Searching Algorithms Comparison: A* and HillClimbing

Dragomir Andrei

April 30, 2024

Contents

1 Introduction

The task of creating optimal timetables for educational institutions is a complex challenge within the field of operations research and artificial intelligence. This problem, known as the "timetable generator problem," requires assigning classes, professors, and rooms to specific timeslots while adhering to various constraints..

The constraints include both hard constraints, such as avoiding time overlaps for professors or rooms, and soft constraints, such as accommodating professors' day preferences and time slots. The inherent complexity of this problem makes it well-suited for solution using heuristic and metaheuristic algorithms, which can find practical solutions in reasonable amounts of time for problems that are otherwise NP-hard.

This paper presents a comprehensive analysis and comparison of two prominent approaches to the timetable generator problem: the Hill Climbing and A* algorithms. Hill Climbing is a local search algorithm that continuously moves towards higher value states, based on a defined fitness function, until no improvements can be made. It is simple and efficient but may easily become trapped in local optima, making it less effective for more complex or constraint-dense environments.

On the other hand, A* is a best-first search algorithm that uses an informed heuristic to estimate the cost from the current state to the goal, providing a balance between path costs and heuristic guidance to explore the most promising paths efficiently.

The effectiveness of these algorithms hinges not only on their general design but also on their implementation details such as choice of heuristic, handling of constraints, and problem-specific adaptations. The following document aims to detail these implementations, providing a direct comparison in terms of efficiency, effectiveness, and the quality of solutions generated.

2 Constraints

The timetable generation process is guided by a set of constraints, which are categorized into hard and soft constraints. Each type of constraint plays a critical role in shaping feasible solutions. These constraints are designed to ensure that the timetable is not only theoretically valid but also practically viable and acceptable to the end user, whether it be an individual or an institution.

2.1 Hard Constraints

Hard constraints are non-negotiable and must be strictly adhered to, as their violation renders a timetable operationally unfeasible. These include:

- Each classroom can host only one subject taught by one instructor during any given time slot.
- An instructor can teach only one subject in one classroom during any given time slot.
- An instructor is limited to teaching in no more than seven time slots per week.

- The number of students in any classroom during a time slot must not exceed its maximum seating capacity.
- All students enrolled in a subject must be scheduled to attend the subject; specifically, the combined capacity of all classrooms during the time slots allocated to the subject must be at least equal to the number of students enrolled.
- Instructors can only teach subjects for which they are specialized.
- Classrooms are assigned only to the subjects for which they are designated.

2.2 Soft Constraints

Soft constraints reflect the preferences of instructors and are flexible to some extent. While it is preferable to meet these where possible, violating soft constraints is permissible if it leads to a viable timetable:

- Instructors may have preferences for or against teaching on specific days. For example, an instructor might prefer to teach on Fridays but not on Tuesdays.
- Instructors may prefer or avoid certain time slots throughout the week. For instance, they might prefer teaching between 12-14 but not between 16-20.
- **[Bonus]** Instructors prefer not to have breaks in their schedule longer than a specified duration, such as no breaks longer than 0 hours or no longer than 2 hours, to avoid having large unoccupied gaps in their daily schedule.

These constraints are essential for developing a functional and efficient timetable that accommodates the needs and preferences of all parties involved. By understanding and carefully managing these constraints, it is possible to optimize the timetable to achieve a balance between operational feasibility and personal preferences.

3 State Representation

The success and efficiency of a search algorithm depend critically on how states are represented. In the context of the timetable generation problem, a state includes the current configuration of all scheduled classes, as well as details like assigned instructors, classrooms, and timeslots. This representation forms the basis for evaluating the feasibility and optimality of a timetable as the algorithm navigates through its search space.

3.1 Defining a State

A state in the timetable problem is defined as a configuration of assignments where each assignment is a tuple consisting of a day, a timeslot, a classroom, a professor, and a subject. This structure allows the algorithm to easily check constraints and make decisions about where and how to place future assignments.

3.1.1 Structure

Each state is represented by an instance of the `TimetableNode` class, which contains:

- **Days:** A dictionary where keys are days of the week and values are dictionaries of timeslots assigned on that day.
- **Professors:** A dictionary mapping professors to the number of their current assignments, ensuring easy constraint checks against maximum allowable teaching loads.
- **Students per activity:** A dictionary tracking the number of students yet to be assigned to each subject, aiding in priority decisions based on student needs.

3.1.2 Code Example

Below is a simplified example of how a state might be initialized and represented within our system:

```
# Initialize state with predefined constraints and data
constraints_manager = ConstraintManager(constraints_dict, total_students)
initial_state = TimetableNode(
    constraints_manager,
    students_per_activity={"SO": 120, "PCOM": 100},
    days={
        "Monday": {"8-10": {}, "10-12": {}},
        "Tuesday": {"8-10": {}, "10-12": {}}
    },
    professors={"Ion Pop": 0, "Maria Ionescu": 0}
)

# Example of adding an assignment to the state
initial_state.days["Monday"]["8-10"] = {"Room EG301": ("Ion Pop", "SO")}
```

The additional two fields of the class, chosen_assignment and profs_assignments, are constructed during the algorithms development and are used for optimisation and bonus constraint managing.

This example demonstrates the creation of a `TimetableNode`, with a focus on how days, timeslots, and assignments are structured. Each assignment within the state is clear and allows for quick checks against hard and soft constraints, essential for the functionality of both Hill Climbing and A* algorithms in exploring solution spaces efficiently.

3.2 State Generation

State generation is a crucial operation in search algorithms as it determines the possible future moves from the current state. In the context of timetable generation, each state transition involves evaluating potential placements of courses into specific timeslots, classrooms, and assigning instructors while adhering to the established constraints.

3.2.1 Generation Mechanism

The method `get_next_states()` is responsible for generating all feasible successors of the current node. It iterates over every unassigned slot for each day and applies constraints to filter out infeasible assignments. To enhance the search's diversity and escape potential local minima, a random element is introduced, allowing for the occasional random selection of the next state rather than strictly following a deterministic approach.

Randomness for Diversity Incorporating randomness helps in exploring the search space more broadly and prevents the algorithm from getting stuck by occasionally making unexpected choices. A 10% chance is used to randomly select from the possible states, which introduces variability in the search path and can lead to discovering more optimal solutions that might otherwise be overlooked.

3.2.2 Code Example

Here is a simplified code snippet that demonstrates the mechanism of state generation:

```
def get_next_states(self):
    next_states = []
    for day, intervals in self.days.items():
        for interval_tuple, assignments in intervals.items():
            for place, assignment in assignments.items():
                if assignment is None:
                    possible_states = self.apply_on_possible_states(
                        day, interval_tuple, place
                    )
                    if random.random() < 0.1 and possible_states:
                        random_choice = random.choice(possible_states)
                        next_states.append(random_choice)
                    else:
                        next_states.extend(possible_states)
    return next_states
```

3.2.3 Constraint Application

To ensure that generated states are viable, the `apply_constraints_on_possible_states()` method filters potential states based on various criteria such as classroom capacity, instructor's availability, and compliance with other hard constraints.

Constraints Checking Before a state is accepted, it must pass through a rigorous constraint check, ensuring that no hard rules are violated and the academic and logistical requirements are met.

```
def apply_constraints_on_possible_states(self, day, interval_t, place):
    possible_states = []
    sorted_activities = sorted(
        self.constraints_manager.constraints[SALI][place][MATERII],
        key=lambda act: -self.students_per_activity[act]
    )
    for activity in sorted_activities:
        if self.students_per_activity[activity] > 0:
            for prof, constraints in self.const_mng.constraints[PROFESORI].items():
                if self.check(constraints, day, interval_t, activity, prof):
                    new_spot = self.choose(day, interval_t, place, prof, activity)
                    possible_states.append(new_spot)
    return possible_states
```

This methodology ensures each state transition is both deliberate and exploratory, balancing between following a guided path and allowing for innovative leaps that might yield better overall results.

3.3 Constraint Representation

As previously seen, the constraints are managed by a constraint manager structure, which also has basic methods that are used in constraint application: number of accepted activities per room, number of rooms accepting an activity and number of professors qualified for an activity. An optimisation used here is that the methods output is cached, since they operate through the same set of constraints in any part of the algorithm. Decorator @lru_cache(maxsize=None) was used to accomplish that.

3.4 Constraint managing

The hard constraints are managed by exclusion, meaning the algorithms only generate the next states that respect these constraints. The soft constraints are considered in the evaluation of the node or the heuristic used, meaning they add a cost in the decision making of these two algorithms. In that way, we can guide the search to a solution with 0 soft constraints without imposing them as hard. This way we make the search a lot more explorative and give it better chances to find the solution we want.

4 Optimisations

4.1 General

In both algorithms, the main optimization involves assigning a field named "chosen assignment" to each node. This optimization aims to improve space and time complexity. Every explored node will have a new assignment compared to its parent, but only the best node chosen in both algorithms as the next node will have it applied in memory. This approach helps conserve resources for nodes that will not be explored immediately.

4.2 HillClimbing Optimisations

One effective enhancement to the basic hill climbing algorithm is the integration of the **random restarts** strategy. This method addresses the algorithm's propensity to get stuck in local optima by periodically restarting the search from different, randomly generated initial states. This allows for multiple opportunities to explore various regions of the search space.

4.2.1 Random Restarts in Hill Climbing

The Random Restart Hill Climbing method enhances traditional hill climbing by initiating multiple searches from diverse initial states, thereby increasing the likelihood of finding a global maximum.

The class `RandomRestartHillClimbing` is designed to manage these restarts:

```
class RandomRestartHillClimbing:
    def __init__(self, max_restarts, max_iterations, initial_state):
        self.max_restarts = max_restarts
        self.max_iterations = max_iterations
        self.initial_state = initial_state.clone()
```

The constructor initializes the maximum number of restarts and iterations per restart, and clones the initial state to ensure each restart begins from an unaltered base state.

4.2.2 Restart Mechanism

The method `random_restart_hill_climbing()` manages the process of conducting multiple hill climbing searches:

```
def random_restart_hill_climbing(self):
    best_solution, best_evaluation = None, float("inf")
    for _ in range(self.max_restarts):
        current_state = self.initial_state.clone()
        iterations, solution = self.hill_climbing(current_state)
        current_evaluation = solution.eval_node()
        if current_evaluation < best_evaluation:
            best_evaluation = current_evaluation
            best_solution = solution
```

Each restart is conducted from a fresh state, and the best solution is updated whenever an improvement is found.

4.2.3 Seed Shuffling and Dynamic Exploration

To diversify the exploration of solutions, seed shuffling is used to randomize the order of neighbor exploration, as shown in the following implementation:

```
def hill_climbing(self, initial_state):
    iterations = 0
    current_state = initial_state
    while iterations < self.max_iterations:
```

```
        neighbors = current_state.get_next_states()
        if not neighbors:
            break
        if iterations == 1:
            seed = choice(self.seeds)
            random.seed(seed)
            random.shuffle(neighbors)
            print("First iteration with seed: ", seed)
        best_neighbor = min(neighbors, key=lambda x: x.eval_node())
        if best_neighbor.eval_node() >= current_state.eval_node():
            break
        current_state = best_neighbor
        current_state.apply_assignment_on_best_node()
    return iterations, current_state
```

This code snippet demonstrates how the first iteration uses a randomly chosen seed to shuffle the neighbors, promoting a varied initial exploration.

The seeds used and proven to produce good results: 42, 69, 420, 666, 1337, 9001, 80085, 8008135, 80081355, 800813555

4.2.4 Stopping Condition

A stopping condition is crucial to prevent unnecessary computations once the algorithm ceases to make progress:

```
if best_neighbor.eval_node() >= current_state.eval_node():
    break
```

This condition halts the algorithm if no better neighbor is found, indicating that a local maximum has been reached.

4.2.5 Conclusion

These enhancements to the Hill Climbing algorithm—namely, random restarts, seed shuffling, and intelligent stopping conditions—significantly improve its effectiveness. They allow the algorithm to avoid common pitfalls such as premature convergence and entrapment in local optima, thereby improving its capability to find more optimal solutions.

4.3 A* Optimisations

To adapt A* for complex scheduling tasks like timetable generation, several optimizations were necessary to handle the high dimensionality and constraint-heavy nature of the problem.

4.3.1 Handling Stagnation with Backtracking

One significant challenge in timetable generation is stagnation, where the algorithm repeatedly explores states without making progress towards a solution. To mitigate this, we introduce a backtracking mechanism that allows the algorithm to revert to the most promising state encountered so far when it detects prolonged stagnation.

Stagnation Detection and Response Stagnation is monitored by counting the number of consecutive iterations without improvement in the number of remaining students. If this count exceeds a preset threshold, the algorithm backtracks to the last best state, effectively escaping potential dead-ends.

```
stagnation_counter = 0
max_stagnation = 25
best_so_far = (float("inf"), None)

while open_set:
    explored_nodes += 1
    current_cost, current_node = heapq.heappop(open_set)
    if current_node.get_remaining_students() == 0:
        break
    if current_node.get_remaining_students() < best_so_far[0]:
        best_so_far = (current_node.get_remaining_students(), current_node)
        stagnation_counter = 0
    else:
        stagnation_counter += 1

    if stagnation_counter >= max_stagnation:
        current_node = best_so_far[1]
        stagnation_counter = 0
```

This snippet highlights how the algorithm resets the current node to the best state when stagnation is detected, providing a fresh perspective for further exploration.

4.3.2 Dynamic Exploration and Node Expansion

To ensure thorough exploration of the state space, each state expansion is guided by both the heuristic function and the actual path cost from the start node, balancing between explorative steps and goal-directed navigation.

Node Expansion with Checks Each neighbor of the current node is considered for expansion unless it has been previously explored. This avoids redundant calculations and ensures that all potential paths are considered.

```
# Expanding nodes
for neighbor in current_node.get_next_states():
    expanded_nodes += 1
    if not any(neighbor.__eq__(closed_node) for closed_node in closed_set):
        neighbor_clone = neighbor.clone()
        neighbor_clone.apply_assignment_on_best_node()
        heapq.heappush(open_set, (neighbor_clone.total_cost(), neighbor_clone))
```

The algorithm checks whether a neighbor has already been evaluated before pushing it onto the priority queue, preventing cycles and reducing computational overhead.

4.3.3 Conclusion

These optimizations to the A* search algorithm enhance its ability to efficiently and effectively solve the timetable generation problem. By incorporating techniques such as backtracking and dynamic node exploration, the algorithm can navigate complex constraint landscapes and find optimal or near-optimal solutions within reasonable time frames.

5 Test cases

Dummy A very small test case with few restrictions.

Timetable_small_exact A small exact result test case with more students to assign and more restrictions.

Timetable_medium_relaxed A medium test case with sufficient restrictions.

Timetable_big_relaxed A very big test case with a lot of students to assign and restrictions to take into consideration.

Timetable_constraints_violated A test case which theoretically can support a minimum cost of 1 as solution. Solutions with cost 0 do exist but are harder to find.

Timetable_bonus_exact A test case with bonus restrictions (restriction regarding pause preferences for professors) that is big and computationally heavy.

6 Comparison

6.1 Evaluation and heuristic

The Hill Climbing and A* algorithms employ distinct strategies for evaluating nodes, which significantly influence their behavior and effectiveness.

Hill Climbing

The Hill Climbing algorithm evaluates nodes by focusing on the number of remaining students and the violations of soft and pause constraints, using a dynamic weighting system:

- The penalty for students is $\text{student\_penalty} = (\text{remaining\_students}^2) \times 50$.
- The constraints penalty adjusts dynamically:
  - If the ratio of remaining students to total is less than $\frac{1}{5}$, the penalties for soft and pause violations increase significantly.
  - Otherwise, standard weights are applied.
- This leads to the total node evaluation:

$$\text{Total Evaluation} = \text{student\_penalty} + \text{constraint\_penalty}$$

This evaluation mechanism ensures that priority is dynamically adjusted, heavily penalizing unassigned students as the solution progresses, thus driving the algorithm towards a more complete assignment early in the process. After most assignments are done, the soft constraints and pause constraints become the priority.

A* Algorithm

The A* search algorithm uses a combination of heuristics (‘h‘) and cost function (‘g‘) for node evaluation:

- The heuristic function ‘h‘ considers:

  $$h = \text{remaining\_students} \times 500 + \text{soft\_violations} \times 10000 + \text{pause\_violations} \times 1000$$

- The cost function ‘g‘ is computed as:

  $$g = 4000 \times \text{number\_of\_assignments}$$

- Thus, the total cost for a node is:

  $$\text{Total Cost} = g + h$$

The heuristic function ’h’ is designed to estimate the lowest possible cost from the current node to the goal, prioritizing nodes that may lead to fewer unassigned students and lower violations.

As we approach the solution, the value of ’g’ increases, reflecting the cost incurred up to that point, while ’h’ tends to approach 0 for good solutions as it estimates the remaining cost to reach a solution. The balance between the number of unassigned students and the number of violated constraints makes ’h’ a heuristic that yields very good results in terms of expanded and explored states.

Admissibility of the Heuristic h

A heuristic h is admissible if it never overestimates the actual minimum cost to reach the goal from any node in the search space. For the given heuristic function in the A* algorithm:

$$h = \text{remaining\_students} \times 500 + \text{soft\_violations} \times 10000 + \text{pause\_violations} \times 1000$$

To evaluate admissibility, consider:

- $\text{remaining\_students} \times 500$: This component is a lower bound since assigning each student incurs at least this cost.
- $\text{soft\_violations} \times 10000$ and $\text{pause\_violations} \times 1000$: These components should reflect the minimal possible cost increase due to violations, ensuring they do not exceed the lowest penalties that might be applied when resolving these violations.

The real cost "g" increases by 4000 for each node close to a full timetable. Additionally, "h" estimates a difference of 8 more students per new assignment and an add-on of 10000 times the number of soft constraints violated. This means that it tends to overestimate when soft constraints are violated. However, this heuristic still effectively guides the search to avoid violations of soft constraints and ensures that no students remain unallocated.

If the penalty weights are calibrated correctly against the actual minimal penalties that can be incurred for each violation and assignment in practice, then h is admissible. However, if the heuristic penalties are set too high relative to the true minimal costs of resolving these issues, h could overestimate the costs, rendering it inadmissible. Thus, careful consideration and adjustment of these weights are crucial.

The issue arises when considering the cost "g" required to reach a state that accurately represents the real cost. This cost encompasses the number of assignments completed up to that state and may actually be smaller than the heuristic "h" for a path from node A to node B, which raises questions about the admissibility of the A* heuristic "h".

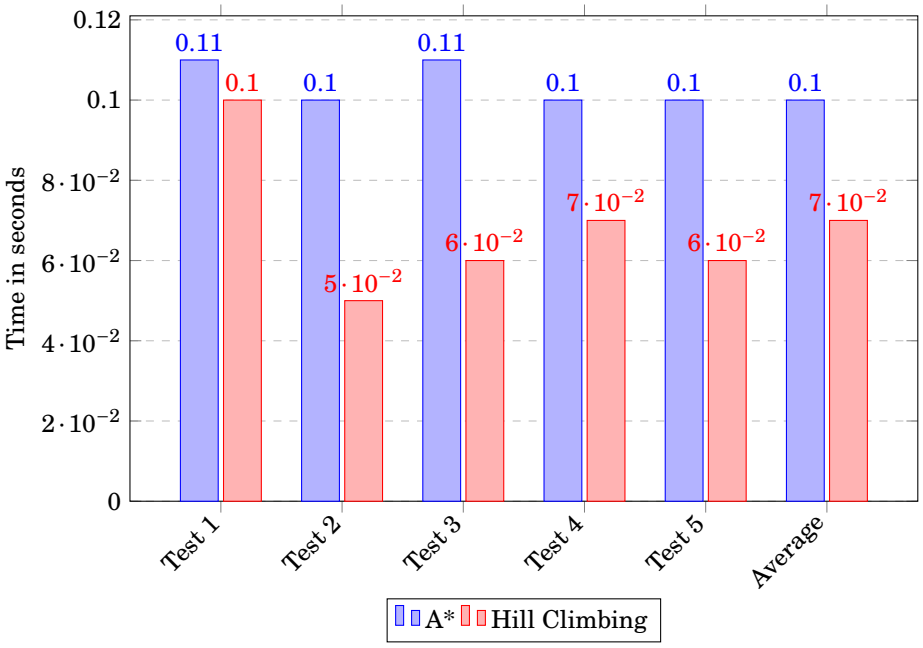
Testing has yielded positive results in A* with this heuristic. However, it can be argued that in a significant number of examples, this heuristic is not admissible and should not produce such good results. It is certain that finding a fully admissible heuristic for this problem is a difficult and uncertain task.

6.2 Time

Note The tests conducted for time measurements do not match the ones for results measurements, but reflect similar scenarios across different test numbers.

Dummy

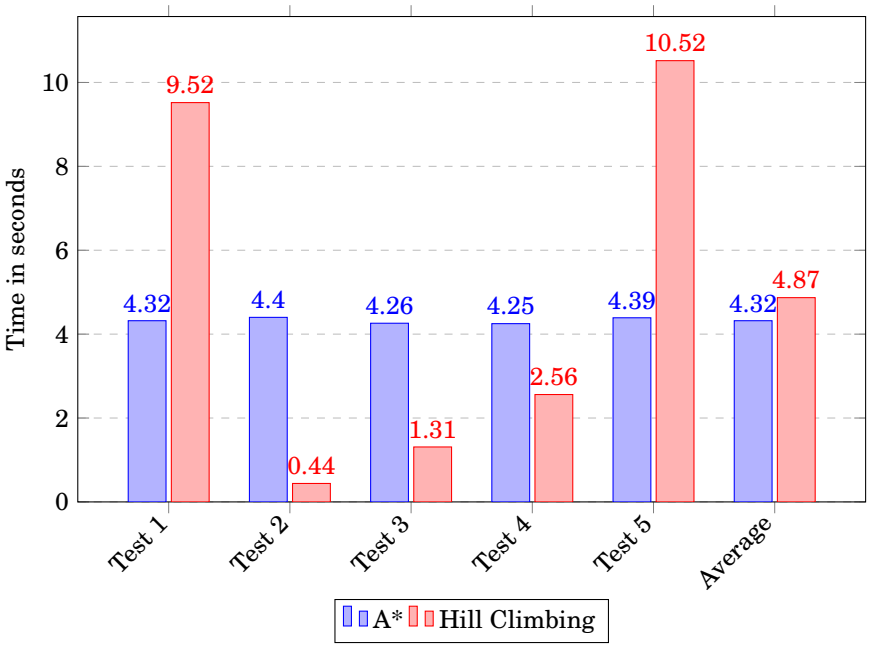
Test	A* Time (s)	Hill Climbing Time (s)
1	0.11	0.10
2	0.10	0.05
3	0.11	0.06
4	0.10	0.07
5	0.10	0.06
Average	0.10	0.07



Both algorithms performed well in the dummy test case. A* demonstrated consistent speed in search and the states it explores, while hill climbing's performance varied depending on the number of restarts, although it only required one or two in this small test.

Timetable_small_exact

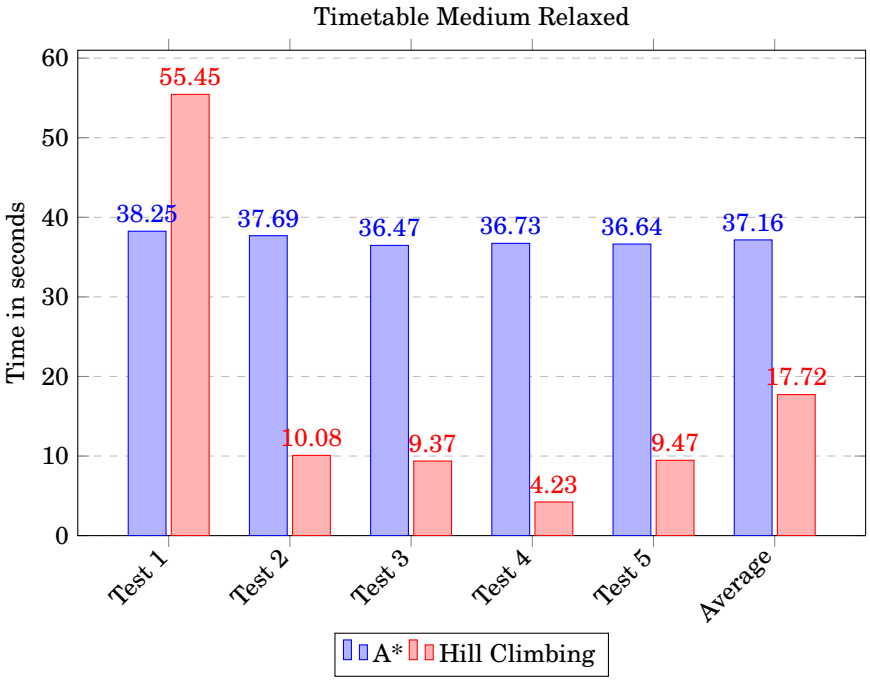
Test	A* Time (s)	Hill Climbing Time (s)
1	4.32	9.52
2	4.40	0.44
3	4.26	1.31
4	4.25	2.56
5	4.39	10.52
Average	4.32	4.87



In this small test, which is larger than the dummy one, we can observe the variability in time of the hillclimbing algorithm with random restarts. The algorithm's speed depends on how quickly it can find a restart with a favorable search starting point. The execution times range from 0.5 to nearly 11 seconds, showing significant variation. Despite these large differences, the average times are relatively close.

Timetable_medium_relaxed

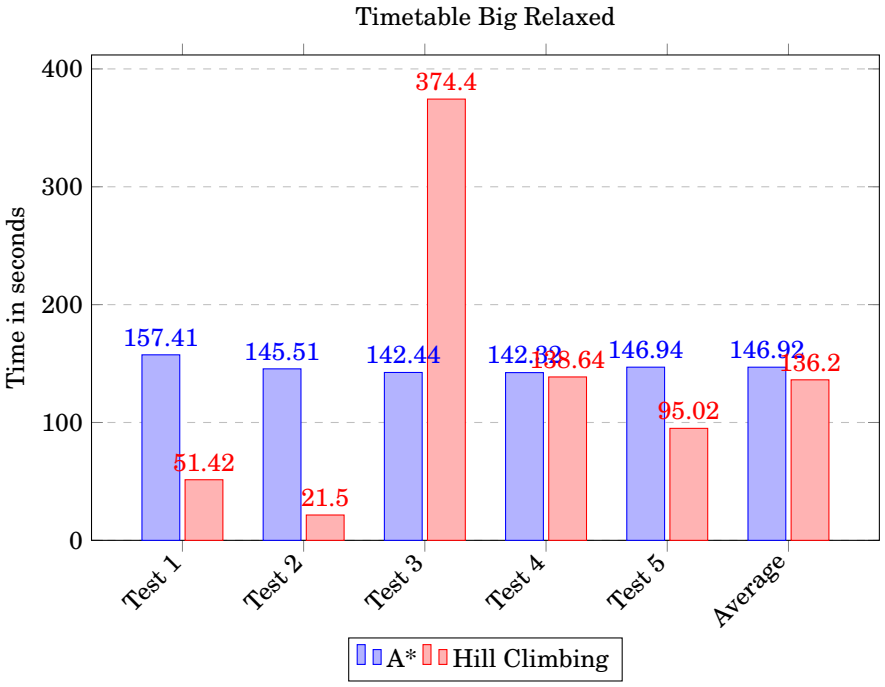
Test	A* Time (s)	Hill Climbing Time (s)
1	38.25	55.45
2	37.69	10.08
3	36.47	9.37
4	36.73	4.23
5	36.64	9.47
Average	37.16	17.72



Now, in a larger test, we observe that hill climbing achieves a significantly lower average. This is attributed to its ability to find solutions in very few restarts and iterations for certain tests. The average remains low, even when some tests do not find the solution in the initial restarts, and the time taken is much longer, almost 1 minute.

Timetable_big_relaxed

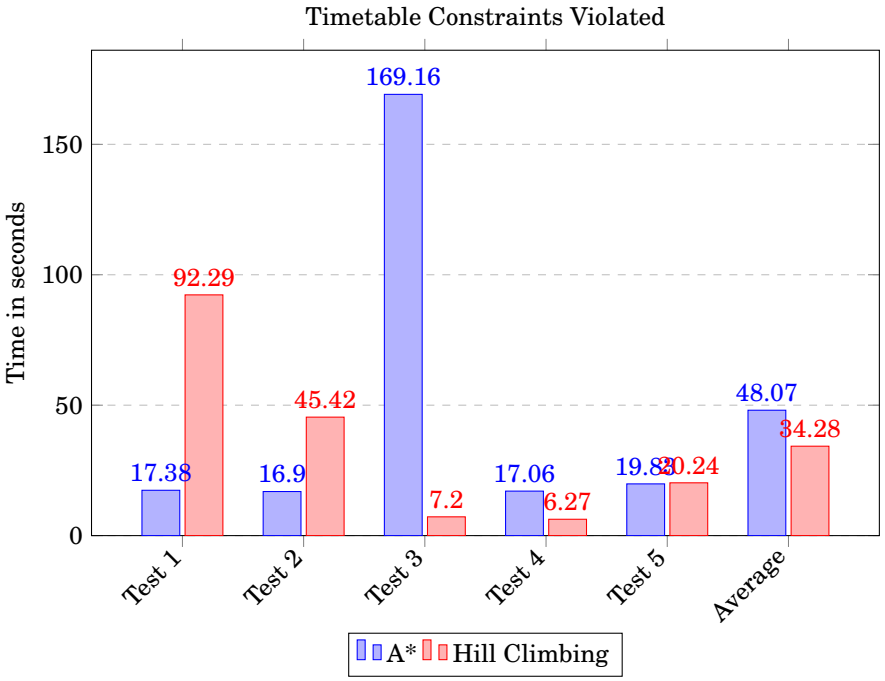
Test	A* Time (s)	Hill Climbing Time (s)
1	157.41	51.42
2	145.51	21.50
3	142.44	374.40
4	142.32	138.64
5	146.94	95.02
Average	146.92	136.20



The larger the test, the greater the disparity in time taken by the hill-climbing algorithm. In the case of the large test, the time ranges from 21 seconds to a disappointing 374 seconds with 7 full restarts, while the other tests had a maximum of 3 restarts. The A* algorithm consistently maintains its time performance, and the average times for the two algorithms are still very close to each other.

Timetable_constraints_violated

Test	A* Time (s)	Hill Climbing Time (s)
1	17.38	92.29
2	16.90	45.42
3	169.16	7.20
4	17.06	6.27
5	19.83	20.24
Average	48.07	34.28



In the test, both algorithms successfully found a hard-to-find solution with a cost of 0. The hill climbing algorithm showed consistent time variations with its restarts, while the A* algorithm had an unexpected performance. Excluding test 3, the A* algorithm was on average twice as fast as hill climbing. However, in one instance, the A* algorithm took almost 3 minutes to complete, compared to the average time of under 20 seconds for hill climbing. This was due to the A* algorithm getting stuck on a bad path near the end of the search, resulting in slow progress towards the 0 cost solution as it had to backtrack multiple times while exploring states with similar costs. A* algorithm typically demonstrates consistent and deterministic time complexity, with rare exceptions.

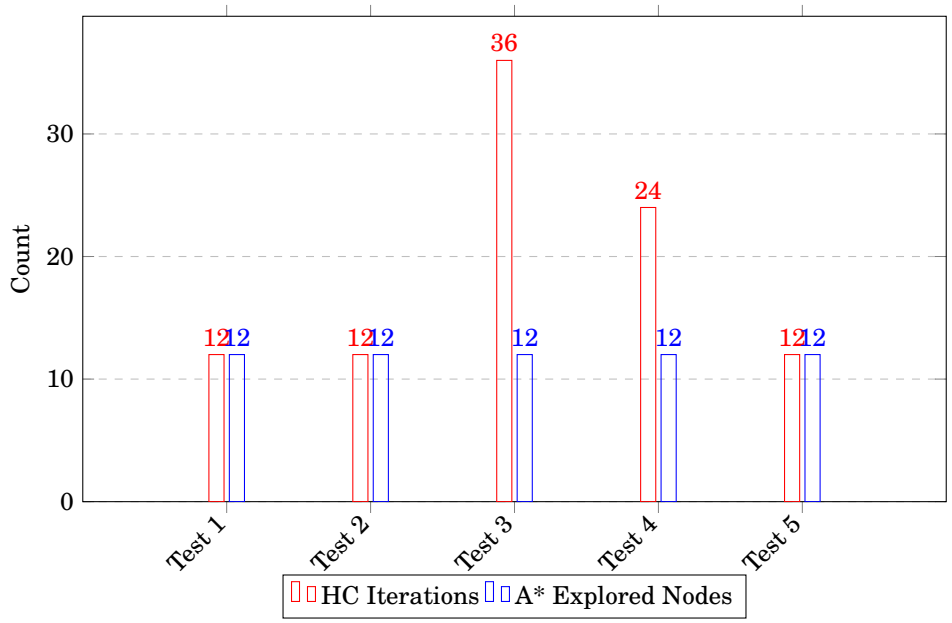
Time comparison conclusions In matters of time complexity, the go to choice for this problem would be the A* for relaxed tests with easier to find solutions because it proves much more consistent results. The hillclimbing can be used for more dense solutions because it has those random restarts that avoid getting stuck in local maximums and help find the global solution faster.

6.3 Exploration

Note The tests conducted for time measurements do not match the ones for results measurements, but reflect similar scenarios across different test numbers.

Dummy

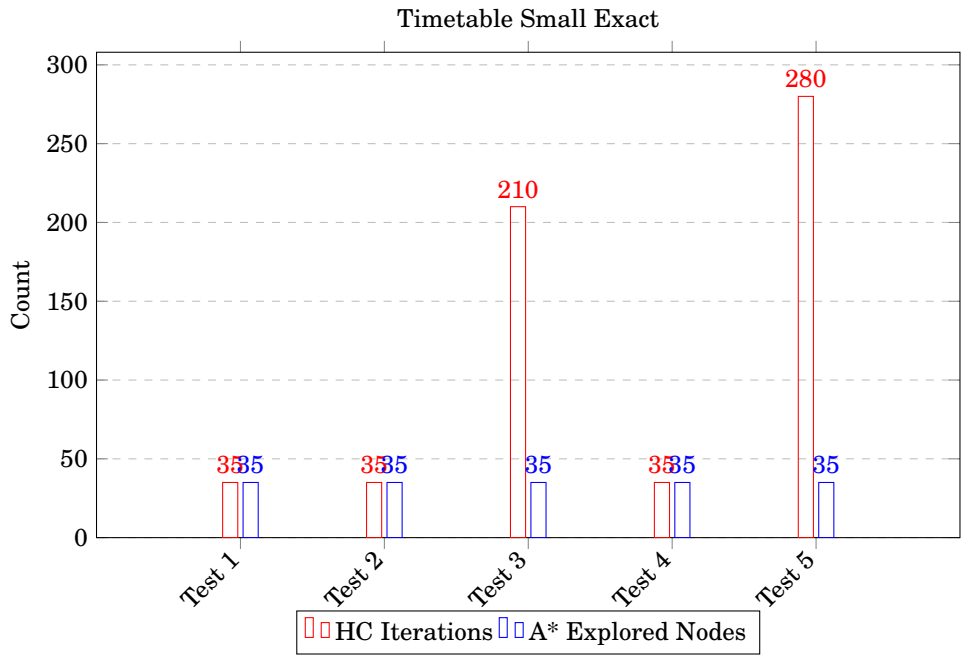
Test	Hill Climbing Iterations	A* Explored Nodes	A* Expanded Nodes	Restarts (HC)
1	12	12	477	0
2	12	12	465	0
3	36	12	483	2
4	24	12	482	1
5	12	12	498	0



This table indicates that for the Dummy test scenario, the Hill Climbing algorithm consistently explored very few nodes across all tests, reflected by low iteration numbers. A* explored and expanded a roughly equivalent number of nodes, indicating a straightforward problem space with minimal complexity. The Hill Climbing algorithm required very few restarts, suggesting that it found satisfactory solutions quickly without needing to explore alternative paths frequently.

Timetable_small_exact

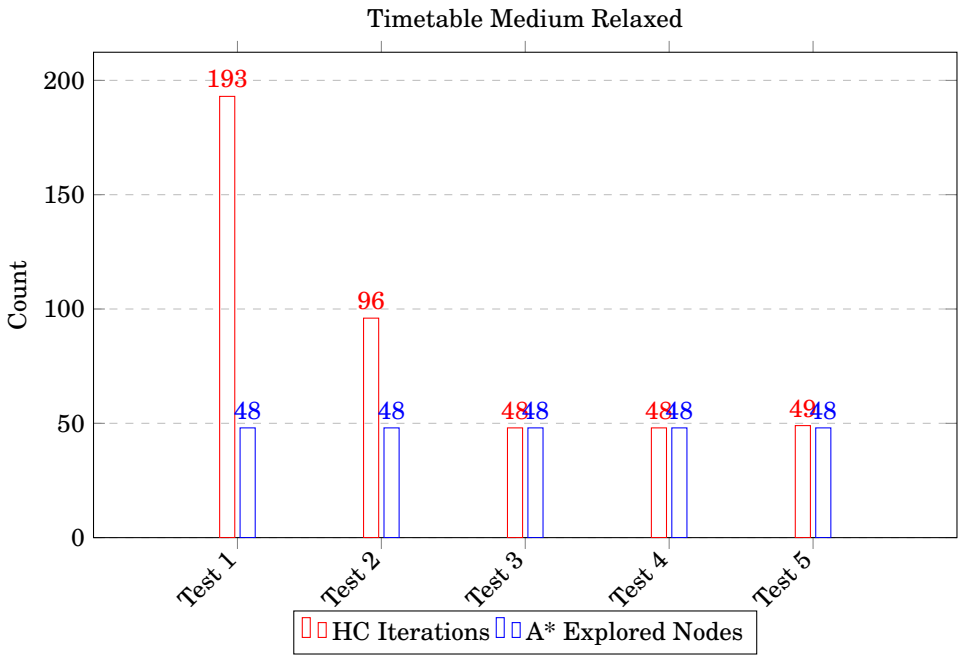
Test	Hill Climbing Iterations	A* Explored Nodes	A* Expanded Nodes	Restarts (HC)
1	35	35	13451	0
2	35	35	14154	0
3	210	35	13243	5
4	35	35	13537	0
5	280	35	13312	7



In the Timetable_small_exact scenario, the Hill Climbing algorithm varied significantly in terms of iterations, particularly in tests 3 and 5 where numerous restarts led to a high number of iterations. This suggests some difficulty in finding a viable solution, likely due to more complex constraints or less straightforward mappings of students to activities. A* explored a consistent number of nodes across all tests but expanded many more, illustrating its thorough search through the problem space to ensure optimality.

Timetable_medium_relaxed

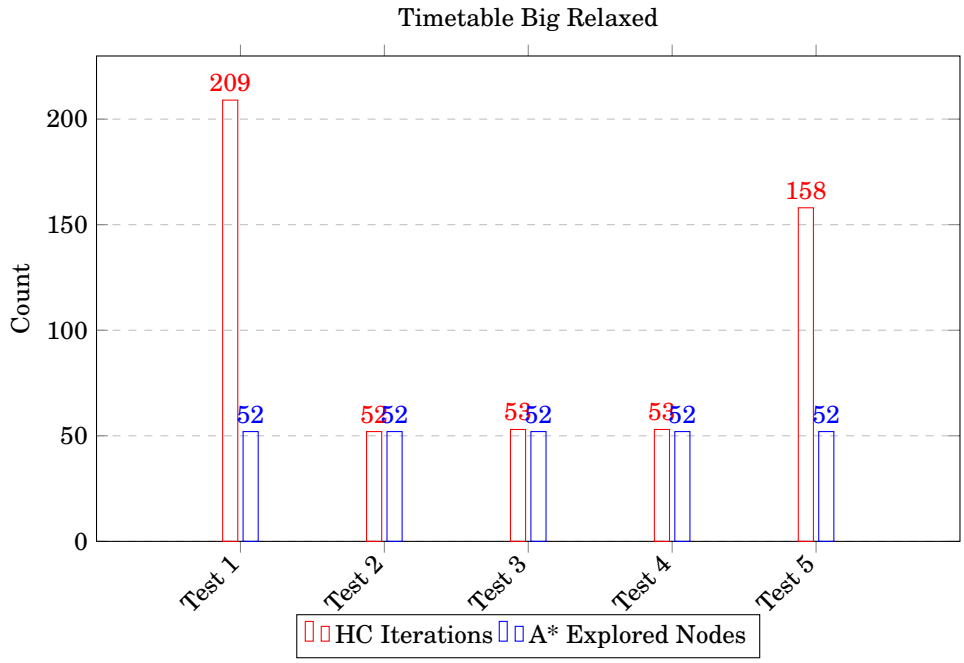
Test	Hill Climbing Iterations	A* Explored Nodes	A* Expanded Nodes	Restarts (HC)
1	193	48	88098	3
2	96	48	89932	1
3	48	48	88599	0
4	48	48	87583	0
5	49	48	89628	0



The Timetable_medium_relaxed tests show a moderate number of iterations for Hill Climbing, with more significant variation than in the simpler Dummy scenario. The expanded nodes for A* are notably high, indicating that while the problems are more complex, A* is efficiently exploring and verifying a large number of potential solutions. The restarts in Hill Climbing, especially in test 1, imply challenges in achieving an initial good fit, possibly due to more relaxed constraints allowing numerous near-optimal solutions.

Timetable_big_relaxed

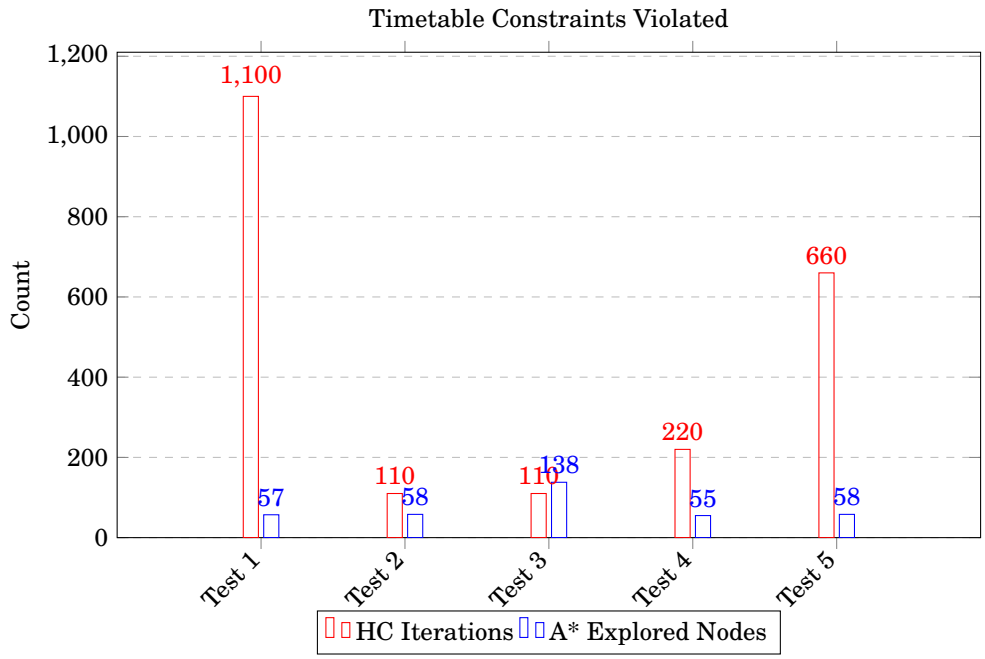
Test	Hill Climbing Iterations	A* Explored Nodes	A* Expanded Nodes	Restarts (HC)
1	209	52	272687	3
2	52	52	272222	0
3	53	52	273779	0
4	53	52	273922	0
5	158	52	275248	2



In the Timetable_big_relaxed scenario, Hill Climbing required a larger number of iterations, particularly in tests 1 and 5, which also saw more restarts. This reflects the algorithm's struggle with a larger and more complex problem space where multiple restarts are necessary to escape local optima. A* again shows a substantial number of expanded nodes, consistent with its goal of exploring all viable paths to ensure the solution's optimality in a significantly complex environment.

Timetable_constraints_violated

Test	Hill Climbing Iterations	A* Explored Nodes	A* Expanded Nodes	Restarts (HC)
1	1100	57	45039	20
2	110	58	44799	1
3	110	138	48460	1
4	220	55	44423	3
5	660	58	44250	11



For the Timetable_constraints_violated tests, Hill Climbing showed a high variation in iterations, particularly notable in test 1 with an extremely high count, accompanied by many restarts. This indicates significant difficulties in finding solutions that meet the stricter constraints, often requiring multiple attempts to refine the search strategy. This reflects in the solution, which does not have cost 0. A* expanded a pretty large number of nodes compared to other complex scenarios, suggesting that while the explored space was smaller, the solution paths required careful consideration and verification, likely due to the tight constraints and penalties for violations.

6.4 Quality

All tests produced solutions where 0 mandatory and 0 optional constraints were violated, with one exception. The only test out of the 50 (25 for A* and 25 for hillclimbing) that had one optional constraint violated was the initial hillclimbing test on the Timetable. Despite 20 full restarts, it was unable to find a solution with a total cost of 0, making it an exceptional case in the tests conducted on this algorithm.

6.5 Conclusions

The comprehensive analysis across various testing scenarios provides clear insights into the strengths and weaknesses of both the Hill Climbing and A* algorithms. Here's a brief overview and recommendations for choosing the appropriate algorithm depending on the specific requirements of the scenario:

- **Simple Scenarios (Dummy)**: Both algorithms perform exceptionally well with minimal time differences and consistent success in finding optimal solutions. For such simple scenarios, Hill Climbing may be slightly preferable due to its slightly faster average execution times.
- **Moderately Complex Scenarios (Small_exact and Medium_relaxed)**: Hill Climbing shows variable performance with its time efficiency heavily dependent on the quality of the restarts. A* demonstrates consistent performance but at a slightly higher computational cost. A* is recommended for cases where consistency and reliability are prioritized over computational resources.
- **Highly Complex Scenarios (Timetable_big_relaxed)**: In these scenarios, A* maintains a consistent performance but with high computational demand. Hill Climbing, despite its unpredictability and potential for long solution times, may still be viable when configured with effective heuristics and restart strategies. The choice here should be guided by the availability of computational resources and the acceptable variance in solution times.
- **Stringent Constraint Scenarios (Timetable_constraints_violated)**: Hill Climbing struggled significantly, particularly when searching for zero-violation solutions, suggesting its limitations in tightly constrained environments. A* shows better control over the exploration of the solution space but may require careful tuning of heuristics to avoid excessive backtracking. A* is recommended for its robustness in handling complex constraint satisfaction problems effectively.

Overall, while Hill Climbing can offer faster solutions in some contexts, it lacks the consistent reliability of A*, which despite its higher computational demands, provides a more predictable and stable performance across a broader range of problem complexities. Therefore, A* is generally the go-to algorithm for most scenarios, especially when solution quality and consistency are critical.

6.6 Additional time tests

Elapsed Time Comparison for Dummy

Test	A* Time (s)	Hill Climbing Time (s)
1	0.10	0.08
2	0.09	0.07
3	0.08	0.06
4	0.08	0.07
5	0.09	0.05
6	0.09	0.05
7	0.09	0.05
8	0.09	0.11
9	0.10	0.06
Average	0.09	0.07

Elapsed Time Comparison for Timetable_small_exact

Test	A* Time (s)	Hill Climbing Time (s)
1	3.59	0.45
2	3.71	5.18
3	3.69	0.93
4	3.47	1.86
5	3.71	0.95
6	3.43	11.05
7	3.77	1.35
8	3.68	1.31
9	3.61	0.88
Average	3.63	2.66

Elapsed Time Comparison for Timetable_medium_relaxed

Test	A* Time (s)	Hill Climbing Time (s)
1	31.20	4.06
2	29.77	3.87
3	29.33	15.48
4	29.89	4.56
5	29.84	42.34
6	28.50	9.43
7	29.24	9.30
8	28.37	4.17
9	30.07	4.31
Average	29.58	10.83

Elapsed Time Comparison for Timetable_big_relaxed

Test	A* Time (s)	Hill Climbing Time (s)
1	117.35	19.01
2	112.65	28.87
3	111.79	84.53
4	113.43	18.45
5	111.75	227.02
6	112.85	45.17
7	113.06	46.36
8	112.32	44.08
9	113.65	18.73
Average	113.20	59.13

Elapsed Time Comparison for Timetable_constrains_violated

Test	A* Time (s)	Hill Climbing Time (s)
1	13.13	27.66
2	12.82	20.25
3	13.09	28.27
4	13.18	79.12
5	15.20	3.80
6	13.22	5.80
7	13.12	77.31
8	12.88	1.85
9	13.09	17.19
Average	13.30	29.03

The additional elapsed time tests reinforce earlier observations about the relative performance of the A* and Hill Climbing algorithms under varying constraints and complexities. For simple scenarios such as the Dummy tests, the marginal time difference further supports the use of Hill Climbing due to its speed and effectiveness in straightforward contexts. However, as the scenario complexity increases, the A* algorithm generally maintains a consistent performance advantage, albeit at a higher computational cost, as seen in the small exact and medium relaxed tests.

In highly complex and constraint-heavy scenarios like Timetable_big_relaxed and Timetable_constraints_violated, the performance gap widens. A* shows far superior control and efficiency, managing the solution space exploration with significantly more predictability than Hill Climbing, which exhibits wide variances in solution times, especially evident in its struggle with stringent constraints where the average times were considerably higher. These tests underscore the necessity of selecting the right algorithm based on the scenario's demand for reliability versus operational efficiency, with A* emerging as a robust choice for handling complexity and constraints effectively.

7 Bonus

Scheduling algorithms rely on pause constraints to create practical timetables for educational settings. These constraints regulate the maximum allowable time between consecutive activities for professors, aiming to prevent inefficient and inconvenient schedules.

The Python function `count_pause_violations` is designed to manage these constraints by calculating instances where the interval between assignments exceeds a defined maximum. It iterates through sorted assignments and checks the gaps against the imposed limit.

```
def count_pause_violations(imposed_max_pause, prof_assignments):
    ...
    if pause > imposed_max_pause:
        bigger_pauses.append(pause)
    ...
```

This function has been externally tested and effectively identifies gaps in professors' timetables that exceed a certain threshold.

In heuristic searches such as A* or hill climbing, pause violations can have a significant impact on the heuristic value of a node. For instance, in the functions `eval_node` and `h`, pause violations result in a penalty being added to the heuristic cost, which favors states with fewer violations. This is demonstrated in the following snippet:

```
pause_violations = self.number_of_pause_restrictions_violated()
student_penalty = (remaining_students**2) * 50
constraint_penalty = soft_violations * 300000 + pause_violations * 1000
return student_penalty + constraint_penalty
```

The bonus tests allocated do not yield good results with the implemented algorithms. The hill climbing algorithm fails to find a solution within a reasonable time using the given strategy, and the A* algorithm fails to assign the last 100 students, indicating that it is not effective for solving this sample.

8 References

1. Geeks for Geeks. (n.d.). A* Search Algorithm. Retrieved from geeksforgeeks/a-search-algorithm
2. Geeks for Geeks. (n.d.). Hill Climbing. Retrieved from geeksforgeeks/introduction-hill-climbing-artificial-intelligence
3. Wikipedia contributors. (2023). A* search algorithm. In *Wikipedia, The Free Encyclopedia*. Retrieved from wikipedia/wiki/A*_search_algorithm
4. Wikipedia contributors. (2023). Hill Climbing. In *Wikipedia, The Free Encyclopedia*. Retrieved from wikipedia/wiki/Hill_climbing