Optimizing Hackathon Team Composition Based on Skill Coverage and Availability Using Branch and Bound

Jason Edward Salim - 13524034
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: jasonedwardsalim@gmail.com , 13524034@std.stei.itb.ac.id

***Abstract*—Hackathons settle their teams in the first hours of a short, fixed event, and a team that lacks a needed skill or cannot find common working hours rarely recovers. This paper formulates hackathon team composition as a combinatorial optimization problem that balances weighted skill coverage against shared availability over a team whose size may vary. Availability serves as both a hard feasibility requirement and a scoring term, and a fixed per-member cost keeps the search from inflating the team. A branch-and-bound method solves the problem exactly. It eliminates candidates with no relevant skill, branches on candidates in order of decreasing skill weight while deciding team size during the search, prunes any partial team whose shared time has dropped below the required threshold, and bounds the achievable coverage to discard subtrees that cannot beat the best team found. Experiments over six scenario families, each built to make a different factor binding, show that the method matches an exhaustive baseline on every comparable instance while exploring a small fraction of the candidate teams, and that it avoids the feasibility failures a greedy heuristic suffers on tightly scheduled instances. Team size emerges as a genuine decision rather than a fixed input.**

***Index Terms*—Branch and Bound, combinatorial optimization, hackathon, skill coverage**

Fig. 1. The hacker, the hipster, and the hustler form a minimum viable team. Source: Brainhub.

I. Introduction

A hackathon packs the whole process of building software into one short, fixed window, usually 24 to 48 hours. It runs from coming up with an idea and splitting the work to writing the code and presenting the result. The format has spread from software firms into classrooms and civic events, partly because this short, intense window also benefits the people who join. In a few days, participants gain technical skills, learn from their teammates, and build professional networks [1]. These benefits still depend on ending up in a team that works. The window is so short that a team has little room to recover from a weak start. The team is fixed before any code is written, and who is on it largely decides what the team can attempt. Studies of corporate and online hackathons report that outcomes depend heavily on how well a team's combined skills fit the project and on how well its members coordinate under time pressure [2], [3]. A team that finds a missing skill halfway through the event rarely has time to fix it.

A popular rule of thumb in the startup and hackathon community describes a minimum viable team. It pairs a *hacker*, who builds the product, with a *hipster*, who shapes its design, and a *hustler*, who sells it and manages users, as shown in Fig. 1. In this view, three members with different strengths already make a complete team. A fourth or fifth person helps only when they add a skill that is still weak, not when they repeat one the team already has. The labels matter less than the idea behind them. A team does well when its members cover different skills, and it struggles when several people fill the same role and leave the others empty. This tension between team size and skill coverage is what an optimizer must weigh, and it is why our formulation lets the team size vary instead of fixing it in advance.

Two ways of forming teams are common in practice, and both fail in their own way. When participants pick their own teams, they tend to group by who they already know rather than by complementary skills, which gives uneven team sizes and gaps in skill coverage. When an organizer assigns teams

by an algorithm to balance skills, participants often reject the result and regroup on their own. A comparison of two civic hackathons documented this: purely algorithmic assignment caused dissatisfaction, so the authors proposed a two-stage assign-then-swap process instead [4]. Neither extreme treats team composition as what it really is, a constrained optimization problem in which skill coverage and member availability must both be satisfied.

Computational work on team formation has taken the optimization view, but it relies on heuristic and metaheuristic search. Multi-objective genetic algorithms build diverse teams from social-network data [5], and reinforcement-learning-assisted genetic programming has been used for person-job matching [6]. These methods scale to large candidate pools, but they return solutions with no guarantee of optimality, and they rarely model scheduling availability as a hard constraint. That constraint is central to hackathons, where members must be free during the same hours for the team to work. Exact methods such as branch and bound can prove optimality and cut the search space sharply. They have been studied for related problems, including bi-objective set covering [7], but not for hackathon team composition with availability handled explicitly.

This paper addresses that gap. It makes three contributions. First, it formulates hackathon team composition as a two-factor combinatorial optimization problem. It uses weighted skill coverage as the objective and schedule overlap as both a hard feasibility constraint and a partial scoring term, over a team whose size can vary. Second, it presents a branch-and-bound design built for this problem, with a coverage-based bounding function, a branching scheme that decides team size during the search, and two pruning rules, one based on feasibility and one based on the coverage bound. Third, it tests the method against a brute-force baseline, which confirms that the search returns optimal solutions, and a greedy baseline, which measures the trade-off between solution quality and running time.

The rest of the paper is organized as follows. Section II reviews work on hackathon team dynamics, computational team formation, and branch and bound. Section III formalizes the problem. Section IV develops the branch-and-bound method. Section V describes the experimental setup and Section VI reports the results. Section VII discusses implications and limitations, and Section VIII concludes.

II. Related Work

A. Team Formation in Hackathons

Research on hackathons has repeatedly identified team formation as a decisive and difficult stage. A comparison of two civic hackathons found that letting participants choose their own teams produced teams biased by prior acquaintance [4]. Assigning teams by an algorithm to enforce diversity instead caused dissatisfaction and pushed participants to leave or regroup. That work proposed a two-stage process that assigns teams and then allows limited swaps. Case studies of corporate hackathons make the same point from another angle: project outcomes depend on whether a team has the mix of skills its idea needs, and on how its members divide and coordinate the work during the event [2], [8]. When skills are missing or repeated, teams stall or scale back their goals. These findings support treating skill coverage as a main objective rather than a side effect of who happens to sign up together.

B. Computational Approaches to Team Formation

A separate line of work treats team formation as an optimization problem and solves it with search. One approach uses a multi-objective genetic algorithm to build teams that are both diverse and well connected within a social network [5]. Another combines reinforcement learning with genetic programming to evolve rules that assign people to jobs under skill requirements [6]. These metaheuristics handle large candidate pools and several objectives at once, but they have two limitations in the hackathon setting. They return a solution without proving it is optimal, so the size of the quality gap stays unknown, and they usually model skills and preferences while leaving scheduling availability implicit. This paper takes the opposite approach: it solves a smaller, tightly constrained instance exactly, which guarantees optimality and lets availability enter the model as a hard requirement.

C. Branch and Bound for Combinatorial Optimization

Branch and bound is a standard exact method for combinatorial optimization. It explores a tree of partial solutions, computes an optimistic bound on the best completion of each node, and discards any node whose bound cannot beat the best solution found so far [9]. Its efficiency depends on the quality of the bounding function and the order in which branches are explored. It has been adapted to problems close to ours. One scheme handles bi-objective integer programs and applies it to the bi-objective set covering problem [7]. Another gives a branch-and-bound procedure for the assignment problem with a maximization objective [10]. These two problems frame our formulation. Choosing a set of members to cover a weighted collection of skills is a weighted set covering problem, and requiring those members to share enough free time adds a side constraint on top of the selection. The next section makes this link precise and states the optimization problem.

III. Problem Formulation

A. Basic Notation

Let $P = \{1, \ldots, n\}$ be the pool of candidate participants, and let S be the set of skills the event needs. Each skill $s \in S$ has a weight $w_s > 0$ that reflects how important or how scarce it is, so rarer and more critical skills count for more. Each participant $i \in P$ is described by a skill set $S_i \subseteq S$ and an availability set $A_i \subseteq U$, where U is a set of discrete time slots that divide the event window. For a 48-hour hackathon split into one-hour slots, $|U| = 48$. A team is a subset $T \subseteq P$. Its size $|T|$ is not fixed in advance. The organizer allows any size in the small range $\{3, 4, 5\}$, which covers the team-size rules common across hackathons.

B. Availability Representation

Each participant's schedule is the set A_i of slots in which they can work. The time a team shares is the intersection of its members' schedules,

$$\mathrm{Overlap}(T) = \left| \bigcap_{i \in T} A_i \right|. \tag{1}$$

This counts the slots in which every member is free at the same time, the only slots in which the whole team can work together. Intersecting more schedules can only remove slots, so $\mathrm{Overlap}$ never grows as the team grows: adding a member never raises the shared time and usually lowers it. This property, that larger teams tend to have less common availability, drives much of the pruning in Section IV.

C. Constraints

A team T is feasible when it meets three conditions. The first is size: $|T| \in \{3, 4, 5\}$. The second is availability, where the shared time must reach a minimum threshold,

$$\mathrm{Overlap}(T) \geq h, \tag{2}$$

where h is the smallest number of common slots the organizer treats as workable. A team below h is infeasible no matter how good its skills are. The third condition is skill contribution: every member must supply at least one skill the event needs,

$$S_i \cap S \neq \emptyset \quad \text{for all } i \in T. \tag{3}$$

This rules out a member with no relevant skill, who would only fill a seat without adding any useful ability. It does not stop two members from sharing a skill. That milder kind of redundancy is left to the objective in Section III-D, which penalizes members who add no new coverage. Conditions (2) and (3) are independent, so a team can satisfy one and break the other.

D. Objective Function

The skill coverage of a team is the total weight of the distinct skills its members supply,

$$\mathrm{Cov}(T) = \sum_{s \in S} w_s \mathbb{1}\left[s \in \bigcup_{i \in T} S_i \right], \tag{4}$$

where $\mathbb{1}[\cdot]$ is the indicator function. A skill counts its weight once, no matter how many members hold it, so a duplicate skill adds nothing. This is the diminishing-returns behavior of weighted set cover. We score a team by its coverage, minus a fixed charge for each member, plus a reward for shared availability,

$$\mathrm{Score}(T) = \mathrm{Cov}(T) - c\,|T| + \lambda\,\mathrm{Overlap}(T), \tag{5}$$

where $c \geq 0$ is the cost of adding one more member and $\lambda \geq 0$ sets the value of availability relative to skill coverage.

The size penalty is deliberate. Without a charge per member, coverage can only grow as members are added. A larger team would then never score worse on the skill term, and the search would drift toward a five-member team automatically rather than because the larger team is actually better. The term $-c\,|T|$ puts a price on size. A new member earns their seat only when the new coverage they bring is worth more than c. A small team whose members each hold scarce skills can then beat a larger team whose skills overlap. This makes the choice among the sizes $\{3, 4, 5\}$ a real trade-off rather than a fixed outcome. $\mathrm{Overlap}$ is left out of the penalty because it already shrinks as the team grows through (1), and charging for size twice would hide the signal the model is meant to show. The penalty also gives the minimum viable team idea from Section I a precise form. A member who only repeats skills the team already has leaves $\mathrm{Cov}(T)$ unchanged but raises $|T|$ by one, so the score drops by exactly c, and the search has a reason to leave that seat empty.

E. Formal Optimization Problem

The task is to select the feasible team of highest score,

$$\begin{aligned} \max_{T \subseteq P} \quad & \mathrm{Score}(T) \\ \text{subject to} \quad & |T| \in \{3, 4, 5\}, \\ & \mathrm{Overlap}(T) \geq h, \\ & S_i \cap S \neq \emptyset, \quad \forall i \in T. \end{aligned} \tag{6}$$

Letting the size vary makes the feasible region the union of the three fixed-size problems for sizes 3, 4, and 5, which is larger than any single one. The skill-contribution constraint then trims this region separately from availability by removing any team that includes a member with no relevant skill.

Even without the availability and contribution constraints, the core of (6) is hard. Deciding whether some k members cover a target set of skills is the set covering decision problem, which is NP-complete [11]. The weighted maximization form belongs to the weighted set cover family, for which the classical greedy heuristic gives a logarithmic-factor guarantee [12]. The availability constraint and the per-member objective only add to this already NP-hard core, so a fast exact algorithm for the general case is unlikely. This is why we use branch and bound: it keeps the guarantee of optimality while pruning the large search space in practice.

F. Illustrative Example

Table I shows a small instance; take the per-member penalty to be $c = 2$. Consider the three-member team $\{P_1, P_2, P_3\}$. It covers Dev, Design, and Business for a coverage of $2 + 2 + 1 = 5$, and its members are free together in slots $\{2, 3, 4\}$, so $\mathrm{Overlap} = 3$ and its score is $5 - 2(3) + \lambda(3) = 3\lambda - 1$. Adding P_4 brings the rare Data skill: the four-member team $\{P_1, P_2, P_3, P_4\}$ raises coverage to 8 but the extra member shrinks the shared time to slots $\{3, 4\}$, so $\mathrm{Overlap} = 2$ and the score is $8 - 2(4) + \lambda(2) = 2\lambda$. Which team wins depends on the parameters. The four-member team scores higher exactly when $2\lambda > 3\lambda - 1$, that is when availability is cheap ($\lambda < 1$) so the valuable Data skill is worth the extra seat; when availability is dear, or when the threshold is strict enough that $h = 3$ already makes the four-member team infeasible, the three-member team is preferred. The team $\{P_1, P_2, P_3, P_5\}$

TABLE I
A SMALL INSTANCE WITH FIVE CANDIDATES AND FOUR SKILLS, WITH WEIGHTS $w_{\text{Dev}} = 2$, $w_{\text{Design}} = 2$, $w_{\text{Business}} = 1$, AND $w_{\text{Data}} = 3$. SLOTS ARE NUMBERED 1 TO 6.

Participant	Skills	Available slots
P_1	Dev	1, 2, 3, 4, 5
P_2	Design	2, 3, 4, 5, 6
P_3	Business	1, 2, 3, 4
P_4	Data	3, 4, 5
P_5	Dev, Design	1, 2, 3, 4

shows the milder kind of redundancy. P_5 holds only Dev and Design, both already supplied by P_1 and P_2, so P_5 adds no new coverage: the team stays feasible, coverage is unchanged at 5, but the size grows to four and the score drops to $5 - 2(4) + 3\lambda = 3\lambda - 3$, exactly $c = 2$ below the three-member team, so the optimizer has no reason to prefer it. The skill-contribution constraint (3) removes only a participant with no relevant skill at all; this softer redundancy is left to the size penalty in the objective.

IV. BRANCH AND BOUND METHODOLOGY

This section develops a branch-and-bound search for problem (6). The method builds teams by deciding, one candidate at a time, whether to include each participant. It throws away parts of the search that cannot hold a better team than the one already found. Two pruning rules do most of the work: one based on the availability constraint and one based on an upper bound on the score.

A. State Space

We fix an order on the candidates and build a binary tree of include-or-exclude decisions over them. A node is a partial team T, the set of candidates chosen so far, together with the index of the next candidate to decide. The root is the empty team. Going down one level either adds the next candidate to T or skips it. Any node whose partial team has size 3, 4, or 5 is itself a complete team and is scored right away. The search then continues to check whether adding more members gives something better, until the size reaches five or the candidates run out. Variable team size therefore needs no special handling. The sizes 3, 4, and 5 are simply the depths at which a node also counts as a usable solution.

Fig. 2 shows the general shape of such a search. Branch and bound explores the tree one branch at a time in depth-first order, and it skips whole subtrees that cannot beat the best team found so far. Our two pruning rules, given below, are what decide which subtrees to skip.

B. Branching Strategy

The order in which candidates are decided affects how early a strong team is found, and finding a strong team early is what makes the bound prune well. We sort candidates by decreasing total skill weight $\sum_{s \in S_i} w_s$, so the members who can add the most coverage are decided first and a high-scoring team tends to appear near the top of the tree. Before the search starts, we remove any candidate i with $S_i \cap S = \emptyset$. Such a participant can never satisfy the skill-contribution constraint (3), so keeping them would only widen the tree. This early elimination is separate from the two pruning rules below, since it shrinks the candidate set itself.

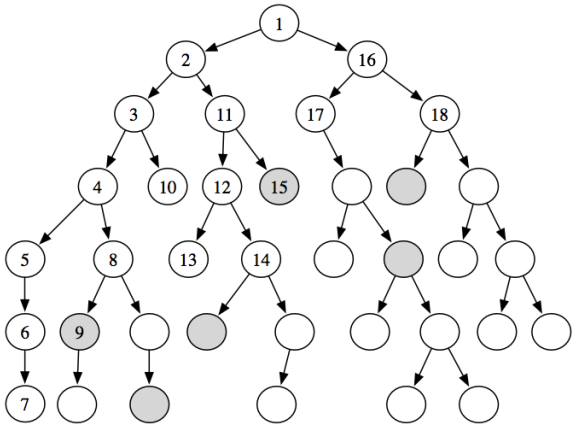

Fig. 2. General shape of a depth-first branch-and-bound search over a tree. Source: Poole and Mackworth, *Artificial Intelligence: Foundations of Computational Agents* (artint.info).

C. Constraint-Based Pruning

Recall from (1) that shared availability cannot increase when a member is added. So the availability of a partial team is an upper bound on the availability of any team that extends it. If a partial team already falls below the threshold,

$$\text{Overlap}(T) < h, \tag{7}$$

then no completion of T can satisfy (2), so the whole subtree below the node is infeasible and can be cut. Checking this at every node, instead of only at full teams, removes whole groups of teams with one test. The effect is uneven across sizes, and this is on purpose. Because overlap shrinks as teams grow, a five-member branch drops below h sooner than a three-member branch, so the larger teams, which are also the most expensive to enumerate, are cut most often.

D. Bound-Based Pruning

The second rule computes the best score any completion of a node can reach and compares it with the best feasible score found so far. Let T be the partial team at a node, let $m = |T|$, and let R be the set of candidates still undecided. A completion adds some members of R to T and ends at a size $|T'| \in \{3, 4, 5\}$.

Two optimistic estimates give the bound. Coverage can grow by at most the weight of the still-uncovered skills that some remaining candidate can supply,

$$\overline{\text{Cov}} = \text{Cov}(T) + \sum_{\substack{s \notin \cup_{i \in T} S_i \\ s \in \cup_{j \in R} S_j}} w_s. \tag{8}$$

Availability can only fall, so $\text{Overlap}(T') \le \text{Overlap}(T)$. The score of a completion is $\text{Cov}(T') - c\,|T'| + \lambda\,\text{Overlap}(T')$, which grows with coverage and falls with both size and lost

Algorithm 1 Branch and bound for hackathon team composition.

Remove every candidate i with $S_i \cap S = \emptyset$
Sort the remaining n candidates by descending $\sum_{s \in S_i} w_s$

best $\leftarrow -\infty$; $\quad T^\star \leftarrow \emptyset$
EXPLORE($\emptyset$, 1)
return $T^\star$
procedure EXPLORE(T, idx)
if $\mathrm{Overlap}(T) < h$ **then**
  return {availability pruning}
end if
if $3 \le |T| \le 5$ **and** $\mathrm{Score}(T) >$ *best* **then**
  best $\leftarrow \mathrm{Score}(T)$; $\quad T^\star \leftarrow T$
end if
if $|T| = 5$ **or** $idx > n$ **then**
  return
end if
if $\mathrm{UB}(T, \{idx, \ldots, n\}) \le$ *best* **then**
  return {bound pruning}
end if
EXPLORE($T \cup \{idx\}$, $idx + 1$) {include candidate idx}
EXPLORE(T, $idx + 1$) {exclude candidate idx}

availability. The penalty $-c\,|T'|$ is smallest at the smallest allowed size, so writing $\ell = \max(3, m)$ for that size, a valid upper bound is

$$\mathrm{UB}(T, R) = \overline{\mathrm{Cov}} - c\,\ell + \lambda\,\mathrm{Overlap}(T). \tag{9}$$

If $\mathrm{UB}(T, R) \le$ *best*, no completion can beat the best team so far, and the subtree is pruned. The bound is optimistic because the largest coverage $\overline{\mathrm{Cov}}$, the smallest size penalty $c\,\ell$, and the largest availability may each come from a different completion, so the three terms need not happen together in any single team. Still, the bound never underestimates the true best completion, which is all that pruning needs.

E. Algorithm

Algorithm 1 states the procedure. It follows the standard recursive template for branch and bound [9], with the two rules above and the rule that any node whose team has three to five members is accepted as a candidate solution.

F. Complexity Analysis

Because the team size is capped at five, the search visits only partial teams of at most five members. So the number of nodes is $O(n^5)$, and each node costs $O(n + |U| + |S|)$ to score and bound. For this fixed size range the problem is therefore polynomial in n, and brute-force enumeration of the $\binom{n}{3} + \binom{n}{4} + \binom{n}{5}$ teams is possible in principle. This cap comes from the hackathon setting, not from the general formulation, which is NP-hard once the team size can grow with the input (Section III-E).

Branch and bound still matters for two reasons, even with the size cap. First, the $O(n^5)$ count has a large constant once n reaches the tens or hundreds, and the two pruning rules cut it sharply in practice rather than only in the worst case. Pruning works best when the availability threshold h is tight, which triggers the constraint rule early and especially on larger teams, and when skill weights vary widely, which lets the bound separate strong nodes from weak ones. It helps least when h is loose and weights are flat, so that most teams are feasible and look alike. Second, the same search extends to the general formulation with no size cap, where brute force becomes impractical and the bound is what keeps the method usable. Section VI measures how many nodes each rule removes in practice.

TABLE II
SYNTHETIC SCENARIO FAMILIES. $|S|$ IS THE NUMBER OF RELEVANT SKILLS, "SCARCE" THE NUMBER OF RARE HIGH-WEIGHT SKILLS, AND DENSITY THE PER-PARTICIPANT FRACTION OF FREE TIME SLOTS.

Family	$\lvert S\rvert$	scarce	avail. density	binding factor
balanced	9	0	0.40–0.70	none (reference)
skill-scarce	12	3	0.55–0.90	skill coverage
avail-tight	8	0	0.15–0.35	availability
abundant	5	0	0.60–0.90	none (dense)
adversarial	12	3	0.30–0.70	coverage vs avail.
irrelevant	9	0	0.40–0.70	skill-less members

V. EXPERIMENTAL SETUP

A. Dataset

No public dataset pairs hackathon participants' skills with their hour-by-hour availability, so we generate synthetic instances from a seeded random model. Every instance can be reproduced from a family name, a pool size, and a seed. An instance fixes a pool of n participants, a set of weighted skills, and an availability window of $|U| = 48$ one-hour slots for a two-day event. Each participant draws a skill set and a free-or-busy schedule over the slots. Instead of varying only the pool size, we define six scenario families, each of which makes a different factor the binding one, so the search runs under genuinely different conditions rather than scaled copies of one setting. Table II lists them. The *skill-scarce* family gives the pool more skills than three members can cover and makes a few rare skills carry high weight, so coverage is what limits the score. The *availability-tight* family makes schedules sparse, so the overlap threshold is what teams struggle to meet. The *abundant* family makes schedules dense and weights flat, so almost every team is feasible and teams look alike. The *adversarial* family places the rare high-weight skills in the participants with the sparsest schedules, so coverage and availability pull against each other. The *with-irrelevant* family adds skill-less participants to test the early-elimination step, and *balanced* is a neutral reference. We use pool sizes $n \in \{12, 16, 20, 24, 30, 40\}$ with three independent seeds per configuration.

B. Baselines

We compare branch and bound against two baselines on the same instances. The first is exhaustive brute force, which lists every team of size three, four, and five, keeps the best

feasible one, and so returns a certified optimum. We run it on the instances with $n \leq 24$, where listing all teams is affordable, and use it to check that branch and bound finds the same optimum. The second is a constraint-aware greedy heuristic. It grows a team one member at a time. At each step it looks only at candidates that keep the shared availability at or above h, and among them it adds the one with the largest gain in weighted coverage. It reports the best feasible team of size three to five that it passes through. Greedy is fast, but it commits to high-coverage members early, so it can miss the optimum or, on tightly scheduled instances, fail to find any feasible team at all.

C. Metrics

For each run we record the score and size of the team returned, whether the method found a feasible team, and the wall-clock time. For branch and bound we also log the number of nodes visited and how many subtrees each pruning rule removes, which lets us separate the work of the availability rule from that of the bound. From these we form three comparisons: whether the branch-and-bound optimum matches brute force, the speedup of branch and bound over brute force on the same instance, and the gap between the greedy score and the optimum. We report the gap only over instances where greedy is feasible, and we also report how often greedy fails to find any feasible team on instances that are in fact solvable.

D. Parameter Sensitivity

Three parameters control the objective and the feasibility test, and we sweep each over two settings to see how the search responds: the availability threshold $h \in \{4, 8\}$ slots, the availability weight $\lambda \in \{0, 0.1\}$, and the per-member penalty $c \in \{1, 2\}$. The full experiment runs every solver on every combination of family, pool size, seed, and these three parameters. All instances and the three solvers are written in Python and run on an ordinary laptop. The program is kept separate from this paper, and only its aggregate output is reported here.

VI. Results and Analysis

We organize the results around the three comparisons from Section V-C: optimality and speed against brute force, where the two pruning rules do their work, and quality against greedy. Table III collects the per-family numbers used throughout.

A. Branch and Bound versus Brute Force

Across all 576 instances small enough to enumerate fully, the team returned by branch and bound matched the brute-force optimum exactly, so the pruning never dropped an optimal solution. Fig. 3 shows what that optimum costs to compute. Brute-force time rises steeply with the pool size, because the number of teams it checks grows as $\binom{n}{3} + \binom{n}{4} + \binom{n}{5}$. Branch-and-bound time grows much more slowly, because the two rules remove most of the tree before it is built. Depending on the family, the median speedup ranges from about 35 to 180 times (Table III). The mean over the enumerable instances

TABLE III
Per-family results. Columns: median branch-and-bound nodes; share of nodes pruned by the constraint and bound rules; brute-force-to-BnB speedup ($n \leq 24$); mean greedy optimality gap where greedy is feasible; and greedy failure rate on solvable instances.

Family	nodes	constr.%	bound%	speedup	gap%	g-fail%
balanced	356	9.4	39.8	65	0.92	0.0
skill-scarce	1805	3.1	37.9	35	1.00	0.0
avail-tight	492	44.2	3.4	47	8.24	55.6
abundant	47	0.0	43.8	178	0.35	0.0
adversarial	289	39.1	10.5	66	2.75	19.4
irrelevant	177	14.6	34.8	55	0.55	0.0

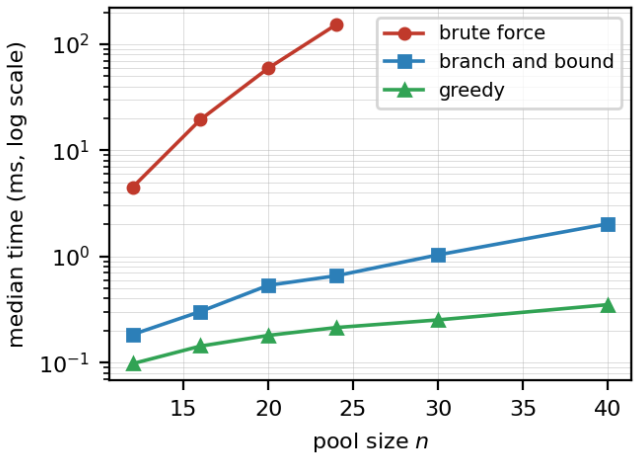

Fig. 3. Median running time against pool size n, on a logarithmic scale.

is about 230 times, pulled up by the largest pools where brute force is slowest, and on the best instances it passes three thousand. Greedy is faster still, but Fig. 3 shows only its running time, not its quality, which Section VI-D covers.

B. Where Pruning Does Its Work

The two rules are not interchangeable, and the family that triggers one barely triggers the other. Fig. 4 splits the search nodes into those cut by the availability rule, those cut by the bound, and those fully explored. In the availability-tight family, where schedules are sparse, the constraint rule removes about 44% of nodes while the bound removes only a few percent, because teams drop below h early and feasibility does the cutting. In the abundant family the picture flips: the bound removes about 44% of nodes and the constraint rule none, because dense schedules keep every partial team feasible and only the coverage bound can tell strong nodes from weak ones. The adversarial family sits between the two, with both rules active at about 39% and 10%, which is the sign of a setting where coverage and availability genuinely compete. This setting is also the most expensive to search: skill-scarce instances, whose large skill set forces the search to weigh many member combinations, visit several thousand nodes on average against a few hundred elsewhere.

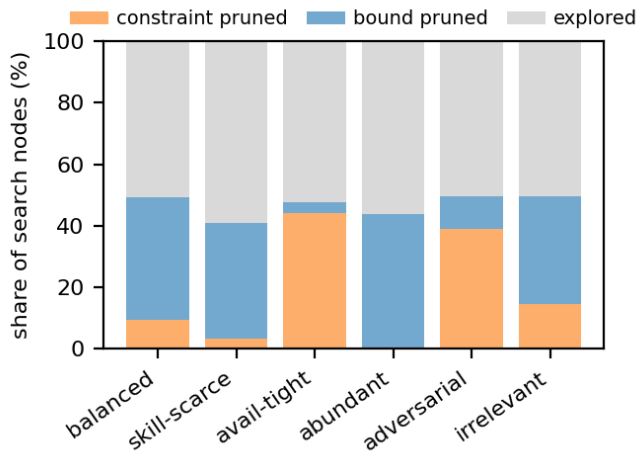

Fig. 4. Share of branch-and-bound search nodes removed by each pruning rule, by scenario family.

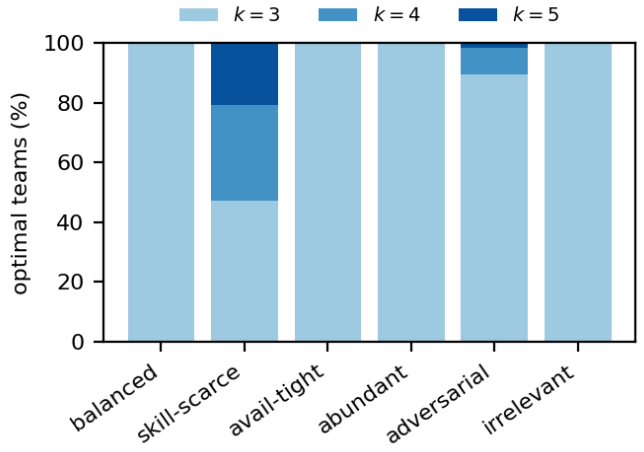

Fig. 5. Distribution of the optimal team size by scenario family.

C. Team Size as a Decision

Because the objective charges c per member, the optimal team size is a result of the search rather than a fixed input, and Fig. 5 shows that it changes only when coverage is the binding factor. In the skill-scarce family, where the skill set is larger than three members can cover, the optimum is a four- or five-member team in more than half of the instances. In the adversarial family it sometimes grows past three. Everywhere else the optimum is a three-member team. In the availability-tight family a fourth member would push the shared time below h, and in the abundant family a fourth member adds no skill the team is missing, so neither is worth the c it would cost. The size penalty thus behaves as intended: it grows the team only when extra coverage is worth more than the seat, and keeps it at the minimum otherwise.

D. Branch and Bound versus Greedy

Greedy's main weakness is not the size of its optimality gap. When it does return a feasible team, its score is within about one percent of the optimum in the unconstrained families and within about eight percent in the availability-limited ones, as the gap column of Table III shows. The bigger problem is feasibility. On availability-tight instances, greedy fails to find any feasible team on 56% of the cases that actually have one, and on adversarial instances on 19%, because it spends its early picks on high-coverage members and only later finds that no remaining candidate keeps the shared time above h. Branch and bound does not have this problem, because it can back out of a coverage-rich but unschedulable partial team. The difference is largest in exactly the availability-limited setting that led us to treat overlap as a hard constraint in the first place.

E. Sensitivity Analysis

The parameter sweep confirms that the search responds to its inputs as designed. Raising the availability threshold from $h = 4$ to $h = 8$ leaves fewer instances solvable, from 428 to 364 of 432, as more teams fall below the stricter overlap requirement. Raising the per-member penalty from $c = 1$ to $c = 2$ shifts the optimum toward smaller teams: the share of three-member optima rises from 84% to 93%, since a more expensive seat is harder for a marginal member to justify. Raising λ has a milder effect in the same direction, since rewarding availability favors the smaller teams that share more time. The availability weight also makes the search larger: with $\lambda = 0.1$ the bound includes the overlap term and prunes a little less, raising the average node count from about 1,200 to about 1,970. None of these changes affects the optimality guarantee. They change only how much of the tree survives pruning and where the optimal size settles.

VII. Discussion

The experiments support a simple reading of where an exact method earns its place in this problem. Branch and bound returned a certified optimum on every instance, and it did so by removing most of the search tree instead of walking through it. So the optimality guarantee came at a cost close to that of the heuristics on the instance sizes a hackathon organizer actually faces. The two pruning rules did not contribute equally. Instead, each one took over in the setting that suited it. When schedules were sparse, the availability rule did most of the pruning. When schedules were dense but skills were unevenly distributed, the score bound did. This split is convenient in practice, because the organizer does not have to know in advance which factor will bind. The same search adapts to the instance, and the share of nodes each rule removes is itself an easy way to see what is limiting team quality.

The comparison against greedy points to the same conclusion from the other side. When greedy returned a team, its score was usually within a couple of percent of the optimum, so on easy instances the heuristic is a reasonable choice. Its

real weakness showed up only once availability was treated as a hard constraint. On the availability-tight family, greedy failed to build any feasible team on more than half of the instances that did have one, because it spent its early picks maximizing coverage and then could not keep the shared time above the threshold. Branch and bound never had that failure, because it can abandon a coverage-rich but unschedulable partial team and try another. The practical point is that the exact method matters most in exactly the setting that motivated the formulation, one where members must share working hours and a team that cannot meet is no team at all.

The model also makes team size an output rather than an input. Because the objective charges a fixed cost c per member, the search grew a team beyond three only when the extra coverage was worth more than the seat, which happened in the coverage-limited families and almost nowhere else. This gives the three parameters a clear meaning. The threshold h sets how much shared time the organizer requires, λ sets how much shared time is rewarded beyond that floor, and c sets how reluctant the search is to add another person. None of them needs tuning against ground truth. They state policy choices the organizer can set directly.

Several limitations bound these claims. The team size is capped at five, which makes the problem polynomial in the pool size, so the experiments measure a large constant-factor speedup rather than a change in complexity class. The same search is meant to extend to the general formulation, where the size can grow and the bound becomes essential, but that case is not measured here. The instances are synthetic, since no public dataset pairs participant skills with hour-by-hour availability, so the scenario families are designed to span many conditions rather than to copy a specific event. The formulation also selects one optimal team rather than splitting a whole pool into several teams at once, and it treats skill weights and availability as given and fixed. Each of these is a natural direction for future work rather than a flaw in the present results.

VIII. Conclusion

This paper formulated hackathon team composition as a combinatorial optimization problem that weighs weighted skill coverage against shared availability over a team of variable size, with availability acting as both a hard feasibility constraint and a scoring term. It developed a branch-and-bound method for the problem. The method combines early removal of skill-less candidates, a weight-ordered branching scheme that decides team size during the search, a constraint rule that cuts subtrees once the shared time falls below the threshold, and a coverage-based upper bound that discards subtrees that cannot beat the best team so far. Across six scenario families and a sweep of the model parameters, the method matched an exhaustive baseline on every comparable instance while exploring only a small fraction of the teams, and it avoided the feasibility failures that a constraint-aware greedy heuristic had on availability-limited instances. The results also showed that the optimal team size is a real decision under the size penalty, growing only when scarce skills make an extra member worth the cost. Future work includes lifting the size cap to the general formulation, where the bound is essential, splitting a whole pool into several teams at once, and validating the model on data from real events.

Appendix

A video that explains this paper and the source code used for the experiments are available online.

A. Explanation Video

YouTube link: https://youtu.be/KuArdNIhSZ8

B. GitHub Repository

https://github.com/jsndwrd/bnb-hacks-optimization

Acknowledgment

The author sincerely thanks God Almighty for providing the strength and opportunity to complete this paper successfully. The author also extends his deep appreciation and gratitude to Ir. Rila Mandala, M.Eng., Ph.D. of K02 IF2211 Strategi Algoritma, whose semester-long support enabled the smooth completion of this paper. The author would also like to extend his personal appreciation to his hackathon team, Shibainu, and to his family, whose support and companionship uplifted the author during the preparation of this paper.

References

[1] K. Oyetade, T. Zuva, and A. Harmse, "Evaluation of the impact of hackathons in education," *Cogent Education*, vol. 11, no. 1, art. 2392420, 2024.

[2] E. P. P. Pe-Than, A. Nolte, A. Filippova, C. Bird, S. Scallen, and J. Herbsleb, "Corporate hackathons, how and why? A multiple case study of motivation, projects proposal and selection, goal setting, coordination, and outcomes," *Human–Computer Interaction*, vol. 37, no. 4, pp. 281–313, 2022.

[3] W. Mendes, A. Richard, T.-K. Tillo, G. Pinto, K. Gama, and A. Nolte, "Socio-technical constraints and affordances of virtual collaboration: A study of four online hackathons," *Proc. ACM Hum.-Comput. Interact.*, vol. 6, no. CSCW2, art. 330, pp. 1–32, Nov. 2022.

[4] D. Hevenstone, N. Endrissat, O. Lavrovsky, and O. Hümbelin, "Designing for diversity: team formation and participatory policy design," *Policy Design and Practice*, vol. 9, no. 2, pp. 232–243, 2026.

[5] D. Gómez-Zará, A. Das, B. Pawlow, and N. Contractor, "In search of diverse and connected teams: A computational approach to assemble diverse teams based on members' social networks," *PLoS ONE*, vol. 17, no. 11, art. e0276061, 2022.

[6] Y. Guo *et al.*, "A reinforcement learning-assisted genetic programming algorithm for team formation problem considering person-job matching," *Neurocomputing*, vol. 650, art. 130917, 2025.

[7] S. N. Parragh and F. Tricoire, "Branch-and-bound for bi-objective integer programming," *INFORMS Journal on Computing*, vol. 31, no. 4, pp. 805–822, 2019.

[8] N. B. Moe, R. Ulfsnes, V. Stray, and D. Šmite, "Improving productivity through corporate hackathons: A multiple case study of two large-scale agile organizations," in *Proc. 55th Hawaii Int. Conf. System Sciences (HICSS)*, 2022, pp. 7310–7319.

[9] A. Levitin, *Introduction to the Design and Analysis of Algorithms*. Boston, MA, USA: Addison-Wesley, 2003.

[10] J. K. Shinzeh and H. A. Lyeme, "An effective branch and bound method to find the optimal solution of an assignment problem with maximization objective function," *Mathematics Open*, vol. 4, art. 2550004, 2025.

[11] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of Computer Computations*, R. E. Miller and J. W. Thatcher, Eds. New York, NY, USA: Plenum Press, 1972, pp. 85–103.

[12] V. Chvátal, "A greedy heuristic for the set-covering problem," *Mathematics of Operations Research*, vol. 4, no. 3, pp. 233–235, 1979.

STATEMENT OF ORIGINALITY

I hereby declare that this paper is my own writing, not an adaptation or a translation of another person's paper, and not an act of plagiarism.

Bandung, June 19th, 2026

Jason Edward Salim - 13524034