An efficient approach to the Hamiltonian path problem

Alexandru Ispir

July 2022

Abstract: In this paper a randomized heuristic $O(|E|\log|V|)$ (average-case time complexity) algorithm is presented which finds a Hamiltonian path in an undirected graph with high frequency of success for randomly generated graphs. The idea behind the algorithm is to repeatedly extend a simple path, such that it eventually becomes Hamiltonian.

Keywords: NP, NP-hard, Hamiltonian path, longest path, DFS, DFS tree, heuristic, dynamic connectivity, probabilistic algorithms

1 Introduction

The undirected Hamiltonian path problem is the problem of finding a path in an undirected graph that visits each vertex exactly once. This one is a part of the Karp's 21 NP-complete problems [1], so it is well known for its difficulty.

Exact statement of the problem

Let $G = (V, E)$ be an undirected graph, in which V represents the set of vertices and E represents the set of edges. Find a simple path which traverses all vertices exactly once or report if it does not exist.

The importance of heuristics

Generally, solutions for NP-complete problems are slow and impractical. Therefore, sometimes, we need to sacrifice the optimality of the answer for efficiency. Those described algorithms are also known as heuristics and they help us find good enough solutions for large instances of these NP-complete problems.

Other approaches to this problem

Other heuristics solving the same task are Angluin's and Valiant's fast probabilistic algorithm [5] and Minram probabilistic algorithm [6]. Those algorithms are efficient and solve large instances of the Hamiltonian path problem for graphs with $O(|V| \log |V|)$ edges. Different from those heuristics, the presented algorithm introduces a novel idea which consists in augmenting the result until it becomes optimal.

Summarising the algorithm

The presented heuristic algorithm extends a simple path many times, which makes it progressively longer. With some optimisations, this method works excellently (both in execution times and the optimality of the returned result) on random graphs (relatively sparse or dense).

Even though the algorithm might fail finding a Hamiltonian path on some special cases, it will return a near-optimal result or optimal result in most cases.

If it returns an unsatisfactory result, it can also be executed any number of times because it will most likely return a different / better result.

2 Concepts and definitions

Before continuing, we need to (re)view some concepts and definitions that are needed for the understanding of the algorithm. Also, every time a "graph" is mentioned, it can be interpreted as an undirected simple graph.

Definition 1. A path in an undirected graph is a finite or infinite sequence of edges which joins a sequence of vertices.

Definition 2. The length of a path is defined as the number of edges it contains.

Definition 3. A simple path is a path that contains no repeated vertices.

Definition 4.1. A Hamiltonian path is a simple path which contains $|V| - 1$ edges, where $|V|$ represents the number of vertices in the graph.

Definition 4.2. A Hamiltonian path is a simple path which traverses all the vertices exactly once in a graph.

Definition 5. Two vertices i and j are considered neighbouring (or adjacent) if and only if they are connected by an edge $e = [i, j]$, in the **initial** graph (the graph before being affected by any edge operations).

Definition 6. The associated (or corresponding) edges of a vertex v are the edges of the form $e = [v, i]$, such that $e \in E$.

Definition 7. A connected graph is a graph in which there exists a simple path between all pairs of vertices $(u, v) \in V \times V$.

Definition 8. A tree is a connected graph with $|V| - 1$ edges, where $|V|$ represents the number of vertices in the graph.

Definition 9. A subgraph is a graph whose vertices and edges are subsets of another graph. Formally, a graph $G' = (V', E')$ is a subgraph of another graph $G = (V, E)$ if and only if $V' \subseteq V$ and $E' \subseteq E$.

Definition 10. A spanning tree of a connected graph G is a subgraph that is a tree which includes all of the vertices of G.

Definition 11. A **DFS** tree of a connected graph G is a spanning tree which contains all the edges used by the **DFS** procedure (applied on G) [5].

Definition 12. The diameter of a tree is the length of the longest simple path between two vertices u and v, $(u, v) \in V \times V$.

Definition 13. A connected component in a graph G is defined as a connected subgraph G' that is not part of any larger connected subgraph.

Definition 14. An articulation point in a connected graph G is defined as a vertex which, when removed along with its associated edges, makes the graph disconnected.

Definition 15. A connected graph is biconnected, if and only if it does not contain any articulation points.

Definition 16. The degree of a vertex v is defined as the number of corresponding edges of v.

Axiom 1. The shortest simple path between two distinct vertices (in a connected graph) contains at least one edge.

Axiom 2. The shortest simple path between two distinct non-neighbouring vertices (if it exists) contains at least two edges.

Axiom 3. Two vertices u and v are in the same connected component if and only if there exists a simple path that connects u and v.

Axiom 4. In a tree, there exists exactly one simple path between two vertices u and v, for any $(u, v) \in V \times V$.

Lemma 1. The extending part of the algorithm will always make a path longer.

The process of extending a path consists in eliminating an edge $e = [i, j]$ from it, making the vertices i and j non-neighbouring and reconnecting the separated paths with another path between i and j, such that the resulting path remains simple.

Proof: Let's assume, at some point, the algorithm found a path with n edges. After the elimination of the edge $e = [i, j]$, the path is separated in two distinct paths, the total length of them being $n - 1$. Using **Axiom 2.**, we find that the algorithm will add at least two edges (if it's possible), such that they connect the separated paths. So, the minimum amount of edges in the new path will be equal to $n - 1 + 2 = n + 1 > n$.

Lemma 2. The maximum number of extensions used by the algorithm is equal to $O(|V|)$.

Proof: Let's assume the algorithm starts with a simple path with only one edge (see **Axiom 1.**). After every extension, the length of the path will increase with at least one edge. So, if the path eventually becomes Hamiltonian, the algorithm will execute at most $(|V| - 1) - 1 = |V| - 2$ (the maximum length minus the minimum length of the path) extensions.

Lemma 3. If there does not exist a valid extension between i and j at a given time, it's pointless to check it if exists in future paths.

Proof: When the algorithm seeks for an extension between i and j, it's not allowed to find a path which intersects with the separated paths (it does not respect the condition of keeping the extended path simple). Let's call the set

of vertices that are connected by the separated paths S. If there is not a valid extension that avoids all vertices in S, it's obvious that there is not a valid extension that avoids all vertices in S', where $S \subset S'$.

3 The initial algorithm

Initially, two random vertices x and y will be chosen. Then, the algorithm will search for a simple path between them. This can be done with a depth first search (**DFS**) [5] procedure.

After this, the algorithm will try to extend the initial path multiple times, until the path becomes Hamiltonian or it's not possible to extend it anymore. If the path becomes Hamiltonian, an optimal answer is given. Otherwise, a suboptimal answer is returned.

The algorithm will look like this:

Step 0: Choose two random distinct vertices x and y. Go to step 1.

Step 1: Find a simple path P, such that $P \subseteq E$ and P connects x and y. Go to step 2.

Step 2: For every edge $e = [i, j]$ in P, find a path P' that connects i and j, such that P' does not contain the edge $e = [i, j]$ and the path $(P \setminus e) \cup P'$ remains simple. If a valid path P' is found, go to step 3. If there is none, go to step 4.

Step 3: The path P becomes $(P \setminus e) \cup P'$. Go to step 2.

Step 4: If $|P| = |V|-1$, an optimal result was found. Otherwise, the algorithm found a suboptimal result. In both cases, return P.

4 Analysis and optimisations

The first part of the algorithm has $O(|E|)$ complexity because only one **DFS** will be executed.

For every path, the algorithm will try to find a valid extension between every two neighbouring vertices. The number of paths that need to be checked is $O(|V|)$ (because the maximum number of extensions is $O(|V|)$, see **Lemma 2.**). Each path has $O(|V|)$ edges. For every edge $e = [i, j]$, the algorithm will try to find a valid extension between i and j (which can be done in $O(|E|)$, using **DFS**). The complexity of this part becomes: $O(|V|^2|E|)$.

The complexity of the algorithm is: $O(|E| + |V|^2|E|) = O(|V|^2|E|)$.

Observation 1. The number of edges (or pairs of neighbouring vertices) the algorithm checks in **3.2.** can be reduced from $O(|V|^2)$ to $O(|V|)$ (see **Lemma 3.**).

The existence of a valid extension between two neighbouring vertices can be checked only once. If an extension between two neighbouring vertices does not exist, it does not make sense to recheck it in future paths. So, the complexity was reduced from $O(|V|^2|E|)$ to $O(|V||E|)$.

Observation 2. The algorithm can be further optimised using **online fully dynamic connectivity** [3] to efficiently update edges / find if there exists a valid extension between two neighbouring vertices in a path.

The data structure described earlier can add / remove edges in $O(\log^2 |V|)$ (amortized complexity) and check if two vertices are in the same connected component in $O(\frac{\log |V|}{\log \log |V|})$ (using B-trees [9]).

When a path connects a vertex, the algorithm will eliminate all the edges adjacent to it. When it will check if there exists a valid extension between two neighbouring vertices in the path, it will add all the edges adjacent to them except the edges that make part of the path. Then, it will query if they are located in the same connected component. If so, the algorithm will extend the path using a **DFS** procedure (furthermore, it is very important to halt the procedure immediately after we find a valid path). After the query, it will remove all the edges that were added just before the query.

The number of updates will be equal to $O(|E|)$ because:

- The number of adjacent edges removed by the connected vertices is at most $O(|E|)$.
- The number of edges added / removed before the queries is also $O(|E|)$ (because an edge adjacent to a vertex will appear at most 4 times in the list of required updates before / after the queries).

The extending part of the algorithm has an average-case complexity of $O(|E|)$ because the number of extensions and the number of operations executed by all **DFS**'s is relatively low (in the average-case).

The number of queries is $O(|V|)$ (see **Observation 1.**).

Including the first **DFS** procedure, the resulted average-case complexity becomes:

$$O(|E| + |E| \log^2 |V| + |E| + |V| \frac{\log |V|}{\log \log |V|}) = O(|E| \log^2 |V|).$$

5 A better algorithm

After applying all the optimisations mentioned in Section **4**, the algorithm becomes:

Step 0: Choose two random distinct vertices x and y. Go to step 1.

Step 1: Find a simple path P, such that $P \subseteq E$ and P connects x and y. Go to step 2.

Step 2: Remove all the existent edges adjacent to vertices that are connected to P. Go to step 3.

Step 3: For every unmarked edge $e = [i, j]$ in P, add all the edges adjacent to i and j, except those that make part of P. Query if there exists a path between i and j in the new graph. If so, find a path P' that connects i and j and go to step 4. Else, mark the edge $e = [i, j]$. If there is none, go to step 5.

Step 4: The path P becomes $(P \setminus e) \cup P'$. Go to step 2.

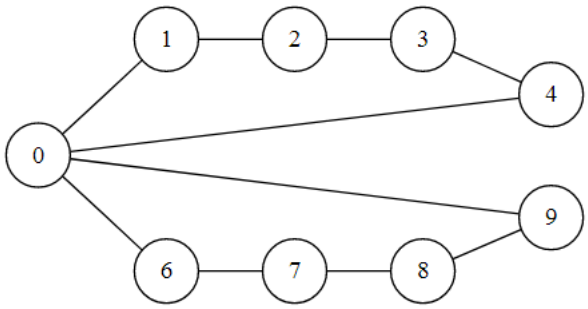
Step 5: If $|P| = |V|-1$, an optimal result was found. Otherwise, the algorithm found a suboptimal result. In both cases, return P.

All the updates / queries were done using the data structure described earlier.

If the algorithm does not return a satisfactory result, we can re-execute it any number of times. Assuming this number is equal to k, the complexity becomes: $O(k\,|E|\log^2|V|)$.

6 Further optimisations

Observation 3. In the first stage of the algorithm, choosing two random vertices x and y is naive. There are some graphs which "confuse" the algorithm, like this one:



For example, the algorithm can choose $x = 2$ and $y = 7$. The initial path could be $P = \{[2, 1], [1, 0], [0, 6], [6, 7]\}$. The path can't be extended, so the algorithm halts and returns a suboptimal result.

For these kinds of "confusing" graphs, we can implement the following trick:

Step 0: If the given graph is not biconnected, pick a random articulation point v (with minimum degree) from the graph. Else, pick a random vertex v (with minimum degree) from the graph. Go to step 1.

The problem of finding all articulation points in a graph can be solved with Hopcroft's and Tarjan's algorithm [10] in $O(|E|)$ time.

Step 1: Find the **DFS** tree of the graph, starting from v. Go to step 2.

Step 2: Find the longest path in the **DFS** tree. This path will represent the initial path (before any extensions). This can be solved with Dijkstra's algorithm (for finding the diameter / longest path in a tree) [7].

This trick not only solves the problem for "confusing" graphs, but also improves the overall result returned by the algorithm for randomly generated graphs.

Observation 4. The algorithm only needs **online decremental dynamic connectivity** [8], instead of **online fully dynamic connectivity**.

This data structure can remove edges in $O(\log |V|)$ (amortized complexity) and query if two vertices are in the same connected component in $O(1)$ (using a table t, t_i representing the index of the connected component which contains i, at a given time, for any $i \in V$).

The algorithm remains essentially the same, but with a slight modification: if we search for a valid extension, for every unmarked edge $e = [i, j]$, instead of adding back all the edges adjacent to i or j (except those that make part of P), we will define two sets S_i and S_j (S_x saves all the distinct values of t_k, for any k - neighbour of x and k is not connected to P, $x \in \{i, j\}$).

Let T be $S_i \cap S_j$. If $T = \emptyset$, then no valid extension exists and we will mark the edge $e = [i, j]$. Else, choose any value l from T, and search two vertices u and v, such that $t_u = t_v = l$, u is a neighbour of i, v is a neighbour of j. Furthermore, u and v must not be connected to P. Next, find a simple path P' between u and v without using the edges eliminated by the data structure (it is guaranteed that there exists at least one path P', see **Axiom 3.**). P' becomes $P' \cup \{[i, u], [v, j]\}$. Finally, we found a valid extension, so P becomes $(P \setminus e) \cup P'$ and we eliminate all the edges corresponding with the new added vertices.

The part of searching for two vertices u and v, can be done in $O(|S_i| + |S_j|)$, using a hash table. In the worst-case, the seeking part of the algorithm will use the set S_i at most twice, for any $i \in V$. Therefore, the complexity of this part is: $O(2 \sum |S_i|) = O(|E|)$.

The complexity of extending the path remains the same as before: $O(|E|)$ time in the average-case.

For the other part (removing of the edges), in the worst-case, the algorithm will eliminate all the edges. As a consequence, the complexity of this part is: $O(|E| \log |V|)$.

Including the articulation point trick, the final average-case complexity of the algorithm becomes: $O(|E| + |E| + |E| + |E| \log |V|) = O(|E| \log |V|)$.

7 The final algorithm

The final algorithm will have the following structure:

Step 0: If the given graph is not biconnected, pick a random articulation point v (with minimum degree) from the graph. Else, pick a random vertex v (with minimum degree) from the graph. Go to step 1.

Step 1: Find the **DFS** tree of the graph, starting from v. Go to step 2.

Step 2: Find the longest path P in the **DFS** tree. This path will represent the initial path (before any extensions). Go to step 3.

Step 3: Remove all the existent edges adjacent to vertices that are connected to P. Go to step 4.

Step 4: For every unmarked edge $e = [i, j]$ in P, query if there exists two vertices u (neighbour of i) and v (neighbour of j), such that u and v are not connected by P and u is in the same connected component as v. If so, find a path P' that connects u and v, using non-removed edges and go to step 5. Else, mark the edge $e = [i, j]$. If there is none, go to step 7.

Step 5: The path P' becomes $P' \cup \{[i, u], [v, j]\}$. Go to step 6.

Step 6: The path P becomes $(P \setminus e) \cup P'$. Go to step 3.

Step 7: If $|P| = |V| - 1$, an optimal result was found. Otherwise, the algorithm found a suboptimal result. In both cases, return P.

To avoid ill-intentioned tests, we can shuffle the edges of the input graph. In addition, just like before, if the algorithm does not return a satisfactory result,

we can re-execute it any number of times. Assuming this number is equal to k, the complexity becomes: $O(k|E|\log|V|)$.

8 Testing the algorithm

The solution was tested on random:

- Dense graphs (containing $O(|V|^2)$ edges)
- Relatively sparse graphs (containing $O(|V|\log|V|)$ edges [2])
- Sparse graphs (containing $O(|V|)$ edges)

On dense graphs, an optimal result was always found. On relatively sparse graphs, the algorithm also returned optimal results with a very high frequency of success. On sparse graphs, the solution failed, but returned a decent suboptimal result.

Note: All the graphs used as tests contained at least one Hamiltonian path.

9 Questions for future research

Question 1. It's evident that the worst-case complexity of the algorithm is $O(|V||E|)$ (the **DFS**'s that help extending the path). Even though the described algorithm is really fast in practice ($O(|E|)$ average-case complexity for all **DFS**'s), is it possible to achieve a better worst-case time complexity for this part?

Question 2. Is it possible to achieve worst-case / average-case $O(|E|\,\alpha(|V|))$ (or better) complexity, by changing the data structure used in this algorithm ($\alpha(n)$ represents the inverse Ackermann function)?

Question 3. Is there a method of assigning each extension a priority, such that it improves the overall result returned by the algorithm?

Question 4. This problem indirectly tries to solve the longest path problem, which is another NP-hard problem. I conjecture that the approximation ratio for this algorithm is 2. Is it true?

Question 5. Same as **Question 4.**, but this time I conjecture a weaker statement: the approximation ratio for this algorithm is $O(1)$. Is it true?

Note: For the last two questions, keep in mind that the edges can be shuffled (to avoid ill-intentioned tests).

10 Conclusion

In this paper, I have described a successful idea / algorithm for finding a Hamiltonian path in an undirected connected graph.

The algorithm starts with a simple path and then extends it multiple times. If the path eventually becomes Hamiltonian, an optimal result is found. Else, the algorithm fails, but finds a suboptimal result.

Also, the algorithm can be adapted to find Hamiltonian circuits in undirected graphs. Initially, instead of finding a path between two random distinct vertices, we can find a cycle and extend it from there using the same concept.

On random graphs with at least $O(|V| \log |V|)$ [2] edges, the algorithm does an excellent job at providing quickly an optimal answer.

11 Special thanks

I would like to thank Dr. Daniela Marcu for constructive criticism of the document and my family and friends for supporting me.

12 References

[1] - Richard M. Karp (1972). "Reducibility Among Combinatorial Problems". In R. E. Miller; J. W. Thatcher; J.D. Bohlinger (eds.). Complexity of Computer Computations. New York: Plenum. pp. 85-103. ISBN 978-1-4684-2003-6.

[2] - L. Posa, Hamiltonian circuits in random graphs, Discrete Math. 14 (1976) 359-364

[3] - Jacob Holm, Kristian de Lichtenberg, Mikkel Thorup: Poly-logarithmic deterministic fully dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. J. ACM 48(4): 723-760 (2001)

[4] - Charles Pierre Trémaux (1859-1882) École polytechnique of Paris (X:1876), French engineer of the telegraph in Public conference, December 2, 2010 - by professor Jean Pelletier-Thibert in Académie de Macon (Burgundy - France) - (Abstract published in the Annals academic, March 2011 - ISSN 0980-6032)

[5] - D. Angluin and L.G. Valiant, Fast probabilistic

algorithms for Hamiltonian circuits and matchings, J. Comput. System Sci. 18 (1979) 155-193.

[6] - Gerald L. Thompson and Sharad Singhal, A successful algorithm for the undirected hamiltonian path problem, Discrete Math. 10 (1985) 179-195

[7] - Bulterman, R.W.; van der Sommen, F.W.; Zwaan, G.; Verhoeff, T.; van Gasteren, A.J.M. (2002), "On computing a longest path in a tree", Information Processing Letters, 81 (2): 93-96

[8] - Shiloach, Y.; Even, S. (1981). "An On-Line Edge-Deletion Problem". Journal of the ACM. 28: 1-4

[9] - Bayer, R.; McCreight, E. (July 1970). "Organization and maintenance of large ordered indices" (PDF). Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control - SIGFIDET '70. Boeing Scientific Research Laboratories. p. 107

[10] - Hopcroft, J.; Tarjan, R. (1973). "Algorithm 447: efficient algorithms for graph manipulation". Communications of the ACM. 16 (6): 372-378