Chapter 1

Bootstrapping Techniques

Chapter by J. Matoušek and A. Steger.

Keywords: randomized algorithms, Monte Carlo algorithms, error probability, edge contraction, minimum cut, minimum spanning tree, bootstrapping.

In computer science we use the term bootstrapping for the process of building complex systems from simple ones[1]. In this chapter we consider two examples that show how such an approach can be used to improve the time complexity of the algorithm. Before we dive into details we outline the two approaches that we will consider.

For a (randomized) algorithm $\mathcal{A}$ we use $t_{\mathcal{A}}(n)$ to denote the maximum running time for inputs of size n. Similarly, we denote by $p_{\mathcal{A}}(n)$ the probability that, for inputs of size n, algorithm $\mathcal{A}$ returns the correct result. If the algorithm is clear from the context, we usually omit the use of $\mathcal{A}$ and simply write $t(n)$ resp. $p(n)$.

Suppose we can show that the function $t(n)$ satisfies the following recursion[2]:

$$t(n) \leq \alpha n + \beta t(n/2), \quad \text{for } n \geq 2 \qquad \text{and} \qquad t(1) = 1.$$

As you may recall from the introductory lectures, the value of β (but not the one of α) has an important influence on the asymptotic value of $t(n)$.

[1] bootstrap = *Schnürsenkel*

[2] In order to be precise we would need to use Gauss brackets; to keep things simple we ignore this issue here and instead tacitly assume that n is a power of 2.

Indeed, we have (recall that we assumed that n is a power of 2):

$$t(n) = \alpha n + \beta\alpha\frac{n}{2} + \beta^2\alpha\frac{n}{4} + \ldots + \beta^{\log_2 n}\alpha \underbrace{t(1)}_{=1}.$$

If $\beta < 2$, then we can use $\sum_{i=0}^{\log_2 n}(\beta/2)^i \leq \sum_{i=0}^{\infty}(\beta/2)^i = 1/(1-\beta/2)$ to deduce that $t(n) = O(n)$ is linear. If $\beta = 2$, then each of the $\log_2 n$ summands is exactly *equal to* αn and we thus have $t(n) = O(n \log n)$. If $\beta > 2$, then the summands are increasing and the asymptotic solution is given by the largest term, i.e. $t(n) = O(n^{\log_2 \beta})$, implying, for example, that for $\beta = 2^k$ we have that $t(n) = O(n^k)$. We conclude: in analyzing recursive algorithms we have to be very careful in keeping track of constants. In fact, our main goal in Section 1.1 is to reduce (in a recursion similar to the one from above) the value of β from 1 to a constant smaller than one (which in turn reduces the runtime of the algorithm by a log-factor).

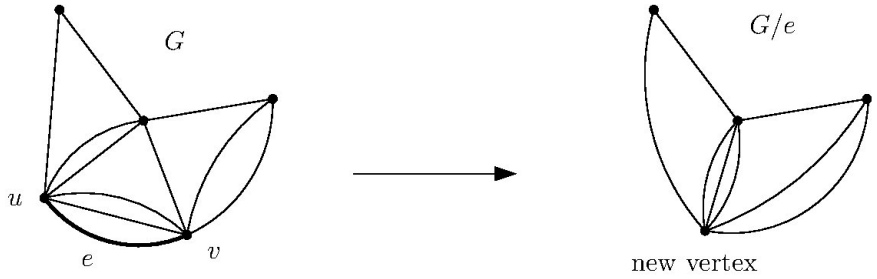
In our second example we will consider a (randomized) algorithm that reduces an instance of size n to an instance of size $n-1$. However, due to the randomness of the algorithm, assume that the probability that this reduction is correct (meaning that the correct solution for the original problem can be read off from the reduced problem) is $p(n) = 1 - 1/n$. If we use this algorithm recursively to compute the solution for a problem of size n, then the probability that we can read off the correct solution from the obtained solution of size 1 is given by the *product* of all the individual probabilities (as we need that *all* reductions are correct). Thus, the overall probability is only

$$(1-\frac{1}{n}) \cdot (1-\frac{1}{n-1}) \cdot \ldots \cdot (1-\frac{1}{2}) = \frac{1}{n}.$$

Again, you may recall from the introductory lectures, that this means that we have to repeat the algorithm a linear number of times, to reduce the error probability to a constant. Our goal for Section 1.2 is to show that we actually can do better.

Notation. As usual, we denote by $G = (V, E)$ an *undirected* graph (without loops) and use $n = |V|$ for the number of vertices and $m = |E|$ for the number of edges. Usually, we consider graphs without multiple edges. If we do allow multiple edges (in this chapter in Section 1.2) we will say so explicitly.

Edge contraction. In this chapter we repeatedly use an operation called *edge contraction*. Let G be a (multi)graph and let $e = \{u, v\}$ be an edge of G. The *contraction* of e means that we "glue" u and v together into a single new vertex and remove loops that may have arisen in this way (multiple edges are retained). The resulting graph is denoted by G/e. This is illustrated in the following picture:



Note that every edge contraction reduces the number of vertices of the graph by exactly 1 and the number of edges by at least 1.

1.1 Computing Minimum Spanning Trees in Linear Time

Given a connected graph $G = (V, E)$ with weights $w : E \to \mathbf{R}$ on the edges, a minimum spanning tree is a subgraph $T = (V, E_T)$ of G on the same vertex set such that T is a tree and such that the sum $\sum_{e \in E_T} w(e)$ of the weights of the edges in T is as small as possible.

It is well known that a minimum spanning tree can be computed in time $O(n \log n + m)$ by Prim's algorithm (together with Fibonacci heaps) and in $O(m \log n)$ by Kruskal's algorithm (together with a union find data structure). In this section we describe a randomized algorithm that computes a minimum spanning tree in $O(m)$.

Before we start we collect some well known (and easy to prove) facts on minimum spanning trees (MST):

- If the weight function $w : E \to \mathbf{R}$ is injective (i.e., no two edges have the same weight), then the graph $G = (V, E)$ contains exactly one MST. To make our life easier we will assume this case from now on. (Note that this is not a strong assumption, as by adding to each edge weight a randomly chosen (small) value ε_i, this property will hold with very high probability.)

- For every vertex $v \in V$ let $e_{\min}(v)$ denote that edge incident to v that has minimum weight (note that by our assumption on the uniqueness of the weights the edge $e_{\min}(v)$ is uniquely defined). Then all edges $e_{\min}(v)$ for $v \in V$ belong to the MST.
- For every cycle C let $e_{\max}(C)$ denote the edge in C that has maximum weight. Then no edge $e_{\max}(C)$ can belong to the MST.

The last observation can easily be turned into an algorithm that checks whether a given tree $T = (V, E_T)$ is a *minimum* spanning tree in the graph $G = (V, E)$. Indeed, every edge $e \in E$ that does not belong to T will close a unique cycle with T that consists of e and tree edges. We call such an edge T-*heavy*, iff all other edges in this cycle have a lower weight. Clearly,

$$T \text{ is a minimum spanning tree} \quad \Leftrightarrow \quad \text{all edges } e \in E \setminus E_T \text{ are } T\text{-heavy}.$$

Turning this idea into an efficient algorithm is tedious and requires clever use of appropriate data structures. But it can be done: there exists an algorithm that finds all T-heavy edges in time $O(m)$, cf. King: *A simpler minimum spanning tree verification algorithm*, Algorithmica, 1997, 263-270. In this section we use FINDHEAVY to denote such an algorithm.

The second observation from above is the basis of Borůvka's algorithm for computing an MST:

BORŮVKA'S ALGORITHM(G):
  $\forall v \in V$: compute $e_{\min}(v)$, insert $e_{\min}(v)$ in the MST and contract $e_{\min}(v)$
    (removing loops and double edges by keeping only the cheapest edge)
  recurse (until the graph contains only one vertex)

Every iteration of Borůvka's algorithm can be implemented in $O(m)$ time and reduces the number of vertices by at least a factor of two. Thus, the number of iterations is bounded by $\log_2 n$ and we thus obtain a total running time of $O(m \log n)$. Our goal for this section is to get rid of the factor $\log n$.

The problem with Borůvka's algorithm is that we reduce the number of vertices by factors of two, but we have no control on the reduction of the number of edges: we can only guarantee that we reduce m edges to at most $m - n/2$ edges, which, for $m \gg n$, is still roughly equal to m.

In order to attack this problem we first observe that Borůvka's algorithm also works for cases in which the graph is not connected. It then returns minimum spanning trees for each component. Henceforth we call such a collection of minimum spanning trees a minimum spanning forest (MSF). With this notation at hand we adapt Borůvka's algorithm as follows:

RANDOMIZED MINUMUM SPANNING TREE ALGORITHM(G):
Perform three iterations of Borůvka's algorithm (which reduces the number of vertices to at most $n/8$)
In the new graph:
Select edges with probability $1/2$ and compute recursively an MSF for the graph consisting of the selected edges.
Call this forest T.
Use FINDHEAVY to find all unselected edges that are not T-heavy.
Add all edges that are *not* T-heavy to T and delete all other edges.
recurse (until the graph contains only one vertex)

The correctness of the algorithm is immediate: we only delete edges during Borůvka steps (which is ok, as they close cycles with edges that we know have to be in the MST) and edges that are T-heavy (which is ok, as we know that they are the heaviest edge in a cycle and thus cannot belong to the MST).

We will show below that in the second step the expected number of edges that are not T-heavy can be bounded by $n/8$. Assuming this for now the expected running time of the modified algorithm can easily be shown to be linear by induction as follows.

Assume we want to show that the expected running time of the modified algorithm is bounded by $C(n+m)$ for an appropriate constant $C > 0$. Then we have to show:

$$C_B(n+m)+m+C(n/8+m/2)+C_{FH}(n/8+m)+C(n/8+n/4) \stackrel{!}{\leq} C(n+m),$$

which is easily seen to be true for an appropriate choice of C. Here $C_B(n+m)$ bounds the running time of the three Borůvka steps, m corresponds to the selection of the edges, $C(n/8+m/2)$ bounds the running time of the recursive call in the second step of the algorithm, $C_{FH}(n/8+m)$ bounds the running time of the FINDHEAVY algorithm for computing the T-heavy

edges, and $C(n/8+n/4)$ corresponds to the recursive call in the third step of the algorithm (for a graph with at most $n/8$ vertices and (in expectation) at most $n/8+n/8$ edges).

It thus remains to show that the expected number of edges that are not T-heavy (and that we therefore have to keep) can indeed be bounded in the claimed way. To this end consider a graph $G=(V,E)$ on n vertices and m edges and assume that the edges are ordered according to their weight: $w(e_1)<\ldots<w(e_m)$. Now we run Kruskal's algorithm on this graph and intertwine the selection of a random subgraph with the run of Kruskal's algorithm. That is, we will compute: *(i)* a subgraph $G'=(V,E')$ of G such that each edge of G belongs to G' independently with probability $1/2$, *(ii)* a minimum spanning forest T for G', and, finally, *(iii)* a subset $F\subset E\setminus E'$ that contains all edges that are not T-heavy (and possibly some more). From the run of the algorithm it will follow that the expected number of edges in F is bounded by n, which will then conclude the proof[3].

Here is a formal description of this algorithm:

```
(1) Let E' = T = F = ∅.
(2) for i = 1,...,m do:
(3)     if e_i connects two components of T then
(4)         Flip a fair coin in order to decide whether e_i belongs to G'.
            If so, let E' := E' + e_i and T := T + e_i, otherwise let F := F + e_i.
(5)     else
(6)         Flip a fair coin in order to decide whether e_i belongs to G'.
            (Observe that e_i is T-heavy and thus cannot belong to F.)
```

Note that we can add at most $n-1$ edges to T during the execution of (4), as T is a spanning forest in a graph on n vertices. We claim that this implies that we add in expectation at most n edges to F. Observe that in each iteration of the if-case we put edges randomly either into T or into F. That is, the expected number of edges in T and in F will be the same at any point in time. Our claim thus seems at least intuitively plausible. Formally, we have to be more careful, as the total number of iterations of

[3] Note that, for simplicity of notation, we assumed here that the graph that we consider contains n vertices. Within the randomized algorithm from above the graph contains at most $n/8$ vertices, the n thus corresponds to $n/8$, which is exactly the bound that we wanted to show.

the if-statement *depends* on the outcome of the coin flips. Exercise 1.1 makes this precise and asks you to complete the proof of our claim.

The algorithm in this section is from Karger, Klein and Tarjan, *A randomized linear-time algorithm to find minimum spanning trees*, Journal of the ACM, 1995, 321 - 328.

Exercise 1.1. *Consider a sequence $(X_i)_{i\in\mathbb{N}}$ of independent Bernoulli random variables with $\Pr[X_i = 1] = 1/2$. For an integer K we define the random variable T as the first point in time where we have seen K zeros. That is, we let*

$$T := \min\{n \in \mathbf{N} \mid |\{i \in \mathbf{N} \mid X_i = 0\}| \geq K\}.$$

By Z we denote the number of ones that we have seen by time T. That is,

$$Z := \sum_{i=1}^{T} X_i.$$

(i) *Show that $\mathbf{E}[Z] = K$.* (ii) *Argue why this implies that the expected number of edges in F is at most n.*

1.2 Computing Minimum Cuts in $\tilde{O}(n^2)$ time

A cut[4] in a graph $G = (V, E)$ is a subset $C \subseteq E$ of edges such that the graph $(V, E \setminus C)$ is disconnected. A *minimum cut* is a cut with the minimum possible number of edges. A graph may have several minimum cuts. We let $\mu(G)$ denote the size of a minimum cut in G.

In the lecture *Algorithmen und Wahrscheinlichkeit* you saw two ways to solve this problem. One can use a flow-based algorithm to find a minimum cut that separates two given vertices s and t and apply this algorithm for an arbitrary but fixed vertex s and all other $n-1$ vertices $t \in V \setminus \{s\}$, resulting in an $O(n^4 \log n)$ algorithm. We also saw a randomized algorithm that solves the problem with high probability in $O(n^4)$ time. Our aim in this chapter is to improve this latter algorithm to achieve a runtime of $O(n^2(\log n)^3)$[5]. For sake of completeness of these lecture notes we first review the $O(n^4)$ algorithm.

[4] More precisely one might speak about *edge cuts*. Sometimes *vertex cuts* are also considered, which are subsets of vertices whose removal disconnects the graph.

[5] To emphasize the main term of the running times of algorithms one often uses the

1.2.1 Basic Version

In this section we tacitly assume that the input graph is connected; we do allow multiple edges. The algorithm maintains a multigraph and performs successive edge contractions. We assume that the current multigraph is represented in such a way that we can

- perform an edge contraction in $O(n)$ time,
- choose an edge uniformly at random among all edges of the current multigraph in $O(n)$ time, and
- find the number of edges connecting two given vertices in $O(1)$ time (actually, $O(n)$ would be sufficient as well).

Designing a suitable data structure is not trivial, but it is not difficult either and we leave it as an exercise.

Minimum cuts and contraction. The correctness of the algorithms will rely on the following simple observation, whose proof is left to the reader (and we strongly advise the reader to think this over carefully).

Observation 1.1. *Let G be a multigraph and e an edge of G. Then $\mu(G/e) \geq \mu(G)$. Moreover, if there exists a minimum cut C in G such that $e \notin C$, then $\mu(G/e) = \mu(G)$.*

The following algorithm repeatedly chooses random edges of the current graph and contracts them, until only two vertices are left:

```
BasicMinCut(G):
  while G has more than 2 vertices do
      pick a random edge e in G
      G ← G/e
  end while
  return the size of the only cut in G
```

The contracted edge is always picked uniformly at random among all edges of the current graph.

notation $\tilde{O}(\cdot)$. This notation is similar to the Big-Oh notation, except that we hide not only constants but also log-factors. Thus, $O(n^2(\log n)^3)$ can be abbreviated by $\tilde{O}(n^2)$, cf. the title of this section.

By the first part of Observation 1.1, the algorithm always returns a number at least as large as $\mu(G)$. If C is a minimum cut in the input graph G, and if we never contract an edge of C during the whole algorithm, then the returned number is exactly $\mu(G)$, by the second part of the observation.

At first sight it looks foolish to hope that no edge of C is ever contracted. After all, to this end, the random choice would have to come out "right" (avoid C) in each of $n-2$ steps. Common sense suggests that making $n-2$ successful random choices in a row is extremely unlikely. The beautiful insight is that in the considered case, a sequence of such "right" choices, while somewhat unlikely, is actually not *extremely* unlikely.

Lemma 1.2. *Let G be a multigraph with n vertices. Then the probability of $\mu(G) = \mu(G/e)$ for a randomly chosen edge $e \in E(G)$ is at least $1 - \frac{2}{n}$.*

Proof. Let us write $k = \mu(G)$ and let us fix a minimum cut C in G, with $|C| = k$. We note that every vertex of G has degree at least k, and thus $|E(G)| \geq \frac{nk}{2}$. By Observation 1.1, the probability of $\mu(G) = \mu(G/e)$ is bounded below by the probability of $e \notin C$, which equals

$$1 - \frac{|C|}{|E(G)|} \geq 1 - \frac{k}{\frac{nk}{2}} = 1 - \frac{2}{n}.$$

□

Next, for a multigraph G, let $p_0(G)$ denote the probability that algorithm BasicMinCut succeeds, i.e., returns $\mu(G)$. Let $p_0(n)$ denote the minimum of $p_0(G)$ over all multigraphs on n vertices (we should really say infimum, since we deal with infinitely many graphs).

We have $p_0(2) = 1$ as the base case (nothing is contracted for $n = 2$). For $n > 2$ we note that BasicMinCut succeeds for G exactly if the following two events occur:

E_1: $\mu(G) = \mu(G/e)$ for the first contracted edge e, and

E_2: BasicMinCut succeeds for G/e.

The probability of E_1 is at least $1 - \frac{2}{n}$ by the above lemma. Given that E_1 occurred, the probability of E_2 is always at least $p_0(n-1)$. Therefore,

we get the recurrence[6]

$$p_0(n) \geq \left(1 - \frac{2}{n}\right) p_0(n-1).$$

Then we compute

$$p_0(n) \geq \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdots \frac{2}{4} \cdot \frac{1}{3} \cdot p_0(2) = \frac{2}{n(n-1)}.$$

Probability amplification. This bound for the success probability doesn't look very optimistic (the algorithm may fail "almost always"). However, we can run the algorithm N times and return the smallest cut size found in all these runs. If the returned cut size is not correct, it means that the algorithm failed N times in a row. The failures in different runs are independent, and hence the probability of N failures in a row is bounded by

$$\left(1 - \frac{2}{n(n-1)}\right)^N \leq e^{-2N/n(n-1)},$$

where we have used the well-known (and important) inequality $1 + x \leq e^x$ with $x = -\frac{2}{n(n-1)}$.

If we set, say, $N = 10n(n-1)$, then the failure probability is bounded above by $e^{-20} < 10^{-8}$. By increasing the number N of repetitions, the failure probability can be further decreased (observe that doubling N squares the bound for the failure probability).

Altogether we have a randomized minimum cut algorithm with running time $O(n^4)$ and with a very small probability of failure. This running time is comparable to some of the simpler flow-based algorithms. In the next section we will improve on it substantially.

We conclude this section with a remark. We have formulated the algorithms in such a way that they compute only a number, the *size* of a minimum cut. What if we also want to find a minimum cut? The algorithms can easily be modified: We keep track of the identity of edges

[6] A formal derivation involves conditional probabilities:

$$\Pr[E_1 \text{ and } E_2] = \Pr[E_1] \cdot \Pr[E_2 \mid E_1] \geq \left(1 - \frac{2}{n}\right) p_0(n-1).$$

Here we use that the probability of E_2 is at least $p_0(n-1)$, no matter which edge e was contracted. Note that E_1 and E_2 need not be independent in general!

during contractions, and then from the cut in the final 2-vertex graph we can reconstruct the corresponding cut in the input graph. We have avoided this issue so far in order to keep the presentation simpler.

1.2.2 Bootstrapping

The first key observation towards an improved running time is that *the probability of contracting an edge in the minimum cut increases as the graph shrinks.* At first the probability is quite small, only $\frac{2}{n}$, but near the end of execution, when the graph has only three vertices, it can be as high as $\frac{2}{3}$. This suggests that we contract edges until the number of vertices decreases to some suitable threshold value t, and then we use some other, perhaps slower algorithm that guarantees a higher probability of success.

What slower but more reliable algorithm do we have at disposal? The key inside is that we actually do not have to use any *different* algorithm, we just use the *same* algorithm, but with a smaller error probability (that we know we can get by repeated calls, cf. previous section).

In the remainder of this section we first provide an abstract argument how such an approach can be used to get an algorithm with a runtime is arbitrarily close to $O(n^2)$. In the later part of this section we will then provide a more explicit statement of such an algorithm.

A sequence of faster and faster algorithms. We claim that there exists a sequence of algorithms $(\mathcal{A}_i)_{i\geq 0}$ such that for all $i \geq 0$ algorithm $\mathcal{A}_i$ finds a minimum cut in time $O(n^{f(i)})$ with probability at least $1/2$, where $f(0) = 4$ and $f(i+1) = 4(1 - 1/f(i))$ for $i \geq 0$. (Recall that the constant $1/2$ is arbitrary: by probability amplification, cf. last section, we can increase it to any constant less than one without increasing the asymptotic runtime of the algorithm.)

We will prove our claim by induction on i. For $i = 0$ there is nothing to show, as our basic version from the last section does the job. So assume we have already proved the existence of algorithm $\mathcal{A}_i$ for some $i \geq 0$. We will show that then also algorithm $\mathcal{A}_{i+1}$ exists. We construct this algorithm explicitly.

$\mathcal{A}_{i+1}(G)$:	RandomContract(G,t):

set parameters t and N suitably
repeat N times:
$H \leftarrow$ RandomContract(G,t)
call $\mathcal{A}_i(H)$
return smallest value
	while $\|V(G)\| > t$ do
	for random $e \in E(G)$
	$G \leftarrow G/e$
	end while
	return G

As before we conclude from Lemma 1.2 that the probability that $\mu(H) = \mu(G)$ is at least

$$(1-\frac{2}{n}) \cdot (1-\frac{2}{n-1}) \cdot \ldots \cdot (1-\frac{2}{t+1}) = \frac{t(t-1)}{n(n-1)}.$$

If this is the case, then – by our induction assumption – $\mathcal{A}_i(H)$ returns the value of $\mu(H)$ with probability at least $1/2$. One iteration of algorithm $\mathcal{A}_{i+1}$ thus finds the size of a minimum cut in G with probability at least $\frac{t(t-1)}{2n(n-1)}$. The probability that none of N iterations return $\mu(G)$ is therefore bounded by

$$\left(1-\frac{t(t-1)}{2n(n-1)}\right)^N \leq e^{-N \cdot \frac{t(t-1)}{2n(n-1)}}.$$

For $N = \frac{2n(n-1)}{t(t-1)}$ this term is equal to $e^{-1} \leq 1/2$, as desired. Next we consider the runtime of the algorithm. In each iteration RandomContract(G,t) can be done in $O(n^2)$ time, while algorithm $\mathcal{A}_i(H)$ requires time $O(t^{f(i)})$. The total runtime of algorithm $\mathcal{A}_{i+1}(G)$ is thus bounded by

$$O(N \cdot (n^2 + t^{f(i)})) = O(\frac{n^2}{t^2} \cdot (n^2 + t^{f(i)})).$$

We now choose $t = n^{2/f(i)}$ (so that both terms in the bracket are equal) and obtain a runtime of $O(n^{4-4/f(i)})$, as claimed. We leave it as an exercise for the reader to show that $f(i) \longrightarrow 2$ for $i \longrightarrow \infty$.

An Explicit Algorithm. The above arguments shows that there *exists* an algorithm with runtime very close to $O(n^2)$. But how does such an algorithm actually look like? The important observation is that within an algorithm $\mathcal{A}_i$ we recursively solve many min cut problems for many small graphs. In fact, the smaller the number of vertices, the larger the number of recursive calls.

Instead of trying to figure this out more precisely from the inductive argument given above, we proceed in this section the other way around.

Namely, we start with the graph G we first *duplicate* this graph and then contract random edges in both copies independently until the number of vertices is reduced to αn (for some appropriate constant $0 < \alpha < 1$ that we will determine later). Then we again *duplicate* both graphs and continue contracting edges in the now four graphs on αn vertices. After the number of vertices is reduced by another factor of alpha, we again duplicate all graphs (so that we now have eight graphs on $\alpha^2 n$ vertices), and so forth. This leads to the following scheme of our improved minimum cut algorithm:

```
MinCut(G):
  if n ≤ 16 then
      compute μ(G) by some deterministic method
  else
      t ← ⌈αn⌉ + 1
      H1 ← RandomContract(G, t)
      H2 ← RandomContract(G, t)
      return min(MinCut(H1), MinCut(H2))
  end if
```

We have introduced the "border case" $n \leq 16$ into the algorithm in order to simplify the analysis (and 16 is just a rather arbitrary constant that will turn out to be convenient in the analysis). In practice, we could actually go on with contractions all the way down to 2-vertex graphs, but we may have to adjust the algorithm for small graphs a little (as for very small n we may have $\lceil \alpha n \rceil + 1 \geq n$).

We begin with estimating the running time, which is routine. Using our assumptions on the representation of the current graph, we know that we can construct the graphs H_j in time $O(n^2)$. We thus get the recurrence

$$t(n) \leq O(n^2) + 2t(\lceil \alpha n \rceil + 1).$$

The alert reader may now see the similarity of this recurrence with the one that we studied in the beginning of this section. Similarly, as we argued there, we see that the solution of this recurrence depends on the value of α. We leave it to the reader (cf. Exercise 1.2) to verify by induction that for $\alpha \leq 1/\sqrt{2}$ we get $t(n) = O(n^2 \log n)$.

Lemma 1.3. *For $\alpha \leq 1/\sqrt{2}$ the running time $t(n)$ of* MinCut *on an n-vertex graph is bounded by $O(n^2 \log n)$.*

The more challenging part is the proof of the success probability. Let $p(n)$ denote the minimum, over all n-vertex multigraphs G, of the probabilities $p(G)$ that MinCut succeeds, i.e., correctly returns $\mu(G)$.

Lemma 1.4. *For $\alpha \geq 1/\sqrt{2}$ there is a real constant c, $c > 1$, such that*

$$p(n) \geq \frac{1}{1 + \log_c n} \quad \text{for all } n \geq 2.$$

We will postpone the proof a little, and first we look what this gives us.

First observe that the bounds on α are complementary in the two lemmas. Our only choice in order to be able to apply both lemmas is $\alpha = 1/\sqrt{2}$. So lets fix this for now.

The success probability $p(n)$ can still be small, but we can achieve success probability close to 1 by repeating MinCut a suitable number of times. Since $p(n)$ is much larger compared to $p_0(n) \approx 1/n^2$, we need much fewer repetitions. Namely, only $O(\log^2 n)$ repetitions suffice to bring the failure probability below n^{-6}, say. The achievements of this section can thus be summarized as follows:

Theorem 1.5. *Let $a \geq 1$ be a parameter and let $\alpha = 1/\sqrt{2}$. The randomized algorithm consisting of $N = Ca \log^2 n$ repetitions of* MinCut*, where C is a suitable constant, has running time $O(n^2 \log^3 n)$ and for every n-vertex input graph it computes the minimum cut size correctly with probability at least $1 - n^{-a}$.*

Proof of Lemma 1.4. In order that MinCut fails for a given G, both of the events $\mathcal{E}_1$ and $\mathcal{E}_2$ must occur, where $\mathcal{E}_j$ means that

$$\mu(H_j) > \mu(G) \quad \textit{or} \quad \text{MinCut fails for } H_j.$$

Since $\mathcal{E}_1$ and $\mathcal{E}_2$ are independent, we have $p(G) = 1 - \Pr[\mathcal{E}_1] \Pr[\mathcal{E}_2]$.

The probability of $\mathcal{E}_j$ *not* occurring is equal to the probability that both $\mu(H_j) = \mu(G)$ occurs *and* MinCut succeeds for H_j. As both events are independent, we therefore get

$$1 - \Pr[\mathcal{E}_j] \geq \Pr[\mu(H_j) = \mu(G)] \cdot p(t).$$

As before, we use Lemma 1.2 to obtain

$$\Pr[\mu(H_j) = \mu(G)] \geq \prod_{i=t+1}^{n} \frac{i-2}{i} = \frac{t(t-1)}{n(n-1)} \geq \frac{(t-1)^2}{n^2}.$$

We now see that t was set up exactly so that the right-hand side is at least α^2. Therefore $\Pr[\mathcal{E}_j] \leq 1 - \alpha^2 p(t)$. Thus, we arrive at the following recurrence:

$$p(n) \geq 1 - \left(1 - \alpha^2\, p(\lceil \alpha n \rceil + 1)\right)^2.$$

Now it is time again to consider the choice of the parameter α. As it turns out, we need that $2\alpha^2 \geq 1$ in order to achieve a good lower bound for $p(n)$. We leave it to the reader to check this.

In the remainder of the proof we fix $\alpha = 1/\sqrt{2}$ to simplify calculations. We again leave it to the reader to check that the proof can in fact easily be generalized to larger values of α.

According to the algorithm we can suppose $p(n) = 1$ for $n \leq 16$. We prove by induction on k that $p(n) \geq 1/k$ for all $n \leq 1.2^k$. (Then, for $1.2^{k-1} < n \leq 1.2^k$, we have $p(n) \geq 1/k > \frac{1}{1+\log_{1.2} n}$ and the lemma follows, since every n has a k to make the conclusion.) The numbers 16 and 1.2 are chosen in such a way that for $n > 16$ we have $\lceil n/\sqrt{2} \rceil + 1 \leq n/\sqrt{2} + 2 \leq n/1.2$. Therefore, assuming that the inductive assumption holds for $k-1$ and that $n \leq 1.2^k$, we have $p(\lceil n/\sqrt{2} \rceil + 1) \geq \frac{1}{k-1}$. Then

$$\begin{aligned} p(n) &\geq 1 - \left(1 - \frac{1}{2} p(\lceil n/\sqrt{2} \rceil + 1)\right)^2 \\ &\geq 1 - \left(1 - \frac{1}{2(k-1)}\right)^2 \quad \text{by induction} \\ &= \frac{k - \frac{5}{4}}{(k-1)^2} > \frac{1}{k}, \end{aligned}$$

since $k(k - \frac{5}{4}) > (k-1)^2$ for $k \geq 2$. This concludes the proof of Lemma 1.4. □

The algorithm of this section is from Karger, Stein, *A new approach to the minimum cut problem*, Journal of the ACM **43**, 1996, 601-640.

We conclude this section with the remark that the best known algorithm for finding a minimum cut in a graph has runtime $O(m \log^3 n)$, cf. Karger, *Minimum cuts in near-linear time*, Journal of the ACM **47**, 2000, 46-76.

Exercise 1.2. *Prove Lemma 1.3 formally. Can we improve the bound on $t(n)$ if we assume that $\alpha < 1/\sqrt{2}$?*

Exercise 1.3. *Prove: a (multi)graph G on n vertices cannot have more than $\binom{n}{2}$ minimum cuts.*

Exercise 1.4. *You recall that the algorithm* BasicMinCut *computes a guess for the size of a minimum cut of a (multi)graph* G *by repeatedly contracting a uniformly random edge until there are only two vertices left and then returning the number of edges running between these two vertices.*

As usual, denote the size of a minimum cut of G *by* $\mu(G)$*. We have derived in the lecture that the number* L_G *which* BasicMinCut *outputs (on input* G*) is at least* $\mu(G)$*, and* $\Pr[L_G = \mu(G)] = \Omega(n^{-2})$*.*

Consider the following slightly modified algorithm BasicMinCut'*: just like* BasicMinCut*, it repeatedly contracts a uniformly random edge until there are only two vertices left. But instead of just returning the number of edges between those two vertices in the very end, it returns the smallest degree of any vertex observed during the execution of the algorithm. That is if* $G_0, G_1, G_2, \ldots, G_{n-2}$ *is the sequence of graphs encountered, with* $G_0 = G$ *and* $|V(G_{n-2})| = 2$*, it returns*

$$L_G := \min_{0 \le i \le n-2} \min_{v \in V(G_i)} deg(v).$$

Prove that

(a) BasicMinCut' *can be implemented so as to run in time* $\mathcal{O}(n^2)$*,*

(b) $L_G \ge \mu(G)$ *always holds,*

(c) for any fixed $\alpha > 0$*, the success probability*

$$p_\alpha(n) := \min_{G \text{ a graph on } n \text{ vertices}} \Pr[L_G \le (1+\alpha)\mu(G)]$$

satisfies the recurrence

$$p_\alpha(n) \ge \left(1 - \frac{2}{(1+\alpha)n}\right) p_\alpha(n-1).$$

Using (c), one can prove that for any fixed $\alpha > 0$*,* $p_\alpha(n) \in \Omega(n^{\frac{-2}{1+\alpha}})$*, but this is just calculation and we do not ask you to do this here.*