Fast algorithm for finding Hamiltonian cycles

Risto A. Paju
http://algoristo.com/
Jyväskylä, Finland
risto.a.paju@iki.fi

March 4, 2024

Abstract

I present a two-stage approach to finding Hamiltonian cycles in low polynomial time. The first stage generates a set of disjoint cycles over all vertices, and the second stage merges them into the full Hamiltonian cycle.

1 Introduction

A Hamiltonian cycle over a graph G is defined as walk that visits every vertex of G exactly once, returning to the starting vertex [1, p. 87]. It is essentially the Traveling Salesman Problem, simplified by having equal edge lengths. The general problem of finding Hamiltonian cycles is considered to be exponentially hard or NP-complete. I present a simple and intuitive method that can solve this problem in a low polynomial time.

2 Cycle finding

2.1 *Initial observations*

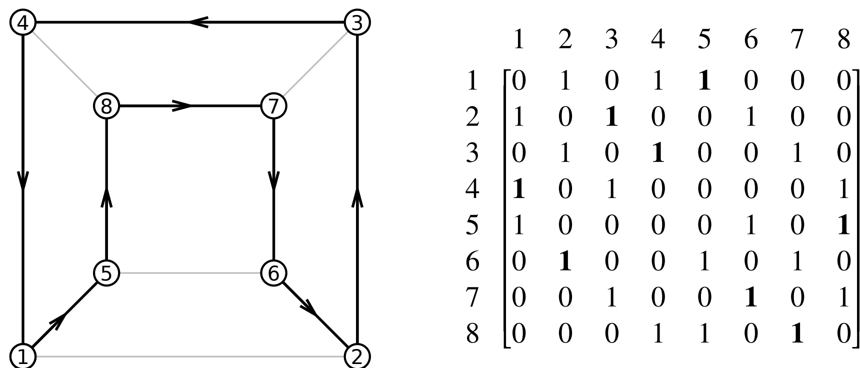

Figure 1: *The cube graph and its adjacency matrix, with a Hamiltonian path highlighted in both*

Consider the graph of a cube and the corresponding adjacency matrix (Figure 1). Hamiltonian cycles are easy to find for the cube, and one is highlighted in the figure. We can also highlight the directed edges of this cycle in the adjacency matrix.

No matter which Hamiltonian path we choose, the bold 1s always form a particular pattern: there is only one per each row and each column. This follows directly from the properties of Hamiltonian cycles, where each vertex has exactly one incoming and one outgoing edge. Visually, the pattern also makes a solution

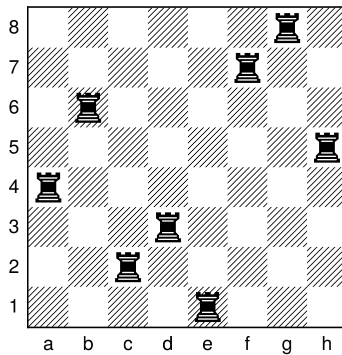

Figure 2: *A solution to the Eight Rooks problem*

to the simple chess-board problem of eight non-attacking rooks (Figure 2). This suggests that by using the solutions of the general problem of N rooks on an $N \times N$ chess board, we could find Hamiltonian cycles in a graph of N vertices.

2.2 Diagonal method

The chess-board problem has a simple solution: choose one of the two diagonals, and place all pieces along it. The solution remains valid under any permutation of columns (or rows). Therefore, by starting with the diagonal solution, we can generate a total of $N!$ valid solutions.

The diagonal solution is not immediately applicable to general graphs, as the choices are limited by their edges, or the 1s in their adjacency matrices. In particular, the main diagonal cannot represent a Hamiltonian cycle, as each vertex would then link only to itself. However, we can use the column permutations to get a diagonal viewpoint on any Hamiltonian graph.

Consider, for example, the complete tetrahedral graph. Let us permute the columns to make a main diagonal full of 1s. The matrix itself remains the same; this is merely a different view, as noted by the column numbers.

$$\begin{array}{c} \\ 1\\2\\3\\4 \end{array}\begin{array}{c} \begin{matrix}1&2&3&4\end{matrix} \\ \begin{bmatrix}0&1&1&1\\1&0&1&1\\1&1&0&1\\1&1&1&0\end{bmatrix}\end{array} \xrightarrow{\text{Permute columns}} \begin{array}{c} \\ 1\\2\\3\\4 \end{array}\begin{array}{c} \begin{matrix}3&1&4&2\end{matrix} \\ \begin{bmatrix}\mathbf{1}&0&1&1\\1&\mathbf{1}&1&0\\0&1&\mathbf{1}&1\\1&1&0&\mathbf{1}\end{bmatrix}\end{array} \xrightarrow[\text{(optional)}]{\text{Permute columns back}} \begin{array}{c} \\ 1\\2\\3\\4 \end{array}\begin{array}{c} \begin{matrix}1&2&3&4\end{matrix} \\ \begin{bmatrix}0&1&\mathbf{1}&1\\\mathbf{1}&0&1&1\\1&1&0&\mathbf{1}\\1&\mathbf{1}&1&0\end{bmatrix}\end{array} \qquad (1)$$

The bold 1s now contain the Hamiltonian cycle solution. For a clearer view, we may return to the original column ordering.

Now we can read the cycle by starting at any vertex, for example $1 \rightarrow 3$; $3 \rightarrow 4$; $4 \rightarrow 2$; $2 \rightarrow 1$. The final permutation is not essential, as long as we follow the column numbers.

In general, this method does not find a full Hamiltonian cycle. Starting with a graph G, the solutions generated thusly are subgraphs of G with the following properties:

1. Each vertex of G is included exactly once.
2. Each vertex is the starting point of exactly one directed edge, and the endpoint of another.

The latter ensures that we get closed cycles and no branches. But it does not guarantee we get a single large

cycle; the solution is generally a collection of disjoint cycles over G (some of these may be 2-cycles or double edges). This may be a useful result by itself, but in many cases we can also merge the cycles into a single Hamiltonian (section 3).

2.3 List formulation

For working with large graphs, adjacency lists are compact alternatives to adjacency matrices. For a graph $G = (V, E)$ of N vertices $V = \{v_i\}$, the adjacency list can be defined as an ordered collection (L_i) of N sets where

$$L_i = \{j | (v_i, v_j) \in E\}$$

i.e. for each vertex, we list its outgoing edges by their endpoint indices. Starting with the adjacency matrix, L_i lists the column numbers containing a 1 for each row i.

The above cycle-finding approach is readily adapted to adjacency lists. The solution can be formed via the sequence of column numbers $K = (k_i)$, a permutation of $(1, \dots, N)$ as seen in the matrix view (1) where $K = (3, 1, 4, 2)$. We thus need to find the numbers $k_i \in 1, \dots, N$ so that

$$k_i \in L_i \; \forall \, i \in \{1, \dots, N\}$$
$$\text{and } k_i \neq k_j \text{ when } i \neq j$$

This can be regarded as a fuzzy sorting problem: we need to order the lists L_i so that we can find the strictly increasing sequence from 1 to N when taking one element of each list. A simple trial-and-error approach can solve this in a reasonable time:

2.4 Simple cycle-finding algorithm

1. Initialize $K = (k_i)$ with $k_i = -1$ for all i to mark them as empty.
2. For each i such that $k_i = -1$, try to find an element of L_i that is not already in K.
   - If one is available, we have a new k_i.
   - If all L_i are in K, pick a random element $r \in L_i$. We know there is a j such that $r = k_j$; find that j. Then swap the values of k_i and k_j with each other. (I.e. set $k_i = r$, $k_j = -1$.)
3. Repeat step 2 until there are no empty/negative k_i left.

Now we can extract the cycles from K in the same way we did with matrix (1): the edges are (i, k_i) for all i. To find all the distinct closed cycles, we can then do the following:

1. For each i, construct the sequence $(a_j) : a_1 = i, a_j = k_{a_{j-1}}$ for $j > 1$.
2. When we find the first $m > 1$ such that $a_m = i$, we can stop, as we have found the closed cycle (a_j) of length $m - 1$.
3. Continue steps 1 and 2 with all i to find new cycles. If i is already in one of the cycles, discard it and continue with the next i.

2.5 Non-Hamiltonian graphs

In some cases, algorithm 2.4 gets stuck in an infinite loop. This suggests that the graph itself is non-Hamiltonian. In order to detect such situations robustly, the algorithm can be implemented with recursive functions to ensure that each permutation is covered exactly once. Unfortunately, the complexity of such recursion is exponential, similarly to full backtracking Hamiltonian solvers. Therefore, this is not a particularly practical method for testing the Hamiltonianness of a graph. Moreover, finding initial cycles does not guarantee that the graph is Hamiltonian.

3 Cycle fusion

The set of cycles found in section 2.4 can often be merged into a single Hamiltonian cycle. While there are existing algorithms for this problem, I have started with a simple approach that turns out effective in many practical cases.

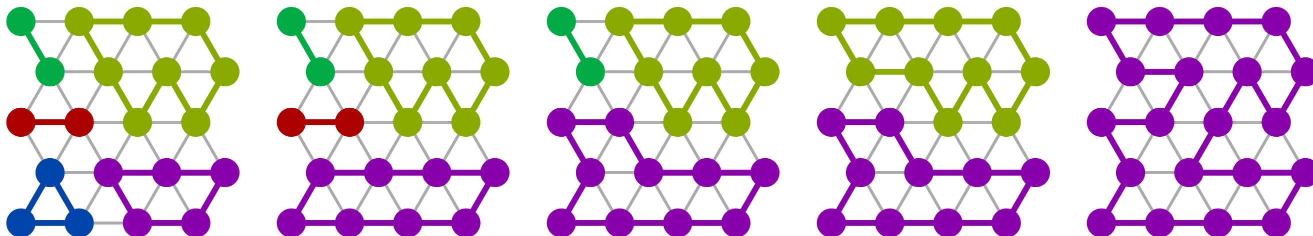

Figure 3: *Cycle stitching progress in a 2D hexagonal lattice*

3.1 Simple algorithm for joining two cycles

This algorithm is to be applied to a pair of cycles at a time, until there is only one left, the full Hamiltonian. A simple example of the process is shown in Figure 3.

1. A joint between cycles (a_i) and (b_i) is possible when all of the following conditions hold:
   (a) (a_i, a_j) is an edge in the cycle (a_i). Note that besides $j = i + 1$, we may also have $i = n, j = 1$ as the cycle of length n wraps around.
   (b) (b_l, b_m) is likewise an edge in the cycle (b_i).
   (c) $b_l \in L_{a_j}$ and $a_i \in L_{b_m}$.
2. To merge the two cycles, we remove the edges (a_i, a_j) and (b_l, b_m) and create new edges (a_j, b_l) and (b_m, a_i). The resulting new cycle is added to our set, and the two old ones are removed.
3. Note the relative directions between the two cycles: in a planar graph, the directions must be opposite to maintain the cyclic flow direction. (Think of U-turns rather than switching lanes – the latter would have the cars crossing paths.) The directions are important even with undirected graphs, as the cycles are written in a certain order, although we are then free to reverse them.

3.2 Problems and solutions

The above pair-joining approach assumes that the original graph contains the 4-cycles (a_i, a_j, b_l, b_m). There are Hamiltonian graphs with no 4-cycles, such as the truncated icosahedron and other Goldberg polyhedra, so this approach is not readily applicable to them. Moreover, a graph might have 4-cycles, but not in a suitable position for the two cycles from the first stage to meet.

As a straightforward solution, this approach can be extended for joining three or more cycles at a time, using recursive functions and backtracking. Care must be taken to limit the recursion depth, as the complexity grows exponentially in the number of cycles. In practice, a depth of three cycles is found appropriate for finding Hamiltonian cycles of Goldberg polyhedra, as expected for their 6-cycles.

Since my initial discovery of this method, I have learned of an alternative approach by Nenadov, Steger and Su that bears a striking similarity to my method with its two stages of initial cycle-finding and subsequent cycle-stitching[2]. They employ very different algorithms at each stage and show the overall complexity to be $O(N)$ with high probability.

Due to the same overall structure, the cycle-joining algorithm of Nenadov et al. can be readily applied to the initial cycles found with algorithm 2.4. Their approach is more complicated but also more thorough, so it serves as a useful backup in the cases where my simple method fails to find a solution.

4 Performance notes

4.1 Cycle finding

In my experiments with polyhedral graphs and simple lattices with $|L_i| \leq 6$ and N up to 4096, algorithm 2.4 takes about $N^{1.2}$ steps to complete.

4.2 Cycle joining

The worst-case complexity of each pair-joining stage is essentially $O\left(N^2\right)$, as we need to find a pair of vertices (a_i, b_l) that fulfil the necessary conditions. However, in my experiments I have found that the full merge usually finishes in less than N steps; in each set of cycles there are multiple opportunities for a joint operation. On the other hand, each step is slower than those of cycle finding, as we need to look up two neighbour lists.

There are also scenarios where one pair-joint of cycles is succesful, but we fail to find a match at a subsequent stage. Insofar as a Hamiltonian solution is possible to find by pairwise joints (see section 3.2), such cases can usually be solved by backtracking, which further increases the computational complexity.

Likewise, joints of 3 or more cycles at a time make this stage computationally heavier. But this is offset to some extent by the cycle count being reduced by 2 or more at a time.

4.3 Practical note for fast results

In my experience, finding Hamiltonian cycles in a reasonable time is a matter of luck: given the random selection aspect at different stages, the time to a final solution varies enormously. In the present approach, the cycle-joining stage appears to be the bottleneck with a bathtub-like distribution of solution times. My previous experiments with various backtracking solvers have followed a similar general pattern.

Therefore, my Hamiltonian production facility has adopted the shotgun approach: launching multiple parallel solvers with different random seeds, picking the early winners, and terminating the remaining processes.

5 Summary and future suggestions

The principal innovation presented here is the diagonal approach to cycle-finding, outlined in section 2. The full implementation requires a kind of fuzzy sorting stage, for which I have described a simple but reasonably fast algorithm. The result of this stage is a set of disjoint cycles, which requires another algorithm to be combined into a full Hamiltonian cycle. For this I have also outlined a simple and somewhat incomplete approach.

In both of the two stages, there remains room for improvement in the form of faster and more thorough algorithms. Nenadov et al. [2] provide interesting alternatives to both stages, with their stage 2 being readily adaptable to my approach.

References

[1] Dénes König. *Theory of Finite and Infinite Graphs*. Birkhäuser, 1990.

[2] Rajko Nenadov, Angelika Steger, and Pascal Su. An o(n) time algorithm for finding hamilton cycles with high probability. In *12th Innovations in Theoretical Computer Science Conference (ITCS 2021)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.