and use it to construct a DAG so that the DAG contains a simple path of length 7 exactly when the original graph does?

It seems like it would be ideal to make a DAG containing all of the vertices of our original graph, with directed versions of all the edges. That way we don't accidentally toss out an edge that belongs to the path of length 7. But how can we figure out which way to direct all of the edges? We can use a permutation on all of the vertices: use the permutation as the topological order, and then make all of the

Using DAGs.
- GOAL: reduce to DAG solution
- each permutation yields a different DAG
- no principled way of orienting edges
- pick a random permutation

edges point from the node earlier in the selected ordering to the node later in the selected ordering.

But how would we pick such an ordering? There are $n!$ different orderings, where $n = |V|$, and each ordering could potentially yield a different DAG. So we certainly can't try all of them. And we don't really know anything about how to pick a "good" permutation — one containing a path of length 7. So rather than trying to figure out what a "good" permutation looks like, we're just going to pick a random one, and see if we get something reasonable when we compute the probabilities.

2.3 Working Out the Details

We can put together the details of everything that we've worked out to get the following pseudocode:

```
SIMPLEPATH7($G = (V, E)$)
 1  create a directed graph $G' = (V', E')$
 2  add all of the vertices in $V$ to $V'$
 3  pick a random ordering $\pi$ of $V$
 4  for each edge $(u, v) \in E$
 5      if $\pi(u) < \pi(v)$
 6          add $(u, v)$ to $E'$
 7      else
 8          add $(v, u)$ to $E'$
 9  for each vertex $v \in V$ in order of $\pi$
10      set $d(v) = 0$
11      for each edge $(u, v) \in E'$
12          set $d(v) = \max\{d(v), 1 + d(u)\}$
13          if $d(v) = 7$
14              return the corresponding path
15  return NIL
```

The runtime of this is $\Theta(V + E)$, which seems pretty good compared to our very high upper bound. But before we can conclude that this algorithm is good, we must analyze the probability of correctness. Note that the structure of this algorithm means that it cannot compute false positives: the only error that it can make is concluding that there is no path when there actually is one.

Suppose that $(v_1, v_2), (v_2, v_3), \ldots, (v_7, v_8)$ is a path of length 7 in our original graph. What's the probability that the path will also exist in the random DAG that we generate? Well, the path will also exist in the DAG if the process of creating the DAG caused all of the edges in the path to point in the same direction. In order for that to be true, the random order that we select should contain as a subsequence either $v_1, v_2, \ldots, v_8$ or $v_8, v_7, \ldots, v_1$. In other words, of the 8! different orderings of $v_1, \ldots, v_8$ within the larger permutation, exactly two orderings give us the result we want. This means

Probability of Correctness.
- worst case: only one path
- $(v_1, v_2), (v_2, v_3), \ldots, (v_7, v_8)$
- need all edges oriented correctly
- need path to occur in permutation in correct order
- probability: $\frac{2}{8!} = \frac{1}{20160}$

that $\frac{2}{8!} = \frac{1}{20160}$ of the $n!$ different permutations are going to preserve the path. The permutation is selected at random, so we have $\Pr[\text{success}] \geq \frac{1}{20160}$.

2.4 Reflect and Improve

The probability of success that we have is pretty small. Usually, it's good to have success probability $\geq \frac{1}{2} + \epsilon$. So let's try to improve our previous algorithm by repeating it t times. If at least one of those t iterations results in a simple path of length 7, we can return the path. Otherwise, we'll return NIL.

So what is the probability of correctness of this amplified algorithm? Well, the only way we could return the wrong answer is by saying there isn't a path when there is one. This would require us to not find that path in all t iterations. So the probability is:

$$\begin{aligned}\Pr[\text{failure of all trials}] &= (1 - \Pr[\text{success of one trial}])^t \\ &= \left(\tfrac{20159}{20160}\right)^t\end{aligned}$$

If we want this failure probability to be $< \frac{1}{3}$, then we can solve for t to get $t \approx 23000$. This yields an algorithm with $\Pr[\text{correctness}] > \frac{2}{3}$ and runtime $\Theta(t(V + E)) = \Theta(V + E)$.

Better Probability.
- run algorithm t times
- probability of always failing: $\left(\frac{20159}{20160}\right)^t$
- when $t \approx 23000$, probability of failure $< 1/3$
- runtime: $O(t(V + E)) = O(V + E)$