Published by
NIGERIAN SOCIETY OF PHYSICAL SCIENCES
Available online @ https://journal.nsps.org.ng/index.php/jnsps

J. Nig. Soc. Phys. Sci. 8 (2026) 3739

Journal of the Nigerian Society of Physical Sciences

A novel approach for constructing a minimum spanning tree

Haribhau R. Bhapkar[a], Rezwan Ul Shaban[b], Shabir Ahmad Mir[a], Junaid Nisar[c,*]

[a]Department of Mathematics, Central University of Kashmir, India
[b]Centre for Distance and Online Education, University of Kashmir, Srinagar, India
[c]Symbiosis Institute of Technology, Pune Campus, Symbiosis International (Deemed University), Pune 412115, India

Abstract

The spanning tree of a graph is obtained when all vertices of a graph are connected in such a way that no cycle is formed. This work proposes a new algorithm that, at each round, selects a maximal independent set of vertices (an inclusion-maximal, not necessarily maximum-cardinality, set of pairwise non-adjacent vertices) and attaches to every vertex of that set its cheapest cycle-safe incident edgeto find a minimum spanning tree of any weighted graph. The procedure organizes safe-edge selections into batches indexed by maximal independent sets; its number of such batching rounds depends on the maximal independent sets selected, and this notion of a round is not directly comparable to a single iteration of Prim's or Kruskal's algorithm without further definition. Using the classical cut property, we prove that the procedure always produces a minimum spanning tree of a connected graph. If r denotes the number of independent-set rounds, a straightforward sequential implementation has worst-case running time $O(r(n+m)+m\log m)$. We do not claim, and this paper does not prove, that the number of rounds is minimized over all possible choices of maximal independent sets, nor that the resulting sequential running time improves on the classical $O(m\log n)$ bounds. This work may be useful for large weighted networks such as communication networks, wiring connections, and transportation networks.

DOI: 10.46481/jnsps.2026.3739

Keywords: Weighted graph, Minimum spanning tree, Maximal independent set, Algorithm, Graph algorithms

Article History:
Received: 12 July 2026
Received in revised form: 14 August 2026
Accepted for publication: 20 August 2026
Available online: 26 August 2026

Communicated by: B. J. Falaye

1. Introduction

Throughout this paper, $G=(V,E)$ stands for a simple undirected graph on vertex set V and edge set E, with $n=|V|$ vertices and $m=|E|$ edges. ("Simple" here rules out loops and repeated edges; this convention is used everywhere below, including in Theorem 2.4 and Lemma 2.2.) Every edge $e\in E$ carries a positive integer label $w(e)>0$, its *weight*; the cut-property argument used below (Lemma 2.2) in fact holds for arbitrary real-valued weights, and we restrict to positive integers only to match the convention used in the worked example. Given another graph H, we call H a *subgraph* of G once $V(H)\subseteq V(G)$ and $E(H)\subseteq E(G)$ both hold. By a *walk* of length k joining x to y we mean a sequence $x=v_0,v_1,\ldots,v_k=y$ of $k+1$ vertices, repetitions allowed, in which consecutive terms v_i,v_{i+1} are always joined by an edge; when no vertex in the sequence repeats, the walk is called a *path* instead.

The distance $d(x,y)$ between two vertices $x,y\in G$ is the length of a shortest walk between them, and the *diameter* of G is $\mathrm{diam}(G)=\max\{d(x,y)\mid x,y\in V(G)\}$ (a maximum rather than merely a supremum, since G is finite). We call G *connected*

*Corresponding Author Junaid Nisar. Corresponding Author Tel. No.: +916234.
Email addresses: drhrbhapkar@cukashmir.ac.in (Haribhau R Bhapkar), rezwanbhat21@gmail.com (Rezwan Ul Shaban), mirshabir967@gmail.com (Shabir Ahmad Mir), junaidnisar73@gmail.com (Junaid Nisar)

when every pair of distinct vertices is joined by some walk; otherwise G is *disconnected*, and we set $d(x, y) = \infty$ for any pair with no connecting walk.

A *component* of G is an inclusion-maximal connected subgraph. A *cycle* is a closed path together with one extra edge returning to its start: formally, a sequence $x = v_0, v_1, \ldots, v_k = x$ with $k \geq 3$ in which $v_0, \ldots, v_{k-1}$ are distinct and consecutive terms are joined by an edge. Vertices no two of which are adjacent form what is called an *independent set*; among all independent sets of G, one of the largest possible size is a *maximum independent set*, whereas an independent set to which no further vertex of G can be added is called *maximal*. A maximum independent set is always maximal, though the converse can fail; finding a *maximum* independent set is NP-hard on general graphs (Ref. [1]), so it is essential for tractability that the construction given in Section 2 relies only on maximality, not on maximum size, and we are careful below never to require a maximum-cardinality set.

A subgraph with no cycles is a *forest*, and a connected forest is a *tree*; a tree on n vertices always has exactly $n-1$ edges and a unique path between any two of its vertices. A tree that touches every vertex of a graph is a *spanning tree* of that graph, and among all spanning trees of G, one with the least total edge weight is called a *minimum spanning tree* (MST). Throughout this work, G denotes a simple weighted graph, and the weight of a tree T is understood to be the sum of the weights of the edges belonging to T.

Graphs, and trees in particular, underlie a great many mathematical models of practical systems, and among the tree-related questions that arise, computing a minimum spanning tree of a weighted graph is one of the best studied. A handful of classical procedures address it.

Borůvka's method (Ref. [2]) proceeds component by component: at every round, each component reaches out along its cheapest available edge, and components joined by such edges are merged, continuing until a single component remains. Prim's procedure (Ref. [3]) instead grows one tree from a chosen starting vertex, at each step attaching whichever outside vertex is nearest, in weight, to the tree built so far; once every vertex has joined the tree, the sum of the weights used gives the MST's total cost (Refs. [4, 5]).

Kruskal's procedure (Ref. [6]) takes a different route: the edges are first sorted from lightest to heaviest, and each edge is admitted, one at a time, whenever doing so would not close a cycle among the edges already chosen; the tree emerges as a byproduct of this filtering pass. Prim's method, implemented with a binary heap, runs in $O(m \log n)$ time, while Kruskal's method runs in $O(m \log m)$ time (equivalently $O(m \log n)$ for simple graphs) once the edges are sorted (Ref. [7]); what they share, however, is that each iteration commits only a single vertex or a single edge, though of course a single such iteration can itself involve nontrivial bookkeeping (e.g. heap updates or union–find operations), so "one vertex/edge per iteration" should be read as a description of the commitment granularity rather than as evidence, by itself, of lower efficiency. This is what motivates the batched, several-vertices-at-once strategy developed in this paper.

The Reverse-Delete method (Ref. [8]) works in the opposite direction from Kruskal's: edges are considered from heaviest to lightest and discarded one by one, an edge being dropped permanently only when its removal leaves the graph connected.

All of these procedures are instances of the greedy paradigm – committing to the locally best choice at each step in the hope of reaching a globally optimal outcome – and in fact all of them, including the algorithm proposed here, can be seen as different scheduling strategies for repeatedly invoking the same underlying *safe-edge / cut property* rule (Lemma 2.2 below; see also the generic greedy-MST framework in Ref. [7]). Once a cost is attached to every edge, the underlying question is always the same: find a spanning tree of least total weight. Because this question surfaces throughout graph-theoretic algorithm design, with applications in network design (Refs. [9, 10]), communications, transport planning, and flow optimisation (Ref. [11]), it has attracted sustained attention; general treatments of the relevant graph theory can be found in Refs. [5, 12–15].

A separate line of work studies not a single MST but the whole family of spanning trees of a graph: algorithms exist to enumerate or list all spanning trees (Refs. [16–20]), typically in increasing order of weight. That is a different (and generally harder) problem from the one addressed here, since it must produce an exponentially large output in general; our concern in this paper remains the single-tree problem. Likewise, the batching idea explored here is loosely in the spirit of parallel MST algorithms, which schedule many safe-edge selections at once under a formal model such as PRAM or distributed message passing (Ref. [21]); unlike that literature, we do not yet analyze the present algorithm under such a model (see Section 3). Finally, approximation algorithms for related NP-hard covering and independent-set problems (Refs. [1, 22, 23]) are relevant background for the general primal-dual and independent-set techniques used across combinatorial optimization, though the MST problem itself is solvable exactly in polynomial time and does not require approximation.

The algorithm developed in this paper departs from the one-at-a-time pattern common to Prim's and Kruskal's methods: at each stage it admits an entire maximal set of pairwise non-adjacent vertices together, with the goal of cutting down the number of *rounds* needed to arrive at a minimum spanning tree. We stress already here that reducing the number of rounds is not automatically the same as reducing running time, work, or depth in a parallel model; Section 3 makes this distinction precise.

2. Algorithm

Where Prim's and Kruskal's procedures commit a single vertex or edge at every step, and Borůvka's commits one edge per component per round, the method introduced here selects a whole batch of vertices, together with a corresponding batch of edges, at each round. On some graphs this can bring the construction to completion in fewer *rounds* than Prim's or Kruskal's approaches, in the precise sense of "number of independent-set phases" fixed in Section 3 below; as discussed there, this is a statement about the number of rounds, not about

total sequential running time; a full complexity comparison, however, is left for future investigation.

Problem statement.

Given a connected, weighted graph $G = (V, E, w)$ with $|V| \geq 2$ (the case $|V| = 1$ is trivial: the unique vertex is itself the minimum spanning tree, with no edges to select), the task is to build a minimum spanning tree of G by repeatedly selecting batches of vertices rather than one vertex at a time.

Working mechanism.

Let $G = (V, E)$ be a weighted graph with $n \geq 2$ vertices; the minimum spanning tree of G is obtained by carrying out the following steps. We first fix the bookkeeping used throughout: set $F_0 = \varnothing$ and $U_0 = V$, where U_{i-1} denotes the set of vertices *not yet covered* entering round i, and F_{i-1} the edges selected so far.

Step 1. At the start of round i ($i = 1, 2, \ldots$), choose $H_i \subseteq U_{i-1}$ to be a maximal independent set of the induced subgraph $G[U_{i-1}]$ (for $i = 1$ this is $G[U_0] = G$), i.e. an inclusion-maximal collection of pairwise non-adjacent vertices of $G[U_{i-1}]$.

Step 2. For every vertex $x \in H_i$, attach to it the lowest-weight incident edge of G that does not close a cycle with edges already chosen; ties among equal-weight candidate edges at x are broken by a fixed rule agreed in advance (for instance, the lexicographically smallest edge label), applied consistently throughout the run. Do this for each vertex of H_i, in any order: because H_i is independent, no vertex of H_i can be the "other endpoint" selected for a different vertex of H_i in the same round (its own incident edges are chosen only when it is itself processed), so every vertex of H_i is still isolated in (V, F_{i-1}) at the moment its own edge is chosen, regardless of the order in which H_i is processed; this is what makes the outcome of Step 2 independent of processing order (see also the proof of Theorem 2.4). Let R_i denote the set of the other endpoints of these newly chosen edges, and let $F_i = F_{i-1} \cup \{\text{these newly chosen edges}\}$; then $S_i = H_i \cup R_i$ and $U_i = U_{i-1} \setminus S_i$.

Step 3. While $U_i \neq \varnothing$, apply Steps 1–2 again (as round $i+1$, on $G[U_i]$) to the vertices not yet covered, and keep repeating until the covered set satisfies $S_1 \cup \cdots \cup S_r = V(G)$ (equivalently $U_r = \varnothing$).

Step 4. Once this leaves a forest of subtrees $T_1, T_2, T_3, \ldots, T_k$, repeatedly choose a globally lowest-weight edge of G that joins two distinct components among $T_1, \ldots, T_k$ (such an edge need not be a bridge of G, since it may still lie on some cycle of G; because its two endpoints lie in different components of the current forest, it is automatically free of creating a cycle within the forest built so far), and use it to merge the two components; ties are again broken by the fixed rule above. Carrying this out until a single component remains yields the required minimum spanning tree. (This phase is exactly Kruskal's algorithm run on G restricted to inter-component edges, starting from the forest F_r built by Steps 1–3.)

How H_1 is chosen matters a great deal, since it governs how many rounds the construction will take (see Remark 2.5 below: an arbitrary maximal independent set need not minimize the number of rounds). We refer to the resulting procedure as Bhapkar's algorithm. □

Remark 2.1 (Disconnected graphs)**.** Theorem 2.4 below is stated, and proved, only for connected G; the proof relies on connectivity in Stage 1 ("one piece, not several") to guarantee that Step 4 can always find an edge linking any two remaining components. To extend the algorithm to a disconnected graph, run it independently on each connected component; the union of the resulting trees is a minimum spanning forest. Step 4 as written should then be understood to run only within a single component, not across the whole graph.

Lemma 2.2 (Cut property)**.** Fix a connected, simple, weighted graph $G = (V, E, w)$ with $w(e) > 0$ for all $e \in E$, as fixed in Section 1, and suppose $F \subseteq E$ already sits inside some minimum spanning tree of G. Take any partition $(C, V \setminus C)$ of the vertex set such that none of the edges of F crosses between the two sides, and let e be an edge of least weight among those that do cross. Augmenting F by e still leaves a set of edges contained in some minimum spanning tree of G (see, e.g., Ref. [5]); the statement remains true for arbitrary real-valued weights, though we phrase it here to match the positive-integer-weight convention used elsewhere in this paper.

Remark 2.3. Lemma 2.2 is the classical MST safe-edge / cut-property rule underlying Prim's, Kruskal's, and Borůvka's algorithms alike (Ref. [7]). Theorem 2.4 below shows correctness by checking that every edge Bhapkar's algorithm ever selects is safe in this same sense; the maximal-independent-set batching does not introduce a new optimality criterion, but rather gives a particular *schedule* for applying the cut property to several singleton cuts $(\{x\}, V \setminus \{x\})$ at once, followed by a Kruskal-type completion phase (Step 4) on the resulting forest. We make this explicit here, rather than presenting the batching as furnishing an independent correctness argument.

Theorem 2.4. For any connected weighted graph $G = (V, E, w)$ of the type fixed in Section 1 (simple, undirected, $w(e) > 0$, $|V| \geq 2$), Bhapkar's algorithm outputs a minimum spanning tree of G.

Proof. The argument splits into two stages: first, that the algorithm's output $X = (V, F)$ is indeed a spanning tree of G; second, that this spanning tree has minimum weight.

Stage 1: X is a spanning tree.

Write $G = (V, E, w)$ for the connected, simple, weighted graph under consideration (as fixed in Section 1), with $n = |V|$, and let F denote the edges Bhapkar's algorithm has selected by the time it terminates.

No cycles. Steps 2 and 4 both admit an edge only when doing so avoids closing a cycle among the edges already present, so (V, F) contains no cycle: it is a forest.

Every vertex is reached. Steps 1–3 continue until the covered set satisfies $S_1 \cup \cdots \cup S_r = V(G)$, so each vertex of G is an endpoint of some edge of F.

One piece, not several. Immediately after Step 3, (V, F) may still fall into several tree-shaped pieces $T_1, \ldots, T_k$. Because G is connected, some edge of G always links two of these pieces whenever $k > 1$ (this is where connectivity of G is used; for disconnected G, see Remark 2.1), and Step 4 keeps adding the cheapest such linking edge until only one piece is left.

Putting these three observations together, (V, F) is connected, has no cycles, and touches every vertex; that is, it is a tree on n vertices, and hence has $n-1$ edges. So $X = (V, F)$ is indeed a spanning tree of G.

Stage 2: X has minimum weight.

We track the following invariant as edges are added, in the order the algorithm adds them: *F is always a subset of the edge set of some minimum spanning tree of G.* Once this invariant is confirmed to hold at termination, minimality of X follows immediately, since Stage 1 already shows $X = (V, F)$ to be a spanning tree, and a spanning tree contained in a minimum spanning tree can only be that minimum spanning tree itself.

The invariant holds vacuously at the outset, when $F = \varnothing$.

Now suppose the invariant holds at some point during Step 2, and let x be a vertex of the current maximal independent set H_i about to acquire its cheapest incident edge $e_x = \{x, y\}$. Because x has not yet been covered, and because independence of H_i rules out any other member of H_i having already been linked to x in this same round (as spelled out in the working mechanism above), x sits isolated in (V, F) at this instant; consequently the cut $(\{x\}, V \setminus \{x\})$ is crossed by none of the edges currently in F, so *every* edge incident with x is automatically cycle-safe at this instant, and the cheapest cycle-safe incident edge that Step 2 selects therefore coincides with the cheapest edge crossing this cut. Hence e_x is the cheapest edge crossing this cut, so Lemma 2.2 guarantees that $F \cup \{e_x\}$ still lies inside some minimum spanning tree; the invariant survives this step.

The same reasoning applies during Step 4: if C denotes a component currently present in (V, F), no edge of F crosses the cut $(C, V \setminus C)$ (components are, by definition, maximal connected pieces). Step 4 selects a globally cheapest edge among *all* current inter-component edges; such an edge is in particular the cheapest edge crossing the cut $(C, V \setminus C)$ for either of the two components C it joins, since every edge crossing that cut is itself an inter-component edge and the global minimum is a lower bound for the minimum over any subset. So the cheapest edge crossing that cut is exactly the edge Step 4 selects, and the invariant remains intact by Lemma 2.2.

Since every edge the algorithm ever adds falls under one of these two cases, the invariant persists until the algorithm halts, at which point F is the edge set of X. Hence X is contained in, and therefore equal to, a minimum spanning tree of G. □

Remark 2.5. The number of rounds used by Steps 1–3 depends on which maximal independent set H_i is chosen at each round, and an *arbitrary* maximal independent set need not minimize that number. For the star $K_{1,r}$ (one center joined to r leaves), the set consisting of the center alone is maximal and covers only the center and one leaf in the first round, forcing further rounds; whereas the set of all r leaves is a maximum independent set and covers every vertex in a single round. Choosing a *maximum*-cardinality independent set at each round is NP-hard in general (Ref. [1]), so it would be inappropriate to build the algorithm's efficiency claim on maximum independent sets; the algorithm as stated (Section 2) uses only maximality, and we make no round-optimality claim (cf. the Abstract and Section 3).

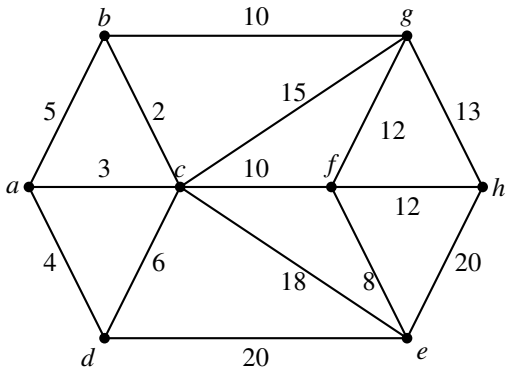

Figure 1: Weighted graph.

Table 1: Complete edge list of G, sorted by weight.

Edge	Weight	Edge	Weight	Edge	Weight
bc	2	cf	10	gh	13
ac	3	fg	12	ce	18
ad	4	fh	12	de	20
ab	5	bg	10	eh	20
cd	6	cg	15		
ef	8				

2.1. Minimum spanning tree of graph shown in Figure 1

Example. Let G be a graph with 8 vertices and 15 edges, as shown in Figure 1.

For clarity and to allow the reader to verify every step independently of the figure, the complete edge list of G, sorted by weight, is given in Table 1.

Solution. We trace through the algorithm above to find a minimum spanning tree of G.

Round 1, Step 1. Take $H_1 = \{a, g, e\}$; no two of these three vertices are adjacent, and no further vertex of G could be added without breaking that property, so H_1 is a maximal independent set.

Round 1, Step 2. Vertex a has three incident edges ($ab = 5, ac = 3, ad = 4$), of which ac is cheapest, so a is joined to c. At vertex g, the incident edges are $gb = 10, gc = 15, gf = 12, gh = 13$, so the cheapest is gb, giving the join g–b. At vertex e, the incident edges are $ec = 18, ef = 8, eh = 20, ed = 20$, so the cheapest is ef, giving the join e–f. Carrying out these three joins produces the forest shown in Figure 2.

$$S_1 = \{a, c, b, g, f, e\}, \qquad U_1 = V \setminus S_1 = \{d, h\}. \tag{1}$$

Round 2, Step 1. Among the vertices not yet covered ($U_1 = \{d, h\}$), d and h are non-adjacent (there is no edge dh in G), so $H_2 = \{d, h\}$ forms the next maximal independent set of $G[U_1]$.

Round 2, Step 2. At d, the incident edges among $\{da = 4, dc = 6, de = 20\}$ give the cheapest edge da, so d joins a; at h, the incident edges among $\{hf = 12, hg = 13, he = 20\}$ give the cheapest edge hf, so h joins f. The forest that results is shown in Figure 3.

$$S_2 = \{a, c, b, g, f, e, d, h\}, \qquad U_2 = \varnothing. \tag{2}$$

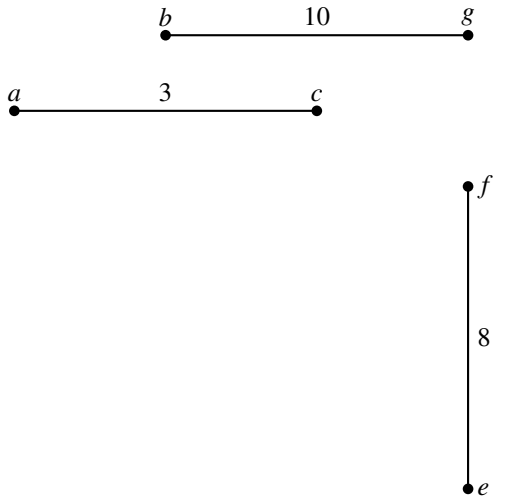

Figure 2: Forest obtained after the first independent-set phase (Steps 1–2).

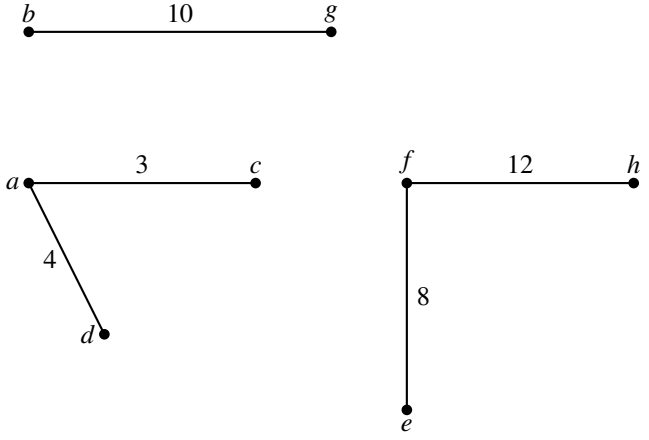

Figure 3: Forest obtained after all vertices have been covered (Round 2, Steps 1–2), consisting of three tree components.

Since $U_2 = \varnicothing$, every vertex of G has now been covered and Steps 1–3 terminate.

Step 4 (completion). The forest (V, F_2) built by Rounds 1–2 has three components: $C_1 = \{a, c, d\}$ (joined by ac, ad), $C_2 = \{b, g\}$ (joined by bg), and $C_3 = \{e, f, h\}$ (joined by ef, fh). Listing every edge of G that runs between two distinct components: $bc = 2$ and $ab = 5$ join C_1–C_2; $cf = 10$, $ce = 18$, and $de = 20$ join C_1–C_3; $fg = 12$ and $gh = 13$ join C_2–C_3 (all other edges of G have both endpoints in the same C_i and so are excluded). We repeatedly add the globally cheapest such inter-component edge: first $bc = 2$ (cheaper than every other listed inter-component edge), merging C_1 and C_2 into $\{a, b, c, d, g\}$, giving the join b–c; then, among the edges linking the two remaining components (ab is now internal and no longer eligible; the candidates are $cf = 10$, $ce = 18$, $de = 20$, $fg = 12$, $gh = 13$), $cf = 10$ is cheapest, giving the join c–f and leaving a single component.

The tree obtained after these two merges, shown in Figure 4,

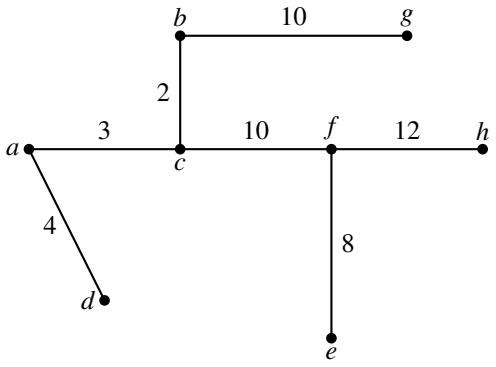

Figure 4: Minimum spanning tree.

is the desired minimum spanning tree.

Summing the weights of its seven edges gives $W(T) = 4 + 3 + 2 + 10 + 10 + 8 + 12 = 49$ for the total weight of this spanning tree.

It is worth stressing that H must be a maximal independent set of whichever vertices remain uncovered at that phase: independence is exactly what allows all of H's vertices to be processed together in Step 2 without interfering with one another (compare the proof of Theorem 2.4). Substituting an edge cover would not even make sense, since H consists of vertices rather than edges; and a vertex cover would not work either, since a vertex cover is generally not independent – rather, it is the complement of an independent set – so using one in place of H would break the argument. As Remark 2.5 shows, however, maximality of H alone does not determine how many rounds the algorithm will take; that depends on which maximal independent set is chosen. A comparative summary is provided in Table 2.

3. Complexity discussion

Number of independent-set phases (rounds), r. This is what Steps 1–3 iterate; by Remark 2.5 it depends on the choice of maximal independent set and is not, in general, minimized by an arbitrary choice. Since each round's independent set H_i is nonempty whenever $U_{i-1} \neq \varnicothing$ (a maximal independent set of a nonempty graph always contains at least one vertex), every round covers at least one new vertex, so trivially $r \leq n$ in the worst case; we do not claim, and have no proof, that any specific graph class (e.g., path-like graphs) forces r close to this trivial bound, so we no longer offer such an example.

Number of inserted edges. This is always $n - 1$ for any spanning tree, including the one this algorithm outputs, and so is not a meaningful point of comparison between algorithms.

Sequential running time. A straightforward implementation computes a maximal independent set of the currently uncovered induced subgraph greedily in time linear in its size, and for each selected vertex scans its incident edge list, together $O(n + m)$ per round in the worst case, giving $O(r(n + m))$ across all rounds in the worst case, which by the trivial bound $r \leq n$ above is $O(n(n + m))$ in the worst case. The completion phase

Table 2: Comparison of Prim's, Kruskal's, and Bhapkar's algorithms.

Prim's algorithm	Kruskal's algorithm	Bhapkar's algorithm
Begins from an arbitrarily chosen vertex	Begins by sorting all edges; processes them in nondecreasing weight order, ties broken by a fixed rule	Begins from a maximal independent set of vertices
Directly grows a single spanning tree; can be restarted once per component for a minimum spanning forest on a disconnected graph	Directly applicable to disconnected graphs, yielding a minimum spanning forest, since it never requires connectivity to pick a next safe edge	Applicable to disconnected graphs only after the modification of Remark 2.1 (run once per component); Theorem 2.4 as proved assumes G connected
Grows a single tree one vertex at a time	Builds up a forest by admitting edges one at a time	Builds up a forest by admitting batches of vertices and their cheapest safe edges together, one maximal independent set per round
With a binary heap, runs in $O(m \log n)$ time	Runs in $O(m \log m) = O(m \log n)$ time after sorting	Worst-case $O(r(n+m) + m \log m)$ (Section 3); whether r can be kept small enough, with an efficient implementation, to beat the above bounds is open
Construction begins at a single chosen root vertex	Scans edges in nondecreasing order of weight, with ties resolved consistently	Construction begins from a set of mutually non-adjacent vertices
A vertex's priority key in the heap may be updated (decrease-key) multiple times over the run	Processes edges, not vertices; each vertex participates in one or more FIND/UNION operations	May be inspected in multiple rounds while independent sets are built (an uncovered vertex is reconsidered next round); may also be an edge endpoint or involved in Step 4 FIND/UNION operations

(Step 4) can be implemented exactly as the tail of Kruskal's algorithm, using a disjoint-set union structure on a single sort of all m edges by weight, in $O(m \log m)$ time. The overall worst-case sequential bound is therefore $O(r(n+m) + m \log m)$, which does *not* improve on the classical $O(m \log n)$ bounds for Prim's or Kruskal's algorithms in the worst case; the independent-set phases add overhead rather than removing it, and any advantage would require r to be sufficiently small, together with a more careful implementation of the independent-set construction than the naive one analyzed here; we do not have a proof that any particular graph class (e.g., bounded-degree graphs) guarantees a small r, and we do not claim one.

Work and depth in a parallel model. If Steps 1–2 are executed with all vertices of H_i processed simultaneously, this is naturally a parallel algorithm, and a rigorous treatment would state a model (e.g., CRCW PRAM, CREW PRAM, or a distributed message-passing model), account for both the total work and the parallel depth of constructing H_i itself (maximal independent set construction is not free in parallel; see, e.g., Ref. [21]), and compare against known parallel MST algorithms. We do not carry out such an analysis here and flag it explicitly as future work, rather than as an established advantage of the method.

In particular, the claim that the algorithm "takes a minimum number of steps" (as stated in earlier drafts of the Abstract) is not established by anything in this paper and has been removed; only the qualitative observation that a whole maximal independent set is processed per round, potentially reducing r below $n-1$, is retained.

4. Comparison with the existing algorithms

This section sets side by side the key characteristics of Prim's algorithm, Kruskal's algorithm, and the algorithm proposed here (Bhapkar's algorithm) in Table 2.

5. Conclusion

This paper has introduced a new procedure for building a minimum spanning tree of a connected graph, which, run once per component (Remark 2.1), also gives a minimum-spanning-forest procedure when the graph is disconnected, by admitting vertices and their cheapest safe edges in coordinated batches drawn from a maximal independent set at each round (Theorem 2.4; see Remark 2.3 for the sense in which this batching is a scheduling of the classical safe-edge rule rather than a new optimality principle). The worked example given above confirms that the procedure produces a correct minimum spanning tree in two rounds on that particular graph; as Remark 2.5 and Section 3 discuss, this does not by itself establish that the number of rounds, let alone the running time, is smaller than Prim's or Kruskal's algorithms on graphs in general, and establishing a general asymptotic comparison against existing algorithms remains a task for future work. A natural next step, left open here, is to identify graph classes or choice rules for the maximal independent set under which the number of rounds r is provably small.

Data availability

No external datasets were generated or analyzed during this study. All data and information used to illustrate and validate the proposed algorithm are included within the article. Therefore, no additional data are required to reproduce the results presented in this work.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this manuscript.

Funding

The authors received no specific funding from any public, commercial, or not-for-profit organisation for this research.

References

[1] M. Xiao, S. Huang & X. Chen, "Maximum weighted independent set: Effective reductions and fast algorithms on sparse graphs", Algorithmica **86** (2024) 1293. https://doi.org/10.1007/s00453-023-01197-x.

[2] J. Nešetřil, E. Milková & H. Nešetřilová, "Otakar Borůvka on minimum spanning tree problem: Translation of both the 1926 papers, comments, history", Discrete Mathematics **233** (2001) 3. https://doi.org/10.1016/S0012-365X(00)00224-7.

[3] R. C. Prim, "Shortest connection networks and some generalizations", Bell System Technical Journal **36** (1957) 1389. https://doi.org/10.1002/j.1538-7305.1957.tb01515.x.

[4] A. Grama, A. Gupta, G. Karypis & V. Kumar, *Introduction to parallel computing*, 2nd ed., Addison-Wesley, Harlow, England, 2003. https://www.cs.purdue.edu/homes/ayg/book/index.html.

[5] K. H. Rosen, *Discrete mathematics and its applications*, 4th ed., WCB/McGraw-Hill, Boston, USA, 1999. https://search.worldcat.org/title/39905655.

[6] J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem", Proceedings of the American Mathematical Society **7** (1956) 48. https://doi.org/10.1090/S0002-9939-1956-0078686-7.

[7] J. Kleinberg & É. Tardos, *Algorithm design*, Addison-Wesley, Boston, USA, 2006. https://books.google.com/books?id=OiGhQgAACAAJ.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest & C. Stein, *Introduction to algorithms*, 3rd ed., MIT Press, Cambridge, MA, USA, 2009. https://mitpress.mit.edu/9780262033848/introduction-to-algorithms/.

[9] O. T. Arogundade, B. Sobowale & A. T. Akinwale, "Prim algorithm approach to improving local access network in rural areas", International Journal of Computer Theory and Engineering **3** (2011) 413. https://doi.org/10.7763/IJCTE.2011.V3.340.

[10] E. O. Effanga & U. E. Edeke, "Minimum spanning tree of city to city road network in Nigeria", IOSR Journal of Mathematics **12** (2016) 41. https://doi.org/10.9790/5728-1204054145.

[11] R. L. Graham & P. Hell, "On the history of the minimum spanning tree problem", IEEE Annals of the History of Computing **7** (1985) 43. https://doi.org/10.1109/MAHC.1985.10011.

[12] B. Bollobás, *Graph theory: An introductory course*, 1st ed., Springer, New York, USA, 1979. https://doi.org/10.1007/978-1-4612-9967-7.

[13] M. C. Golumbic & I. Ben-Arroyo Hartman (Eds.), *Graph theory, combinatorics and algorithms: Interdisciplinary applications*, Springer, New York, USA, 2005. https://doi.org/10.1007/b106672.

[14] R. Diestel, *Graph theory*, 1st ed., Springer, New York, USA, 1997. https://books.google.com/books?id=bu_uAAAAMAAJ.

[15] L. R. Foulds, *Graph theory applications*, Springer, New York, USA, 1992. https://doi.org/10.1007/978-1-4612-0933-1.

[16] H. N. Gabow, "Two algorithms for generating weighted spanning trees in order", SIAM Journal on Computing **6** (1977) 139. https://doi.org/10.1137/0206011.

[17] S. Kapoor & H. Ramesh, "Algorithms for enumerating all spanning trees of undirected and weighted graphs", SIAM Journal on Computing **24** (1995) 247. https://doi.org/10.1137/S009753979225030X.

[18] T. Matsui, "A flexible algorithm for generating all the spanning trees in undirected graphs", Algorithmica **18** (1997) 530. https://doi.org/10.1007/PL00009171.

[19] K. Sörensen & G. K. Janssens, "An algorithm to generate all spanning trees of a graph in order of increasing cost", Pesquisa Operacional **25** (2005) 219. https://doi.org/10.1590/S0101-74382005000200004.

[20] T. Yamada, S. Kataoka & K. Watanabe, "Listing all the minimum spanning trees in an undirected graph", International Journal of Computer Mathematics **87** (2010) 3175. https://doi.org/10.1080/00207160903329699.

[21] D. A. Bader & G. Cong, "Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs", Journal of Parallel and Distributed Computing **66** (2006) 1366. https://doi.org/10.1016/j.jpdc.2006.06.001.

[22] I. Bansal, J. Cheriyan, L. Grout & S. Ibrahimpur, "Improved approximation algorithms by generalizing the primal-dual method beyond uncrossable functions", Algorithmica **86** (2024) 2575. https://doi.org/10.1007/s00453-024-01235-2.

[23] L. Huang, W. Yu & Z. Liu, "Approximation algorithms for the min–max mixed rural postmen cover problem and its variants", Algorithmica **86** (2024) 1135. https://doi.org/10.1007/s00453-023-01187-z.