THE 3-PHASE APPROACH TO FINDING HAMILTON CYCLES IN RANDOM DIGRAPHS

Submitted in Partial Fulfilment of the
Requirements for the Degree of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

Submitted by

ADITYA SUBRAMANIAN

(1610110045)

Under the Supervision of

PROF. SANDEEP SEN

(Professor, Computer Science & Engineering)

SHIV NADAR UNIVERSITY

Department of Computer Science and Engineering

May, 2020

Declaration

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources.
I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission.
I understand that any violation of the above will be cause for disciplinary action by the University and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Aditya Subramanian

May 27, 2020

Certificate

This is to certify that the project titled "The 3-phase approach to finding Hamilton Cycles in Random Digraphs" is submitted by Aditya Subramanian, Shiv Nadar University in the partial fulfilment of the requirements for the award of the degree of
Bachelor of Technology in Computer Science and Engineering.
This work was completed under the supervision of
Prof. Sandeep Sen, Shiv Nadar University.
No part of this dissertation has been submitted elsewhere for award of any other degree.

Sandeep Sen

Professor, Computer Science and Engineering

Shiv Nadar University

Acknowledgments

First and foremost, I would like to thank my advisor Prof. Sandeep Sen for guiding me through this project. He has been generous with his time even during the lockdown, and been very encouraging throughout.

I also with to extend my gratitude to the Project Committee, and Shiv Nadar University, for giving me the opportunity to pursue this project.

I want to thank Praneet Srivastava and Nishank Suresh for their moral support and company, making life at SNU enjoyable.

Finally, I want to thank my family for their continuous support throughout my degree.

Contents

List of Figures

Abstract

The main focus of this thesis is to examine an algorithm for finding Hamiltonian cycles on random digraphs [9], and using that method to experimentally check it's validity on other models of random graphs.

In Chapter 3 we look at the DHAM algorithm closely, and verifying it's performance and accuracy claims experimentally. We further go on to check whether modifying some parameters in the algorithm itself could potentially improve it's runtime.

In Chapter 4 we consider some other models of graphs such as random tournaments, and random regular digraphs, and look at the performance of DHAM on these graphs.

Chapter 1

Introduction

The concept of graphs was first used by Leonhard Euler to study the Königsberg bridges problem [8] in 1735. This laid the foundation for field of mathematics known as graph theory.
These days graphs are used to model and study complex networks. Networks like the internet, electric grid, road networks, social networks etc. are too large to be directly studied. In such situations a mathematical modelling of the structure allows us to study it's properties more efficiently.
In this thesis we shall be studying one such property of a graph known as it's hamiltonicity, on particular classes of graphs.

1.1 Preliminaries

In the following section we shall briefly introduce some of the preliminaries and conventions that shall be used throughout the thesis.

1.1.1 Graph Theory

Definition 1.1.1. We define a **Graph** G as the pair of sets (V, E) where V is the *vertex set*, and the *edge set*, $E \subseteq V \times V$.

In an *undirected graph* the order of vertices in an edge does not matter. In such graphs we represent an edge as $\{u, v\}$. On the other hand, in a *directed graph* (or digraph) the order of vertices in an edge matters. Edges in such a graph are represented as (u, v).

Definition 1.1.2. The number of edges *adjacent* to a vertex, is known as the **Degree** of that vertex.

Note that in a directed graph, we have separate in and out-degrees for each vertex. Henceforth, we shall be using δ to represent the minimum degree of a vertex in a graph, and similarly Δ to represent the maximum degree.

Definition 1.1.3. A sequence of distinct vertices $P = v_0, v_1, \cdots v_k$ in a directed graph $G(V, E)$, where $\forall i < k, e_i = (v_i, v_{i+1}) \in E$ is known as a **Path** in the graph.

Definition 1.1.4. A path $C = v_0, v_1, \cdots v_k$ in graph $G(V, E)$ where $v_0 = v_k$ is known as a **Cycle**.

Definition 1.1.5. A cycle C such that it passes through every vertex of the graph, is called a **Hamiltonian Cycle**. A graph which contains a Hamiltonian cycle, is said to be a **Hamiltonian Graph**.

1.1.2 Random Graph Models

The graphs we shall be looking at in this thesis shall all be sampled from one of the distributions described below.

$D_{n,m}$ **model:** A graph is picked uniformly at random from the set of all digraphs with n vertices and m edges.

k**-in,** k**-out:** In the $D_{k-in,k-out}$ model, for a graph with n vertices, k in-neighbours and l out-neighbours are chosen independently at random from the vertex set V_n.

k**-regular digraph:** In a random regular digraph $D_{n,k}$, each of the n vertices has in-degree and out-degree exactly k.

Powerlaw graphs: Given a fixed degree sequence that has a power law tail, we pick a graph uniformly at random from the set of all graphs that realize the given degree seqence.

Random Tournaments: In a complete graph K_n, we assign a direction uniformly at random to each edge.

Random n-lifts: The *n-lift* of a given graph $G(V, E)$, is obtained by replacing each vertex in V by a set of n vertices, and by adding a random perfect matching (between the corresponding sets of $u\&v$) for every $e_i = (u, v) \in E$, .

Chapter 2

Literature Review

P. Erdős and A. Rényi started the study of Random Graphs [7] in 1960, and in the same paper also asked the question about the existence of a Hamiltonian path in a random undirected graph. This problem has been extensively studied over the years with a recent survey by Frieze [10] giving a very good introduction to all the different directions that this study of Hamiltonian Cycles in Random graphs has taken.

Pósa[14] and Komlós& E. Szemerédi[12] made significant advances in the undirected version of the problem, showing that in $G_{n,m}$, $m \in \mathcal{O}(n \log n)$ is sufficient for a graph to be Hamiltonian.

Bollobas [2] improved this result by showing that

$$\lim_{n\to\infty} Pr(G_{m^*} \text{is hamiltonian}) = 1$$

where G_m is defined as the graph obtained by taking the first m edges from the list $E = \{e_1, e_2, \cdots e_{\frac{n(n-1)}{2}}\}$ obtained by permuting the edges of a complete graph on n vertices, and $m^* = min\{m : \delta(G_m) \geq 2\}$

An analogous result for directed graphs was given by Frieze [9] while also giving a $\mathcal{O}(n^{1.5})$ algorithm(DHAM) showing that

$$\lim_{n\to\infty} Pr(\text{DHAM finds a hamilton cycle in } D_{m^*}) = 1$$

where D_m is the directed analogue of G_m defined above, and $m^* = min\{m : \delta^+(D_m) \geq 1, \delta^-(D_m) \geq 1\}$

The algorithm is divided into 3 phases, the first obtains a permutation digraph from the given graph (which is obtaining a permutation on the set of vertices such

that they form $\mathcal{O}(\log n)$ cycles). The second merges some of the these cycles to get an asymptotically larger cycle. And the final phase merges all cycles to obtain the desired Hamiltonian cycle.

A somewhat similar 3-phase method has been used to obtain existential results on more classes of graphs. These results do not usually explicitly give an algorithm, or try to bound the runtime of the same. In our thesis, we attempt to use the above 3-phase algorithm, to experimentally see, if the DHAM algorithm also runs successfully with polynomial growth on these classes of graphs.

Chapter 3

The DHAM Algorithm

In this chapter we will take a closer look at the 3-phase DHAM algorithm[9], reviewing each phase and consider some of the implementation and runtime details of the same. We shall then move on to analysing how the algorithm performs, in terms of accuracy and runtime on the intended class of random digraphs.

3.1 The Algorithm

Let $e_1, e_2, \cdots e_{n(n-1)}$ be a random permutation of the edges of the complete digraph DK_n with vertex set V_n. Let $E_m = e_1, e_2, \cdots e_m$ for $1 \leq m \leq n(n-1)$ and $D_m = (V_n, E_m)$. Then it is claimed that

Theorem 3.1.1. *Let $m^* = min\{m : \delta^+(D_m) \geq 1, \delta^-(D_m) \geq 1\}$. Then, there is a (randomized) polynomial ($\mathcal{O}(n^{1.5})$) time algorithm DHAM, which satisfies*

$$\lim_{n\to\infty} Pr(DHAM \text{ finds a hamilton cycle in } D_{m^*}) = 1$$

3.1.1 Phase 1

The first phase takes the edge set E_{m^*} as input and constructs a permutation ϕ on the set of vertices such that, $\forall i \in V_n, (i, \phi(i)) \in E_{m^*}$. This is called a permutation digraph. It has been shown that,

Lemma 3.1.2. *The following hold, with high probability*

(a) Phase 1 succeeds.

(b) ϕ has at most $2 \log n$ cycles.

In terms of runtime, the construction of the subset of edges and then the Bipartite graph from it, takes $\mathcal{O}(n \log n)$ time, and then finding the perfect matching in the graph takes $\mathcal{O}(n^{1.5})$ time[6].

Algorithm 1: Psuedocode for DHAM Phase 1.

Input: edge set E_{m^*}
Output: permutation digraph, ϕ

// We construct Edge sets $\hat{E^+}$ and $\hat{E^-}$ with average out/in degree 10
$\sigma \leftarrow$ random permutation on V_n
for *each edge* $e_i = (v_i, w_i) \in E_{m^*}$ **do**
    $\hat{e_i} = (v_i, \sigma(w_i))$
    if *out-degree of* v_i *in* $\hat{E^+} \leq 9$ **then**
        $\hat{E^+} \leftarrow \hat{E^+} \cup \hat{e_i}$
    if *in-degree of* $\sigma(w_i)$ *in* $\hat{E^-} \leq 9$ *AND* $\hat{e_i} \notin \hat{E^+}$ **then**
        $\hat{E^-} \leftarrow \hat{E^-} \cup \hat{e_i}$
// We now construct a bipartite graph, from the edges selected above
$\hat{E} \leftarrow \hat{E^+} \cup \hat{E^-}$
define: Graph $BIP(V_n, W_n, E')$, where W_n is a disjoint copy of V_n, and
$\{u, v\} \in E' \iff (u, v)$ or $(v, u) \in \hat{E}$
use Dinic's max-flow algorithm [6] to find a maximum matching, M in BIP
if *M is not a perfect matching* **then**
    return no solution
else
    define permutation ψ of V_n by $M = \{(v, \psi(v) : v \in V_n\}$
return $\phi = \sigma^{-1}\psi$

3.1.2 Phase 2

Phase 2 attempts to 'patch' together some of the cycles in ϕ using 2 edge exchanges (see Figure 3.1). Let $C_1, C_2, \cdots C_r$ denote the cycles in ϕ, at the end of Phase 2, where $|C_1| \geq |C_2| \geq \cdots \geq |C_r|$

Lemma 3.1.3. *At the end of Phase 2, with high probability*

$$|C_1| \geq n_0 = \left\lceil n - \frac{\sqrt{n}}{\log n} \right\rceil$$

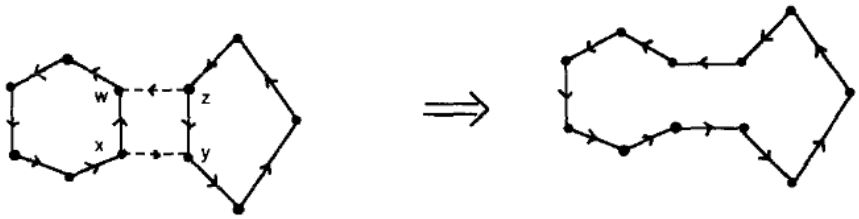


Figure 3.1: 2 edge rotations to patch a cycle

Algorithm 2: Psuedocode for DHAM Phase 2.

Input: permutation ϕ, set of associated edges
Output: ϕ with some cycles patched

```
1: m_2 ← ⌈5n log n / 6⌉
2: flag ← true
3: while flag == true do
4:     flag ← false
5:     for each e_i = (x, y) ∈ E_{m_2} do
6:         e ← (φ^{-1}(y), φ^{-1}(x)) = (z, w)
7:         if x and y are in different cycles of φ AND e ∈ E_{m_2} then
8:             φ(x) ← y
9:             φ(z) ← w
10:            flag ← true
```

This means that by the end of Algorithm 2, we have one large cycle of size $n - o(n)$, and $\mathcal{O}(n)$ others.

The outer loop has been shown to execute only $\mathcal{O}(\log n)$ times and the inner loop runs in $\mathcal{O}(m_2) \in \mathcal{O}(n \log n)$ time. Inside the loop, checking if 2 vertices belong to the same cycle can be done in constant time (using Union-Find) while checking if the edge is in the graph takes another $\mathcal{O}(\log n)$ time. Hence the overall runtime can be bounded by $\mathcal{O}(n(\log n)^3)$

3.1.3 Phase 3

In this phase we attempt to patch all of the smaller cycles, into the larger cycle by using the process of 'double rotations' (See Figure 3.2). It has been shown that

Lemma 3.1.4. *For some constant* $\epsilon > 0$

$$Pr(FindCycle\ fails) = \mathcal{O}(n^{-\epsilon})$$

It follows that,

$$Pr(\textit{Phase 3 fails}) = \mathcal{O}(n^{-\epsilon}\log n) + o(1) = o(1)$$

From which Theorem 3.1.1 follows.

A successful run of Algorithm 3, having patched all the cycles returns a Hamiltonian path.

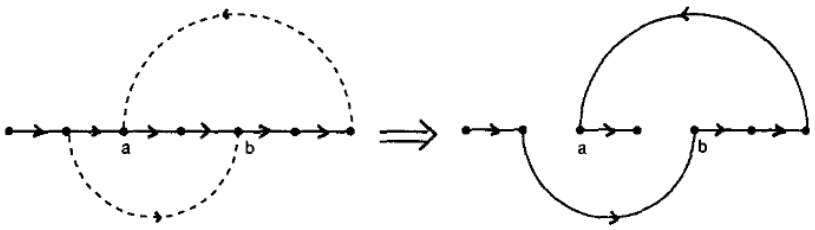


Figure 3.2: double rotation, to merge all cycles

For the double rotation operations, we have implemented a data structure storing the path as a tree, which allows us to get all the necessary operations in $\mathcal{O}(\log n)$ time (For further details of this implementation, refer to Appendix A).

It has been shown that each execution of the *FindCycle* routine generates $\mathcal{O}((6\log n)^T) \in \mathcal{O}(n^{4/3+o(1)})$ paths altogether. Now, a double rotation, and hence each traversal in the tree takes $\mathcal{O}(\log n)$ time, and *FindCycle* itself is called $\mathcal{O}(\log n)$ times, all adding up to $\mathcal{O}(n^{4/3+o(1)})$ runtime for Phase 3.

3.2 Experimentally analysing Performance

The runtime of the algorithm is expected $\mathcal{O}(n^{1.5})$ when the input graphs are selected from the required uniform distribution. We generate multiple instances of such graphs of different sizes, and try to verify if the runtime grows as function proportional to $n^{1.5}$. of the size of the input graph. Note that here we use the $D_{n,m}$ model, where m is set to $m = n\log n + cn$

In Figure 3.3a we can see that the runtime curve closely resembles the plot for $y = 0.0012n^{1.5}$, which is in accordance with the results of Frieze[9].

We also see that the result that $\lim_{n\to\infty} Pr(\text{DHAM finds a hamilton cycle in } D_{m^*}) = 1$ is also observed to be true, with our implementation finding a Hamiltonian cycle in every graph instance. In Figure 3.3b we can see the average number of runs of

Algorithm 3: Psuedocode for DHAM Phase 3.

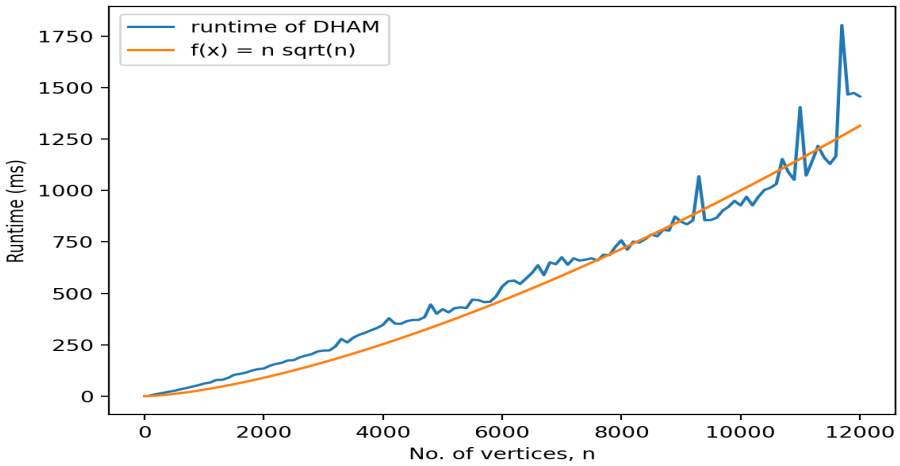
Input: Permutation digraph ϕ from phase2
Output: Hamiltonian Cycle, C

Function `Phase3(`ϕ`)`**:**
  $C_1 \leftarrow$ Largest cycle in ϕ
  foreach *Cycle* $C_i \in \phi$ **do**
    let $C_i = (x_1, x_2, \cdots x_k)$
    for $j = 1$ *to* k **do**
      $outcome =$ `FindCycle(`C_1, C_i, x_j`)`
      if $outcome == true$ **then**
        break
    if $outcome == false$ **then**
      return No Solution

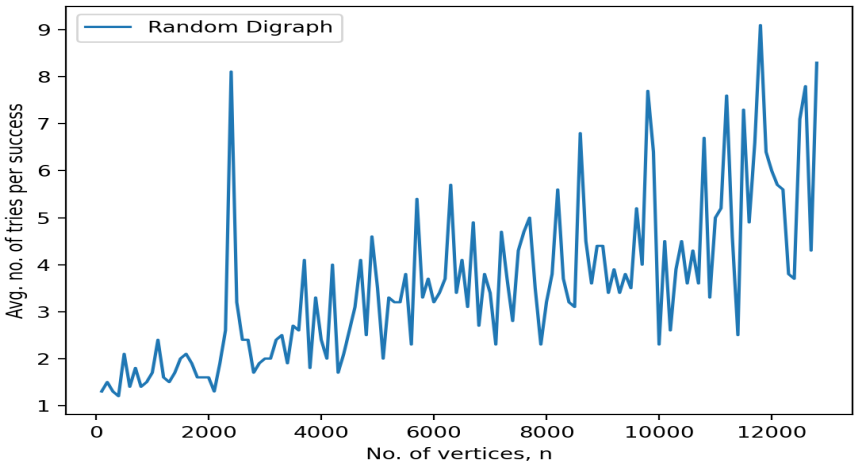
Input: Large cycle C_1, smaller cycle C_i, vertex $x_j \in C_i$
Output: Cycle C_1 spanning vertices of $C_1 \& C_i$, boolean *success*

Let $C_1 = (y_1, y_2, \cdots, y_p)$
$T \leftarrow \lceil \frac{2 \log n}{3 \log \log n} \rceil$
Function `FindCycle(`C_1, C_i, x_j`)`**:**
  Set $\rho_0 \leftarrow$ NULL
  foreach *out-neighbour* y_i *of* x_j, *in* C_1 **do**
    construct path, $P = (x_{j+1}, x_{j+2}, \cdots, x_1, \cdots, x_j, y_i, y_{i+1}, \cdots, y_{i-1})$
    $\rho_0.insert(P)$
  for $t = 1$ *to* T **do**
    $\rho_t \leftarrow NULL$
    foreach *Path* $P_r \in \rho_{t-1}$ **do**
      suppose $P_r = (u_1, u_2, \cdots, u_q)$
      if $(u_q, u_1) \in E_{m^*}$ **then**
        update ϕ, with the cycle P_r
        return $true$
      else
        if $(u_q, u_a) \in E_{m*}$ AND $(u_{a-1}, u_b) \in E_{m*}$ AND $b > a$ **then**
          $\rho_t.insert(ROTATE(P_r, a, b)$
          where $ROTATE((P_r, a, b) = (u_1, u_2, \cdots, u_{a-1}, u_b, u_{b+1}, \cdots, u_q, u_a, \cdots u_{b-1})$ (See Figure 3.2)
  return $false$

DHAM required on a graph for each success plotted against the size of the graph. Though there is a slight increase in the no. of attempts required, it is not noticeable enough to draw any conclusions.



(a) Runtime plot



(b) No. of Attemps plot

Figure 3.3: DHAM running on instances from the $D_{n,m}$ model

We now try to increase the number of edges in the input graph (by varying the value of c). From Figure 3.3 we can see that the runtime does not seem to vary much, w.r.t the number of edges in the graph. This is what we would have predicted, since the algorithm considers only the first m^* edges to find the cycle, so extra edges should not affect the runtime.

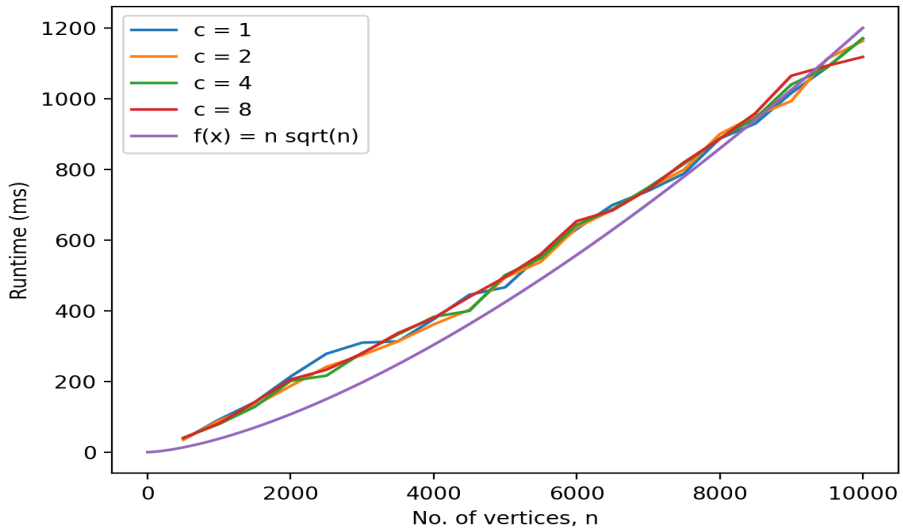
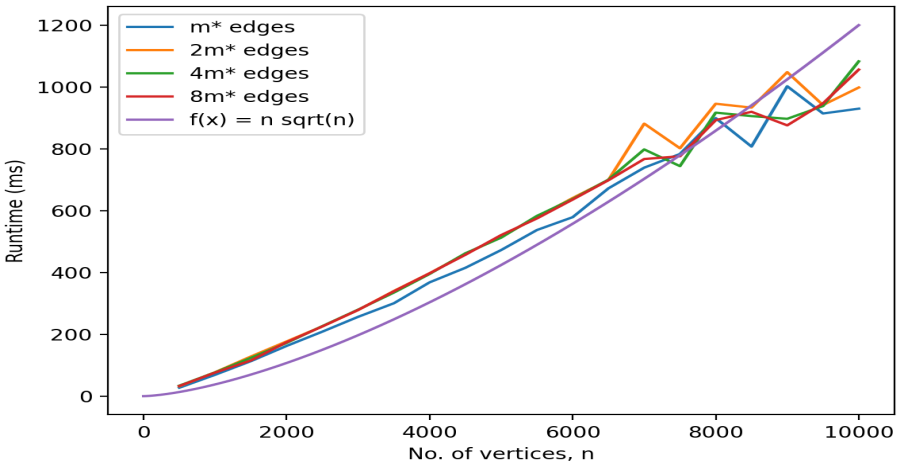


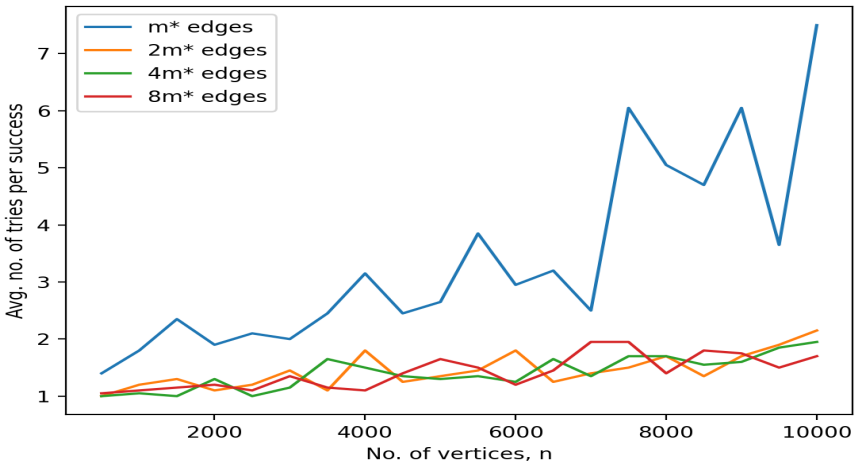
Figure 3.4: Varying the value of c

Another interesting thing to study is to modify the algorithm to consider more than just the first m^* edges. If we increase the number of edges by a constant factor, we could potentially see some constant factor improvements in runtime, since having access to a bigger set of edges could make a lot of our procedures terminate successfully sooner.

Running the test (Figure 3.5a) shows us that there is no such improvement in the runtime. But, what we see in Figure 3.5b is that the average number of times we need to run the algorithm to find a cycle drops significantly, down to almost 1, just by considering the first $2m^*$ edges instead of just the first m^*.



(a) Modifying DHAM to consider more edges - runtime



(b) Modifying DHAM to consider more edges - avg no.of attempts

Figure 3.5: modified DHAM

Chapter 4

Experimenting with Other Random Models

A method similar to the 3-phase method in DHAM has been adopted and used to show existential/constructive results in some other models of random digraphs. Now that we are familiar with the DHAM algorithms, we shall attempt to inpnut graphs from some of these models, and see how DHAM performs on them. We shall try to look for patterns in terms of accuracy, runtime, or number of attempts in the following models.

4.1 k-in, k-out Graphs

4.1.1 Graph Generation

The way $D_{k-in,k-out}$ is defined, it is pretty straightforward to generate instances of this graph. At every vertex $u \in V_n$, we simply have to generate 2 lists of k vertices each (as in-neighbours and out-neighbours), sampling uniformly at random from the set of all vertices. Now we add an edge from u to each out-neighbour and an edge from each in-neighbour to u.

4.1.2 Observations

Cooper and Frieze [4] have shown that $D_{2-in,2-out}$ is Hamiltonian with High Probability, using a 3-phase analysis.

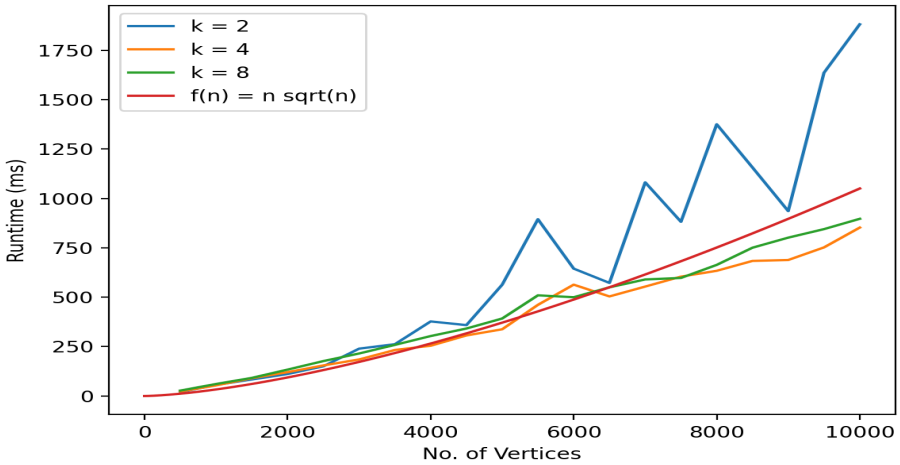
Phase 1 Show that $D_{2-in,2-out}$ contains a permutation digraph, Π of size at most

$2 \log n$.

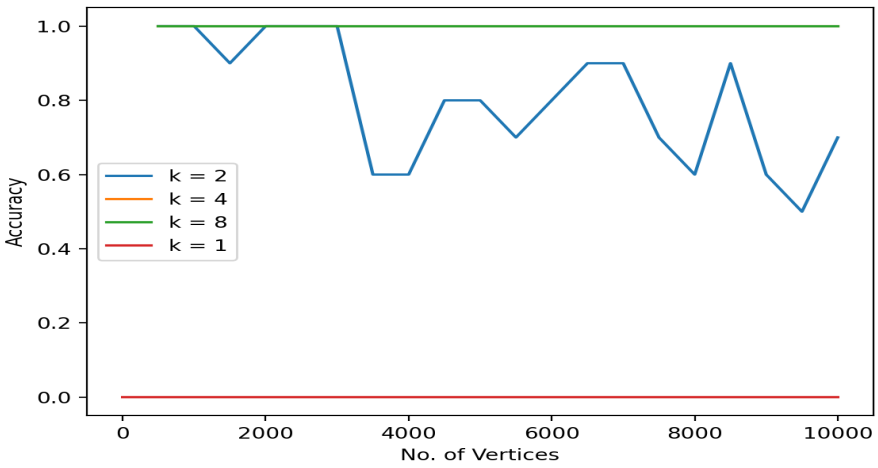
Phase 2 Increase the minimum cycle length in Π to $n_0 = \left\lceil \frac{1000n}{\log n} \right\rceil$.

Phase 3 Convert the Phase 2 permutation digraph to a Hamilton Cycle.

It is interesting to note that the proof follows a very similar outline to the one for Random Digraphs $D_{n,m}$, showing the existence of a Hamiltonian Cycle but, a similar bound on the runtime has not been obtained. It is still not known if there is a polynomial time algorithm to find the same.



(a) Runtime plot of $D_{k-in,k-out}$



(b) Probability of finding a Hamilton cycle in $D_{k-in,k-out}$

Figure 4.1: DHAM on $D_{k-in,k-out}$

We try to test if DHAM might be successful in finding this Hamiltonian cycle in polynomial time. From Figure 4.1a we can see that (to the extend of n that is feasible for testing) that runtime grows more or less the same as on $D_{n,m}$. This is suggestive of the fact that it might be possible to give a polynomial bound the runtime of 3-phase method on this class of graphs.

We also see some spikes in the runtime and also a lower probability of finding the Hamiltonian cycle in Figure 4.1b for $D_{2-in,2-out}$, even though we know that this graph is Hamiltonian with high probability. This phenomenon is observed again in the next model we take a look at and seems to suggest that the algorithm does not run as well on sparse graphs.

4.2 k-regular Graphs

4.2.1 Graph Generation

To generate a k-regular digraph, we need to make sure that the in-degree and the out-degree of each vertex are k. But simply choosing $2k$ neighbours independently like for $D_{k-in,k-out}$ does not work, since we cannot guarantee that exactly k other vertices will also be in-neighbours.

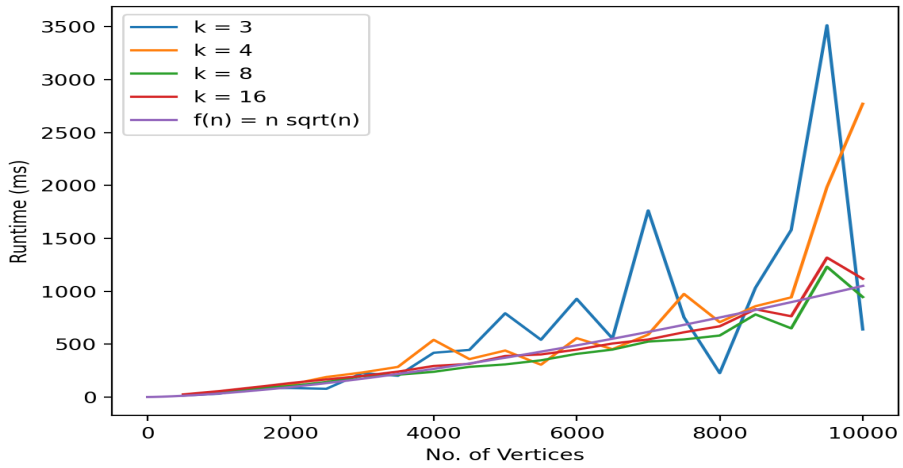
Instead, we notice that in a permutation digraph, each vertex has exactly 1 in/out neighbour. We use this fact and take the union of k random permutation digraphs (which have no mutual overlap) to obtain a k-regular digraph.

4.2.2 Observations

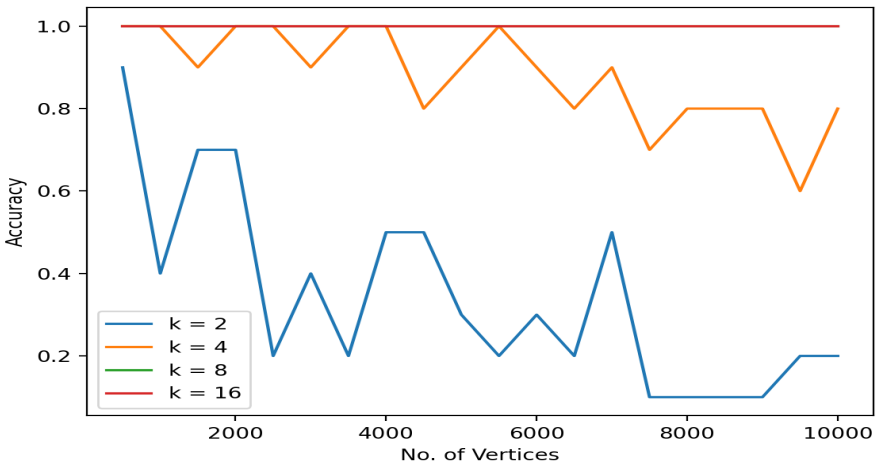
Theoretically, it has been shown that $D_{n,r}$ is Hamiltonian with high probability for $r \geq 3$ [5]. The proof for this is again based on a 3-phase method and is very similar to that for $D_{k-in,k-out}$, since large k-regular graphs look very similar to $D_{k-in,k-out}$ graphs. Hence it is not very surprising that their runtime plots look similar. The runtime very closely follows the same $n^{1.5}$ curve that $D_{n,m}$ had, suggesting that it might be possible to give a polynomial bound for DHAM on this class of graphs too.

Once again we notice the spikes in runtime and the lower probabilities of success for $k = 2, 3$ in Figure 4.2b, reiterating the idea that DHAM performs poorly on sparse graphs. We can trace this bad performance to the first phase of the algorithm,

where we construct a subgraph of average degree 10. The bipartite matching that follows depends on the degree of vertices in this subgraph, and a lower degree limits the success probability of the matching, and hence bottle-necking DHAM.



(a) Runtime plot of $D_{n,r}$



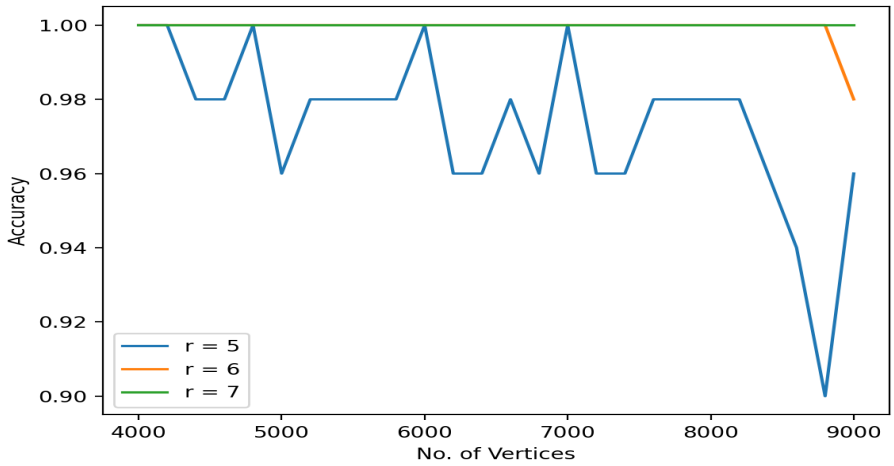
(b) Probability of finding a Hamilton cycle in $D_{n,r}$

Figure 4.2: DHAM on $D_{n,r}$

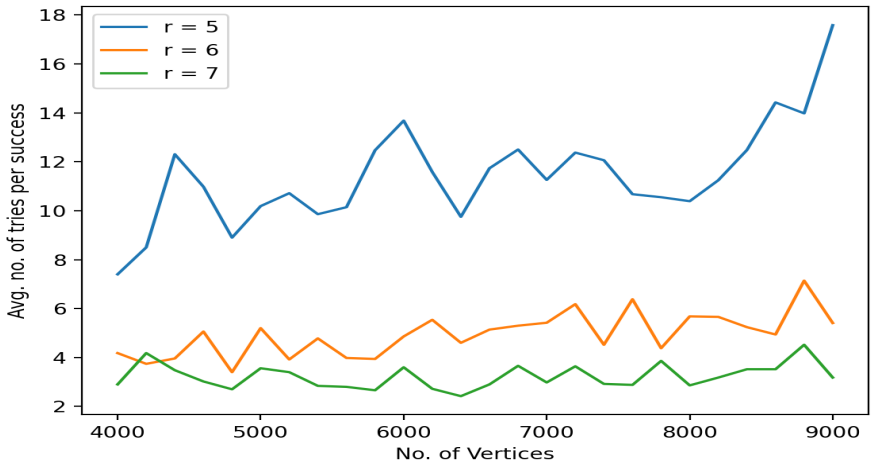
4.2.3 Closer look at sparse graphs

Since we notice the DHAM performs poorly on sparse graphs, but still works just fine at value below the mentioned degree 10, in the original paper, we take a

closer look to see above what value of r, does DHAM find a cycle in $D_{n,r}$, with high probability and low number of attempts.



(a) Probability of Finding a cycle in sparse $D_{n,r}$



(b) Average no. of attempts to find the cycle

Figure 4.3: $D_{n,r}$ with low values of r

From Figure 4.3 we see that for $r \geq 6$, the probability of finding a cycle is almost at 1, and the number of attempts also stabilises around 5.

4.3 Random Tournaments

4.3.1 Graph Generation

We can generate every possible pair of vertices, using a nested loop, and assign a direction to every edge uniformly at random to obtain the desired tournament.

4.3.2 Observations

Random Tournaments have been shown to have $\delta = min(\delta^+, \delta^-)$ edge disjoint Hamiltonian cycles[13]. Though the proof does not rely on the 3-phase method, DHAM using that method, still does find cycles in such graphs (Figure 4.4), with high probability (probability 1 in our tests).

Tournaments grow in size as a function of n^2, limiting the range of n feasible for testing. From the range of sizes that we tested on it is not possible to comment on the runtime on these graphs (since the implementation overhead might be affecting the runtime more than the algorithm itself). But we do notice that every graph generated returned a valid Hamiltonian cycle, which seems to suggest that the probability of success of the 3-phase method is also very high on random tournaments. Also, since tournaments are dense graphs, we do not run into the bottlenecks that we observed in the last 2 models.

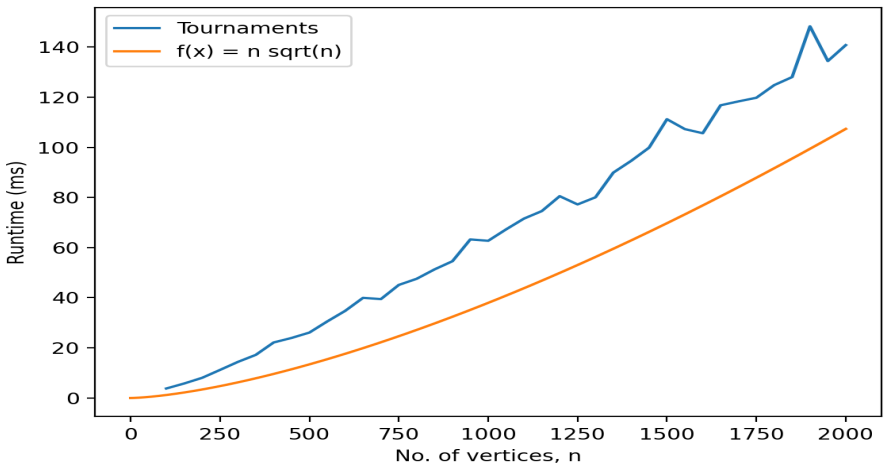


Figure 4.4: Runtime plot of Tournaments

4.4 Random n-lifts of complete digraph K_h

4.4.1 Graph Generation

We first create the complete digraph, K_h on h vertices. Now, we start constructing a new digraph D, with $h \times n$ vertices, with the range of vertices $[i \times n, i \times (n+1) - 1]$ corresponding to each vertex $i \in V(K_h)$.

Now, for each edge $(u, v) \in E(K_h)$, we construct arrays U & V, of the vertices corresponding to $u, v \in V(K_h)$. After shuffling the arrays, we obtain a random matching as $\forall i < n, (U[i], V[i])$, which is now added to the graph D, to obtain the required n-lift of K_h.

4.4.2 Observations

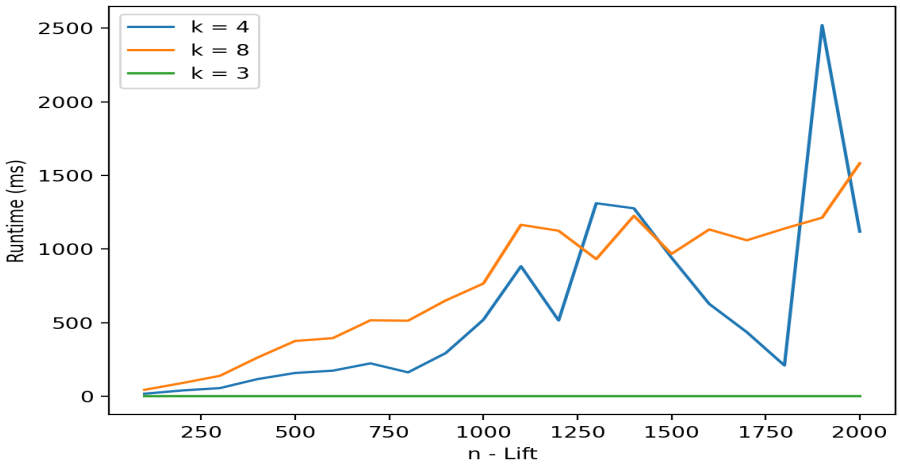
This is another class of graphs that proved to be Hamiltonian [3] (for sufficiently large values of h) by using the 3-phase method, with appropriate modifications.

Phase 1 Show that with high probability the lift D contains a directed permutation digraph, Π of size at most $2 \ln n$.

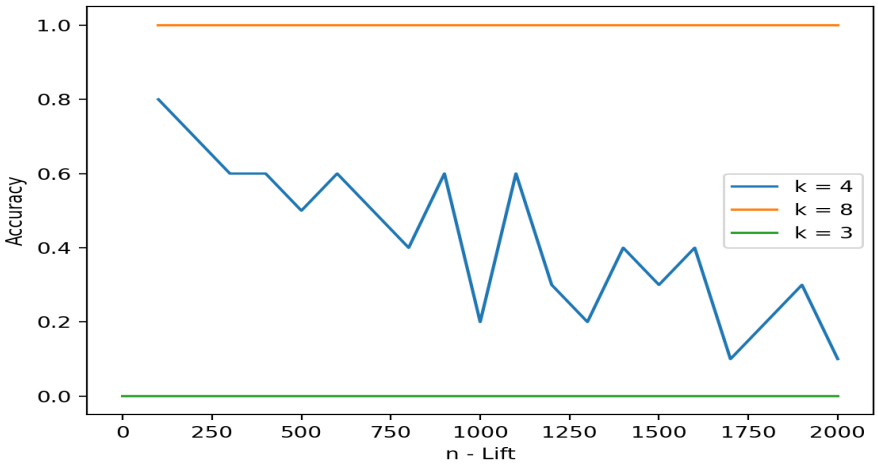
Phase 2 Increase the minimum cycle length in Π to $n_0 = \left\lceil \frac{100nh^3}{\ln n} \right\rceil$.

Phase 3 Convert the Phase 2 permutation digraph to a Hamilton Cycle.

We make some interesting observations here. The first is that for $k = 8$ we see that all the generated graphs are Hamiltonian, which could be considered a more precise lower bound for h than the *sufficiently large h* . But more importantly, none of the n-lifts of K_3 are Hamiltonian. Frieze in his survey[10], has posed the problem to show that n-lifts of K_3 are Hamiltonian. What we observe with DHAM, is that the probability of lifts of K_3 being Hamiltonian is zero. Though this particular algorithm failing, need not imply that n-lifts of K_3 are not Hamiltonian (indeed it could be because of the phenomenon of DHAM doing poorly on sparse graphs, with us also observing the runtime spikes, and low probability at $k = 4$), it is an interesting observation nevertheless and worth taking a deeper look into.



(a) Runtime plot of n-lifts



(b) Probability of finding a Hamilton cycle in lifts of K_h

Figure 4.5: DHAM on n-lifts

Chapter 5

Conclusion and Future Work

5.1 Conclusion

After having looked at DHAM with respect to different classes of graphs, we did not observe a exponential rate of growth for any of these models (upto the sizes of n, that are practicall to simulate). Our results suggest that DHAM successfully finds the cycle in most Hamiltonian graphs, though the probability of finding such a cycle starts to drop with the minimum degree in the graph dropping below 7.

We also verified the original claims made by the algorithm and noted that slightly modifying the algorithm to consider a constant multiple more edges could reduce the number of attemps required to find a Hamilton Cycle without any significant impact on the runtime.

5.2 Future Research Directions

With encouraging results like described above, one future direction of research is to theoretically bound the runtime of this algorithm on the models of graphs we looked at.

Another interesting direction to look at is to see if the $\mathcal{O}(n \log n)$ bipartite matching algorithm given by Karp, Rinnooy-Kan, and Vohra[11] can be used to improve the runtime bound on phase 1 of DHAM. Though the runtime does look better at first sight, we are not sure if the runtime/correctness analysis will work out cleanly.

Bibliography

[1] C. Aragon and R. Seidel, "Randomized search trees", *Proc. 30th IEEE Symposium on Foundations of Computer Science*, 1989.

[2] B. Bollobas, "The evolution of sparse graphs, graph theory and combinatorics", *Cambridge Combinatorial Conference in Honor of Paul Erdős*, pp. 335–357, 1984.

[3] P. Chebolu and A. Frieze, "Hamilton cycles in random lifts of complete directed graphs", *SIAM Journal on Discrete Mathematics 22*, pp. 520–540, 2008.

[4] C. Cooper and A. Frieze, "Hamilton cycles in random graphs and directed graphs", *Random Structures and Algorithms*, vol. 16, pp. 369–401, 2000.

[5] C. Cooper, A. Frieze, and M. Malloy, "Hamilton cycles in random regular digraphs", *Combinatorics, Probability and Computing*, vol. 3, pp. 39–50, 1994.

[6] Y. Dinitz, "Algorithm for solution of a problem of maximum flow in networks with power estimation", *Soviet Math. Dokl.*, vol. 11, pp. 1277–1280, Jan. 1970.

[7] P. Erdős and A. Ŕenyi, "On the evolution of random graphs", *Publ. Math.Inst. Hungar. Acad. Sci.*, vol. 5, pp. 17–61, 1960.

[8] L. Euler, "Solutio problematis ad geometriam situs pertinentis", *Commentarii academiae scientiarum Petropolitanae*, vol. 8, pp. 128–140, 1741.

[9] A. Frieze, "An algorithm for finding hamilton cycles in random directed graphs", *Journal of Algorithms*, vol. 9, no. 2, pp. 181–204, 1988, ISSN: 0196-6774.

[10] A. Frieze, "Hamilton cycles in random graphs: A bibliography", 2019. arXiv: `1901.07139 [math.CO]`.

[11] R. M. Karp, A. H. G. R. Kan, and R. V. Vohra, "Average case analysis of a heuristic for the assignment problem", *Mathematics of Operations Research*, vol. 19, no. 3, pp. 513–522, 1994.

[12] J. Komlós and E. Szemerédi, "Limit distributions for the existence of hamilton circuitsin a random graph", *Discrete Mathematics*, vol. 43, pp. 55–63, 1983.

[13] Kuhn and Osthus, "Hamilton decompositions of regular expanders: Applications", *Journal of Combinatorial Theory*, pp. 1–27, 2014.

[14] L. Pósa, "Hamiltonian circuits in random graphs", *Discrete Mathematics*, vol. 14, pp. 359–364, 1976.

Appendix A

Storing a Path as a Tree

In phase 3 (Algorithm 3) of the DHAM, we construct a path out of the 2 cycles being patched and search (DFS/BFS) the tree formed on this path by double rotation operations. Now, there are 2 parts to this problem: One is to store the path in such a way that the double rotations operations can be done efficiently, and the other is to identify valid double rotations, given such a path.

If we make the obvious decision to store the path as a link list to make the double rotation operation constant time (by just swapping 2 of the links), the process of identifying a valid operation ends up being linear due to having to traverse the path.

Instead, if we store the path as an array that making generation of valid operations faster, we realize that executing the operation itself now needs updating up to linear number of elements in the array.

To get around this difficulty, we store our path as a Randomized Search Tree [1] to get a $\mathcal{O}(\log n)$ bound on both of the operations.

A.1 Randomized Search Tree

The Randomized Search Tree (or treap) as given by Aragon and Seidel [1] is based on 2 auxiliary operations - split (around a given key) and merge (assuming elements within the trees are in order of keys), to implement the main operations like insertion, deletion, union etc. The reason we choose this data structure is because we can use these auxiliary operations to also implement a double rotation by 'splitting' out a sub-array, and 'merging' it at the end.

To attain all the desired functionality, we make some modifications to the data structure

A.1.1 Implicit Keys

Doing a double rotation changes the position of elements in the structure. In this situation the keys(or index) of these elements becomes inaccurate. So, instead of storing the key as a value within the node, we maintain the keys of all nodes implicitly.

This is achieved by storing the size of the tree rooted at a node, and using this value to compute keys on the fly.

The updated node looks like

```
struct Node {
    Node *left, *right, *parent;
    int val, y, subtree_size = 1;
    void recalc();
};
int cnt(Node* n) { return n ? n->subtree_size : 0; }
void Node::recalc() { c = cnt(l) + cnt(r) + 1; }
```

The key of a node is the number of elements with a smaller key, which can be looked on as the number of elements in the left subtree (because of the BST structure). So while traversing down the tree, we can keep track of the sum of their sizes (every time we traverse to the right) to implicitly maintain the key of a node we are currently visiting. Modifying the split function accordingly, we get

```
pair<Node*, Node*> split(Node* n, int k)
        if (!n) return {};
        if (cnt(n->l) >= k) {
                auto pa = split(n->l, k);
                n->l = pa.second;
                n->recalc();
                return {pa.first, n};
        } else {
                auto pa = split(n->r, k - cnt(n->l) - 1);
                n->r = pa.first;
                n->recalc();
                return {n, pa.second};
        }
```

The merge function remains largely the same since we do not access the keys of a node, assuming them to already be in order.

A.1.2 key $\iff$ value conversions

With keys being automatically updated, we only need to add the functionality to return the value given a key (accessing an element) and return a key given the value (search operation). These operations will allow us to generate valid operations efficiently. These can simply be implemented in $\mathcal{O}(\log n)$ as shown below

```
int key(Node* root, Node* x) {
    if (tree==nullptr || x==nullptr) return -1;
    int ans = cnt(x->l);
    while(x != root) {
        auto par = x->p;
        if (par->r == x)
            ans += 1 + cnt(par->l);
        x = par;
    }
    return ans;
}

int value(Node *n, int key) {
    if (!n) return -1;
    if (cnt(n->l) == key) return n->val;
    else if (cnt(n->l) > key) return value(n->l, key);
    else return value(n->r, key - cnt(n->l) - 1);
}
```

A.2 Double Rotation

Armed with the tools described above, we can implement the double rotation operation to move the subarray $[l, r)$ to the index k in $\mathcal{O}(\log n)$ time.

```
void move(Node*& t, int l, int r, int k) {
    Node *a, *b, *c;
    tie(a,b) = split(t, l); tie(b,c) = split(b, r - l);
    if (k <= l) t = merge(ins(a, b, k), c);
    else t = merge(a, ins(c, b, k - r));
}
```