An O(N) Time Algorithm for Finding Hamilton Cycles with High Probability

Rajko Nenadov
ETH Zürich, Switzerland
raikon@gmail.com

Angelika Steger
ETH Zürich, Switzerland
asteger@inf.ethz.ch

Pascal Su
ETH Zürich, Switzerland
sup@inf.ethz.ch

Abstract

We design a randomized algorithm that finds a Hamilton cycle in $\mathcal{O}(n)$ time with high probability in a random graph $G_{n,p}$ with edge probability $p \geq C \log n/n$. This closes a gap left open in a seminal paper by Angluin and Valiant from 1979.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis; Mathematics of computing → Random graphs; Mathematics of computing → Graph algorithms; Mathematics of computing → Matchings and factors; Theory of computation → Random walks and Markov chains

Keywords and phrases Random Graphs, Hamilton Cycle, Perfect Matching, Linear Time, Sublinear Algorithm, Random Walk, Coupon Collector

Digital Object Identifier 10.4230/LIPIcs.ITCS.2021.60

Funding *Pascal Su*: This author was supported by grant no. 200021 169242 of the Swiss National Science Foundation.

1 Introduction

A Hamilton cycle is a cycle in a graph that visits every vertex exactly once. Determining whether a graph has a Hamilton cycle is a notoriously difficult problem that has been tackled in various ways. In general, it is known to be $\mathcal{NP}$-hard, putting it in a bag of complexity theory together with colorability or SAT, problems for which one has tried to find polynomial time algorithms for a long time without any success so far.

While the Hamilton cycle problem is a difficult problem in general, it turns out that for most graphs it is actually not. To illustrate this, we take a closer look at the Erdős-Rényi random graph $G_{n,p}$ which is an n-vertex graph with each edge being present independently with probability p. The existence question of the Hamilton cycle problem is very well understood, cf. the comprehensive survey by Frieze [13]. Let $\mathcal{H}$ be the set of Hamiltonian graphs, then for $G_{n,p}$ it holds that (Komlós and Szemerédi [19] and Korshunov [20])

$$Pr[G_{n,p(n)} \in \mathcal{H}] = \begin{cases} 0, & p(n) = \frac{\log(n)+\log\log(n)-\omega(1)}{n} \\ e^{-e^{-c}}, & p(n) = \frac{\log(n)+\log\log(n)+c+o(1)}{n} \\ 1, & p(n) = \frac{\log(n)+\log\log(n)+\omega(1)}{n}, \end{cases}$$

which is limitwise the same as the threshold for when $G_{n,p}$ has minimum degree 2. So really vertices of degree one are the bottleneck for random graphs. In fact, it is known that if we add the edges randomly one by one, the moment we reach minimum degree 2 is the same as

the moment the graph becomes Hamiltonian with high probability [1]. And this threshold is also robust (e.g. [22, 23]). For other random graph models like the random graph with m edges $G_{n,m}$, the random regular graph $G_{n,r}$ or the k-out which takes k random edges from every vertex the corresponding thresholds for Hamiltonicity are also known [7, 9, 11, 26, 29]. Similar to the classical random graph case also in these cases the thresholds coincides with a local obstruction such as minimum degree two or any two vertices have a neighborhood of size at least 3. And this is not a coincidence. Randomness gives us such nice expansion properties that only the small structures can be an obstruction to the Hamilton cycle. This phenomenon has been observed also for other properties such as connectivity, containing a perfect matching or colorability.

The proofs of Komlos and Szemeredi and Korshunov are just existential, i.e. they determine the threshold for the existence of Hamilton cycle, but do not provide an efficient algorithm for finding it. In a seminal paper, Angluin and Valiant [4] show that with the input given as a random adjacency list one can find Hamilton cycles in $G_{n,p}$ for $p \geq C \log n/n$ in $\mathcal{O}(n \log^2 n)$ time with high probability. There are two ways in which this result is possibly non-optimal: the lower bound on p and the runtime. The first point was considered by Shamir and then Bollobas, Fenner and Frieze, who brought the bound down to the existence threshold of $G_{n,p}$. In more recent works the runtime has also been optimized for graphs given in adjacency matrix form, assuming a pair of vertices can be queried in constant time. We summarize these results in the table below. There are various related results that are hard to compare, as their setting is slightly different [2, 12, 14, 15]. Some of the results are assuming the graph is given as an adjacency matrix with black box queries and the runtime $\mathcal{O}(n/p)$ is optimal in that model.

Authors	**Year**	**Time**	**p(n)**	Graph Model
Angluin, Valiant [4]	'79	$\mathcal{O}(n \log^2(n))$	$p \geq \frac{C \log(n)}{n}$	adj. list
Shamir [28]	'83	$\mathcal{O}(n^2)$	$p \geq \frac{\log(n)+(3+\varepsilon)\log\log(n)}{n}$	adj. list
Bollobas, Fenner, Frieze [8]	'87	$n^{4+o(1)}$	$p \geq$ *Existence threshold*	adj. list
Gurevich, Shelah [16]	'87	$\mathcal{O}(n/p)$	p const.	adj. matrix
Thomason [30]	'89	$\mathcal{O}(n/p)$	$p \geq Cn^{-1/3}$	adj. matrix
Alon, Krivelevich [3]	'20	$\mathcal{O}(n/p)$	$p \geq 70n^{-1/2}$	adj. matrix

In this paper we consider the second question that was left open in the Angluin-Valiant paper: can the runtime be improved. Note that a graph with $p \geq C \log n/n$ has $\Theta(n \log n)$ edges. Thus, improving the runtime below this bound requires a *sublinear* algorithm, i.e. sublinear in the input size. These are algorithms that produce an output without reading the input completely (see e.g. [27] for an overview of the topic). Such algorithms are less restrictive than those designed for online or a (semi-)streaming model as they allow some control over which part of an input is used. However for graphs with n vertices and $m \gg n$ edges the algorithm is only allowed to read $o(m)$ edges, i.e., a negligible fraction of the input – but nevertheless has to compute the desired output correctly.

1.1 Our contribution

In this paper we show that given a random graph with edge probability $p \geq C \log n/n$, for an appropriately chosen constant C, we can find a Hamilton cycle in $\mathcal{O}(n)$ time with high probability. This time is clearly optimal, as the algorithm has to return $\Omega(n)$ edges. We assume that the graph is given to us with randomly ordered adjacency lists, such that we can query the next neighbor in those lists for any vertex in constant time.

Figure 1 Algorithm uses a random walk like strategy, blue edge is the `newneighbor()`.

▶ **Theorem 1.** *There exists a randomized algorithm $\mathcal{R}$ which finds a Hamilton cycle in a random graph $G_{n,p}$ in $\mathcal{O}(n)$ time with high probability, provided $p \geq C \log n/n$ for a sufficiently large constant C.*

Note that "with high probability" is always meant to mean with probability $1 - o(1)$ tending to one as n tends to infinity and takes into account all sources of randomness: i.e., the randomness of the algorithm, the random graph and the randomness of the datastructure used to store the graph (random ordering of the adjacency lists).

Our paper is organized as follows. Section 2 contains the algorithm and the proof of Theorem 1. It is based on three technical lemmas that we prove in Section 3.

2 Algorithm

The most commonly used technique for efficient cycle extensions is Posa rotations. This is also the case for the original algorithm of Angluin and Valiant [4], which we outline in Section 2.1 below, cf. also Figure 2.To reduce the runtime to $\mathcal{O}(n)$ we reduce the total *number* of Posa rotations that are required and simultaneously also restrict ourselves to certain types of Posa rotations so that we can realize each of them in $\mathcal{O}(\log n)$ time.

2.1 Finding A Hamilton Cycle via Posa Rotations

We sketch here the algorithm of Angluin and Valiant. The main idea of their algorithm is to perform a greedy random walk until all vertices are incorporated in the path/cycle. This means we start from an arbitrary vertex and query a neighbor of that vertex. If the neighbor is already contained in the path we have built so far we consider this a failure and we query a new neighbor. Otherwise we add the neighbor to the path and continue from the new endpoint vertex (see Figure 1).

Once the path is long enough (at least $n/2$) we add possible Posa rotations. Assume we start with a path $P = (v_1, \dots, v_s)$, then if we find two edges such that for some index $i \in [s]$ the edges are of the form $\{v_{i+1}, v_s\}$ and $\{v_i, v_j\}$ for some $j > i + 1$, we can rearrange the path to form a new path $P' = (v_1, \dots, v_i, v_j, v_{j+1}, \dots, v_s, v_{i+1}, \dots, v_{j-1})$ and now the new path has the same vertex set but a different endpoint vertex. This we call a Posa rotation. Additionally we will always want *long* Posa rotations meaning $s - i$ must be at least $n/2$ to ensure that we can find the second edge needed quickly with high probability.

So during our Algorithm if the neighbor (v_{i+1}) of the endpoint of the path (v_s) has distance at least $n/2$ from the endpoint along the path we use that edge to build a cycle and continue from the vertex preceding the neighbor (v_i) on the path (see Figure 2). This leaves a cycle of size at least $n/2$ and if we ever find one of the vertices on the cycle to be the neighbor of the current endpoint we reincorporate the large cycle by appending it to the path (again giving a new endpoint).

Many details need to be considered on how random variables interact, etc., but leaving those aside one can easily convince oneself that on average the current vertex changes after a constant number of queries to a new random vertex, and that the number of queries until

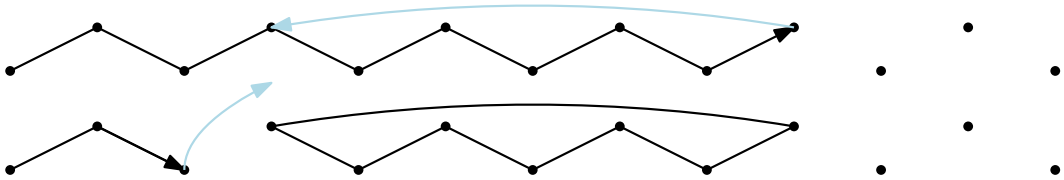

Figure 2 Posa rotation, detaching a large cycle.

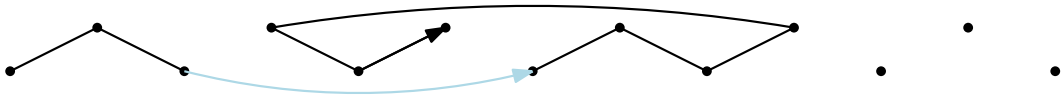

Figure 3 Reincorporating the large cycle.

the path length increases by one is geometrically distributed and has an expectation of $n/(n-i)$ where i is the current length of the path. The total number of Posa rotations is thus bounded by

$$\mathcal{O}\left(\sum_{i=1}^{n} \frac{n}{i}\right) = \mathcal{O}(n \log n).$$

As each Posa rotation takes time $\log n$ to realize this gives a total running time of $\mathcal{O}(n \log^2 n)$.

2.2 Our Algorithm

We give a short overview of the new algorithm we propose. The algorithm comes in two phases. In phase 1 we find two random perfect matchings. The union of these two random perfect matchings forms a two regular graph, i.e., a set of disjoint cycles or double edges covering all vertices. It is not difficult to show that the number of cycles is with high probability bounded by $2 \log n$. In phase 2 of the algorithm we stitch these $2 \log n$ cycles together.

For the analysis of the algorithm it is very helpful to assume that a query for a new neighbor of some vertex v returns a vertex w that is *uniformly* distributed over all vertices in $V - v$ and *independent* from all previous queries. Of course such an assumption a priori does not hold if we simply return the next vertex from the adjacency list of v. We realize this by directing the edges and resampling. More formally, we will show the following lemma in Section 3; in the remainder of Section 2 we will use the corresponding function `newneighbor()` as a black box.

▶ **Lemma 8** (newneighbor). *It is possible to interact with the graph $G_{n,p}$, $p \geq \frac{C \log n}{n}$, with an algorithmic procedure* ***`newneighbor(v)`*** *which has the following properties with high probability:*

- **(i)** *Calling* ***`newneighbor(v)`*** *returns a neighbor of v distributed uniformly among $V - v$ and independent of all calls so far – as long as we make at most $\mathcal{O}(n)$ calls to* ***`newneighbor()`*** *altogether and every vertex is queried at most $100 \log n$ times.*
- **(ii)** *The total run time of all $\mathcal{O}(n)$ calls is $\mathcal{O}(n)$.*

Note that this algorithm uses both internal randomness as well as the randomness of $G_{n,p}$. If `newneighbor(`v`)` ever returns 'there are no more neighbors' we immediately terminate the entire algorithm and return failure. To avoid this, we will prove that we query `newneighbor(`v`)` from any vertex at most $100 \log n$ times w.h.p. and choose C large enough so that with high probability the minimum degree of the random graph is large enough.

2.2.1 Phase 1: Perfect Matching

In the first phase of the algorithm we show that we can find a perfect matching in $\mathcal{O}(n)$ time. We call the algorithmic procedure described in this section `FastPerfectMatching`, see Algorithm 1. In fact, for an easier understanding of the required ideas, we work in this section with a random *bipartite* random graph. This can easily be done by partitioning the vertex set V into two equal sets A and B arbitrarily (if n is odd we set one vertex aside and include it in phase 2) and only considering the edges between A and B. Formally, the function `newABneighbor(v)` calls `newneighbor(v)` until we receive a neighbor which is in B (resp. A).

▷ Claim 2. If we call `newABneighbor()` for a sequence of $\mathcal{O}(n)$ vertices, in which every vertex $v \in A \cup B$ occurs at most $\log n$ times, then with high probability this results in at most $\mathcal{O}(n)$ calls to `newneighbor()` with at most $6 \log n$ calls per vertex.

The claim holds because any call of `newneighbor()` has probability at least $1/2$ to be in the correct partition and, by our assumptions on `newneighbor()`, the calls are independent. We can thus apply concentration bounds for binomial distributions and union bound for every vertex. Clearly, `newABneighbor()` still has a uniform and independent distribution over all vertices of the opposite partition.

Let G be the balanced bipartite graph with partitions A and B. During the algorithm we will maintain a matching M which covers some of the vertices and is empty at first. At any point in time, we denote by A_M the vertices in A that are covered by the matching and with A_0 the unmatched vertices. Equivalently for B_M and B_0.

Additionally we need a set of edges that expand well from the vertices of A. And we need to be able to keep track of them efficiently and on the fly. So for any vertex v we define the d-neighborhood of v, $N_d(v) \subseteq V(G)$, to be the set of the first $\lceil d \rceil$ calls to the function `newABneighbor(v)`. In particular this implies that for any $d' < d$ the d'-neighborhood is contained in the d-neighborhood of v. Similarly, the d-neighborhood of a set of vertices S, denoted by $N_d(S)$, is defined as the union of the d-neighborhoods of all vertices in S. We expose and keep track of the d-neighborhood of the unmatched vertices A_0, $N_{d(|A_0|)}(A_0)$, for the function $d(t) = \min(\sqrt{n/t}, \log n)$. This gives us a neighborhood large enough for the random walks to be effective, but small enough so that we do not need too much time to update/expose.

To increase the matching we call a subroutine `IncreaseMatching`. `IncreaseMatching` takes as argument the current matching M and an unmatched vertex $v \in B_0$. It proceeds as follows. If v is in $N_d(A_0)$ we add the corresponding neighbor in A_0 and v to the matching. If not we take $w =$ `newABneighbor(v)`. If w is in A_0 we add the edge $\{w, v\}$ to M. If neither of the two is the case, then $w \in A_M$ and there exists a unique u such that $\{w, u\}$ is currently in M. We swap $\{w, u\}$ for $\{w, v\}$, thereby making u a new unmatched vertex, and repeat `IncreaseMatching` with u, cf. Figure 4.

Clearly, during the run of the algorithm we also have to dynamically update the d-neighborhood of A_0. In particular this means removing $N_d(w)$ of a newly matched vertex w and, if $d(|A_0|)$ increases, adding vertices from additional calls to `newABneighbor()` for every vertex in A_0.

To bound the runtime of Algorithm 2, `FastPerfectMatching`, we observe first that we increase the matching exactly n times, which is inline with our desired bound of $\mathcal{O}(n)$. We can thus concentrate on bounding the *recursive* calls to `IncreaseMatching` in line 13 of `IncreaseMatching`.

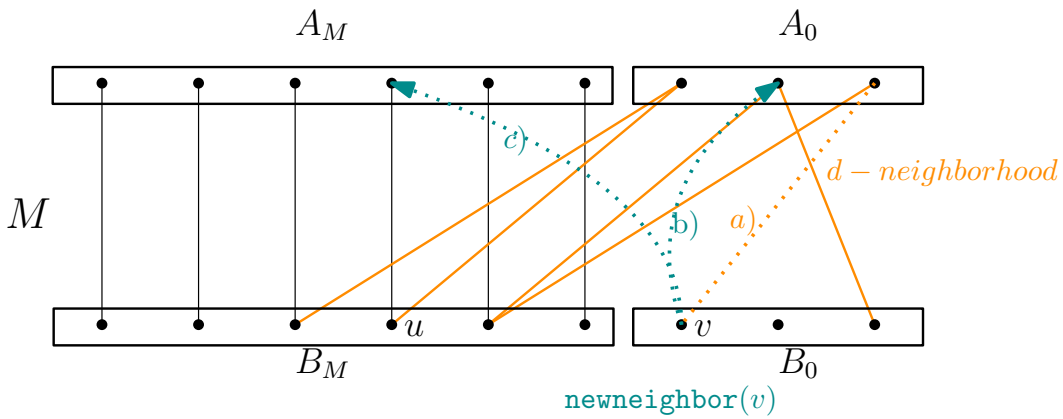

Figure 4 For `IncreaseMatching` three things can happen. Either a) the vertex is already in the neighborhood of A_0, in which case we match immediatly, b) the vertex `newABneighbor(`v`)` is in A_0, which also gets matched, or c) `newABneighbor(`v`)` is in A_M. Then we swap the matching and continue from the partner of the `newABneighbor(`v`)`.

Algorithm 1 `FastPerfectMatching`(G).

1: $B_0 \leftarrow B$; $B_M \leftarrow \{\}$; $A_0 \leftarrow A$; $A_M \leftarrow \{\}$;
2: d $\leftarrow 0$; $M \leftarrow \{\}$
3: **while** $B_0 \neq \{\}$ **do**
4: $v \leftarrow$ arbitrary vertex from B_0 ▷ and remove from B_0
5: IncreasingMatching(G, M, v); ▷ see Algorithm 2
6: **while** $d < \min\left(\sqrt{\frac{n}{|A_0|}}, \log(n)\right)$ **do**
7: $d \leftarrow d + 1$
8: Add `newABneighbor(`v`)` to the d-neighborhood for every vertex in $v \in A_0$
9: **return** Matching M

▶ **Lemma 3.** *Let $\mathcal{L}_i$ denote the number of calls `IncreaseMatching` in line 13, while $|A_0| = i$ for any $i \in [n]$. Then the $\mathcal{L}_i$ are dominated by independent geometric distributions with success probability $p_i = \frac{i \cdot d(i)}{100n}$.*

Proof. Whenever we are at a vertex v in B we expose an edge to a random neighbor in the set A. If that vertex is in A_0 we match v and $|A_0|$ decreases by one so we end the count of $\mathcal{L}_{|A_0|}$. Otherwise we swap with a matched vertex and get a new starting point in B_0. As `newABneighbor()` is independent and uniform, and the matching forms a bijection between A_M and B_M, the fact that the vertex is not in A_0, implies that we get a new *random* vertex u in B_M for the next call. If this vertex is in the exposed d-neighborhood of A_0 we stop and match to a vertex in A_0 also ending the count of $\mathcal{L}_{|A_0|}$.

To assess the probability of stopping, we use the *expansion properties* of the d-neighborhood of A_0 that are inherited from the random graph. This means in particular that the exposed neighborhood of A_0, $N_{d(|A_0|)}(A_0)$, has size at least $\frac{1}{100}|A_0| \cdot d(|A_0|)$, cf. Lemma 9 in Section 3 for a proof. The probability of hitting a vertex in A_0 or the d-neighborhood of A_0 (while looking at the matched vertex of w in B_M) is thus at least $\frac{|A_0| \cdot d(|A_0|)}{100n}$. Every new call of `newABneighbor()` is independent by Lemma 8, thus $\mathcal{L}_i$ is dominated by an independent geometric distribution with success probability as claimed. ◀

Algorithm 2 `IncreaseMatching`(G, M, v).

1: **if** $v \in N_d(A_0)$ **then**
2: $w \leftarrow$ [neighbor of v] $\in A_0$
3: Add $\{v, w\}$ to M
4: Remove w from A_0 and update $N_d(A_0)$
5: **return**
6: $w \leftarrow$ `newABneighbor(`v`)`
7: **if** $w \in A_0$ **then**
8: Add $\{v, w\}$ to M
9: Remove w from A_0 and update $N_d(A_0)$
10: **return**
11: $u \leftarrow$ unique vertex with $\{u, w\} \in M$
12: Remove $\{u, w\}$ from M and replace with $\{v, w\}$
13: IncreaseMatching(G, M, u)
14: **return**

We are now ready to proof the desired complexity bound:

▶ **Proposition 4.** *`FastPerfectMatching` finds a perfect matching in a balanced random bipartite graph in time $\mathcal{O}(n)$ with high probability.*

Proof. There are two main contributions to the running time of the Algorithm. First the subroutine `IncreaseMatching`, which we prove to be fast with the help of Lemma 3, and secondly the updating and revealing of the d-neighborhood.

Recall that $\mathcal{L}_i$ is the random variable corresponding to the number of calls of `IncreaseMatching` in line 13, while $|A_0| = i$ for any $i \in [n]$. We set $\mathcal{L} = \sum_{i=1}^{n} \mathcal{L}_i$. Note that we can ignore the calls in line 5 of `FastPerfectMatching`, as these add only at total of $\mathcal{O}(n)$ to the run time. From Lemma 3 we know that there exists a coupling to a geometrically distributed random variable $\mathcal{L}'$ such that ${\mathcal{L}'}_i \succeq \mathcal{L}_i$ and ${\mathcal{L}'}_i$ *is* geometrically distributed with $p_i = \frac{i \cdot d(i)}{100n}$.

From the definition of ${\mathcal{L}'}_i$ we know that $\mathbb{E}[{\mathcal{L}'}_i] = \frac{100n}{i \cdot d(i)}$ and $Var[{\mathcal{L}'}_i] = \frac{1-p_i}{p_i^2} \leq \frac{1}{p_i^2} \leq (\frac{100n}{i \cdot d(i)})^2$. Recall that $d(i) = \sqrt{n/i}$ whenever $i \geq \frac{n}{(\log n)^2}$. The total time used for those sets can thus be bounded in expectation by

$$\sum_{i=\frac{n}{(\log n)^2}}^{n} \mathbb{E}[{\mathcal{L}'}_i] = \mathcal{O}\left(\sum_{i=1}^{n} \frac{\sqrt{n}}{\sqrt{i}}\right) = \mathcal{O}(n),$$

as $\sum_{i=1}^{n} i^{-1/2} \leq \int_0^n x^{-1/2}\,\mathrm{d}x = 2\sqrt{n}$. If $i \leq \frac{n}{\log(n)^2}$, then $d(i) = \log n$, and the total expected time used for these sets is thus bounded by

$$\sum_{i=1}^{\frac{n}{(\log n)^2}} \mathbb{E}[{\mathcal{L}'}_i] = \mathcal{O}\left(\sum_{i=1}^{n} \frac{n}{i \cdot \log(n)}\right) = \mathcal{O}(n).$$

We thus have that $\mathbb{E}[\mathcal{L}'] = \Theta(n)$ as well. To show that the actual run time is concentrated around the expectation we apply Chebyshev's inequality. A similar case distinction as above gives us

$$Var[\mathcal{L}'] \leq \sum_{i=\frac{n}{(\log n)^2}}^{n} \frac{10000n}{i} + \sum_{i=1}^{\frac{n}{(\log n)^2}} \frac{10000n^2}{i^2 \cdot (\log n)^2} = \mathcal{O}\left(\frac{n^2}{(\log n)^2}\right).$$

By Chebyshev's inequality we thus get

$$Pr[\mathcal{L} \geq 2\mathbb{E}[\mathcal{L}']] \leq Pr[\mathcal{L}' \geq 2\mathbb{E}[\mathcal{L}']] \leq \frac{Var[\mathcal{L}']}{(\mathbb{E}[\mathcal{L}'])^2} \leq \mathcal{O}\left(\frac{1}{(\log n)^2}\right),$$

which concludes the first part of the proof.

To bound the time needed to expose the d-neighborhoods, we observe first that we can order the vertices in A by the order in which they join the matching. As $N_{d'}(v) \subseteq N_d(v)\ \forall d' \leq d$, we thus have to expose for the i-th vertex in this ordering at most $d(n-i)+1$ edges, where $d(x) = \min\{\sqrt{n/x}, \log n\}$. Thus, the total number of exposed edges is bounded by

$$\begin{aligned}\sum_{i=1}^{n}(d(n-i)+1) &= \sum_{i=1}^{\frac{n}{(\log n)^2}} (d(i)+1) + \sum_{i=\frac{n}{(\log n)^2}}^{n} (d(i)+1) \\ &\leq \sum_{i=1}^{\frac{n}{(\log n)^2}} (\log n + 1) + \sum_{i=\frac{n}{(\log n)^2}}^{n} \sqrt{\frac{n}{i}} \leq 4n.\end{aligned}$$

Additionally we show the number of calls to the `newABneighbor()` function is at most $\log n$ for every vertex w.h.p.. For any $v \in B$ we call `newABneighbor(`v`)` exactly once for each time it appears as the matched partner of `newABneighbor(`v`)`. As the distribution on the neighbors is uniform on A and we only use `IncreaseMatching` $\mathcal{O}(n)$ many times in total, the probability that $v \in B$ occurs at least $\log n$ times is at most

$$\binom{\mathcal{O}(n)}{\log n}\left(\frac{1}{n}\right)^{\log n} = \mathcal{O}(n^{-2}),$$

with room to spare. We can thus apply a union bound over all vertices in B to see that w.h.p. no vertex in B has more than $\log n$ calls to `newABneighbor()`. Clearly the same holds for vertices in A, as we only expose the d-neighborhood and $d(|A_0|) \leq \log n$ always. This concludes the proof of Proposition 4. ◀

2.2.2 Phase 2: Incorporating the Cycle Factor

In the previous section we have seen that we can find a perfect matching in $\mathcal{O}(n)$ time. In this section we show how we can extend this algorithm to find a Hamilton cycle. To do this we first call the perfect matching algorithm *twice*, reseting the d-neighborhoods after the first run. By our assumption on the independence on the calls to the function `newneighbor()`, we thereby get two *independent* random perfect matchings. Their union forms a union of cycles (or double edges) covering all vertices (if the number of vertices was odd we add the single vertex excluded in phase 1 here back as a cycle with one vertex). Our task in this phase is to join these cycles into a single cycle. We start with a lemma that bounds the number of cycles that we need to join.

▶ **Lemma 5.** *The union of two random independent perfect matchings in a bipartite graph contains at most* $2\log n$ *cycles with high probability.*

Proof. We claim that the two independent perfect matchings can be seen as a random permutation of $[n/2]$. Indeed, without loss of generality we may assume that M_1 is just the identity (by renumbering the vertices appropriately). M_1 and M_2 are independent which implies M_2 corresponds to a random assignment of B to A. The union of the two matchings thus defines a random permutation of A.

For random permutations the number of cycles has been well studied and is related to the Stirling numbers of the first kind. Using a double counting argument one can easily see that the expected number of cycles of length $2k$ will be $1/k$. The total expected number of cycles is thus equal to the nth harmonic number. It is also well-known that this random variable is concentrated, see e.g. [5] or [6, 21]. Thus, with high probability the number of cycles is bounded by $2\log n$, as claimed. ◀

Description of Algorithm 3 `JoinCycles`. To glue the cycles together we proceed in three phases. First we greedily combine cycles into a path, until this path has length at least $3n/4$. Then we incorporate the remaining cycles one by one using Algorithm 4 `AddSingleCycle`. Finally, we close the Hamilton path into a Hamilton cycle (Lemma 7).

The idea behind the first phase is straightforward. We start with an arbitrary cycle and break it apart into a path P. Consider the endvertex p_{end} of that path. We use `newneighbor()` to query a new neighbor of p_{end}. If that neighbor is in a new cycle (which will happen with probability at least $1/4$, as long as the path P contains at most $3n/4$ vertices), we attach that cycle to P, thereby also getting a new endpoint p_{end}. If the latter did not happen, we query a new neighbor. In order to ensure that we do not query to many vertices from a single vertex, we repeat the query for new neighbors at most $40\log n$ times. If we have not been successful by then, we give up. It is easy to see that the probability for ever giving up at this stage of the algorithm is bounded by $o(1)$. It is also easy to see that the total time spent until the path has length at least $3/4n$ is bounded by $\mathcal{O}(n)$.

Once the path has length at least $3n/4$, the probability that a new neighbor is in one of the remaining cycles gets too small (for our purpose) and we thus change strategy. In particular, we add long Posa rotations, so that we can try various endpoints. This is the purpose of the procedure `AddSingleCycle` (Algorithm 4).

We use a set U to keep track of *used* vertices. Those are vertices for which we already queried neighbors within the algorithm `JoinCycles`. We denote the current path by $P = (p_{start}, .., p_{end})$. We also assume that we have access to a function $pred_P(v)$ that determines the vertex before v on the path (null for p_{start}), and a function $half_p(v)$ which is true iff v is in the first half of P. We denote the cycle C that we want to add as $C = (c_{start}, ..., c_{end})$, where c_{start} is an arbitrary vertex at which we cut C into a path. We now explore neighborhoods of vertices at once. To do this we denote by $\texttt{newneighbor}(v, 40\log n)$ the set of vertices that we obtain if we apply $\texttt{newneighbor}(v)$ $40\log n$ times. Let $N_{start} = P \cap \texttt{newneighbor}(c_{start}, 40\log n)$ and $N_{end} = P \cap \texttt{newneighbor}(c_{end}, 40\log n)$ denote the intersections of these neighborhood vertices with the path P. Until the cycle C is part of the path P we do the following (Figure 5). Let $N(p_{end}) = \texttt{newneighbor}(p_{end}, 40\log n)$ and check for all $v \in N(p_{end})$ if $pred_P(v) \in N_{start}$. If so we also check if $half_p(v)$ is true.

If we find a vertex v for which both conditions hold, we join the cycle here. To do this look at N_{end} and take a vertex $q \in \texttt{newneighbor}(c_{end}, 40\log n)$ such that $half_P(q)$ is false and $pred_P(q) \notin U$. Then we add the cycle to the path by constructing the new path $P_{new} = (p_{start}, ..., pred_P(v)) + (c_{start}, ..., c_{end}) + (q, ..., p_{end}) + (v, ..., pred_P(q))$. Then add p_{end}, c_{start} and c_{end} to the used vertices U. (If we cannot find q we abort the algorithm; it will be easy to show that the probability that this happens is negligible.)

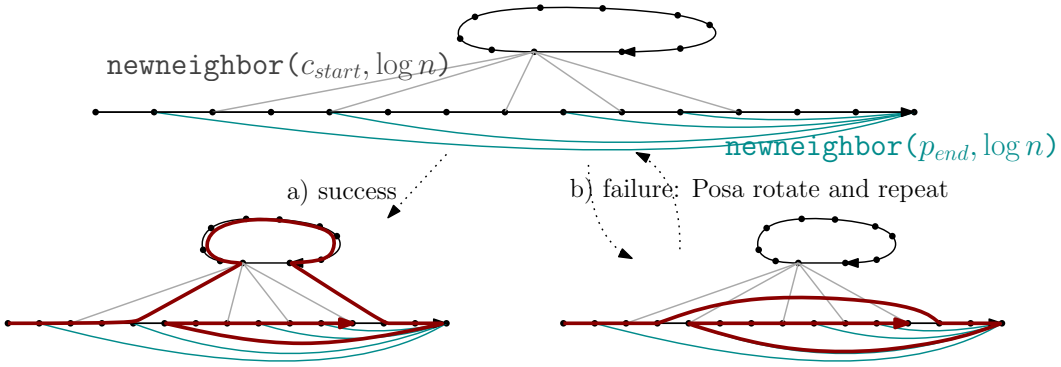

Figure 5 Incorporating a single new cycle with `AddSingleCycle`. The dark red path indicating the new Path after an iteration of the while loop.

Algorithm 3 `JoinCycles`$(G, \mathcal{C} = M_1 \cup M_2)$.

$U \leftarrow \{\}$
$C_0 \leftarrow$ first cycle of $M_1 \cup M_2$, $(c_{0,start}, ..., c_{0,end})$;
$P \leftarrow (c_{0,start}, ..., c_{0,end})$;
$p_{end} \leftarrow$ last vertex of P;
while $|P| \leq \frac{3n}{4}$ **do**
    $N \leftarrow$ `newneighbor`$(p_{end}, 40 \log n)$;
    $U \leftarrow$ add p_{end};
    $v \leftarrow$ Search N for v such that $v \notin P$
    $(v, ..., c_{i,end}) \leftarrow$ cycle of v;
    $P \leftarrow P + (v, ..., c_{i,end})$;
    $p_{end} \leftarrow c_{i,end}$;
while $|P| \neq n$ **do**
    $C_i \leftarrow$ any cycle not in P
    $AddSingleCycle(G, P, C_i, U)$ ▷ See Algorithm 4
return
// If any of the "Search" parts of the algorithm fail, we abort the algorithm and return failure.

If the check fails for all $v \in N(p_{end})$ we perform a Posa rotation. To do this is we take a $v \in N(p_{end}), v \neq p_{start}$, such that $half_P(v)$ is true and such that $pred_P(v) \notin U$, and then take a $q \in$ `newneighbor`$(pred_P(v), 40 \log n)$ such that both $half_p(q)$ is false and $pred_P(q)$ is unused. We then use v and q to construct a new path with a new endpoint, namely $P_{new} = (p_{start}, ..., pred_P(v)) + (q, ..., p_{end}) + (v, ..., pred_P(q))$. Now we can repeat the above procedure with P_{new} and the new endpoint $p_{newend} = pred_P(q)$. (If we cannot find v or q we abort the algorithm; again it will be easy to show that the probability that this happens is negligible.)

To store the path and cycles we use AVL trees with a linked list. The linked list just stores the vertices in the order as they appear in the path resp. cycle. For the AVL tree we take the ordering in the path/linked list as an ordering of the vertices. With this ordering at hand, the AVL tree is well defined, and it allows for searching resp. answering the query $half(v)$ in $\mathcal{O}(\log n)$ time. In addition, splitting the path resp. concatenating two paths correspond to splitting an AVL tree at a given vertex (into a tree containing all smaller vertices and a

Algorithm 4 `AddSingleCycle`(G, P, C_i, U).

// Function $halfP(v)$ returns true if and only if v is in the first half of P;
// For any vertex $v \in P$, $pred_P(v)$ denotes the vertex before v on the path P ;
$p_{end} \leftarrow$ last vertex of P;
$N_{start} \leftarrow$ `newneighbor`$(c_{start}, 40 \log n) \cap P$;
$N_{end} \leftarrow$ `newneighbor`$(c_{end}, 40 \log n)$;
$U \leftarrow$ add c_{start} and c_{end};
while true **do**
    $N \leftarrow$ `newneighbor`$(p_{end}, 40 \log n)$;
    $U \leftarrow$ add p_{end};
    if $\exists v \in N$ *s.t.* $pred_P(v) \in N_{start}$ and $halfP(v) = true$ **then**
        $q \leftarrow$ Search N_{end} for q such that $halfP(q) = false$ and $pred_P(q) \notin U$;
        $P \leftarrow (p_{start}, ..., pred_P(v)) + (c_{start}, ..., c_{end}) + (q, ..., p_{end}) + (v, ..., pred_P(q))$;
        return
    else
        $v \leftarrow$ Search N for v such that $halfP(v) = true$;
        $N \leftarrow$ `newneighbor`$(pred_P(v), 40 \log n)$;
        $U \leftarrow$ add $pred_P(v)$;
        $q \leftarrow$ Search N for q such that $halfP(q) = false$ and $pred_P(q) \notin U$;
        $P \leftarrow (p_{start}, ..., pred_P(v)) + (q, ..., p_{end}) + (v, ..., pred_P(q))$;
        $p_{end} \leftarrow pred_P(q)$;
// If any of the "Search" parts of the algorithm fail, we abort the algorithm and return failure.

tree containing the remaining vertices) resp. concatenate two AVL trees in which the largest vertex in one tree is smaller than the smallest vertex in the other tree. It is well known that both of these operations can be done for AVL trees in $\mathcal{O}(\log n)$ time, cf. Lemma 10 in Section 3 for more details.

▶ **Proposition 6.** *Applying the procedure* `AddSingleCycle` *at most* $2 \log n$ *times will run in time* $\mathcal{O}(n)$ *with high probability.*

Proof. We want to bound the number of Posa rotations we need to perform while we add at most $2 \log n$ cycles. Each Posa rotation occurs at the end of a while loop in the pseudocode.

To incorporate a cycle we want to find a vertex v which, in the order of the path, is right after a vertex in N_{start} and is in the first half of P. P has size at least $3n/4$ so the number of vertices in the first half is at least $n/4$. A random vertex therefore has a chance of at least $1/4$ to be in the first half of P. So every vertex in `newneighbor`$(c_{start}, 40 \log n)$ has probability at least $1/4$ independently of being in the first half of P and different from the other vertices. This implies that the number of vertices in N_{start} which are also in the first half of P dominates a binomial distributed random variable $F \sim Bin(40 \log n, 1/4)$. For F we know the expectation to be $10 \log n$ and by a Chernoff bound (11) the probability that F is less than $\log n$ is $\mathcal{O}(n^{-2})$. We observe that where the Posa rotation happens is independent of N_{start}. So we apply a union bound that on fixed $\mathcal{O}(n)$ many rotations of P the probability that there are less than $\log n$ vertices of N_{start} in the first half of P is in $\mathcal{O}(n^{-1})$. This implies that any call to `newneighbor`(p_{end}) has a chance of at least $\log n/n$ to be right after a vertex in N_{start} and also in the first half of P. As each call to `newneighbor()` is independent, the number of tries we must make is geometrically distributed with success probability $\log n/n$ and we must succeed at most $2 \log n$ many times. This means the number of Posa rotations

is dominated by a negative binomial distributed random variable $R \sim NB(2\log n, \log n/n)$. So by the concentration of the negative binomial distribution (Lemma 14) the probability that we need to try more than $4n$ times is at most $\mathcal{O}(\log^{-1} n)$. Before every Posa rotation we try `newneighbor(`$p_{end}, 40\log n$`)` so $40\log n$ tries. This proves an upper bound on the number of Posa rotations of $\mathcal{O}(n/\log n)$ with high probability.

Posa rotation. We summarize the operations we need to do per Posa rotation. This assumes that we already failed to find v which is both after a vertex in N_{start} and also in the first half of P. We expose $40\log n$ new neighbors of p_{end} and $40\log n$ of the vertex before v on the path, we need to Posa rotate by splitting the path twice and then joining twice. Checking whether a vertex is in U and adding vertices to U is a constant time operation with a lookup table. All of these operations by choice of proper datastructure (Lemma 8 and 10) are done in $\mathcal{O}(\log n)$. So over all Posa rotations these sum up to a runtime of at most $\mathcal{O}(n)$. Additionally we need to find the vertex v in the first half of P with $pred_P(v) \notin U$. Since U is much smaller than $n/8$ and $|P| \geq 3n/4$ the number of possible vertices is at least $n/4$. This means that if we test a random vertex, the probability that $half_p()$ returns true and its predecessor is not in U is at least $1/4$. So the number times we need to call $half_P()$ is dominated by a geometric distribution with success probability $1/4$. Similarly to find the vertex q in the second half of P with $pred_P(q) \notin U$, the number of times we need to call $half_P()$ is also dominated by a geometric distribution with success probability $1/4$. So over all rotations, the number of times we need to call $half_P()$ is dominated by a negative binomial distribution $H \sim NB(2 \cdot \mathcal{O}(n/\log n), 1/4)$. So by the concentration of the negative binomial distribution (Lemma 14) the probability that we need to call $half_P$ more than $\mathcal{O}(n/\log n)$ times is $\mathcal{O}(\log n/n)$. And since we can perform $half_P()$ in time $\mathcal{O}(\log n)$ by Lemma 10 these have a total runtime of $\mathcal{O}(n)$ with high probability.

Incorporating cycles. Very similarly we bound the time we need to incorporate the cycles. To find the vertex v which in the order of the path is right after a vertex in N_{start} and is in the first half of P we need to call $half_P$ until we succeed. Note that since $|N_{start}| \leq 40\log n$ and as we proved above at least $\log n$ vertices of them are in the first half of P, every call to $half_P()$ from a random vertex after a vertex in N_{start} has a chance of succeeding of at least $1/40$. This means the number of times we call $half_P$ is again dominated by a negative binomial distribution $NB(2\log n, 1/40)$ and this runtime is negligible with high probability. As we only incorporate a cycle $2\log n$ times, also the join and split operations as well as the exposing of N_{end} and searching for q are negligible compared to the $\mathcal{O}(n)$ runtime.

Note also that we only call `newneighbor()` of vertices we then add to U and then not again during the entire algorithm so no vertex has `newneighbor()` called more than $40\log n$ times. At most $\mathcal{O}(n/\log n)$ many vertices are added to U, and U is small enough so that it is always much smaller than $n/8$.

This concludes the proof of Proposition 6. ◀

▶ **Lemma 7.** *Given a Hamilton path we can transform it to a Hamilton cycle in $\mathcal{O}(n)$ time.*

Proof. Calling the algorithm `AddSingleCycle` with the cycle being p_{start}, but instead looking for v such that a vertex after v is in the neighborhood of p_{start} instead of a predecessor gives us a cycle $C = (p_{start}, ..., v) + (p_{end}, ..., after_P(v))$. Analysis of runtime equivalent to the analysis of `AddSingleCycle`. ◀

Propositions 4 and 6 as well as Lemma 7 show that all components of the algorithm run in time $\mathcal{O}(n)$. It is also easy to check that both phases together require at most $50\log n$ calls to `newneighbor()` from any fixed vertex, so the assumptions of Lemma 8 do hold. So choosing C large enough, say $C = 200$, suffices to guarantee that with high probability the random graph is such that all vertices have more neighbors than we query. This thus concludes the proof of Theorem 1.

3 Datastructures

In this section we give the details of the data structures that we used within our algorithm.

3.1 Querying a new vertex

As explained above, we assumed throughout the analysis of our algorithm that we have access to a function `newneighbor(v)`, that returns for a given vertex v a neighbor w that is *uniformly* distributed in $V - v$ and whose result is *independent* from all previous calls.

▶ **Lemma 8** (newneighbor). *It is possible to interact with the graph $G_{n,p}$, $p \geq \frac{C \log n}{n}$, with an algorithmic procedure `newneighbor(v)` which has the following properties with high probability:*

- **(i)** *Calling `newneighbor(v)` returns a neighbor of v distributed uniformly among $V - v$ and independent of all calls so far – as long as we make at most $\mathcal{O}(n)$ calls to `newneighbor()` altogether and every vertex is queried at most $100 \log n$ times.*
- **(ii)** *The total run time of all $\mathcal{O}(n)$ calls is $\mathcal{O}(n)$.*

Proof. To realize such a function `newneighbor()`, it is important to make the adjacency lists independent of each other. To realize this we transform the graph G (which is distributed as a random graph $G_{n,p}$) into a directed graph G' distributed as $D_{n,p/2}$ (in which each directed edge is present independently with probability $p/2$). It is well know how this can be done. In particular, we can sample $D_{n,p/2}$ from $G_{n,p}$ as a subgraph such that every edge in the directed graph is also an undirected edge in the $G_{n,p}$. More precisely, we do the following for every edge $\{i, j\}$ of G: with probability

$$\begin{aligned}
&\frac{1}{2} - \frac{p}{4} \text{ set } (i,j) \in G' \text{ and } (j,i) \notin G' \\
&\frac{1}{2} - \frac{p}{4} \text{ set } (i,j) \notin G' \text{ and } (j,i) \in G' \\
&\frac{p}{4} \text{ set } (i,j) \in G' \text{ and } (j,i) \in G' \\
&\frac{p}{4} \text{ set } (i,j) \notin G' \text{ and } (j,i) \notin G'
\end{aligned}$$

In order to be consistent with the transformation from G to G' and to not lose too much time we only direct the edges once we see it for the first time. To recall the made decision, we store the random choices of the edges that we have we encountered so far into a hashtable. Thus, we can check for each edge that we obtain from querying the adjacency list of a vertex in G, whether we have seen this edge already and if so, which orientation we have chosen. The hash table has size n and we use a hashfunction which is 4-wise independent. This way the variance of the number of collisions is equal to a random function, and therefore the time we need for hashing is $\mathcal{O}(n) + \mathcal{O}(\textit{number of collisions}) = \mathcal{O}(n)$, which can be seen by applying Chebyshev's inequality. A more detailed argument of why linear probing with hash functions works in this context can be found in [24, 31].

Finally, we want the distribution of the next edge to be uniform among the vertices. For this we need to resample from the already seen edges. Assuming we have revealed d many edges from v we flip a biased coin. With probability $d/(n-1)$ we retake an old neighbor and output it, one chosen uniformly at random, and with probability $1 - d/(n-1)$ we take the next vertex in the adjacency list (which is also in $D_{n,p}$). Otherwise, we return one of the previously seen neighbors uniformly at random. In this way any vertex has probability exactly $1/(n-1)$ to be returned by `newneighbor(v)`. ◀

3.2 Expansion

What we need from the random graph are properties of good expansion. Given the adjacency list of a vertex v we define the d-neighborhood $N_d(v) \subseteq V(G)$ to be the set of the first $\lceil d \rceil$ calls to the function `newABneighbor(`v`)`. In the analysis of the algorithm we make use of the following lemma.

▶ **Lemma 9** (Neighborhood Lemma). *Let $G_{n/2,n/2,p}$ be a random bipartite graph with $p \geq \frac{C \log(n)}{n}$ and partitions A and B. Then with high probability we have for all subsets $A' \subseteq A$ that the d-neighborhood of A' is of size at least*

$$|N_{d(A')}(A')| \geq \frac{1}{100}|A'| \cdot d(A'), \tag{1}$$

where $d(A') = min(\sqrt{\frac{n}{|A'|}}, \log(n))$.

Proof. The proof follows from a straight forward calculation of probabilities. Let us assume by contradiction there exists a set $A' \subseteq A$ with $|N_{d(A')}(A')| < \frac{1}{100}|A'| \cdot d(A')$. Then there is a set $B' \subseteq B$ of size $|B'| = \frac{1}{100}|A'| \cdot d(A')$ containing this d-neighborhood, $N_{d(A')}(A') \subseteq B'$. So this is a probability we want to bound from above. The probability for a single vertex in A' to have its d-neighborhood contained in a fixed set B' is $\left(\frac{|B'|}{n}\right)^{d(A')}$ since the d-neighborhood is $d(A')$ vertices chosen from B uniformly and independently at random. The probability for two specific sets $A' \subseteq A$ and $B' \subseteq B$ to have this property is $(|B'|/n)^{|A'| \cdot d(A')}$. Now take the union bound over all possible sets A' and B' (with $|B'| = \frac{1}{100}|A'| \cdot d(A')$):

$$Pr[(1) \text{ false}] \leq \sum_{A',B'} Pr[B' \text{ contains } N_{d(A')}(A')] = \sum_{i=1}^{n} \binom{n}{i} \binom{n}{\frac{1}{100} i \, d(i)} \left(\frac{\frac{1}{100} i \cdot d(i)}{n}\right)^{i \cdot d(i)}.$$

Then we apply an approximation for the binomial coefficients: $\binom{n}{k} \leq \left(\frac{en}{k}\right)^k$. We see that $\frac{1}{4} \log \frac{100n}{i \cdot d(i)} \geq \log(e \cdot n/i)/d(i)$ so

$$Pr[(1) \text{ false}] \leq \sum_{i=1}^{n} exp\left(i \cdot d(i) \cdot \left(-\frac{1}{2} \log\left(\frac{100n}{i \cdot d(i)}\right)\right)\right).$$

Now $d(i)$ is a known function of i. So we distinguish between two cases. When $i \geq n/\log^2(n)$, then $d(i) = \sqrt{n/i}$. And we can calculate ($i \leq n$)

$$\sum_{i=1}^{n} \left(\frac{i^{1/4} \cdot n^{1/4}}{10 \cdot n^{1/2}}\right)^{\sqrt{n\ i}} \leq \sum_{i=1}^{n} \left(\frac{1}{10}\right)^{\sqrt{n\ i}} \leq \mathcal{O}(n^{-2}).$$

On the other hand if $i \leq n/\log^2(n)$, then $d(i) = \log(n)$. And we can calculate

$$\sum_{i=1}^{n/\log^2(n)} \left(\frac{i^{1/2} \cdot \log(n)^{1/2}}{10 \cdot n^{1/2}}\right)^{i \log(n)} \leq \sum_{i=1}^{n/\log^2(n)} \left(\frac{1}{10 \cdot \log(n)^{1/2}}\right)^{i \log(n)} \leq \mathcal{O}(n^{-2}).$$

Together this implies that the lemma holds for random graphs with probability $\geq 1 - \mathcal{O}(n^{-2})$. ◀

3.3 AVL Trees

▶ **Lemma 10** (AVL Trees). *We can store a path (or cycle) in an AVL tree joint with a linked list datastructure and can perform the following operations (where we view the cycle as a path split at an arbitrary point):*

- *For any vertex v find the vertex preceding or succeeding it in the path in constant time $\mathcal{O}(1)$*
- *For any vertex v searching the path it is in and determining whether it is in the first or second half of it in time $\mathcal{O}(\log n)$*
- *Split the path into two paths in time $\mathcal{O}(\log n)$*
- *Concatenate two paths into one by adding the endpoint of one to the start of the other in time $\mathcal{O}(\log n)$*

Proof. We combine an AVL tree, which is a balanced binary search tree, with a linked list. The AVL tree is built on the order sequence of the path as if numbering the vertices along the path from 1 to $|P|$. The linked list ensures that going forward and backward on the path is done in constant time, where the AVL tree can perform search (for the half function) in $\mathcal{O}(\log n)$. A split of the path is nothing other than splitting the AVL tree at a leaf node into two trees such that all the nodes smaller go into one tree and all the nodes larger go into the other. The concatenate is the inverse of the split and only requires attaching the smaller tree to the larger one at the appropriate node and rebalancing up to the root. Both operations run in $\mathcal{O}(\log n)$ time.AVL trees are by now a part of basic datastructure lectures and in particular the split and concatenate operations can be found e.g. in the book by Knuth [18] see page 473, which also cites from [10] or more generally on AVL trees see [25]. ◀

4 Concluding remarks

In this paper we presented a simple randomized algorithm based on iterative random walks that construct a Hamilton cycle in time linear in the number of vertices. Our algorithm is based on first building (two) random perfect matchings. The key idea here is to expose more and more edges of the currently unmatched vertices, where the exact number of these exposed neighbors is a function of the currently unmatched vertices. Our analysis requires that the density of the random graph $G_{n,p}$ is at least $p \geq \frac{C \log n}{n}$. C is chosen such that we have a sufficient minimum degree with high probability.

We leave it as an open question whether our approach can be modified to also find Hamilton cycles in $G_{n,p}$ for p at the threshold for existence of Hamilton cycles. This certainly requires additional ideas.

References

1. Miklós Ajtai, János Komlós, and Endre Szemerédi. First occurrence of Hamilton cycles in random graphs. *North-Holland Mathematics Studies*, 115(C):173–178, 1985.
2. Peter Allen, Julia Böttcher, Yoshiharu Kohayakawa, and Yury Person. Tight hamilton cycles in random hypergraphs. *Random Structures & Algorithms*, 46(3):446–465, 2015.
3. Yahav Alon and Michael Krivelevich. Finding a Hamilton cycle fast on average using rotations and extensions. *Random Structures & Algorithms*, 57(1):32–46, 2020.
4. Dana Angluin and Leslie G Valiant. Fast probabilistic algorithms for Hamiltonian circuits and matchings. *Journal of Computer and System Sciences*, 18(2):155–193, 1979.
5. Richard Arratia, Andrew D Barbour, and Simon Tavaré. *Logarithmic combinatorial structures: a probabilistic approach*, volume 1. European Mathematical Society, 2003.

6 Richard Arratia and Simon Tavaré. The cycle structure of random permutations. *The Annals of Probability*, pages 1567–1591, 1992.

7 Tom Bohman and Alan Frieze. Hamilton cycles in 3-out. *Random Structures & Algorithms*, 35(4):393–417, 2009.

8 Bela Bollobas, Trevor I. Fenner, and Alan M. Frieze. An algorithm for finding Hamilton paths and cycles in random graphs. *Combinatorica*, 7(4):327–341, 1987.

9 Bela Bollobás, Trevor I. Fenner, and Alan M. Frieze. Hamilton cycles in random graphs with minimal degree at least k. *A tribute to Paul Erdos, edited by A. Baker, B. Bollobás and A. Hajnal*, pages 59–96, 1990.

10 C.A. Crane. *Linear Lists and Priority Queues as Balanced Binary Trees*. Computer Science Department. Department of Computer Science, Stanford University., 1972.

11 Trevor I Fenner and Alan M Frieze. Hamiltonian cycles in random regular graphs. *Journal of Combinatorial Theory, Series B*, 37(2):103–112, 1984.

12 Asaf Ferber, Michael Krivelevich, Benny Sudakov, and Pedro Vieira. Finding hamilton cycles in random graphs with few queries. *Random Structures & Algorithms*, 49(4):635–668, 2016.

13 Alan Frieze. Hamilton cycles in random graphs: a bibliography. *arXiv preprint*, 2019. `arXiv:1901.07139`.

14 Alan Frieze, Mark Jerrum, Michael Molloy, Robert Robinson, and Nicholas Wormald. Generating and counting hamilton cycles in random regular graphs. *Journal of Algorithms*, 21(1):176–198, 1996.

15 Alan M Frieze. Finding hamilton cycles in sparse random graphs. *Journal of Combinatorial Theory, Series B*, 44(2):230–250, 1988.

16 Yuri Gurevich and Saharon Shelah. Expected computation time for Hamiltonian path problem. *SIAM Journal on Computing*, 16(3):486–502, 1987.

17 Svante Janson, Tomasz Luczak, and Andrzej Rucinski. *Random graphs*, volume 45. John Wiley & Sons, 2011.

18 Donald E Knuth. *The art of computer programming: Volume 3: Sorting and Searching*. Addison-Wesley Professional, 1998.

19 János Komlós and Endre Szemerédi. Limit distribution for the existence of Hamiltonian cycles in a random graph. *Discrete mathematics*, 43(1):55–63, 1983.

20 Aleksei Dmitrievich Korshunov. Solution of a problem of erdős and renyi on Hamiltonian cycles in nonoriented graphs. *Doklady Akademii Nauk*, 228(3):529–532, 1976.

21 Kenneth Maples, Ashkan Nikeghbali, and Dirk Zeindler. On the number of cycles in a random permutation. *Electron. Commun. Probab.*, 17:13 pp., 2012.

22 Richard Montgomery. Hamiltonicity in random graphs is born resilient. *Journal of Combinatorial Theory, Series B*, 139:316–341, 2019.

23 Rajko Nenadov, Angelika Steger, and Miloš Trujić. Resilience of perfect matchings and hamiltonicity in random graph processes. *Random Structures & Algorithms*, 54(4):797–819, 2019.

24 Anna Pagh, Rasmus Pagh, and Milan Ruzic. Linear probing with constant independence. In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 318–327, 2007.

25 Ben Pfaff. An introduction to binary search trees and balanced trees. *Libavl Binary Search Tree Library*, 1:19–20, 1998.

26 Robert W. Robinson and Nicholas C. Wormald. Almost all regular graphs are Hamiltonian. *Random Structures & Algorithms*, 5(2):363–374, 1994.

27 Ronitt Rubinfeld and Asaf Shapira. Sublinear time algorithms. *SIAM Journal on Discrete Mathematics*, 25(4):1562–1588, 2011.

28 Eli Shamir. How many random edges make a graph Hamiltonian? *Combinatorica*, 3(1):123–131, 1983.

29 Benny Sudakov and Van H Vu. Local resilience of graphs. *Random Structures & Algorithms*, 33(4):409–433, 2008.

30 Andrew Thomason. A simple linear expected time algorithm for finding a hamilton path. *Discrete Mathematics*, 75(1-3):373–379, 1989.
31 Mikkel Thorup and Yin Zhang. Tabulation based 4-universal hashing with applications to second moment estimation. In *SODA*, volume 4, pages 615–624, 2004.

A Concentration Inequalities

We mention here some well-known inequalities used to show concentration on random variables. By the concentration of a random variable X we usually mean there exist constants c and C such that with high probability $c\mathbb{E}[X] \leq X \leq C\mathbb{E}[X]$. Often also we ask C to be 2. Although these are by no means new insights, we present them here for completion and as a help for the reader. The Chernoff and Chebyshev inequality can be found e.g. in the book [17].

▶ **Theorem 11** (Chernoff Inequality). *If X is distributed as a binomial random variable $X \sim Bin(n,p)$ and $0 < \varepsilon \leq 3/2$, then*

$$Pr[|X - \mathbb{E}[X]| \geq \varepsilon\mathbb{E}[X]] \leq 2e^{-\frac{\varepsilon^2\mathbb{E}[X]}{3}}.$$

▶ **Theorem 12** (Chebyshev Inequality). *For any random variable X for which the variance $Var[X]$ exists,*

$$Pr[|X - \mathbb{E}[X]| \geq t] \leq \frac{Var[X]}{t^2}.$$

We use the term negative binomial distribution in the analysis and since this is defined slightly differently sometimes we give here the definition we use.

▶ **Definition 13.** *Let X_i be independent bernoulli random variables with probability of being one is p for any $i \in \mathbb{N}$. For any $r \in \mathbb{N}$ let Y be index of the r-th X_i which evaluates to 1. Then Y has a negative binomial distribution $NB(r,p)$.*

We observe that a negative binomial distribution $Y \sim NB(r,p)$ is equivalent to Y being distributed as the sum of r geometric random variables with success probability p. Further a simple corollary from the Chebyshev inequality:

▶ **Corollary 14.** *For a negative binomial distributed variable $Y \sim NB(r,p)$*

$$Pr\left[Y \geq 2\frac{r}{p}\right] \leq \frac{1}{r}.$$

Proof. We calculate $\mathbb{E}[Y] = r/p$ and $Var[Y] = r(1-p)/p^2$ and apply Chebyshev.

$$Pr\left[Y \geq 2\frac{r}{p}\right] \leq Pr\left[|Y - \mathbb{E}[Y]| \geq \frac{r}{p}\right] \overset{Chebyshev}{\leq} \frac{1-p}{r} \leq \frac{1}{r}$$ ◀