Fırat Üniversitesi Müh. Bil. Dergisi
38(2), 1-16, 2026

Araştırma Makalesi
https://doi.org/10.35234/fumbd.1926890

A Novel Edge-Efficiency-Based Algorithm for Hamiltonian Cycle and Path Detection in Graphs

Fatih OKUMUŞ[1], Ahmet KARADOĞAN[2*]
[1,2] Department of Software Engineering, Faculty of Engineering, İnönü University, Malatya, Türkiye
[1] fatih.okumus@inonu.edu.tr, [2] ahmet.karadogan@inonu.edu.tr

(Geliş/Received: 09/04/2026; Kabul/Accepted: 16/05/2026)

Abstract: Searching for the existence of a Hamiltonian cycle and path connecting all nodes in a graph is an NP-complete problem. This article proposes the E^2 Algorithm for constructing the Hamiltonian cycle in an arbitrary graph without edges' weights. The Kmax and Kmin that are particular spanning trees are generated first to obtain the fundamental cuts. Then, each edge's total number in the fundamental cuts is obtained to state edge efficiency. Next, all nodes are navigated with a method that determines priority, starting with the highest degree node at the most efficient edge. Thus, when the greedy traversal succeeds, a Hamiltonian cycle or path is constructed between all nodes. Since E^2 is a deterministic greedy algorithm without backtracking, it does not guarantee finding a Hamiltonian cycle in every Hamiltonian graph; however, it always terminates in polynomial time. These methods are used for the first time to obtain the Hamiltonian cycle in this study. In addition, we present an object-oriented construction to avoid getting exponential algorithm complexity. Finally, to prove the correctness of the method, we show whether some general graphs are Hamiltonian using the proposed method.

Keywords: Graphs, hamiltonian cycle, hamiltonian path, np-complete problems.

Çizgelerde Hamilton Çevrimi ve Yolunun Tespiti İçin Kenar Etkinliğine Dayalı Yeni Bir Algoritma

Özet: Bir çizgede tüm düğümleri birbirine bağlayan Hamilton çevrimi ve Hamilton yolunun varlığını araştırmak NP-tam bir problemdir. Bu makalede, kenar ağırlıkları olmayan keyfi bir çizgede Hamilton çevrimini oluşturmak için E^2 Algoritması önerilmektedir. Öncelikle, temel kesitleri elde etmek amacıyla özel örten ağaçlar olan Kmax ve Kmin oluşturulmaktadır. Daha sonra, kenar etkinliğini belirlemek için her bir kenarın toplam temel kesit sayısı elde edilmektedir. Ardından, en etkin kenar üzerindeki en yüksek dereceli düğümden başlayarak önceliği belirleyen bir yöntemle tüm düğümler dolaşılmaktadır. Böylece, açgözlü geçiş başarılı olduğunda, tüm düğümler arasında bir Hamilton döngüsü veya yol oluşturulur. E^2 geri izleme içermeyen deterministik bir açgözlü algoritma olduğundan, her Hamilton grafiğinde bir Hamilton döngüsü bulmayı garanti etmez; ancak her zaman polinom zamanında sonlanır. Bu çalışmada, Hamilton çevrimini elde etmek için bu yöntemler ilk kez kullanılmaktadır. Ayrıca, üstel algoritma karmaşıklığının ortaya çıkmasını önlemek amacıyla nesne yönelimli bir yapı sunulmaktadır. Son olarak, yöntemin doğruluğunu göstermek için bazı genel çizgelerin önerilen yöntem kullanılarak Hamiltonyen olup olmadığı gösterilmektedir.

Anahtar kelimeler: Çizgeler, hamilton çevrimi, hamilton yolu, np-tam problemler.

1. Introduction

The Hamilton cycle (a cycle that traverses each node exactly once) connects all the nodes in a graph. However, searching for the existence of this cycle in any graph is an NP-complete problem [1, 2]. The NP-complete problem requires exponential complexity. Due to this, many studies are on constructing the Hamiltonian path/cycle for the given graph. All the studies on this subject are based on approximation approaches.

Hamiltonian connected graphs were first introduced in 1963 by Ore [3]. Since then, many different approaches have been proposed for Hamiltonian path and cycle problems. A light-based solution that uses a light filter to find valid Hamiltonian paths of a graph was proposed in [4]. A linear-time algorithm for finding a Hamiltonian path in rectangular grid graphs was presented in [5]. Dybizbański and Szepietowski [6] consider Hamiltonian cycles in hypercubes with pairwise disjoint faulty edges.

DeBiasio and Spanier studied computing the Hamiltonian cycles in balanced k-partite graphs [7]. DeBiasio et al. studied the powers of the Hamiltonian cycle in multipartite graphs [8]. Agueda et al. studied the proper Hamiltonian cycle in edge-colored multigraphs with parallel and loop edges [9]. If the line graph of any graph can be decomposed into two Hamiltonian cycles, what is the type of this graph? The answer to this question was investigated in [10]. The Hamiltonian cycles and paths in faulty twisted hypercubes were investigated in [11]. The

* Corresponding author: ahmet.karadogan@inonu.edu.tr ORCID Information: [1] 0000-0003-3046-9558, [2] 0000-0003-0954-5958

Hamiltonian cycles of covering graphs of trees were studied in [12]. The number of Hamiltonian cycles in planar triangulations was computed in [13]. The edge-disjoint cycles in balanced hypercubes [14] and the cycles of edge-faulty hypercubes [15] were another studies on Hamiltonian cycles. More recently, Ahmed [16] proposed a spanning-tree-based routing algorithm for Hamiltonian cycles in planar graphs using quasi-spanning trees, and Biton et al. [17] presented a distributed CONGEST algorithm for finding Hamiltonian paths in Dirac graphs. Gonzalez-Martinez et al. [18] addressed the multi-objective Hamiltonian cycle problem using a branch-and-fix heuristic outperforming genetic algorithms on large graphs. These studies confirm that spanning-tree structures and structural graph properties continue to be productive sources of algorithmic insight for Hamiltonian problems.

Motivation: Most of the studies mentioned above are about hypercubes which are regular graphs. The detailed studies are not valid for arbitrary graphs. Due to this case, there should be new studies to fill this gap. Most of the studies mentioned above also include analytical content; applying the computational method to these studies requires new efforts. This study is a part of these new studies. The novel approach to constructing the Hamiltonian cycle/path includes Kmax and Kmin proposed by Karci [19-23].

This paper aims to propose a novel deterministic greedy method to search for a Hamiltonian cycle in a given connected graph. Since Hamiltonicity is NP-complete, no polynomial-time algorithm can be guaranteed to solve all instances unless P = NP. Accordingly, E^2 is not an exact decision procedure; it is a deterministic, polynomial-time greedy algorithm that follows a fixed priority rule based on edge efficiency and node degree, without backtracking. When the greedy traversal successfully visits all nodes and returns to the starting node, a Hamiltonian cycle is found. Otherwise, the algorithm reports a Hamiltonian path or signals the absence of a cycle for that particular traversal order. This approach is based on two particular spanning trees of the graph, namely Kmax and Kmin. These structures were introduced in Karcı's studies in the context of graph-theoretic problems such as dominating and independent sets [19-23]. In the present study, they are adapted to the Hamiltonian cycle/path problem.

Constructing Kmax tree: $G = (V, E)$ is a given graph where $V = \{v_1, v_2, \dots, v_n\}$. The node with the maximum degree is selected as the root node. If there is more than one node with maximum degrees, then one of them can be arbitrarily selected as a root. Assume that the selected node is v_i, then $N(v_i)$ are added to the tree as the root's children. The children of v_i are expanded from maximum to minimum remaining degrees. The obtained tree is called as Kmax Tree (Karcı Maximum Spanning Tree). The details of Kmax can be found in [19-23].

Constructing Kmin tree: $G = (V, E)$ is a given graph where $V = \{v_1, v_2, \dots, v_n\}$. The node with the minimum degree is selected as the root node. If there is more than one node with minimum degrees, then one of them can be selected as a root arbitrarily. Assume that the selected node is v_i, then $N(v_i)$ are added to the tree as the root's children. The children of v_i are expanded from minimum to maximum remaining degrees. The obtained tree is called as Kmin Tree (Karcı Minimum Spanning Tree). The details of Kmin can be found in [19-23].

Both trees are used to construct the fundamental cut-sets. The number of edge occurrences in each fundamental cut-sets represents the effectiveness of the corresponding edge. The effectiveness of all edges is used to construct the Hamiltonian path for the given graph. The fundamental cut-sets are used to obtain the Hamiltonian cycle for the given connected graphs for the first time in this study.

In this paper, we proposed the E^2 Algorithm built on edge efficiency using the fundamental cut-sets obtained from the spanning trees mentioned above. The proposed method's algorithms are stated in the second part of the paper, and their mathematical proofs are shown. Pseudocodes of the algorithms are also given in this section. The results obtained by applying the proposed method to some general graphs are given in the third part of the paper. The fourth section states general evaluations of the paper and the basic foundations of future studies.

2. Proposed Method

We propose a 4-step algorithm based on edge efficiency to find the path forming the Hamilton cycle between the graph nodes. In the first step, the Kmax tree is constructed, prioritizing the nodes with the maximum degree of nodes. In the second step, the Kmin tree is constructed, prioritizing the nodes with the minimum degree. In the third step, cut-sets are generated for the edges separately from the Kmax and Kmin trees. For each edge, the total number of its occurrences in the generated cut-sets is taken as its edge efficiency number. In the last step, starting from the highest degree node of the most efficient edge, the Hamilton path and cycle are found if it exists. The proposed method is called as E^2 (edge efficiency) Algorithm, and its flow chart is given in Figure 1.

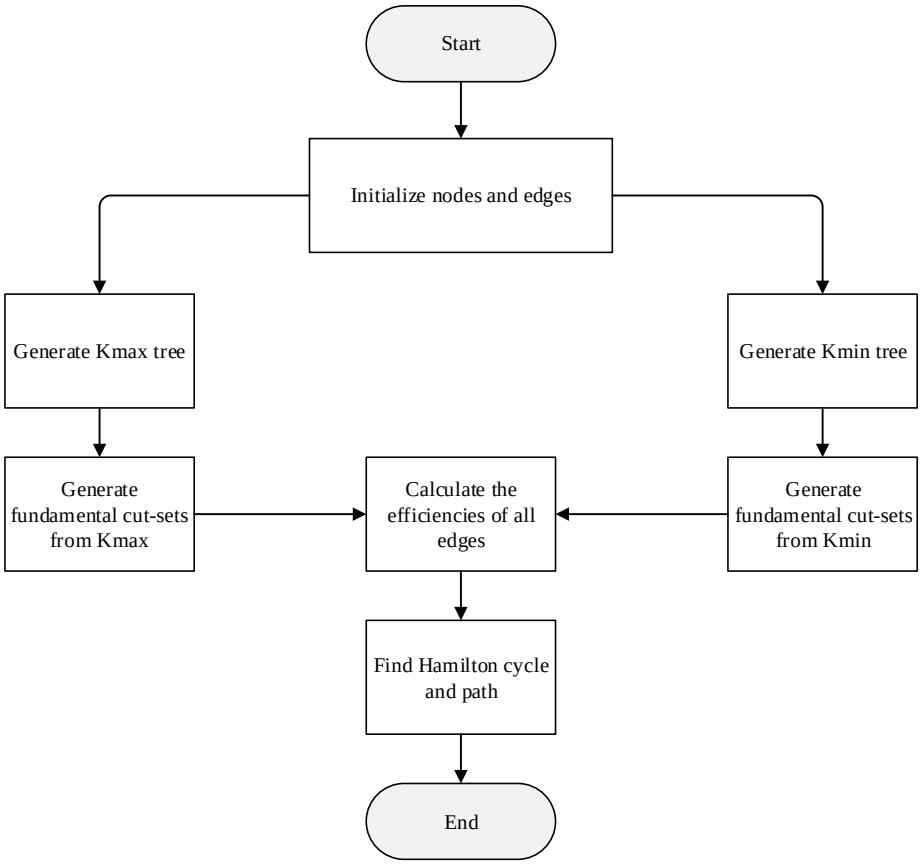

Figure 1. Flow chart of E^2 Algorithm.

Definition 1: Assume that $G = (V, E)$ is a graph and $Kmax = (V, E_1)$ and $Kmin = (V, E_2)$. The fundamental cut-sets obtained from $Kmax$ and $Kmin$, denoted by $Cmax$ and $Cmin$, respectively, are defined in Eq. (1):

$$Cmax = \{ {}_xC_1,\ {}_xC_2, \ldots,\ {}_xC_{|V|}\}\ and\ Cmin = \{{}_mC_1, {}_mC_2, \ldots, {}_mC_{|V|}\} \quad (1)$$

The Edge Efficiency number of an edge (e_i) is defined in Eq. (2):

$$\varrho(e_i) = \sum_{j=1}^{|V|}[e_i \in {}_xC_j] + \sum_{j=1}^{|V|}[e_i \in {}_mC_j] \quad (2)$$

where $[P]$ denotes the Iverson bracket: $[P] = 1$ if predicate P is true, and $[P] = 0$ otherwise. Accordingly, each sum counts the number of fundamental cut-sets (from Cmax and Cmin respectively) in which edge e_i participates. The total edge efficiency $\varrho(e_i)$ thus reflects the structural importance of e_i across both spanning trees.

Definition 1 gives how the $\varrho(e_i)$ is computed. After computation of $\varrho(e_i)$ for all edges, the edge with maximum $\varrho(e_i)$ value is selected as the first edge (assume that this edge is e_1) for the Hamiltonian cycle. Assume that the nodes on which e_1 is incident are v_1 and v_2. If $d(v_1) > d(v_2)$, then the edge e_1 will have direction from v_1 to v_2. $C = \{v_1, v_2\}$ is the cycle node set and E_C is the cycle edge set, $E_C = \{e_1\}$. After this step, $N(v_2)$ is the set of neighbor nodes set, and assume that e_2 is an edge from v_2 to the node minimum degree in $N(v_2)$ where this node v_3. $E_C = E_C \cup \{e_2\}$, and $C = C \cup \{v_3\}$. In the case of the number of minimum degree nodes in $N(v_2)$ being more than one, the edge with minimum $\varrho(e_i)$ value is selected as the cycle (path) edge. If more than one node in $N(v_2)$ has the same minimum degree, the next edge is chosen as the one with the minimum $\varrho(e_i)$ value among the corresponding candidate edges. If a tie still remains, the edge whose end node has the fewest neighbors is selected. This process carries on in this manner until C set contains all nodes in the given graph or there is no edge/node to be selected for E_C / C. In the last step, a path is followed starting from the highest degree node on the maximum cut edge and proceeding towards the minimum degree neighbor. This path will be a cycle reaching all nodes in the graph.

2.1. E^2 Algorithm

The first stage of the E^2 Algorithm includes the construction of the Kmax tree. The Kmax tree is based on the idea of constructing the tree by selecting the nodes with the maximum degree. The purpose of constructing this tree is to bring to the fore the highest degree nodes. The algorithm creates a copy of the node list (in order to not corrupt the original node list) and appoints the highest degree node as the root. The root node is placed in the first row of the matrix by constructing an $(n\ x\ n)$ matrix representing the Kmax tree. The root node is determined as the parent node, and all its neighbors are placed on the next row, and then the highest degree node within these nodes is determined as the new parent. This process is continued so that all nodes are added to the tree. All nodes added to the tree are placed in a queue to select the highest degree node. In this queue, nodes are placed in the front according to their degree. If there are nodes of the same degree, the one with the lower level in the tree takes priority. When a node is added to the tree, all its neighbors' degrees are decreased by one. This process plays an essential role in selecting the highest-degree node. The pseudo-code for constructing the Kmax tree for a graph is given in Algorithm 1.

Algorithm 1. Construction of Kmax Tree

```
Input: G = (V, E): undirected graph; nodeList: copy of V with degree information
Output: Tmax: Kmax spanning tree represented as an n × n level matrix
Data: Q: priority queue (max-degree, min-level order); T: two-dimensional array
1     T ← ∅; Q ← ∅
2     r ← node with maximum degree in nodeList ∴ designated as tree root
3     r.level ← 0
4     Enqueue(Q, r)
5     for all u ∈ N(r) do deg(u) ← deg(u) − 1 ∴ update neighbor degrees
6     nodeList ← nodeList \ {r}
7     parent ← r
8     while nodeList ≠ ∅ do
9         ℓ ← parent.level + 1
10        Dequeue(Q, parent)
11        for all v ∈ N(parent) do
12            if v ∉ T then
13                for all u ∈ N(v) do deg(u) ← deg(u) − 1
14                T[ℓ] ← T[ℓ] ∪ {v}
15                Enqueue(Q, v)
16                nodeList ← nodeList \ {v}
17            end if
18        end for
19        parent ← Dequeue-Max(Q) ∴ highest-degree node; ties broken by min level
20    end while
21    return T
```

Theorem 1: *Algorithm 1 (construction of Kmax) has polynomial complexity.*

Proof: Assume that the given graph is $G = (V, E)$. The proof can be handled in more than one step.

- Assume that G is a star or complete graph. The node with the degree of |V|-1 will be the root node, and the remaining nodes will be children of the root. The complexity of Algorithm 1 is O(|V|) in this case since the first step is to find a node of maximum degree. This process requires O(|V|), and after this step, |V|-1 children will be added to the root node. Figure 2 illustrates this case.

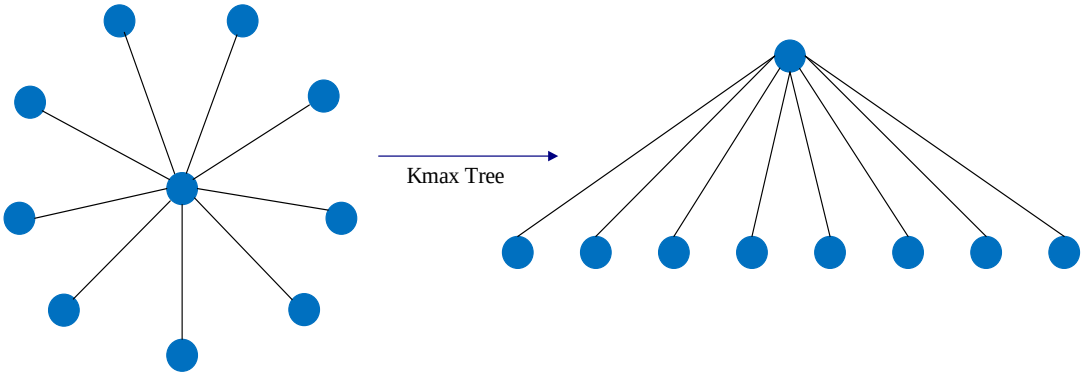

Figure 2. Star graph and its Kmax tree.

- Assume that G=(V,E) is not a star graph or complete (fully connected) graph. In order to find the node with the maximum degree, this process requires (|V|) steps, and the node with the highest degree is used for the root node (Figure 3: root node). There will be Δ(G) children of the root where Δ(G) is the maximum node degree in G. The number of remaining nodes not added to Kmax is |V|-Δ(G)-1. The nodes in the second level of Kmax that is seen in Figure 3 have the maximum degree as |V|-Δ(G)-1. The children of any node with this remaining degree will be added to Kmax, and the construction will be completed in this way. Thus, the complexity of Algorithm 1 for this case can be expressed as given in Eq. (3):

$$T(|V|,|E|) = |V| + \Delta(G) + |V| - \Delta(G) - 1 = O(|V|) \quad (3)$$

where $p = \Delta(G)$, and $q = |V| - p - 1$.

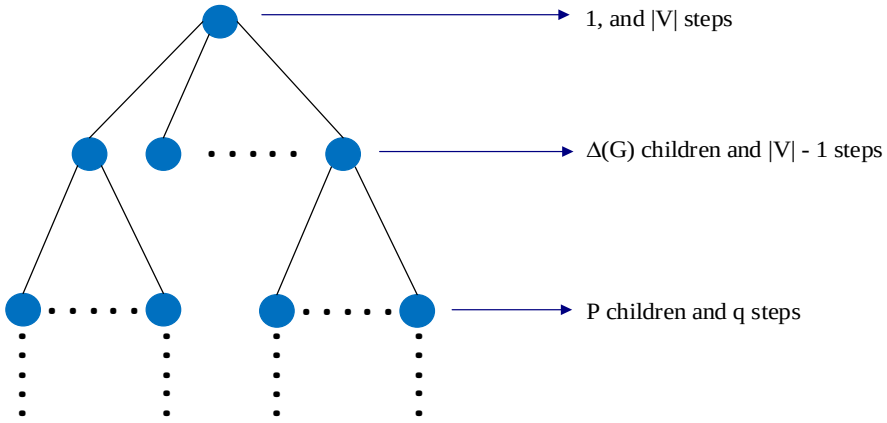

Figure 3. Kmax tree of any graph.

If the maximum degree in the second level of Kmax is not equal to q, at least two nodes will have children. Accordingly, the computational cost of Algorithm 1 for this case is given in Eq. (4):

$$T(|V|,|E|) = |V| + \Delta(G) + \Delta(G) - 1 + |V| - \Delta(G) - 1 = O(|V| + \Delta(G)). \quad (4)$$

where $p = 2\Delta(G)$, and $q = |V| - p - 1$.

General case. In the general case, Algorithm 1 maintains a priority queue Q ordered by node degree (descending), with ties broken by tree level (ascending). Each node $v \in V$ is enqueued exactly once and dequeued exactly once, contributing $O(log\ |V|)$ per operation due to the priority queue. The total cost of all enqueues and dequeue operations is therefore $O(|V|\ log\ |V|)$.

When a node v is added to the tree, the degrees of all its neighbors in nodeList are decremented. The total number of such decrements across all nodes equals the sum of degrees, which by the handshaking lemma satisfies as shown in Eq. (5):

$$\sum_{v \in V} deg(v) = 2|E| \tag{5}$$

Therefore, the total cost of all degree update operations is $O(|E|)$. Combining both terms, the overall time complexity of Algorithm 1 is given in Eq. (6):

$$T(|V|, |E|) = O(|V| \log |V| + |E|) \tag{6}$$

Since $|E| \leq |V|^2$ for simple graphs, this is bounded by $O(|V|^2)$ in the worst case, confirming polynomial complexity. ■

The next stage of the algorithm is to construct the Kmin tree. The basic logic of the Kmin tree is that it is built starting from the lowest degree node. Essentially, all the steps are pretty similar to the construction of the Kmax tree. However, selecting the lowest degree node is applied instead of the highest degree node when determining the parent node. The purpose of constructing Kmax tree is to bring to the fore low-degree nodes. The pseudo-code for constructing the Kmin tree for a graph is given in Algorithm 2.

Algorithm 2. Construction of Kmin tree

```
Input: G = (V, E): undirected graph; nodeList: copy of V with degree information
Output: T_min: Kmin spanning tree represented as an n × n level matrix
Data: Q: priority queue (min-degree, min-level order); T: two-dimensional array
    T ← ∅; Q ← ∅
    r ← node with minimum degree in nodeList ∴ designated as tree root
    r.level ← 0
    Enqueue(Q, r)
    for all u ∈ N(r) do deg(u) ← deg(u) − 1
    nodeList ← nodeList \ {r}
    parent ← r
    while nodeList ≠ ∅ do
        ℓ ← parent.level + 1
        Dequeue(Q, parent)
        for all v ∈ N(parent) do
            if v ∉ T then
                for all u ∈ N(v) do deg(u) ← deg(u) − 1
                T[ℓ] ← T[ℓ] ∪ {v}
                Enqueue(Q, v)
                nodeList ← nodeList \ {v}
            end if
        end for
        parent ← Dequeue-Min(Q) ∴ lowest-degree node; ties broken by min level
    end while
    return T
```

Corollary 1*: Algorithm 2 (construction of Kmin) has polynomial complexity.*

Proof: Algorithm 2 is structurally identical to Algorithm 1, with the sole difference that the priority queue Q is ordered by node degree in ascending order (min-degree first) rather than descending. All asymptotic arguments from Theorem 1 apply without modification: each node is enqueued and dequeued exactly once in O(log |V|) per

operation, and degree updates across all nodes cost $O(|E|)$ in total by the handshaking lemma. Therefore, the time complexity of Algorithm 2 is also $O(|V| \log |V| + |E|)$, bounded by $O(|V|^2)$ in the worst case. ■

After the Kmin and Kmax trees are constructed, the construction of fundamental cut-sets takes place and is applied to remove all nodes from the trees. The edge connecting the node to the parent node is cut in the cut-set generation. In addition, the chord connections of all the node's children in the hierarchical order that is not in the tree are also cut. For each cutting process, the $\varrho(..)$ value of the relevant edge is increased by one. The pseudocode for construction cut-sets is given in Algorithm 3.

Algorithm 3. Construction of fundamental cut-set

```
Input: T: spanning tree (Kmax or Kmin) as level matrix; G = (V, E): original graph
Output: ϱ: edge efficiency array where ϱ(eᵢ) is updated for each eᵢ ∈ E
Data: children: set of subtree nodes for a given root; i: level index (reset per node)
1     for all rows r ∈ T do
2         for all node v ∈ T[r] do
3             children ← {v}
4             i ← v.level + 1 ∴ reset i for each node (was set once outside the loop)
5             while i < |T| do
6                 for all child c ∈ T[i] do
7                     if parent(c) ∈ children then
8                         children ← children ∪ {c}
9                     end if
10                end for
11                i ← i + 1
12            end while
13            for all u ∈ children do
14                ϱ(edge(parent(u), u)) ← ϱ(edge(parent(u), u)) + 1
15                for all e = (u, w) ∈ E do
16                    if e is a chord and w ∉ children then
17                        ϱ(e) ← ϱ(e) + 1
18                    end if
19                end for
20            end for
21        end for
22    end for
```

Theorem 2: *Algorithm 3 (construction of fundamental cut-sets) has polynomial complexity.*

Proof: The proof can be handled in more than one step.

- Assume that $G = (V, E)$ is a complete graph. In this case, the neighbors' list has $|V| - 1$ members (Figure 4). One of them is selected as a root node, and the remaining nodes will be the tree's leaves. So, there is $|V| - 1$ fundamental cut-sets, and each cut-set consists of $|V| - 1$ edges. In this case, the complexity of Algorithm 3 is given in Eq. (7):

$$T(|V|, |E|) = |V|(|V| - 1) = O(|V|^2) \tag{7}$$

General case. For a general graph $G = (V, E)$, Algorithm 3 iterates over every node $v \in V$. For each node, it constructs the set children by traversing the levels of the spanning tree below v. This traversal visits at most $|V|$ nodes in total per root node, giving an inner loop cost of $O(|V|)$ per node.
After constructing children, the algorithm increments the efficiency counter ϱ for each edge incident on the nodes in children. The number of such edges is at most $|E|$ in the worst case (complete graph). However, summed over all root nodes v, each edge $(u, w) \in E$ is counted at most twice per spanning tree (once when u is the root, once when w is the root). Since Algorithm 3 is called once for Tmax and once for Tmin, the total number of ϱ increments is bounded as shown in Eq. (8):

$$2 \times 2|E| = 4|E| = O(|E|) \tag{8}$$

Combining the subtree construction cost $O(|V|)$ per node over all $|V|$ nodes, and the edge counting cost $O(|E|)$ is given in Eq. (9):

$$T(|V|,|E|) = O(|V|^2 + |E|) \quad (9)$$

Since $|E| \leq |V|^2$ for simple graphs, this simplifies to $O(|V|^2)$, confirming polynomial complexity. ■

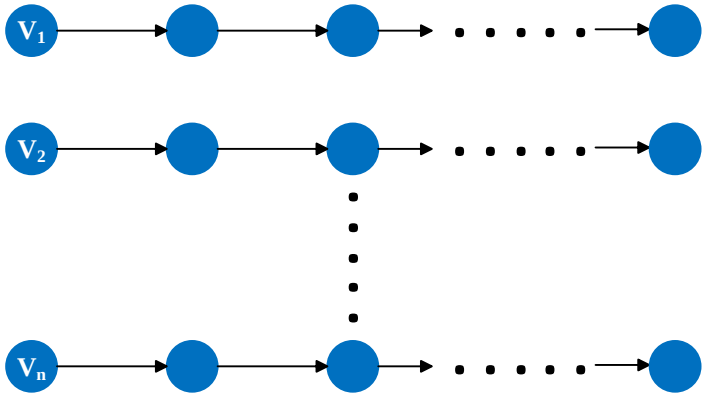

Figure 4. The nodes and their neighbors list.

In Algorithm 3, retrieving all descendants of a node in the spanning tree is a prerequisite for constructing each fundamental cut-set. A recursive traversal would achieve this naturally, but its cost grows with the depth of the tree and re-evaluates subtrees multiple times, leading to redundant computation.

To avoid this, we employ an iterative object-oriented node structure in which each node object stores the following fields explicitly: its level in the tree, a reference to its parent node, and a list of its children. The spanning tree is simultaneously represented as a two-dimensional array T, where $T[\ell]$ holds all nodes at level ℓ. This dual representation, object references for parent/child navigation, and a level array for top-down traversal, allows the subtree rooted at any node v to be collected iteratively by scanning levels $\ell = v.level + 1, v.level + 2, \ldots, |T| - 1$ and retaining only nodes whose parent is already in the collected set (Algorithm 3, lines 5–12).

Since each node is visited at most once during this scan, the cost of subtree collection for a single root node is $O(|V|)$. Summed over all $|V|$ root nodes, the total cost of cut-set construction remains $O(|V|^2)$, consistent with the bound established in Theorem 2. No recursive function calls are made, ensuring that the call stack depth remains $O(1)$ regardless of tree structure.

The final stage of the proposed method is to find the path through which a Hamiltonian cycle is constructed by traversing all the nodes. The first node of the path to be constructed is the highest-degree node of the edge with the highest edge efficiency number. In other words, the edge with the maximum ϱ(..) value. After selecting the first node, the neighboring node connected with the relevant edge is determined as the second node. Next, among the neighboring nodes, the lowest degree one is selected. This selection continues until there are no unreached nodes. If there is more than one neighbor with the lowest degree, the connected edges are checked, and the neighbor node with the lowest edge efficiency number is selected. If there is an equality in the edge efficiency number, either one is selected and continued. Every node added to the path is removed from the list, so a node is visited at most once. When the last node is added to the tree, if there is an edge between it and the first node, the Hamiltonian cycle will be formed. A direction occurs because nodes are added to the cycle in a specific order. This direction also represents the Hamilton path. The searching method of the Hamilton cycle and path is shown in Algorithm 4.

Algorithm 4. Finding the cycle and path traversing all nodes

```
Input: G = (V, E): undirected graph; ϱ: edge efficiency values computed by Algorithm 3
Output: P: ordered node sequence
Data: visited: set of visited nodes; currentNode: active traversal node
    P ← ∅; visited ← ∅
    e* ← argmax_{e ∈ E} ϱ(e) ∴ edge with highest efficiency
    Let e* = (v1, v2) where deg(v1) ≥ deg(v2)
    P ← ⟨v1, v2⟩; visited ← {v1, v2}
    currentNode ← v2
    while ∃ u ∈ N(currentNode) such that u ∉ visited do
        C ← { u ∈ N(currentNode) : u ∉ visited }
        next ← argmin_{u ∈ C} deg(u) ∴ lowest-degree unvisited neighbor
        if |argmin_{u ∈ C} deg(u)| > 1 then ∴ degree tie exists
            next ← argmin_{u ∈ C, deg(u)=min} ϱ(currentNode, u) ∴ break by min edge efficiency
        end if
        if tie still remains then
            next ← argmin_{u ∈ C} |N(u) \ visited| ∴ break by fewest remaining neighbors
        end if
        for all u ∈ N(next) do deg(u) ← deg(u) − 1
        P ← P ‖ ⟨next⟩; visited ← visited ∪ {next}
        currentNode ← next
    end while
    if (P[1], P[|P|]) ∈ E then
        return P ∴ Hamiltonian cycle
    Else
        return P ∴ Hamiltonian path
    end if
```

Theorem 3: *Algorithm 4 (construction of path) has polynomial complexity*.

Proof: The edge with maximum ϱ(..) value is selected for the Hamiltonian path in the given graph, and this step requires the complexity as O(|E|) due to the linear search. Figure 5 illustrates this case, and the endpoints of the selected edge are v_i and v_j. If $d(v_i) > d(v_j)$, then the direction of the selected edge is from v_i to v_j. The next step for selecting the second edge requires complexity as $O(|N(v_j)|) = O(d(v_j))$, since the edges incident on to v_j are investigated as the second edge option. The same process can be performed for the third and fourth edges, and they require the complexities as $O(|N(v_k)|) = O(d(v_k)), O(|N(v_m)|) = O(d(v_m))$, respectively. This process goes on until all nodes are added to cycle set C.

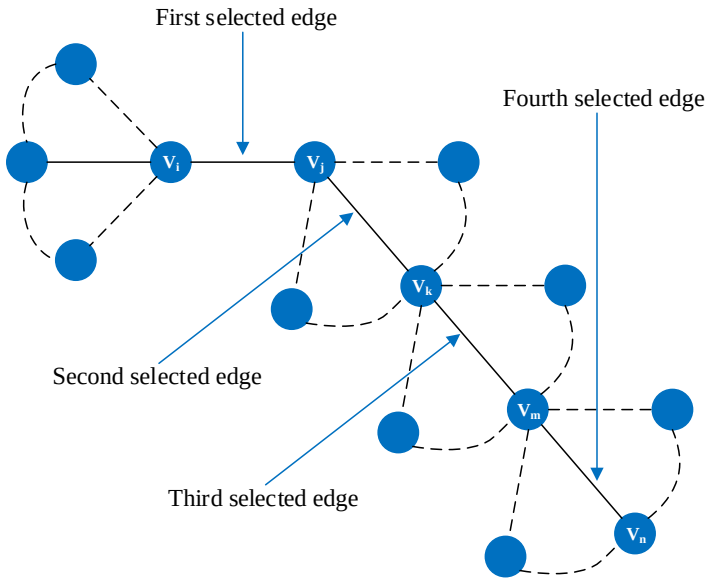

Figure 5. Construction of Hamiltonian path.

As the fundamental counting principle: if the first task can be done in n_1 ways and the second task can be done in n_2 ways, and if these tasks cannot be done simultaneously, then there are n_1+n_2 ways to do either task.

General case. Algorithm 4 consists of the following steps whose costs are analyzed separately.

Step 1 — Selecting the starting edge. Finding the edge e* with maximum ϱ value requires a linear scan over all edges: $O(|E|)$.
Step 2 — Greedy traversal. At each iteration of the while loop, the algorithm examines the unvisited neighbors of the current node. Each node is visited exactly once and removed from consideration; the total work across all iterations equals the sum of degrees of all visited nodes. By the handshaking lemma, this sum is given in Eq. (10):

$$\sum_{v \in V} deg(v) = 2|E| \tag{10}$$

Therefore, the traversal cost is $O(|E|)$.

Step 3 — Tie-breaking. When multiple neighbors share the minimum degree, the algorithm compares their edge efficiency values ϱ. This comparison is performed within the neighbor set of the current node, adding at most $O(deg(v))$ per step, which is already accounted for in Step 2.
Step 4 — Cycle check. Checking whether an edge exists between the first and last nodes of P requires $O(1)$ with an adjacency structure, or $O(deg(v))$ with an adjacency list.

Summing the costs of all steps, the total time complexity of Algorithm 4 is obtained as shown in Eq. (11):

$$T(|V|,|E|) = O(|E|) + O(|E|) + O(1) = O(|E|) \tag{11}$$

Since $|E| \leq |V|^2$, this is bounded by $O(|V|^2)$ in the worst case, confirming polynomial complexity. ■

Overall complexity of E². The four algorithms are executed sequentially. Their individual complexities are $O(|V|\ log\ |V|\ +\ |E|)$ for Algorithms 1 and 2, $O(|V|^2\ +\ |E|)$ for Algorithm 3 (called twice), and $O(|E|)$ for Algorithm 4. The dominant term is $O(|V|^2 + |E|)$, which simplifies to $O(|V|^2)$ for simple graphs. Therefore, the overall time complexity of the E² Algorithm is $O(|V|^2)$, confirming that it runs in polynomial time.

3. Experimental Results

The proposed algorithm has been applied to some common and small graphs to measure its reliability. In Table 1, graphs on which the algorithm was applied and their results were given in detail. Hypercube, Wheel Graph, and Petersen Graph examples are examined and shown in Figures 6, 7, and 8 to show the algorithm's evolution. The Kmax and Kmin tree formation and the Hamilton cycle/path are shown in the Figures, respectively.

Hypercubes have been the topology of choice for parallel machines due to their regular structure and diverse edge connections [24]. These features enable the development of efficient algorithms for inter-node communication. In graph theory, the hypercube is expressed as Q_n; it has 2^n nodes and $n2^{n-1}$ edges. Figure 6 shows the working steps of the E² Algorithm for hypercube Q_4, which has 16 nodes and 32 edges. The hypercube is a Hamilton graph, and the E² Algorithm found the Hamilton cycle and path, as shown in Figure 6. The algorithm determined that the most effective edge was between the15th and 16th nodes. Also, the 15th node became the first node of the Hamilton path.

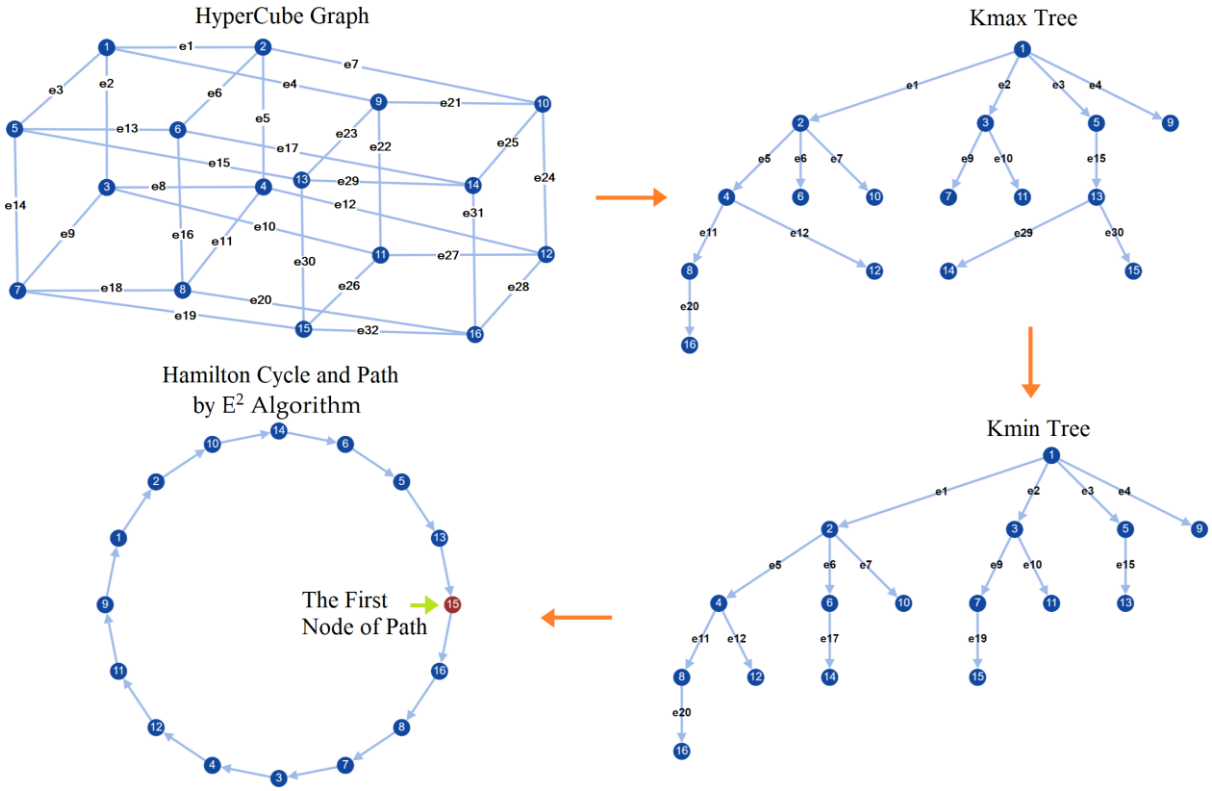

Figure 6. The results of the E² Algorithm were applied to the Hypercube graph.

An example of the Wheel Graph, another Hamiltonian graph, is shown in Figure 7 to illustrate the E² Algorithm's performance. In Wheel Graphs, all nodes are combined under a single top node. The Kmax tree is also illustrated to support this idea. Figure 7 shows the application of the E² Algorithm to a 9-node Wheel Graph. Also, the algorithm finds the correct Hamiltonian cycle in all Wheel Graphs.

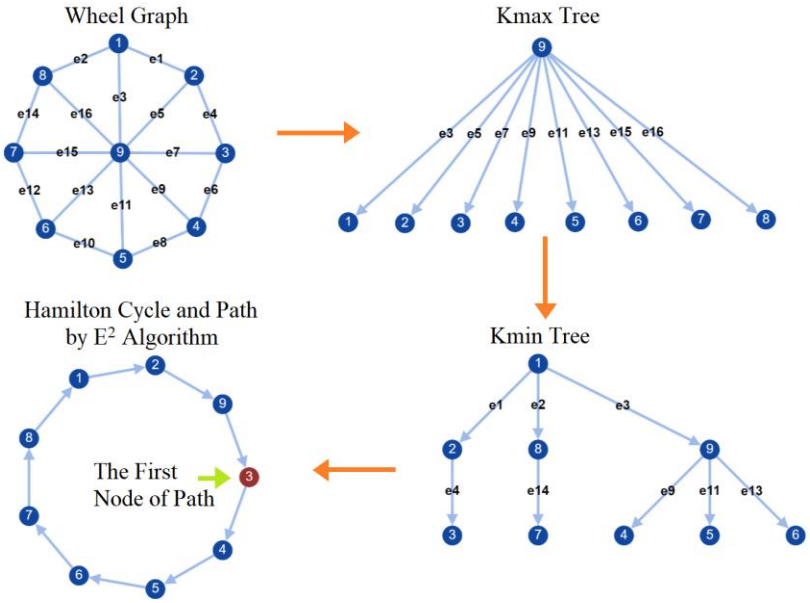

Figure 7. The results of the proposed method on the wheel graph

The algorithm gives effective results on non-Hamilton graphs as well as Hamilton graphs. If the graph does not contain a complete cycle but contains a full path that visits all nodes at once, this path can be found by the proposed algorithm. It was seen in the applications made in common graphs such as Petersen and Sousselier that the proposed algorithm confirmed that there was no Hamiltonian cycle, and the full path was found. Figure 8 shows the results of the application on the Petersen graph example. All the nodes are traversed from the 4th node, and finally, the full path is obtained at the 10th node.

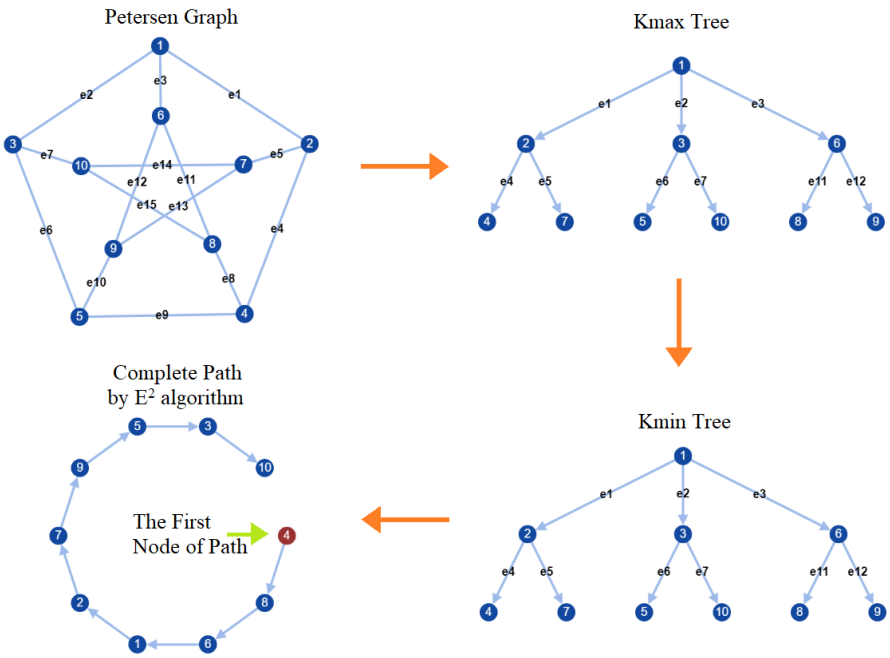

Figure 8. The results on Petersen graph.

The application results of the E^2 Algorithm in some common and small graphs (Wheel, HyperCube, Grötzsch, Frucht, Errera, Folkman, Franklin, Petersen, and Sousselier) were given in Table 1. The algorithm has been applied effectively to all given graphs. The Hamilton cycle and path have been successfully obtained for most Hamiltonian graphs such as Wheel, Hypercube, Grötzsch, Errera, Frucht and Folkman Graphs by E^2 Algorithm. A complete path including all nodes was formed in non-Hamilton graphs, such as Petersen and Sousselier. The Franklin graph, although Hamiltonian, yielded only a Hamiltonian path under E^2. This outcome is consistent with the nature of E^2 as a deterministic greedy algorithm: the fixed priority rule based on minimum degree and edge efficiency does not incorporate backtracking, and therefore cannot revise earlier decisions when a dead end is reached. In regular graphs such as Franklin, where all vertices share the same degree, the tie-breaking mechanism becomes the dominant selection factor, which can lead the traversal into a configuration from which the starting node cannot be re-reached. This identifies regular and near-regular graphs as instances where E^2 is more susceptible to failure, and motivates future work on limited backtracking extensions.

Table 1. Application results of the proposed algorithm in common graphs.

Graph	Hamiltonian	Number of Nodes	Number of Edges	Hamilton Path Created	Hamilton Cycle Created	Most Efficient Edge	First Node
Wheel	Yes	9	16	✓	✓	e_6 (V_3→V_4)	3
Hyper Cube	Yes	16	32	✓	✓	e_{32} (V_{15}→V_{16})	15
Grötzsch	Yes	11	20	✓	✓	e_{10}(V_4→V_5)	4
Errera	Yes	17	44	✓	✓	e_{27} (V_6→V_7)	6
Folkman	Yes	20	40	✓	✓	e_{19} (V_{13} →V_6)	13
Frucht	Yes	12	18	✓	✓	e_7 (V_4→V_3)	4
Petersen	No	10	15	✓	-	e_8 (V_4→V_8)	4
Sousselier	No	16	27	✓	-	e_{27} (V_{15}→V_{16})	15
Franklin	Yes	12	18	✓	-	e_{10} (V_7→V_5)	7

To contextualise the performance of E^2, Table 2 provides a conceptual comparison with established approaches for the Hamiltonian cycle and path problem. Unlike metaheuristic methods such as simulated annealing [25], genetic algorithms [25], and ant colony optimization [26], E^2 is a fully deterministic algorithm: it requires no hyperparameter tuning, no population initialization, and produces identical output for the same input. This makes it more predictable and computationally lighter per run. The primary trade-off is the absence of backtracking: metaheuristic methods implicitly explore multiple solution candidates through iterative restarts or population-based search, whereas E^2 commits to a single greedy traversal. Additionally, E^2 is currently designed for unweighted graphs, while most metaheuristic approaches naturally accommodate weighted graphs and TSP-like objectives. Among exact methods, dynamic programming on Hamiltonian cycle runs in $O(2^n \cdot n^2)$ time [1], and brute-force search requires $O(n!)$ — both infeasible for large graphs. E^2 achieves $O(|V|^2)$ deterministic polynomial complexity at the cost of completeness guarantees.

Table 2. Conceptual comparison of Hamiltonian cycle/path methods.

Method	Time Complexity	Deterministic	Guarantees Cycle	Weighted Graphs	Hyperparameters
E² (this study)	$\boldsymbol{O(\lvert V\rvert^2)}$	Yes	No	No	None
Simulated Annealing [25]	$O(\lvert V\rvert^2 \cdot iter)$	No	No	Yes	Temperature, cooling rate
Genetic Algorithm [25]	$O(pop \cdot \lvert V\rvert^2)$	No	No	Yes	Pop. size, mutation rate
Ant Colony Optimization [26]	$O(\lvert V\rvert^2 \cdot iter)$	No	No	Yes	Pheromone params, ants
Snakes and Ladders Heuristic [27]	$O(\lvert V\rvert^3)$	Yes	No	No	None
Dynamic Programming [1]	$O(2n \cdot n^2)$	Yes	Yes	Yes	None
Brute Force	$O(n!)$	Yes	Yes	Yes	None

4. Conclusions

The problem of finding Hamiltonian path or Hamiltonian cycle for any connected graphs is an NP-complete problem. There are many studies to solve Hamiltonian path/cycle finding problem in literature. However, none of them has used spanning trees and fundamental cut-sets of given connected graphs. In this paper, we proposed an innovative approach to find Hamiltonian path/cycle for given connected graphs based on constructing two special spanning trees (Kmax and Kmin) of given graphs, and fundamental cut-sets of this graph with respect to both of spanning trees.

The E^2 Algorithm is based on the edge efficiencies obtained by constructing spanning trees. We applied E^2 to search for the Hamiltonian cycle in given graphs using these edge efficiencies. A four-step method was created to detect the Hamilton cycle. In the first stage, the Kmax tree; in the second stage, the Kmin tree; and in the third stage, the fundamental cut-sets were created for both spanning trees. In the last step, a method was applied that determines the order in which the nodes should be traversed for the Hamiltonian path using the edge efficiency numbers. The last step was completed when all nodes were visited once, so The Hamilton path was formed. The Hamilton cycle was also formed if there was a connection between the first node and the last node.

The proposed algorithm was applied in some small and general graphs, and its effectiveness was shown in the results section. When the results are examined, it is seen that the cycle and path of Hamiltonian graphs such as Wheel, HyperCube, Grötzsch, Frucht, Errera, and Folkman were determined. In Petersen and Sousselier graphs, which are known to be non-Hamiltonian, a cycle naturally did not occur, but a Hamiltonian path was observed. As an exception, although the Franklin graph is Hamiltonian, a cycle was not formed, but the Hamiltonian path was formed. This four-step method contains fully iterative algorithms, so it is guaranteed that the time complexity of method is bounded by a polynomial function.

E^2 Algorithm was designed for unweighted graphs; extending it to weighted graphs constitutes ongoing work. One natural approach is to define a weighted edge efficiency as $\varrho w(e_i) = \varrho(e_i) / w(e_i)$, assigning higher priority to edges that appear frequently in fundamental cut-sets but carry low weight, a property desirable for shortest Hamiltonian path variants. The traversal rule in Algorithm 4 would then select the neighbor minimizing a combined score of degree and weighted edge efficiency. A formal analysis of this extension, including its behavior on TSP-like objectives, is left for future work. Although the algorithm produces effective results in this state, there is uncertainty about whether it is optimal or not. Our proof works continues on this issue.

4.1. Limitations

Although E^2 produces correct results on a wide range of graph families, three structural conditions are identified under which the algorithm is more susceptible to failure.

Regular and near-regular graphs. In graphs where all vertices have the same degree, such as the Franklin graph, the minimum-degree selection rule in Algorithm 4 provides no discrimination between candidate neighbors. As a result, tie-breaking via edge efficiency becomes the sole determinant of the traversal order. If the edge efficiency values are also closely distributed, as tends to occur in regular graphs with symmetric structure, the traversal may reach a configuration from which the starting node cannot be revisited, producing only a Hamiltonian path. This is the condition observed in the Franklin graph experiment.

Dense graphs. In highly connected graphs, the number of fundamental cut-sets in which any given edge participates increases, causing the edge efficiency values $\varrho(e_i)$ to converge toward similar magnitudes. When efficiency values are nearly uniform, the priority signal provided by ϱ weakens, and the traversal relies more heavily on degree-based selection. This reduces the structural guidance that distinguishes E^2 from a plain greedy degree traversal.

Absence of backtracking. E^2 makes irrevocable greedy decisions at each step. Once a node is added to the path, the algorithm does not reconsider earlier choices even if a dead end is encountered. This is a fundamental property shared by all greedy algorithms without backtracking and is the primary reason why polynomial complexity is achievable. Incorporating even limited backtracking, for example by reverting the last k steps upon reaching a dead end, would increase the worst-case complexity but could substantially improve success rates on the graph families described above. This constitutes the primary direction for future work.

Declarations

Source Code and Data Availability. Code for E^2 Algorithm is provided as part of the replication package. It is available at https://github.com/fatihokumus/e2algorithm for reviewers. Moreover, there is an application site for E^2 algorithm at https://fatihokumus.github.io/e2algorithm. Finally, there are some sample graph data for testing at https://github.com/fatihokumus/e2algorithm/tree/master/data. Reviewers can reproduce graph data samples.

References

[1] Garey MR, Johnson DS. Computers and Intractability: A Guide to the Theory of NP-Completeness. USA: W. H. Freeman & Co., 1990.
[2] Karp RM. Reducibility among combinatorial problems. In: Miller RE, Thatcher JW, editors. Complexity of Computer Computations. New York: Plenum Press, 1972. pp. 85-103.
[3] Ore O. Hamilton-connected graphs. J Math Pure Appl 1963; 42: 21-27.
[4] Sartakhti JS, Jalili S, Rudi AG. A new light-based solution to the Hamiltonian path problem. Future Gener Comput Syst 2013; 29(2): 520-527.
[5] Keshavarz-Kohjerdi F, Bagheri A. A linear-time algorithm for finding Hamiltonian (s,t)-paths in even-sized rectangular grid graphs with a rectangular hole. Theor Comput Sci 2017; 690: 26-58.
[6] Dybizbański J, Szepietowski A. Hamiltonian cycles and paths in hypercubes with disjoint faulty edges. Inf Process Lett 2021; 172: 106157.
[7] DeBiasio L, Spanier N. On Hamiltonian cycles in balanced k-partite graphs. Discrete Math 2021; 344: 112583.
[8] DeBiasio L, Martin RR, Molla T. Powers of Hamiltonian cycles in multipartite graphs. Discrete Math 2022; 345: 112747.
[9] Agueda R, Borozan V, Diaz R, Manoussakis Y, Montero L. Proper Hamiltonian cycles in edge colored multigraphs. Discrete Math 2017; 340: 1897-1902.
[10] Sivaraman V, Zaslavsky T. Two Hamiltonian cycles. Discrete Math 2022; 345: 112797.
[11] Liu H, Hu X, Gao S. Hamiltonian cycles and paths in faulty twisted hypercubes. Discrete Appl Math 2019; 257: 243-249.
[12] Hell P, Nishiyama H, Stacho L. Hamiltonian cycles in covering graphs of trees. Discrete Appl Math 2020; 282: 271-281.
[13] Liu X, Wang Z, Yu X. Counting Hamiltonian cycles in planar triangulations. J Combin Theory Ser B 2022; 155: 256-277.
[14] Lu H, Wu T. Edge-disjoint Hamiltonian cycles of balanced hypercubes. Inf Process Lett 2019; 144: 25-30.
[15] Liu JJ, Wang YL. Hamiltonian cycles in hypercubes with faulty edges. Inf Sci 2014; 256: 225-233.
[16] T. Anuradha, T. L. Surekha, P. Nuthakki, B. Domathoti, G. Ghorai, F. A. Shami, and M. T. Rahim. Graph theory algorithms of Hamiltonian cycle from quasi-spanning tree and domination based on Vizing conjecture. Journal of Mathematics, pp. 1–7, 2022, doi: 10.1155/2022/1618498.
[17] Biton N, Levi R, Medina M. Distributed CONGEST algorithm for finding Hamiltonian paths in Dirac graphs and generalizations. arXiv preprint 2023; arXiv:2302.00742.
[18] M. Murua, D. Galar, R. Santana. Solving the multi-objective Hamiltonian cycle problem using a Branch-and-Fix based algorithm. Journal of Computational Science 2022; 61: 101618. doi: 10.1016/j.jocs.2022.101578
[19] Karci A. New Algorithms for Minimum Dominating Set in Any Graphs. Journal of Computer Science 2020a; 5: 92-70.
[20] Karci A. Finding Innovative and Efficient Solutions to NP-Hard and NP-Complete Problems in Graph Theory. Journal of Computer Science 2020b; 5: 137-143.
[21] Karci A. Efficient Algorithms for Determining the Maximum Independent Sets in Graphs. Journal of Computer Science 2020c; 5: 144-149.

[22] Karci A, Karci Ş. Determination of effective nodes in graphs. In: International Conference on Science, Engineering and Technology; 2020; Mecca, Saudi Arabia. pp. 127-134.
[23] Karci Ş, Ari A, Karci A. Pençesiz çizgelerde maksimum-yakın bağımsız küme ve üst sınırları için yeni algoritma. Gazi Üniversitesi Mühendislik-Mimarlık Fakültesi Dergisi 2021. doi: 10.17341/gazimmfd.902093.
[24] Boals, A.J., Gupta, A.K. & Sherwani, N.A. Incomplete hypercubes: Algorithms and embeddings. *J Supercomput* **8**, 263–294 (1994). https://doi.org/10.1007/BF01204731
[25] Smaili F. A hybrid genetic-simulated annealing algorithm for multiple travelling salesman problems. Decision Science Letters 2024; 13: 1–14. doi: 10.5267/j.dsl.2024.4.001
[26] Cheng J, et al. Dynamic path optimization based on improved ant colony algorithm. Journal of Advanced Transportation 2023; 2023: 7651100. doi: 10.1155/2023/7651100
[27] Clares B, Roura S. Deterministic "Snakes and Ladders" heuristic for the Hamiltonian cycle problem. arXiv preprint 2019; arXiv:1902.10337.