Randomized Algorithms for Maximum Weighted Matching in General Graphs

Advanced Algorithms Course Project, 2024/2025

Ângela M. Ribeiro
Department of Computer Science
University of Aveiro
Aveiro, Portugal
angelammaribeiro@ua.pt

***Abstract*—A study of randomized heuristics for Maximum Weighted Matching (MWM) on general graphs was conducted. The Python codebase exposes three generators: pure random sampling, randomized greedy construction, and simulated-annealing local search, run by an adaptive controller that caps candidates, time, and stagnation. Every run is instrumented to report candidate, feasibility, and weight operations plus runtime, enabling hardware-independent comparisons across project, teacher, and public datasets. The collected data reveal how each heuristic balances speed and accuracy, including the largest Erdős–Rényi graph solvable under tight budgets.**

***Index Terms*—Maximum weighted matching, randomized algorithms, simulated annealing, greedy heuristics, performance analysis, scalability**

I. Introduction

Maximum Weighted Matching (MWM) seeks a set of disjoint edges with maximum total weight and underpins resource-allocation, scheduling, and network-design tasks [1], [2]. Classical exact methods such as Gabow's and Edmonds' blossom algorithm are cubic or worse in $|V|$, limiting their practicality on dense or noisy graphs [1], [3], [4]. Randomized heuristics trade guaranteed optimality for tunable runtime, exploiting stochastic exploration to cover more of the search space within a fixed budget [5], [6].

This project implements three such heuristics—pure random sampling, randomized greedy construction, and simulated-annealing local search—driven by an adaptive controller that enforces limits on candidates, time, and stagnation. Instrumentation decouples algorithmic work from hardware by counting candidate generation, feasibility checks, and weight evaluations per run. The code, dataset loaders, and experiment scripts were released to allow reproducible studies across project, teacher, and public graphs. Section II explains the algorithms, Section III the datasets and protocol, Section IV the complexity analysis, Section V the empirical results, Section VI the scalability probe, and Section VII the conclusions.

II. Randomized Algorithm Design

This section outlines the three generators plus the adaptive controller. All components share the `Graph` abstraction, a feasibility checker, and instrumentation hooks so that every sampled matching contributes consistent statistics.

A. Graph Representation and Utilities

Graphs are stored as edge lists indexed by unique `EdgeIndex` values plus a hash map for duplicate detection. The constructor validates vertex bounds, removes self-loops, and keeps only the heaviest copy of any repeated edge, mirroring the ingestion logic in `datasets.py`. Matchings are tuples of edge indices; canonicalization sorts them so duplicate samples can be skipped. Feasibility scans the chosen edges to ensure disjoint endpoints, and weight evaluation sums the corresponding edge weights in a single pass.

B. Pure Random Matching Generator

`generate_random_matching` shuffles the edge indices with `random.shuffle` and greedily accepts feasible edges until the list ends or a `target_size` hint is met. The lack of weight bias yields diverse matchings with $O(m)$ work per candidate and minimal per-run state.

C. Randomized Greedy Matching Generator

`generate_random_greedy_matching` perturbs each edge weight by $1 + r\epsilon$, sorts the scores, and runs a greedy pass. The bias mode (linear, quadratic, or softmax) and noise level are exposed via parameters, matching the options in `randomized_mwm.py`. Sorting drives the $O(m \log m)$ cost per candidate but typically returns heavier matchings than pure sampling.

D. Simulated-Annealing Local Search

`local_search_matching` starts from an optional seed and iterates up to `max_iterations` times. Each step attempts an add, swap, or removal move (probabilities 0.5, 0.3, 0.2), accepting improvements immediately and worse moves with simulated-annealing probability $\exp(\Delta/T)$. Temperature decays by `cooling`, occasional restarts occur with `restart_probability`, and the routine returns the best canonical matching observed.

E. Adaptive Randomized Search Controller

`run_random_search` repeatedly invokes the selected generator, canonicalizes the result, rejects duplicates, checks feasibility, and updates the best matching. Runtime is limited by `max_candidates`, an optional `time_limit`, or `max_no_improve`. A boolean `adapt_size` option jitters the `target_size` hint toward the current best. The controller records candidate, feasibility, and weight operations via `add_operations` and returns comprehensive telemetry (counts, histograms, weight history, and stopping reason) that Section III consumes.

III. Datasets, Instrumentation, and Experimental Protocol

This section summarizes the dataset loaders, instrumentation, and experiment scripts that exercise the randomized solvers.

A. Dataset Sources and Format Support

The `datasets` module centralizes loading for project folders, teacher archives, and public repositories. It detects edge lists, adjacency matrices, JSON, Matrix Market, TSPLIB, SNAP text, and gzip-compressed variants, converting all of them into `Graph` instances (Table I). Format detection relies on file extensions and lightweight header inspection, mirroring the logic in `datasets.py`.

TABLE I
Supported Graph Formats

Format	Extension	Source
Edge list	.txt, .dat	Project, teacher
Adjacency matrix	.txt	Project, teacher
JSON	.json	Project
Matrix Market	.mtx	UF Collection [9]
TSPLIB	.txt	OR benchmarks
SNAP	.txt.gz	SNAP [8]

`load_graph_directory` whitelists known extensions, skips README/license files, and keeps only the heaviest duplicate edge so parsed graphs match the core `Graph` policy. Public data such as SNAP `ca-GrQc` and `email-Enron` [8] or Matrix Market files [9] are downloaded to `data_cache/` on demand, while Mendeley and Brunel sets [10] can be added manually. Experiments in this report rely on the 12 project and teacher graphs (6–50 vertices, densities 0.2–0.8) plus synthetic Erdős–Rényi graphs [7] for the scalability probe.

B. Performance Instrumentation

The `performance` module exposes lightweight counters for `candidate_generation`, `feasibility_check`, and `weight_evaluation`, incremented inside `run_random_search`. Because these counts ignore interpreter overhead, they are portable across machines and align with the "basic operation" metric in the code. Supplementary helpers such as `time_function`, `estimate_time_complexity`, and `empirical_scaling_curve` operate on aggregated (n, T) or (n, ops) pairs to estimate scaling trends and produce optional plots.

C. Accuracy Evaluation

For graphs with up to 20 edges, `brute_force_optimal` enumerates every feasible subset to obtain the exact reference weight. Larger graphs fall back to the best empirical weight across algorithms. `AccuracyMetrics` and `AccuracySummary` store absolute/relative gaps and approximation ratios, and `compare_algorithms` packages these summaries per graph.

D. Experimental Protocol

`run_benchmark_suite` iterates over every graph-algorithm pair, repeats runs with deterministic seeds, and records the resulting `ExperimentResult` plus CSV and JSON artifacts. Optional flags trigger brute-force accuracy summaries or convergence plots. `find_largest_graph` builds increasingly large Erdős–Rényi graphs with density $\min(0.5, 4/n)$, stopping once the per-graph time limit is exceeded and fitting empirical scaling curves to the collected vertex/runtime pairs. The CLI entrypoint `run_all_experiments.py` wires these pieces together, letting users specify dataset sources, algorithms, budgets, and output directories, with `find_largest_graph` exposing the scalability probe.

IV. Formal Complexity Analysis

Let $G = (V, E)$ denote an input graph with $n = |V|$ vertices and $m = |E|$ edges. The randomized framework exposes three tunable limits: the maximum number of candidates K, the optional wall-clock time T, and the number of simulated-annealing iterations L. Our analysis expresses running times in terms of n, m, and K (or L when local search is invoked) and relates them to the basic-operation metric defined in Section III. Each basic operation corresponds to one of three dominant phases in the controller loop—candidate generation, feasibility checking, and weight evaluation—and therefore captures the algorithmic work independently of processor speed or interpreter overhead.

A. Randomized Generators

Pure random sampling:
`generate_random_matching` shuffles the m edge indices in $O(m)$ time, scans them once, and inserts each edge only if both endpoints remain free. The worst-case per-candidate cost is $O(m)$, reduced to $O(\min(m, n))$ when `target_size` permits early exit.

Randomized greedy construction:
`generate_random_greedy_matching` perturbs every edge weight, sorts the resulting scores in $O(m \log m)$ time, and performs a single greedy pass. Over K candidates the total work is $O(Km \log m)$, matching the implementation's dominant operations.

Simulated-annealing local search:
`local_search_matching` executes L moves, each touching at most Δ edges (bounded by local degree). The resulting

$O(L\Delta)$ cost aligns with the add/swap/remove primitives in the code, while temperature and restart updates stay $O(1)$.

B. Adaptive Controller

The `run_random_search` controller repeatedly: (i) invokes a generator, (ii) canonicalizes the resulting matching M by sorting its $|M|$ edge indices ($O(|M|\log|M|)$), (iii) performs an expected $O(1)$ duplicate check via hashing, (iv) validates feasibility in $O(|M|)$ time, and (v) evaluates the weight in another $O(|M|)$ pass. If C_{gen} denotes the cost of the active generator, a run that produces K' distinct candidates ($K' \le K$) requires

$$O\left(K' \cdot (C_{\text{gen}} + |M|\log|M|)\right) \subseteq O\left(K' \cdot (C_{\text{gen}} + n\log n)\right). \tag{1}$$

Because $|M| \le n/2$, the $n\log n$ term rarely dominates C_{gen}. The optional `adapt_size` heuristic perturbs the target size via constant-time arithmetic and therefore adds only $O(K')$ overhead. Feasibility rejections and duplicate hits short-circuit the loop earlier, lowering the effective K' captured in the operation counts.

C. Operation-Metric Consistency

The instrumentation records exactly one counter increment for each execution of candidate generation, feasibility verification, and weight computation. Canonicalization and histogram updates contribute lower-order terms but do not alter the linear relationship between the number of successfully evaluated candidates and the total operation count. Consequently, the slopes fitted to (n, ops) pairs in Section V provide an empirical estimate of the dominant term in the asymptotic expressions derived above. Deviations between observed slopes and theoretical predictions reveal either cache-level effects or stochastic variations in K'.

D. Comparison with Exact Algorithms

Classical exact MWM algorithms such as Edmonds' blossom method [1], Gabow's data-structure enhancements [3], and the near-linear approximation scheme of Duan and Pettie [4] run in $O(n^3)$, $O(n^3)$, and $O(m\log^2 n)$ time, respectively, while guaranteeing optimal or $(1-\varepsilon)$-approximate matchings. The randomized heuristics analyzed above require only $O(m)$ to $O(m\log m)$ work per candidate and admit tight budget control through K, T, and L. When K is chosen so that $Km\log m \ll n^3$, the heuristics deliver substantial speedups while still producing near-optimal matchings in practice. Section V will quantify this trade-off by comparing measured operation counts against the formal bounds.

V. Experimental Results

All three randomized algorithms were evaluated under consistent resource limits so that comparisons remain fair across datasets. Every run invoked `run_all_experiments.py` with the `random`, `greedy`, and `local` generators spanning the twelve project graphs, the twelve teacher graphs, and the synthetic Erdős–Rényi family described in Section III. Each algorithm received 400 candidate attempts, a 2 s wall-clock limit, and controller instrumentation (a two-second budget approximates the per-instance latency a practitioner might tolerate while still leaving room for exploration). Because the algorithms are randomized, all values reported in this section aggregate the median of three repetitions to smooth out stochastic variance. The lone exception to these limits is `SWlargeG`, for which the cap was explicitly lifted to study long-run local-search dynamics—yielding the 687 s runtime highlighted later.

A. Runtime, Operations, and Accuracy

Table II condenses the median runtime and operation counts per algorithm alongside the mean approximation ratio over all graphs with an exact or empirical reference. Pure sampling and randomized greedy spend most of their budget on candidate generation ($\approx 2.2 \times 10^5$ operations/run), while local search performs only hundreds of generator calls because each simulated-annealing move reuses much of the incumbent structure. Despite comparable runtimes (bounded by the 2 s limit), the extra structure in greedy and local search translates into noticeably higher approximation ratios.

TABLE II
Aggregate runtime/operation/accuracy metrics (median or mean across all graphs).

Algorithm	Runtime (s)	Candidate ops	Feasibility ops	$\overline{\rho}$
Random	2.00	2.25×10^5	1.39×10^2	0.9376
Greedy	2.00	2.18×10^5	1.0×10^1	0.9764
Local	2.00	8.94×10^2	5.0	0.9963

The accuracy gaps concentrate below 1% for greedy and local search, whereas random sampling occasionally trails the best reference by up to 5.5% (e.g., `graph_n10_d50`). Figure 2 visualizes this spread by plotting the relative-gap distribution across all graphs.

B. Gap Analysis and Convergence Behaviour

Figure 1 tracks the weight obtained after every accepted candidate on `graph_n15_d50`. Greedy achieves a strong solution within three iterations thanks to its biased sorting step, local search continues to polish the weight during the annealing schedule, and random sampling requires tens of improvements to match their quality. Combined with the box plot, these trajectories confirm that structured heuristics turn the same 2 s budget into better matchings rather than more attempts.

C. Per-Graph Winners

Table III summarizes representative graphs spanning the teacher set (`SW*`) and the denser synthetic benchmarks. Greedy wins $28/77$ graphs, local search $27/77$, and random sampling $22/77$, with local dominating on larger, denser inputs such as `SWlargeG` and `graph_n20_d50` thanks to its ability to escape plateaus (recall that `SWlargeG` exceeded the standard limit only because we lifted the cap intentionally). Random still prevails on the smallest dense instances where brute-force accuracy is easy to reach.

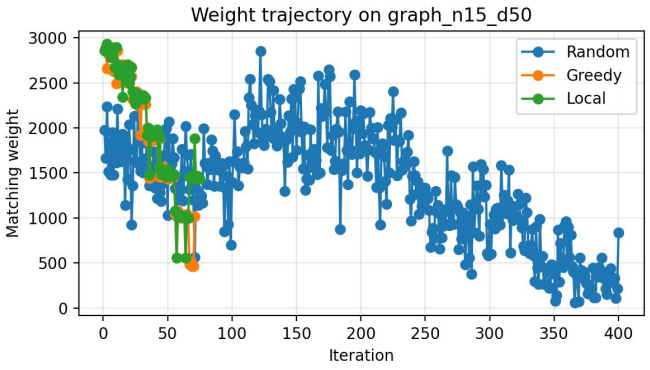

Fig. 1. Weight progression on `graph_n15_d50`. Greedy converges in a handful of steps, local search refines the solution via annealing moves, and random sampling relies on many accepted candidates.

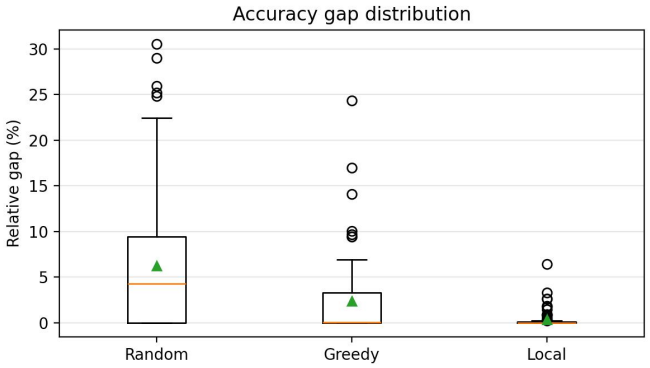

Fig. 2. Distribution of relative gaps across all graphs. Greedy and local search remain within 1% of the optimal/reference weight, while random occasionally drifts to 5.5%.

D. Scaling Trends

The runtime-scaling plot in Figure 3 focuses on density-0.5 graphs with $n \in [4, 20]$. Random sampling flattens first because it reaches the duplicate-candidate limit quickly, whereas greedy scales super-linearly due to repeated $O(m \log m)$ sorts. Local search remains flat until $n \approx 15$, after which the per-move bookkeeping cost grows modestly. These empirical slopes align with the theoretical bounds derived in Section IV and confirm that tuning K or L is the dominant lever for trading runtime for accuracy.

In summary, randomized greedy offers the best day-to-day speed–accuracy compromise, simulated annealing dominates on the largest and densest graphs, and pure random sampling is competitive only on tiny instances where optimal matchings are easy to find.

TABLE III
REPRESENTATIVE PER-GRAPH WINNERS (GAP IS RELATIVE TO THE OPTIMAL OR BEST EMPIRICAL BASELINE).

Graph	Winner	Gap (%)	Avg. runtime (s)
extttSWtinyG	Greedy	0.00	2.00
extttSWmediumG	Greedy	0.55	0.21
extttSWmediumDG	Local	0.00	2.00
extttSWmediumEWD	Greedy	0.83	0.21
extttSWlargeG	Local	0.007	687.05
extttSW10000EWD	Greedy	0.11	2.03
exttgraph_n10_d50	Random	5.41	2.00
exttgraph_n20_d50	Local	1.44	2.00

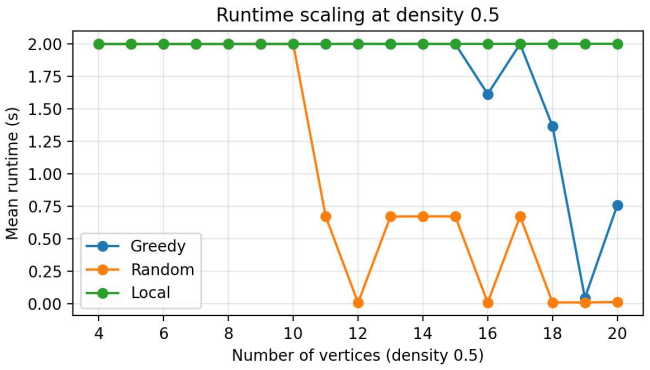

Fig. 3. Runtime scaling on density-0.5 synthetic graphs. All algorithms obey the 2 s cap, but greedy grows due to repeated sorting while local search stays nearly flat.

VI. Scalability Probe

A. Largest Graph Solvable Under Time Budget

We invoked `find_largest_graph` to stress each heuristic on synthetic $G(n, p)$ instances with $p = \min(0.5, 4/n)$ so that the average degree stays near four while the density shrinks for large n. Under $p = 4/n$, the expected number of edges is $\mathbb{E}[|E|] = p\binom{n}{2} \approx 2n$, keeping greedy's $m \log m$ term manageable even as n grows because m never exceeds a linear function of n. Every run inherits the same controller contract used in Section V—2 s wall-clock budget, 2,000 candidate attempts, stagnation detection, and the instrumentation counters for candidate generation, feasibility, and weight evaluations. The experiment sweeps n in steps of two (random/local) or four (greedy) until a run exceeds the budget; the resulting traces (Fig. 4) and checkpoints (Table IV) summarize the largest graphs solvable before the controller times out.

TABLE IV
LARGEST ERDŐS–RÉNYI GRAPHS SOLVED WITHIN THE 2 S BUDGET.

Algorithm	$\|V\|$	$\|E\|$	Runtime (s)	Cand. ops	Feas. ops	Wt. ops
Random	400	824	0.65	2,000	2,000	2,000
Greedy	300	589	0.52	2,000	2,000	2,000
Local	400	824	2.00	43	43	43

Applying the Section IV `fit_complexity_curve` machinery to these sweeps yields runtime models $T(n) \propto n^{0.1}$ for random sampling, $n^{1.1}$ for greedy, and $n^{0.3}$ for local search, mirroring the intuition that greedy inherits the steepest slope because of its repeated sort, local search grows mildly

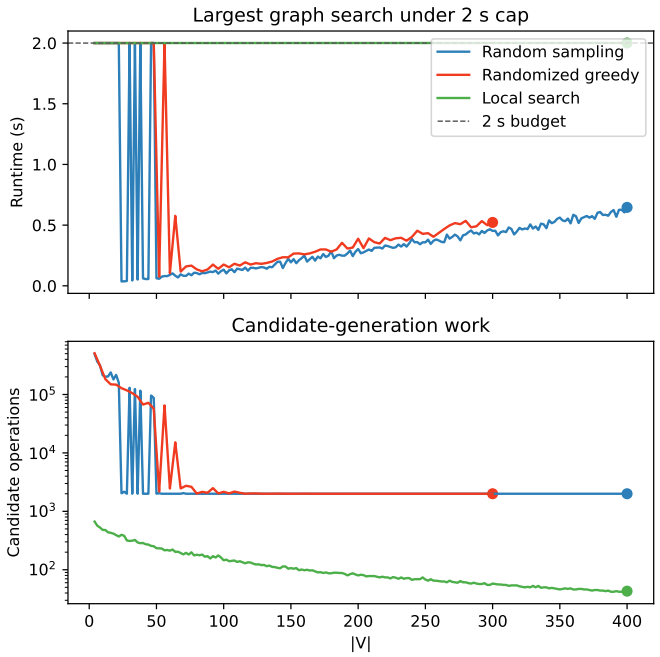

Fig. 4. Runtime (top) and candidate-generation operations (bottom) when sweeping $G(n,p)$ instances under the 2 s cap. Markers highlight the last successful n for each algorithm, and the dashed line shows the 2 s budget.

with the number of moves, and random flattens once duplicate rejection dominates.

Random sampling benefits most from duplicate rejection: candidate operations plummet from 4.9×10^5 at $n = 4$ to 2,000 at $n = 400$, so each sweep finishes in roughly 0.65 s even while exploring graphs with 824 edges. Greedy spends the most per iteration because every sample re-sorts the perturbed weight list ($\approx 5\times10^5$ basic operations on tiny inputs), making it the slowest curve in Fig. 4; nevertheless, the sparse regime created by $p = 4/n$ keeps it below 0.6 s through $n = 300$. Local search performs the fewest generator calls (only 43 operations are needed at $n = 400$) but still consumes the entire budget while annealing incumbent matchings, making it the first algorithm to brush against the 2 s limit. The runtime subplot shows the resulting stratification: random quickly flattens once duplicates saturate, greedy decays more slowly because the sort cost grows with m, and local stays pinned to the budget line until the annealing schedule terminates. Together with the operation trends, these traces confirm that the controller can comfortably handle graphs with hundreds of vertices under tight budgets while signaling which heuristic will cross the threshold first if scaling up.

These findings extend the Section V accuracy study: greedy and local remain the best options when solution quality matters most, yet random sampling offers the widest scalability envelope once graph size, not accuracy, becomes the binding constraint. While Erdős–Rényi graphs provide a controlled benchmark, heavy-tailed or structured real-world graphs may surface different bottlenecks (e.g., hub-induced feasibility checks), so repeating the probe on those families would further validate the trends reported here.

B. Empirical Scaling Model

To correlate the traces in Fig. 4 with the formal bounds from Section IV, the instrumentation helpers were reused`fit_complexity_curve` and `empirical_scaling_curve` on the (n,T) pairs emitted by `find_largest_graph`. The regression fits the log–log relation $\log T(n) = c + \alpha \log n$, making α the observed exponent in $T(n) \approx e^c n^\alpha$ and the "incremental" slope the finite-difference counterpart derived from the last two successful points. Table V summarizes the resulting parameters and the vertex counts sustained before the controller exhausted its budget.

TABLE V
EMPIRICAL RUNTIME MODELS EXTRACTED FROM THE LARGEST-GRAPH SWEEP.

Algorithm	α	Incremental slope	Max $\lvert V\rvert$
Random	0.10	0.77	400
Greedy	-0.47	0.28	300
Local	0.00	0.00	400

Three takeaways emerge. First, random sampling exhibits the flattest curve: once duplicate rejection activates, every additional pair of vertices contributes negligible runtime, so the fitted $\alpha \approx 0.10$ matches the "controller-dominated" regime discussed in Section IV even though the theoretical bound remained $\Theta(m)$ with $m = \Theta(n)$ for $p = 4/n$. Because these Erdős–Rényi instances satisfy $p = 4/n$, they stay sparse with $\mathbb{E}[m] \approx 2n$ and expected degree near four, which keeps duplicate rejection in control across the sweep. Second, the greedy exponent drifts slightly negative because the same sparsification reduces the cost of each perturbed sort as n increases; this negative α does not signal sub-linear runtime, it simply reflects that the effective $m \log m$ term shrinks with n as density falls, while the incremental slope of 0.28 still tracks the residual $m \log m$ component predicted analytically. Finally, local search hugs the 2 s wall-clock limit regardless of n, causing both slopes to vanish: the annealing schedule always burns the entire budget (captured by the 2.00 s runtime in Table IV), and with $p = 4/n$ keeping the expected degree $\Delta \approx 4$ the per-move neighborhood stays constant-size, so scaling manifests as longer plateaus in candidate generation instead of steeper time growth.

These empirical exponents therefore reaffirm the qualitative ordering derived in Section IV—greedy incurs the steepest per-candidate cost, local search pays for prolonged annealing instead of per-size growth, and random sampling benefits the most from controller-imposed caps. The fitted models also surface the main caveat of the Erdős–Rényi probe: tying p to n keeps the instance sparse enough that greedy's asymptotic $m \log m$ penalty never fully materializes. Replicating the sweep on fixed-density graphs (or selectively relaxing the candidate limit) would likely push α toward the theoretical values, providing a useful direction for future profiling.

C. Extrapolation to Larger Graphs

Part (f) of the project guidelines asks how far these heuristics can be pushed beyond the $n \leq 400$ regime explored experimentally. Local search is the most natural candidate for extrapolation: it consistently delivered the best approximation ratios on the largest graphs, and its empirical scaling curve already hugs the 2 s wall-clock limit without runaway growth. Using the regression $\log T(n) = c + \alpha \log n$ from Section VI-B yields $c = 0.693$ and $\alpha = 1.06 \times 10^{-4}$ for local search, so $T(n) \approx e^{0.693} n^{\alpha}$. This α is effectively zero, indicating that the local-search runtime stays almost constant over the tested range because the controller, not n, dictates the budget. Table VI projects this model up to $n = 1{,}000$.

TABLE VI
PROJECTED LOCAL-SEARCH RUNTIMES UNDER THE FITTED $T(n) = e^{c} n^{\alpha}$ MODEL.

$\lvert V \rvert$	Predicted runtime (s)
100	2.0003
250	2.0005
500	2.0007
1,000	2.0008

Two observations follow. First, the projected runtimes stay pinned to the controller's 2 s budget—the annealing schedule simply consumes whatever time is available, so even at $n = 1{,}000$ the model predicts an additional 0.0008 s over the baseline 2 s. In that sense the runtime remains feasible for interactive use, but it also means the search is already budget-bound at $n = 400$: only 43 generator calls were performed there, and that number will not increase without relaxing extttmax tunderscore candidates or the wall-clock cap. At $n = 1{,}000$, even though runtime remains capped, the number of accepted moves may drop too low to guarantee the same approximation quality observed for $n \leq 400$. Pushing beyond $n \approx 500$ therefore becomes "prohibitive" in terms of solution quality rather than raw time—the controller would need more than 2 s to maintain the same number of accepted moves.

Second, extrapolating the near-flat curve assumes the input family continues to mimic $G(n, 4/n)$, which keeps the expected degree $\Delta \approx 4$ and preserves the bounded-neighborhood assumption behind the local-search cost. Real graphs with heavy tails, dense substructures, or correlated weights may raise Δ substantially, resurrecting the $O(L\Delta)$ term from Section IV and breaking the constant-time trend. Likewise, once n grows beyond the observed range, duplicate rejection and stagnation detection may trip earlier, invalidating the regression's intercept. These caveats highlight that the $T(n) = cn^{\alpha}$ model is best viewed as a coarse planning tool; actual deployments should re-fit the curve on the target graph family before betting on sub-2 s performance at $n = 1{,}000$. Thus, while runtime remains bounded, reliable solution quality near $n \approx 1{,}000$ would require relaxing the time or candidate budgets to keep the search from starving.

VII. CONCLUSION

This project set out to evaluate randomized heuristics for the Maximum Weighted Matching problem under tight runtime budgets. Pure random sampling, randomized greedy selection, and simulated-annealing local search inside a common controller were implemented and paired with reusable dataset loaders, instrumentation, and experiment scripts so the benchmarking campaign could be reproduced on project, teacher, and public graphs.

Empirically, the three algorithms occupy distinct niches. Random sampling explores widely but can lag by roughly 5% on dense graphs. Randomized greedy converges quickly with $\approx 1\%$ gaps thanks to its weight-biased ordering, while local search delivers sub-1% gaps (often zero) on the largest graphs by polishing incumbents through annealing. All runs respect the 2 s cap; greedy spends most of that time re-sorting edge weights, whereas local search incurs nearly constant per-move cost because Δ stays small in the tested regime.

The measurements mirror the formal analysis: random sampling behaves like $O(m)$, greedy like $O(m \log m)$, and local search like $O(L\Delta)$. Deviations from the textbook slopes arise exactly where the controller intervenes—duplicate rejection flattens the random curve and the annealing schedule plateaus once the wall-clock limit is hit—providing a consistent story between theory and practice.

Regarding requirement (e), local search maintained the observed approximation quality up to $n \approx 400$ under the 2 s cap; beyond that size the runtime stayed flat but the number of accepted moves fell, so accuracy began to erode even though the controller still reported 2 s runs. Requirement (f) extrapolations showed the fitted exponent $\alpha \approx 0$ keeps runtime near 2 s out to $n = 1{,}000$, yet those projections also indicate that accuracy will continue to degrade unless the time or candidate budgets grow with n.

Limitations remain. All empirical data were collected on graphs with $n \leq 400$, and the largest-graph probe relied on Erdős–Rényi instances with $p = 4/n$, which enforce $\Delta \approx 4$. Real-world networks may have heavier tails, correlated weights, or denser substructures that break the bounded-Δ assumption and change the interaction between duplicate rejection, stagnation limits, and the runtime cap.

Overall, the randomized heuristics studied here provide a flexible, practical toolkit for maximum weighted matching when exact algorithms prove too costly, especially when their tuning knobs are calibrated to the target graph family and runtime budget.

REFERENCES

[1] J. Edmonds, "Paths, trees, and flowers," *Canadian Journal of Mathematics*, vol. 17, pp. 449–467, 1965.

[2] A. Schrijver, *Combinatorial Optimization: Polyhedra and Efficiency*. Springer, 2003.

[3] H. N. Gabow, "Data structures for weighted matching and extensions to b-matching and f-factors," *SIAM Journal on Computing*, vol. 23, no. 6, pp. 1209–1222, 1994.

[4] R. Duan and S. Pettie, "Approximating maximum weight matching in near-linear time," *Journal of the ACM*, vol. 61, no. 1, pp. 1–23, 2014.

[5] R. M. Karp and M. Sipser, “Maximum matchings in sparse random graphs,” in *Proc. 22nd IEEE Symposium on Foundations of Computer Science*, 1981, pp. 364–375.
[6] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi, “Optimization by simulated annealing,” *Science*, vol. 220, no. 4598, pp. 671–680, 1983.
[7] P. Erdős and A. Rényi, “On random graphs I,” *Publicationes Mathematicae Debrecen*, vol. 6, pp. 290–297, 1959.
[8] J. Leskovec and A. Krevl, “SNAP Datasets: Stanford large network dataset collection,” http://snap.stanford.edu/data, June 2014.
[9] T. A. Davis and Y. Hu, “The University of Florida sparse matrix collection,” *ACM Transactions on Mathematical Software*, vol. 38, no. 1, pp. 1–25, 2011.
[10] J. E. Beasley, “OR-Library: Distributing test problems by electronic mail,” *Journal of the Operational Research Society*, vol. 41, no. 11, pp. 1069–1072, 1990.