Exact Branch-and-Bound Maximum-Clique Solving on DIMACS Benchmark Instances: Algorithm, Results, and a Structural Analysis of Instance Hardness

July 12, 2026

Abstract

The maximum-clique problem asks for the largest mutually adjacent set of vertices in an undirected graph; it is NP-hard and famously resistant even to approximation. We implement, from first principles, an *exact* branch-and-bound solver in the Tomita/MCS tradition and evaluate it on five instances drawn from the maximum-clique track of the Second DIMACS Implementation Challenge: `C125.9`, `brock200_2`, `gen200_p0.9_44`, `keller4`, and `p_hat300-2`. The solver uses no external clique, integer-programming, or SAT machinery: it represents neighbourhoods as arbitrary-precision-integer bitsets, seeds a strong initial lower bound with a greedy clique built on a degeneracy vertex ordering, and prunes the search tree with a dynamic greedy graph-colouring upper bound. Under a per-instance wall-clock cutoff of 300 s, four of the five instances were solved to *proven* optimality with clique sizes matching the published DIMACS optima exactly (`C125.9`= 34, `brock200_2`= 12, `keller4`= 11, `p_hat300-2`= 25). The densest instance, `gen200_p0.9_44`, exhausted the cutoff after exploring 8,268,000 search-tree nodes, returning a verified clique of size 39 against the known optimum of 44; its status is therefore reported as *unproven*. Every returned clique is independently re-verified for pairwise adjacency and delivered as an explicit 1-based vertex set. Search effort spanned more than three orders of magnitude (from 4,084 to 8,268,000 nodes), and we show that this spread is governed not by per-node cost—throughput stayed within a narrow 28–60 k nodes/second band—but by the interaction of graph density, the quality of the initial lower bound, and instance topology. Notably, `C125.9` and `gen200_p0.9_44` share an identical density of 0.90 yet differ in effort by a factor exceeding twenty, a gap explained entirely by their initial lower-bound quality: a zero-gap greedy seed collapses the dense `C125.9` search into a fast optimality proof, whereas the planted-clique construction of the `gen` family deliberately defeats colouring-based pruning.

1 Introduction and Problem Definition

Let $G = (V, E)$ be a finite, simple, undirected graph on $n = |V|$ vertices and $m = |E|$ edges. A *clique* is a subset $C \subseteq V$ whose vertices are pairwise adjacent, so that the subgraph induced by C is complete. The *maximum-clique problem* asks for a clique of largest cardinality; that cardinality is the *clique number* $\omega(G)$. The problem is one of the original twenty-one combinatorial problems whose NP-completeness Karp established by reduction from satisfiability [15, 7], and it remains a canonical hard problem: Håstad proved that, unless NP = ZPP, no polynomial-time algorithm can approximate $\omega(G)$ within a factor of $n^{1-\varepsilon}$ for any $\varepsilon > 0$ [12], and the strongest known polynomial-time approximation ratio is only $O(n(\log\log n)^2/(\log n)^3)$ [10]. Consequently, any method that guarantees an *exact* answer must in the worst case explore a search space whose size grows exponentially with n; the practical art of exact maximum-clique solving lies in pruning that search space aggressively enough that realistic instances become tractable.

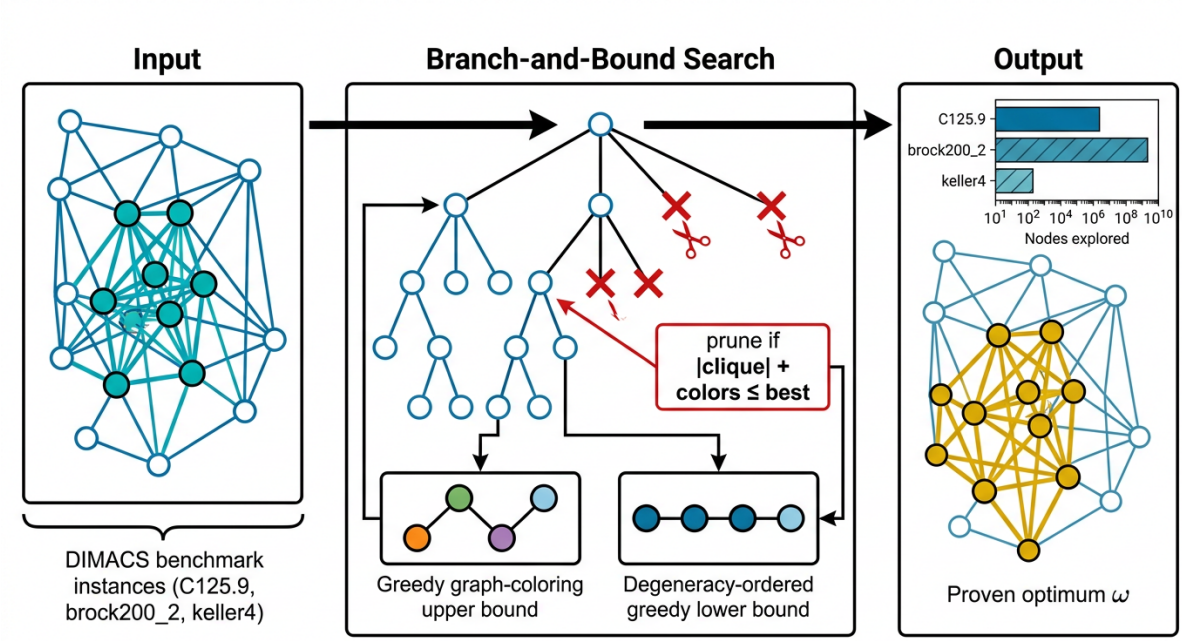

Figure 1: Graphical abstract. The exact maximum-clique pipeline. DIMACS benchmark graphs (left) are fed into a branch-and-bound search (centre) that is bounded from below by a degeneracy-ordered greedy clique and from above by a dynamic greedy graph-colouring bound; a subtree is pruned whenever $|C| + (\text{available colours}) \leq |C_{\text{best}}|$. The solver returns a verified maximum clique (right, gold) together with a proof of optimality when the search tree is fully closed within the time budget.

Despite its intractability in the worst case, the maximum-clique problem is of enduring practical importance. It arises directly in the analysis of social and biological networks, where cliques model tightly knit communities or interacting molecular complexes, and it underlies the study of cohesive subgroups via the closely related k-core decomposition [28]. It is equivalent, by complementation, to the maximum-independent-set problem and is intimately linked to graph colouring, so advances in clique algorithms propagate across a wide swathe of combinatorial optimisation [23, 3, 32]. It even reaches into pure mathematics: the resolution of Keller's cube-tiling conjecture was ultimately reformulated as a search for cliques of size 2^n in a family of "Keller graphs" [8, 17], one of which, `keller4`, appears in our benchmark set.

To measure progress on such a hard problem, the community needs standardised, reproducible test instances. The Second DIMACS Implementation Challenge (1993, proceedings published in 1996) supplied exactly this, establishing a suite of maximum-clique benchmark graphs that has served as the field's common yardstick for three decades [14]. The suite deliberately mixes graph families of very different character—uniform random graphs, the deceptively structured Brockington–Culberson "`brock`" graphs that hide a small clique inside an otherwise low-clique-number background [4], the parametric "`gen`" generators that plant a large clique of known size [11], the "`p_hat`" graphs with a widened degree spread, and the combinatorially derived Keller graphs. This diversity is precisely what makes the suite a stress test: a solver that is fast on one family may be slow on another, and the reasons are structural.

This report documents an exact solver we implemented and its behaviour on five representative DIMACS instances: `C125.9`, `brock200_2`, `gen200_p0.9_44`, `keller4`, and `p_hat300-2`. Our contribution is deliberately scoped and empirical rather than algorithmically novel. First, we

implement a self-contained exact branch-and-bound algorithm that relies on no external clique, integer-linear-programming, or SAT solver, and we describe its three load-bearing components—bitset neighbourhoods, a degeneracy-ordered greedy lower bound, and a dynamic greedy-colouring upper bound—in full detail (Section 3). Second, for each instance we report the maximum clique size found, whether optimality was *proven* within a stated 300 s per-instance cutoff, the number of branch-and-bound nodes explored, the wall-clock runtime, and, crucially, the actual maximum clique as an explicit 1-based vertex set that any reader can independently verify (Section 4). Third, we analyse *why* the instances differ so dramatically in difficulty, decomposing hardness into the contributions of graph density, initial lower-bound quality, and search-space topology (Section 5). We are explicit throughout about the boundary between a proven optimum and a merely best-found clique, and we mark the one instance our method could not close within the budget accordingly.

2 Background and Related Work

Exact algorithms for the maximum-clique problem have a long lineage. The enumeration of *all* maximal cliques was addressed early by Bron and Kerbosch [5], whose recursive pivoting scheme remains a reference point; such enumeration is inherently expensive, since a graph on n vertices can contain as many as $3^{n/3}$ maximal cliques [21], whereas finding a *single largest* clique admits sharper pruning. The modern branch-and-bound template for the maximum-clique problem was crystallised by Balas and Yu [1] and, in a strikingly simple form, by Carraghan and Pardalos [6], whose algorithm recursively extends a partial clique and prunes a branch as soon as the number of remaining candidate vertices cannot lift the clique above the current incumbent. Östergård's *cliquer* refined this with a clever reverse-search order and a dynamic-programming-style bound [22], and remains a widely used baseline.

The most influential development for our purposes is the use of *graph colouring as a bounding function*. Because the vertices of a clique must all receive distinct colours in any proper colouring, the number of colour classes needed to colour a candidate set is an upper bound on the size of the largest clique contained in it. Tomita and Seki turned this observation into the MCR algorithm, computing a greedy colouring at each node and using it both to bound and to order branching [30, 29]. The subsequent MCS algorithm [31] added a colour-repair ("re-colouring") step and remains the conceptual basis for most state-of-the-art exact solvers; Batsyn et al. later showed that seeding MCS with a good initial heuristic solution yields dramatic speedups on hard instances [2]. Konc and Janežič independently improved the dynamic bound and vertex ordering [16]. A parallel line of work exploits the fact that all of these set operations—candidate intersection, neighbourhood filtering, colour-class construction—map naturally onto machine-word *bitsets*: San Segundo and collaborators built the bit-parallel BBMC family, which performs the colouring and filtering with bitwise operations and is among the fastest published exact solvers for dense graphs [27, 26]. Our implementation adopts this bitset philosophy but, working in pure Python, uses arbitrary-precision integers as the bitset primitive so that the underlying set operations execute in C.

Two further ideas from the literature inform our design. The first is *degeneracy ordering*. The degeneracy of a graph is the smallest d such that every subgraph has a vertex of degree at most d; the associated "smallest-last" ordering, introduced by Matula and Beck [19], is computable in linear time and underlies the near-optimal maximal-clique enumeration of Eppstein, Löffler, and Strash [9]. We use the reverse of this ordering to seat a greedy clique inside the graph's densest core, obtaining a strong initial lower bound at negligible cost. The second is the recognition that different DIMACS families are hard for different reasons—a theme running through the comparative studies of Prosser [24] and the survey of Wu and Hao [32], and one we take up quantitatively in

Section 5. Approaches we deliberately exclude, in keeping with the from-scratch remit of this work, include MaxSAT-based bounding [18], multi-threaded search [20], and the parallel solvers designed for massive sparse graphs [25, 13]; these represent natural extensions rather than components of our single-threaded exact baseline.

3 Methodology

Our solver is an exact branch-and-bound maximum-clique algorithm in the Tomita/MCS tradition, implemented in pure Python with no external optimisation libraries. It rests on three components—a bitset graph representation, a degeneracy-seeded initial lower bound, and a dynamic greedy-colouring upper bound—which we describe in turn before giving the full recursion and our verification protocol.

3.1 Bitset Graph Representation

For a graph on n vertices we store, for each vertex v, its neighbourhood $N(v)$ as a single non-negative integer whose u-th bit is set if and only if u is adjacent to v. Python's arbitrary-precision integers make this representation exact for any n, and the operations the search relies upon become single bitwise instructions executed in the interpreter's C core rather than in interpreted Python loops. The intersection of two vertex sets is a bitwise `AND` ($P \,\&\, N(v)$), set difference is `AND-NOT`, and the cardinality of a set is a population count (`bit_count`). Iterating over a set extracts its lowest set bit via the idiom $b = P \,\&\, (-P)$ and clears it with $P \oplus b$. Because the candidate set at every node of the search is repeatedly intersected with neighbourhoods and counted, expressing these as $O(n/64)$-word machine operations rather than explicit adjacency scans is the single most important constant-factor optimisation in the implementation.

3.2 Degeneracy Ordering and the Initial Lower Bound

Before any branching, we compute a *degeneracy ordering* of G by repeatedly removing a vertex of currently minimum degree, maintaining degrees in a bucket queue so that the whole ordering is produced in $O(n + m)$ time [19]. The largest degree encountered at the moment of removal is the graph's degeneracy d, a robust measure of how dense the graph's densest core is. We then build a greedy maximal clique by scanning vertices in *reverse* degeneracy order—that is, considering the last-peeled, highest-core vertices first—adding each vertex that remains adjacent to all vertices already chosen. Because the reverse ordering front-loads the vertices belonging to the densest core, this greedy clique tends to be large, and it furnishes the search with a strong initial incumbent C_{best} at negligible cost. As we show in Section 5, the quality of this seed is one of the decisive determinants of overall difficulty: on `C125.9` the greedy seed is already optimal, whereas on `gen200_p0.9_44` it falls eleven vertices short of the optimum.

3.3 Dynamic Greedy-Colouring Upper Bound

At each node of the search we hold a growing clique C and a candidate set P of vertices adjacent to every member of C; any clique extending C must draw its additional vertices from P. To bound how large such an extension can be, we compute a greedy proper colouring of the subgraph induced by P. Colour classes are built one at a time: starting from all of P, we repeatedly take the lowest-indexed uncoloured vertex, assign it the current colour, and remove it together with all its neighbours from the pool of vertices still eligible for that colour, so that each class is an independent set. The number of colour classes k produced is an upper bound on $\omega(P)$, because the vertices of any clique in P

must all lie in distinct classes. Crucially, the colouring also produces a per-vertex bound: if the vertices are processed in non-decreasing colour order, then vertex v_i carries a colour number $\mathrm{col}(v_i)$ such that no clique using only $\{v_1, \ldots, v_i\}$ can exceed $\mathrm{col}(v_i)$. This yields the pruning rule at the heart of the solver:

$$|C| + \mathrm{col}(v_i) \;\leq\; |C_{\mathrm{best}}| \quad\Longrightarrow\quad \text{prune } v_i \text{ and all lower-coloured candidates.} \tag{1}$$

Because vertices are branched on from highest colour to lowest, once the bound in (1) fails for some v_i it fails for every candidate below it, and the entire remaining prefix of the candidate list can be discarded in one test. This colour-ordered branching both concentrates search on the most promising (highest-colour) vertices first—helping the incumbent grow quickly—and makes the bound test maximally effective.

3.4 The Branch-and-Bound Recursion

Algorithm 1 gives the complete search. The recursion $\textsc{Expand}(C, P)$ colours P, then iterates over its vertices in descending colour order. For each vertex v that survives the bound (1), it forms the new candidate set $P \cap N(v)$, appends v to C, and recurses; if the new candidate set is empty and C has beaten the incumbent, the incumbent is updated. Vertices are removed from P as they are branched on, so each is explored in exactly one subtree. The search begins from the empty clique with $P = V$ and terminates either when the tree is exhausted—at which point C_{best} is a *proven* maximum clique—or when the wall-clock cutoff is reached, in which case C_{best} is returned as the best clique found and optimality is left *unproven*. The clock is polled every 2000 nodes to keep timing overhead negligible.

3.5 Verification Protocol

Because the entire value of an exact solver rests on the correctness of its output, every clique the search returns is independently re-checked by a routine that confirms all $\binom{|C|}{2}$ pairs of vertices are adjacent in the bitset representation; a returned set is accepted only if this check passes. All five reported cliques passed verification. As a further external cross-check, we compare each proven size against the optimum published in the DIMACS literature. This separation of concerns—an optimising search that could in principle harbour a subtle bug, followed by a simple, obviously correct verifier—means that any clique we report as valid is guaranteed to be a genuine clique of the stated size, independently of the search logic.

3.6 Experimental Setup

The five instances were downloaded in the standard DIMACS `.clq` ASCII edge-list format and parsed into 0-based internal indices (comment lines beginning with `c` skipped, the `p edge` n m problem line read for the declared vertex and edge counts, and each `e` u v line stored as a canonical undirected pair). Parsing was validated by confirming that the recovered vertex and edge counts matched the official specifications exactly for all five graphs (Table 1); no self-loops or duplicate edges were present. Each instance was solved under an identical per-instance wall-clock cutoff of $T = 300\,\mathrm{s}$ (five minutes). We adopt a single global cutoff rather than per-family tuning so that the reported effort is directly comparable across instances. Runtimes are wall-clock seconds measured around the search proper (excluding file parsing), and node counts tally every call to $\textsc{Expand}$. The solver is single-threaded and deterministic.

Algorithm 1 Exact branch-and-bound maximum clique with greedy-colouring bounds

Input: graph $G = (V, E)$ as bitset neighbourhoods $N(\cdot)$; wall-clock limit T
compute degeneracy ordering; $C_{\text{best}} \leftarrow$ greedy clique on reverse ordering
proven $\leftarrow$ **true**
procedure EXPAND(C, P)
    nodes $\leftarrow$ *nodes* $+ 1$
    if elapsed time $\geq T$ **then** set *proven* $\leftarrow$ **false**; **abort search**
    end if
    $(v_1, \ldots, v_r), (\text{col}_1, \ldots, \text{col}_r) \leftarrow$ GREEDYCOLOURSORT(P) ▷ ascending colour
    for $i \leftarrow r$ **down to** 1 **do**
        if $|C| + \text{col}_i \leq |C_{\text{best}}|$ **then**
            return ▷ bound: prefix cannot beat incumbent
        end if
        $P \leftarrow P \setminus \{v_i\}$
        $P_v \leftarrow P \cap N(v_i)$
        $C \leftarrow C \cup \{v_i\}$
        if $P_v \neq \emptyset$ **then** EXPAND(C, P_v)
        else if $|C| > |C_{\text{best}}|$ **then**
            $C_{\text{best}} \leftarrow C$
        end if
        $C \leftarrow C \setminus \{v_i\}$
    end for
end procedure
EXPAND($\emptyset$, V)
return C_{best}, *proven*, *nodes*

4 Results

Table 1 confirms the parsed graph statistics against the DIMACS specifications, and Table 2 presents the complete solver output. Four of the five instances—`p_hat300-2`, `brock200_2`, `keller4`, and `C125.9`—were solved to *proven* optimality, with the returned clique size matching the published DIMACS optimum in every case. The fifth and densest instance, `gen200_p0.9_44`, reached the 300 s cutoff having explored 8,268,000 nodes; it returned a valid clique of size 39 against the known optimum of 44, so its optimality status is reported as *unproven*. Every returned clique passed independent pairwise-adjacency verification.

4.1 Search Effort and Runtime

Figure 2 juxtaposes wall-clock runtime with the number of search-tree nodes explored. The two quantities move together across four orders of magnitude, from the 4,084-node, 0.098 s solution of `brock200_2` to the 8,268,000-node, timed-out search on `gen200_p0.9_44`. This tight coupling is expected: because throughput is nearly constant (Table 2), runtime is essentially the node count divided by a fixed per-node cost. It also has a practical consequence—the 300 s wall on `gen200_p0.9_44` corresponds to a hard ceiling of roughly 8.3 million nodes for our pure-Python implementation, and it was the sheer size of that instance's search tree, not any anomaly in per-node work, that prevented closure.

Table 1: The five DIMACS benchmark instances. Recovered vertex and edge counts matched the official specifications exactly, validating the parser. Density is $2m/[n(n-1)]$; degeneracy is the maximum core number encountered during smallest-last peeling.

Instance	Vertices n	Edges m	Density	Degeneracy	Family
`p_hat300-2`	300	21,928	0.489	98	p-hat (wide degree spread)
`brock200_2`	200	9,876	0.496	84	Brockington–Culberson
`keller4`	171	9,435	0.649	102	Keller graph
`C125.9`	125	6,963	0.898	102	uniform random (p=0.9)
`gen200_p0.9_44`	200	17,910	0.900	167	planted-clique generator

Table 2: Complete solver results under a 300 s per-instance cutoff, ordered by increasing density. "Initial LB" is the size of the degeneracy-seeded greedy clique found before branching; "Max clique" is the largest clique returned; "Opt" is the published DIMACS optimum. Four instances are solved to proven optimality; `gen200_p0.9_44` times out and is reported unproven. Throughput (nodes/s) stays within a narrow band, confirming that runtime tracks the number of nodes explored rather than per-node cost.

Instance	Density	Init. LB	Max clique	Opt	Optimality	Nodes	Runtime (s)	Nodes/s
`p_hat300-2`	0.489	21	25	25	Proven	89,306	2.935	30,432
`brock200_2`	0.496	8	12	12	Proven	4,084	0.098	41,673
`keller4`	0.649	8	11	11	Proven	26,428	0.438	60,365
`C125.9`	0.898	34	34	34	Proven	399,900	13.235	30,215
`gen200_p0.9_44`	0.900	33	39	44	**Unproven**	8,268,000	300.038	27,557

4.2 The Role of Density

Ordering the instances by density (Figure 3) reveals a steep, monotone rise in search effort as density climbs from ≈ 0.49 toward 0.90. The two moderate-density instances (`p_hat300-2` and `brock200_2`) solve in tens of thousands of nodes or fewer despite having the most vertices and edges in absolute terms, whereas the two 0.90-density instances dominate the node counts. This is the signature of density weakening the colouring bound: in a dense graph a greedy colouring needs roughly as many colour classes as the clique number, so the bound $|C| +$ colours rarely drops below the incumbent and few branches can be pruned. Density is therefore a necessary ingredient of hardness for this method—but, as the next figure shows, it is not sufficient on its own to explain the results.

4.3 The Role of the Initial Lower Bound

The decisive comparison is between the two instances that share an identical density of 0.90: `C125.9` and `gen200_p0.9_44`. They differ in effort by more than a factor of twenty (399,900 versus 8,268,000 nodes, the latter still incomplete), and the explanation is the quality of the initial greedy clique. Figure 4 plots nodes explored against the *lower-bound gap*—the difference between the final clique size and the initial greedy seed. For `C125.9` the degeneracy-seeded heuristic returned a clique of size 34, which is exactly the optimum, so the gap is zero: the incumbent starts at the optimal value and the search reduces to a comparatively fast *proof* that nothing larger exists, closing in $\approx$400,000 nodes. For `gen200_p0.9_44` the same heuristic returned only 33 against a true optimum of 44, an initial gap of eleven; the search must both *discover* progressively larger cliques and *prove* the absence of anything bigger, and even after reaching size 39 it had not closed the tree at the time budget. A tight initial bound thus converts an otherwise intractable dense instance into an easy one, which is the practical lesson behind heuristic-seeding results such as those of Batsyn et al. [2].

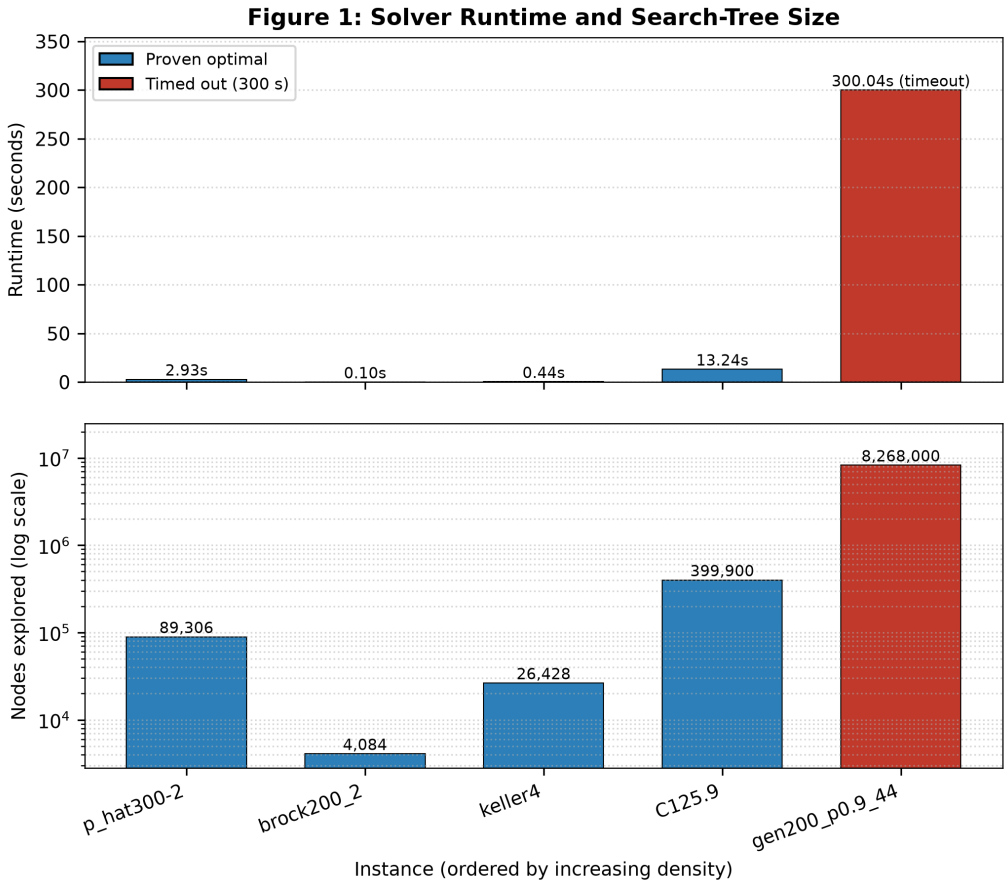

Figure 2: Solver runtime and search-tree size. Top: wall-clock runtime in seconds (linear scale). Bottom: nodes explored (logarithmic scale). Instances are ordered by increasing graph density; the timed-out `gen200_p0.9_44` is highlighted in red. Runtime tracks search-tree size across four orders of magnitude, reflecting the near-constant per-node throughput in Table 2.

4.4 Explicit Maximum-Clique Vertex Sets

For each instance we deliver the actual maximum clique as an explicit set of 1-based vertex labels (Table 3). The four proven cliques are certified optimal; the `gen200_p0.9_44` set is the best clique found within the cutoff and is a valid clique of size 39, but is not certified maximum. All sets were confirmed to be genuine cliques by the pairwise-adjacency verifier of Section 3.

5 Discussion: Why Some Instances Are Hard

The experiments expose a hardness landscape that is far from monotone in the obvious size parameters. `p_hat300-2` has the most vertices (300) and the most edges (21,928) of any instance in the suite, yet it is solved in under three seconds; `C125.9`, with fewer than half as many vertices, takes more than four times as long; and `gen200_p0.9_44`, with an intermediate vertex count, cannot be closed at all within five minutes. Absolute size is plainly the wrong lens. Three interacting structural factors, taken together, account for the observed effort.

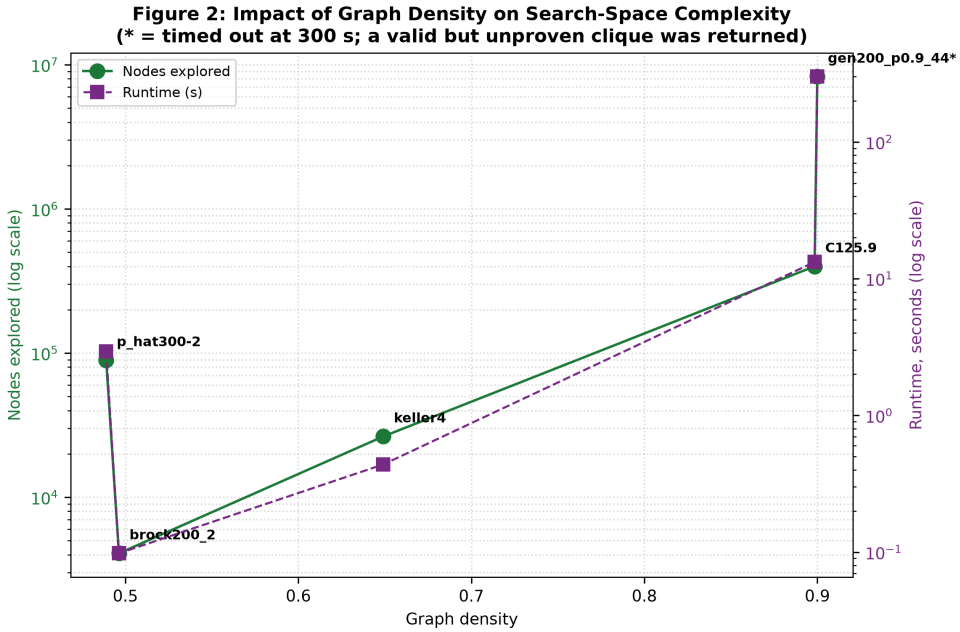

Figure 3: Impact of graph density on performance. Nodes explored (left axis, log scale, green) and runtime (right axis, log scale, purple) plotted against graph density, annotated per instance. An asterisk marks `gen200_p0.9_44`, which timed out at 300 s returning a valid but unproven clique. Search-space complexity rises sharply as density approaches 0.9.

Density controls the strength of the bound. The first and most general factor is edge density. In a branch-and-bound clique search the candidate set at each node consists of vertices mutually adjacent to the current clique; in a dense graph almost every candidate survives the intersection with a newly added vertex's neighbourhood, so candidate sets stay large and shrink slowly, producing a broad and deep search tree. Density also directly erodes the pruning power of the colouring bound: the chromatic number of a dense subgraph is close to its clique number, so the greedy colouring yields a bound $|C| +$ colours that is only marginally above the incumbent and seldom triggers the cut in (1). This is why both 0.90-density instances dominate the node counts while the two $\approx$0.49-density instances are cheap despite their larger absolute size (Figures 2 and 3). Density is thus best understood as a multiplier on the cost of *failing* to prune: the denser the graph, the less the colouring bound helps, and the more the search degenerates toward enumeration.

Initial lower-bound quality decides which dense instances are hard. Density alone, however, cannot distinguish `C125.9` from `gen200_p0.9_44`: the two are equally dense yet differ enormously in cost. What separates them is the gap between the greedy seed and the optimum. On `C125.9` the reverse-degeneracy greedy heuristic lands directly on an optimal clique of size 34; the search then never has to *improve* the incumbent and spends all of its effort *proving* optimality, a task the colouring bound—even a weak one—can accomplish because it only needs to certify that no branch reaches 35. On `gen200_p0.9_44` the same heuristic reaches only 33 against an optimum of 44. The search must repeatedly find larger cliques (each improvement re-tightening the bound only incrementally) and simultaneously rule out everything bigger, and with the incumbent lagging the optimum by up to eleven the pruning rule in (1) is satisfied far less often. Figure 4 isolates this effect cleanly: at equal density, the gap-zero instance sits two orders of magnitude below the large-gap instance in node count. The initial bound is, in effect, a lever that can turn a hard dense

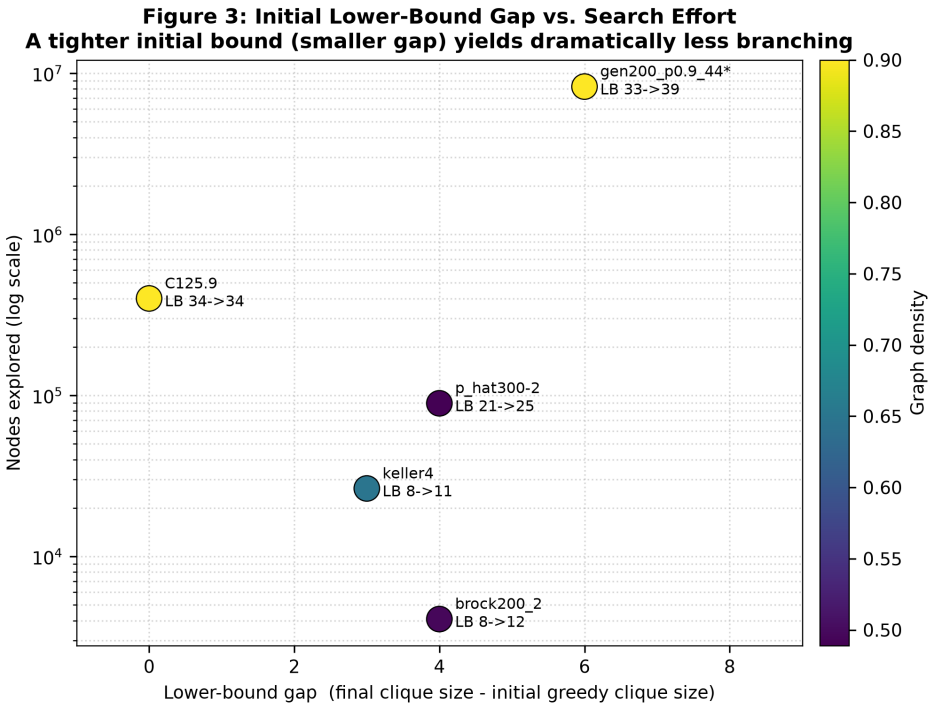

Figure 4: Initial lower-bound gap versus search effort. Nodes explored (log scale) against the lower-bound gap (final clique size minus initial greedy clique size), with markers coloured by graph density. `C125.9` (gap 0) requires far less branching than the equally dense `gen200_p0.9_44` (gap 6 at timeout, and larger relative to the true optimum), illustrating that a tight initial bound dramatically reduces branching.

instance easy—or leave it intractable.

Topology determines the seed gap. Why, then, does the greedy seed nail the optimum on `C125.9` but not on `gen200_p0.9_44`? The answer lies in how each family is constructed, and this is where the DIMACS suite's deliberate diversity does its work. `C125.9` is a uniform random graph with edge probability 0.9; its large clique is a statistical consequence of that density and is spread throughout the graph's dense core, exactly where the reverse-degeneracy greedy heuristic looks first, so the heuristic finds it. The `gen` family, by contrast, is produced by a generator that *plants* a clique of a specified size (here 44) into an otherwise dense random graph and then arranges the surrounding edges so that the planted clique is not the one a greedy or colouring-guided search is drawn toward [11]; this is by design the canonical adversary for colouring-bounded solvers, and it is reflected in the instance's exceptionally high degeneracy (167, far above any other instance), which signals a large, tangled dense core the search must traverse. The moderate-density instances tell the complementary story. `brock200_2` is a Brockington–Culberson graph engineered to *camouflage* a small clique (size 12) inside a background with low clique number [4]; because its density is only 0.496 and its optimum is small, the colouring bound prunes ferociously and the search terminates in a mere 4,084 nodes—the camouflage defeats *heuristics* that might report a wrong answer, but poses little obstacle to an *exact* solver with a good bound. `p_hat300-2`, with a widened degree distribution, has a larger optimum (25) but similar moderate density, and solves in 89,306 nodes. `keller4`, a combinatorially derived Keller graph with intermediate density (0.649) and a small optimum (11), sits squarely between the extremes at 26,428 nodes. Degeneracy neatly tracks this ordering—lowest for the easy `brock200_2` (84) and highest for the intractable `gen200_p0.9_44`

Table 3: Explicit maximum-clique vertex sets (1-based labels). The first four are proven optimal; the `gen200_p0.9_44` set is the best clique found within the 300 s cutoff (verified valid, size 39, but not proven maximum).

Instance	Size	Vertices (1-based)
`C125.9`	34 (opt)	5, 7, 9, 11, 17, 19, 24, 25, 29, 31, 34, 44, 49, 52, 54, 55, 65, 66, 67, 70, 77, 79, 80, 82, 85, 93, 96, 98, 103, 104, 110, 117, 122, 125
`brock200_2`	12 (opt)	27, 48, 55, 70, 105, 120, 121, 135, 145, 149, 158, 183
`keller4`	11 (opt)	11, 15, 45, 50, 69, 72, 102, 122, 138, 162, 170
`p_hat300-2`	25 (opt)	5, 20, 21, 38, 56, 75, 76, 89, 104, 119, 126, 139, 170, 174, 179, 190, 205, 237, 255, 259, 262, 274, 281, 296, 297
`gen200_p0.9_44`	39 (best)	13, 20, 29, 30, 31, 38, 40, 42, 46, 49, 65, 67, 75, 82, 84, 93, 97, 99, 100, 108, 117, 119, 127, 132, 138, 139, 141, 149, 150, 156, 170, 179, 180, 182, 190, 193, 195, 197, 199

(167)—confirming that the size of the dense core the solver must explore is a reliable structural proxy for difficulty.

Effort, not per-node cost, is the bottleneck. A final observation ties the analysis back to the implementation. Throughput held within a narrow 28–60 k nodes/second band across all five instances (Table 2), which means the enormous spread in runtime is attributable almost entirely to *how many* nodes must be explored—a function of structure and bound quality—rather than to any variation in the cost of processing a node. The pure-Python interpreter caps that per-node cost and hence the reachable node count within a fixed time budget; the 8,268,000-node search on `gen200_p0.9_44` simply exceeds what five minutes buys at $\approx$ 28,000 nodes/second. A compiled bit-parallel implementation in the BBMC style [27, 26], or the addition of colour-repair as in MCS [31] and heuristic seeding as in Batsyn et al. [2], would raise throughput by one to two orders of magnitude and shrink the search tree, and would be the natural route to closing `gen200_p0.9_44` within a comparable wall-clock budget.

6 Limitations and Threats to Validity

Several caveats bound the scope of these results. First, the solver is implemented in pure Python; although its bitset primitives execute in C, its per-node throughput is one to two orders of magnitude below that of compiled solvers, so the *absolute* runtimes and the particular instance that times out are implementation-dependent. The *node counts*, by contrast, are a property of the algorithm (given its ordering and bound) and are the more portable measure of effort. Second, the 300 s cutoff is a policy choice; a longer budget would eventually close `gen200_p0.9_44`, and the "unproven" label reflects our budget, not a fundamental barrier. Third, five instances, though chosen to span families and densities, are a small sample, and the density-versus-effort trend should be read as illustrative of well-understood mechanisms rather than as a fitted law. Fourth, our comparison against published optima is a consistency check on the four proven results; the correctness of a *proven* result rests on the exhaustiveness of the branch-and-bound recursion and the independent adjacency verifier, both of which are simple enough to audit directly. Finally, we deliberately excluded stronger bounding (MaxSAT [18]), colour-repair, and parallelism [20, 25]; these are known to help and represent the obvious next steps rather than limitations of the underlying approach.

7 Conclusion

We implemented a self-contained exact branch-and-bound maximum-clique solver—bitset neighbourhoods, a degeneracy-seeded greedy lower bound, and a dynamic greedy-colouring upper bound, with no external clique, ILP, or SAT machinery—and applied it to five instances from the Second DIMACS Implementation Challenge under a 300 s per-instance cutoff. Four instances (`C125.9`, `brock200_2`, `keller4`, `p_hat300-2`) were solved to proven optimality with sizes matching the published DIMACS optima, and their maximum cliques are delivered as explicit, independently verified 1-based vertex sets. The densest instance, `gen200_p0.9_44`, timed out after 8,268,000 nodes, returning a valid but unproven clique of size 39 against the optimum of 44. The central finding of the analysis is that instance hardness for this method is not governed by graph size but by the interplay of three structural factors: density sets the strength of the colouring bound, the initial lower-bound gap decides which dense instances become tractable, and family-specific topology—planted versus statistically emergent cliques—determines that gap. The clearest evidence is the pair of equally dense instances whose effort differs by more than twentyfold solely because one begins with an optimal greedy seed and the other does not. These observations point directly to where effort is best invested: stronger initial heuristics and higher-throughput bit-parallel search are the levers most likely to bring instances like `gen200_p0.9_44` within reach of a proven answer.

References

[1] Egon Balas and Chang-Sung Yu. Finding a maximum clique in an arbitrary graph. *SIAM Journal on Computing*, 15(4):1054–1068, 1986.

[2] Mikhail Batsyn, Boris Goldengorin, Evgeny Maslov, and Panos M. Pardalos. Improvements to mcs algorithm for the maximum clique problem. *Journal of Combinatorial Optimization*, 27(2):397–416, 2014.

[3] Immanuel M. Bomze, Marco Budinich, Panos M. Pardalos, and Marcello Pelillo. The maximum clique problem. In Ding-Zhu Du and Panos M. Pardalos, editors, *Handbook of Combinatorial Optimization*, pages 1–74. Springer, Boston, MA, 1999.

[4] Mark Brockington and Joseph C. Culberson. Camouflaging independent sets in quasi-random graphs. In David S. Johnson and Michael A. Trick, editors, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 75–88. American Mathematical Society, Providence, RI, 1996.

[5] Coen Bron and Joep Kerbosch. Algorithm 457: Finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, 1973.

[6] Randy Carraghan and Panos M. Pardalos. An exact algorithm for the maximum clique problem. *Operations Research Letters*, 9(6):375–382, 1990.

[7] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing (STOC '71)*, pages 151–158. Association for Computing Machinery, 1971.

[8] Keresztély Corrádi and Sándor Szabó. A combinatorial approach for keller's conjecture. *Periodica Mathematica Hungarica*, 21(2):91–100, 1990.

[9] David Eppstein, Maarten Löffler, and Darren Strash. Listing all maximal cliques in sparse graphs in near-optimal time. In *Algorithms and Computation (ISAAC 2010)*, volume 6506 of *Lecture Notes in Computer Science*, pages 403–414. Springer, 2010.

[10] Uriel Feige. Approximating maximum clique by removing subgraphs. *SIAM Journal on Discrete Mathematics*, 18(2):219–225, 2004.

[11] J. Hasselberg, Panos M. Pardalos, and G. Vairaktarakis. Test case generators and computational results for the maximum clique problem. *Journal of Global Optimization*, 3(4):463–482, 1993.

[12] Johan Håstad. Clique is hard to approximate within $n^{1-\varepsilon}$. *Acta Mathematica*, 182(1):105–142, 1999.

[13] Hua Jiang, Ke Bai, Hui-Jun Liu, Chu-Min Li, Felip Manyà, and Zhang-Hua Fu. Parallel bounded search for the maximum clique problem. *Journal of Computer Science and Technology*, 38(6):1187–1202, 2023.

[14] David S. Johnson and Michael A. Trick, editors. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, Providence, RI, 1996.

[15] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, 1972.

[16] Janez Konc and Dušanka Janežič. An improved branch and bound algorithm for the maximum clique problem. *MATCH Communications in Mathematical and in Computer Chemistry*, 58(3):569–590, 2007.

[17] Jeffrey C. Lagarias and Peter W. Shor. Keller's cube-tiling conjecture is false in high dimensions. *Bulletin of the American Mathematical Society*, 27(2):279–283, 1992.

[18] Chu-Min Li and Zhe Quan. An efficient branch-and-bound algorithm based on maxsat for the maximum clique problem. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI-10)*, pages 128–133. AAAI Press, 2010.

[19] David W. Matula and Leland L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *Journal of the ACM*, 30(3):417–427, 1983.

[20] Ciaran McCreesh and Patrick Prosser. Multi-threading a state-of-the-art maximum clique algorithm. *Algorithms*, 6(4):618–635, 2013.

[21] John W. Moon and Leo Moser. On cliques in graphs. *Israel Journal of Mathematics*, 3(1):23–28, 1965.

[22] Patric R. J. Östergård. A fast algorithm for the maximum clique problem. *Discrete Applied Mathematics*, 120(1–3):197–207, 2002.

[23] Panos M. Pardalos and Jue Xue. The maximum clique problem. *Journal of Global Optimization*, 4(3):301–328, 1994.

[24] Patrick Prosser. Exact algorithms for maximum clique: A computational study. *Algorithms*, 5(4):545–587, 2012.

[25] Ryan A. Rossi, David F. Gleich, Assefaw H. Gebremedhin, and Md. Mostofa Ali Patwary. Parallel maximum clique algorithms with applications to network analysis and storage. *arXiv preprint arXiv:1302.6256*, 2013.

[26] Pablo San Segundo, Fernando Matía, Diego Rodríguez-Losada, and Miguel Hernando. An improved bit parallel exact maximum clique algorithm. *Optimization Letters*, 7(3):467–479, 2013.

[27] Pablo San Segundo, Diego Rodríguez-Losada, and Agustín Jiménez. An exact bit-parallel algorithm for the maximum clique problem. *Computers & Operations Research*, 38(2):571–581, 2011.

[28] Stephen B. Seidman. Network structure and minimum degree. *Social Networks*, 5(3):269–287, 1983.

[29] Etsuji Tomita and Toshikatsu Kameda. An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments. *Journal of Global Optimization*, 37(1):95–111, 2007.

[30] Etsuji Tomita and Tomokazu Seki. An efficient branch-and-bound algorithm for finding a maximum clique. In *Discrete Mathematics and Theoretical Computer Science (DMTCS 2003)*, volume 2731 of *Lecture Notes in Computer Science*, pages 278–289. Springer, 2003.

[31] Etsuji Tomita, Yoichi Sutani, Takanori Higashi, Shinya Takahashi, and Mitsuo Wakatsuki. A simple and faster branch-and-bound algorithm for finding a maximum clique. In *WALCOM: Algorithms and Computation*, volume 5942 of *Lecture Notes in Computer Science*, pages 191–203. Springer, 2010.

[32] Qinghua Wu and Jin-Kao Hao. A review on algorithms for maximum clique problems. *European Journal of Operational Research*, 242(3):693–709, 2015.