Reed Andreas and Matt Schwarz

CS 4260

Final Project Report

4 December 2023

Strengths and Weaknesses of Genetic Algorithms

1. Introduction

1.1 Motivation

Our team's project was motivated by a desire to critically evaluate the effectiveness of genetic algorithms across various problem-solving scenarios. Recognizing the growing importance of algorithmic efficiency and adaptability, we aimed to explore the strengths and weaknesses of genetic algorithms compared to traditional methods. Through our work, we sought to demonstrate the practical applications and limitations of these algorithms by applying them to three distinct problems: the 0/1 Knapsack problem, the N-Queens problem, and a class scheduling challenge. Our objective was not only to solve these problems but also to gain deeper insights into when and how genetic algorithms can be most effectively utilized in real-world situations. Each of these three problems has a distinct set of inherent complexities as well as commonly understood traditional approaches to solving them.

1.2 Background

Genetic algorithms are an approach to solving optimization problems that have been inspired by natural selection (MathWorks). The algorithm has several steps. First, an initial population is randomly created–these random versions of a solution are almost certainly suboptimal. Then, the population is ranked according to some utility-like fitness function. Based upon this fitness function, certain elements from the population are selected to be parents for the next generation; the rest of the population is discarded. Next, these parents are bred together in some fashion (often with some mutation) to result in a new population, and the entire process is repeated. Over many generations, the population evolves toward an approximation of an optimal solution.

Our first example problem–the 0/1 Knapsack problem–is a fundamental problem in combinatorial optimization. It models a scenario where one must maximize the total value of items placed in a knapsack of limited capacity. Items can either be taken whole or left, hence the name 0/1. The classical approach to this problem is dynamic programming.

Next, the N-Queens problem is a well-known puzzle, originating from the game of chess. It involves placing N queens on an NxN chessboard in such a way that no two queens threaten each other by lying in the same row, column, or diagonal. A common formulation of this problem is the 8-Queens problem. Traditionally, this problem is approached using recursive backtracking, a method that incrementally builds candidates to the solutions and abandons a candidate as soon as it determines that the candidate cannot possibly be completed to a valid solution.

Lastly, the class scheduling problem, while not as historically prominent as the other two, is a practical issue faced in educational and organizational contexts. It involves scheduling classes over a set period without violating prerequisites. A common approach to this problem is using a topological sort.

1.3 Challenges & Novelty

In our project, we embarked on a novel exploration. For each problem, we compared the traditional problem-solving approach with a more dynamic and adaptable genetic algorithm solution. This allowed us to illuminate the strengths and weaknesses inherent in both classical and genetic algorithmic methods. In our brief review of the existing academic literature, we did not encounter any papers which empirically investigated the tradeoffs between genetics and traditional approaches for a myriad of different problem types. Our approach was an applied endeavor that required us to adapt and tailor the genetic algorithms specifically for each problem, ensuring their relevance and efficacy. By doing so, we were able to draw meaningful comparisons.

2. Methods and Results

For each problem, we implemented the genetic algorithm and traditional approach using Python (with the exception of the class scheduling problem). Then, we empirically tested the performance of each approach by timing the runtime of each algorithm over a variety of input sizes and examples while ensuring the correctness of the solution. Finally, we identified and interpreted any relationships that we observed within our experiment.

2.1 0/1 Knapsack

2.1.1 Methods.

In tackling the 0/1 Knapsack problem, our team employed both the classical dynamic programming approach and a customized genetic algorithm. We consulted online resources to plan the simple dynamic programming approach, including much inspiration from Geeks for Geeks regarding the structure of the code (GeeksforGeeks, 2023). We wrote the genetic algorithm ourselves while, of course, borrowing some core ideas about implementation approaches from resources such as Medium articles.

```python
def knapsack(values, weights, capacity, verbose=False):
    num_items = len(values)

    dp = []
    for i in range(num_items + 1):
        dp.append([0] * (capacity + 1))

    for i in range(1, num_items + 1):
        for w in range(1, capacity + 1):
            if weights[i - 1] <= w:
                value_including_item = values[i - 1] + dp[i - 1][w - weights[i - 1]]
                value_excluding_item = dp[i - 1][w]
                dp[i][w] = max(value_including_item, value_excluding_item)
            else:
                dp[i][w] = dp[i - 1][w]

        if verbose and i % 100 == 0:
            print(f"Finished {i} items")

    return dp[num_items][capacity]
```

Figure 1. Dynamic programming implementation of 0/1 Knapsack problem

```python
def knapsack_gen(values, weights, capacity, num_generations = 50, verbose=False, log_times = False):
    # Parameters
    num_items = len(values)
    population_size = 1000
    times = {}

    population = [generate_knapsack(num_items) for _ in range(population_size)]

    for gen in range(num_generations):
        fitness_scores = [calculate_fitness(k, values, weights, capacity) for k in population]

        # Sort the population based on fitness and select the top knapsacks
        sorted_population = [x for _, x in sorted(zip(fitness_scores, population), reverse=True)]
        parents = sorted_population[:50]

        # Generate new population through crossover and mutation
        new_population = parents[:]
        while len(new_population) < population_size:
            parent1, parent2 = random.sample(parents, 2)
            child1, child2 = crossover(parent1, parent2)
            new_population.extend([mutate(child1), mutate(child2)])

        population = new_population

        if verbose and gen % 10 == 0:
            best_solution = max(population, key=lambda k: calculate_fitness(k, values, weights, capacity))
            best_fitness = calculate_fitness(best_solution, values, weights, capacity)
            print(f"Generation: {gen} | Best fitness: {best_fitness}")

        if log_times and gen % 10 == 0:
            times[time()] = [gen, best_fitness]

    best_solution = max(population, key=lambda k: calculate_fitness(k, values, weights, capacity))
    best_fitness = calculate_fitness(best_solution, values, weights, capacity)
```

Figure 2. Genetic algorithm implementation of 0/1 Knapsack Problem

2.1.1 Results.

Our dynamic programming solution effectively handled smaller-capacity examples. However, as the size of the problem increased, we observed the inherent limitations of this approach in terms of scalability and computation time. In contrast, our genetic algorithm, which was specifically tailored for this problem, provided a more flexible and scalable solution. Although it did not always guarantee an optimal solution–especially in larger instances–it demonstrated a significant advantage in finding approximate solutions quickly. This comparison led us to an important takeaway: while traditional methods like dynamic programming are powerful in solving certain scales of problems, genetic algorithms offer a valuable alternative in dealing with larger, more complex scenarios where approximate solutions are acceptable and time efficiency is crucial. Nevertheless, if absolute accuracy was a key factor, then our genetic algorithm–as currently devised–did not stand up to the rigor of the classic approach.

2.2 N-Queens

2.2.1 Methods.

For the N-Queens problem, we wanted to understand how a genetic algorithm solution would perform against the traditional recursive backtracking approach. To this end, we implemented both approaches, then tested them for several different N-sized boards. Since the purpose of our project was gaining insight into the tradeoffs between genetic algorithms and traditional approaches and not simply coding an implementation to these approaches, we borrowed code snippets from online resources such as Educative in order to expedite this portion of the project (Bondar, 2023; Kazmi, n.d.). To get a sense of how each approach handled both small and large inputs, we tested each algorithm on boards of sizes eight through twenty in increments of two. Additionally, we conducted five trials for each N-sized board, collecting the average and standard deviation for each batch of trials. Each of these batches then became a data point which we plotted in two separate graphs: Average Algorithm Runtime vs. [Board Size] N and Stdev of Algorithm Runtime vs. [Board Size] N.

```
'''
N-Queens solver
- Solves N-Queens problem using genetic algorithm
'''
def n_queens_genetic(N, POPULATION_SIZE=50, MUTATION_RATE=0.1, MAX_GENERATIONS=100):
  MAX_FITNESS = N * (N - 1) / 2

  # Initial population
  population = [(generate_board_state(N), 0) for _ in range(POPULATION_SIZE)]

  # Loop over generations...
  for generation in range(MAX_GENERATIONS):
      # Calculate fitness for each board state
      population = [(board_state, calculate_fitness(board_state)) for board_state, _ in population]

      # Check if solution is found
      best_board_state = max(population, key=lambda x: x[1])[0]
      if calculate_fitness(best_board_state) == MAX_FITNESS:
          #print("Solution found in generation", generation)
          break

      # Create the next generation
      new_population = []

      # Elitism: Keep the best board state from the previous generation
      new_population.append(max(population, key=lambda x: x[1]))

      # Perform selection, crossover, and mutation
      while len(new_population) < POPULATION_SIZE:
          parent1 = tournament_selection(population)
          parent2 = tournament_selection(population)
          child = crossover(parent1[0], parent2[0])
          if random.random() < MUTATION_RATE:
              child = mutate(child)
          new_population.append((child, 0))

      # Update the population
      population = new_population

  return best_board_state
```

Figure 3. Genetic algorithm implementation of N-Queens problem

2.2.2 Results.

Our findings from this experiment were that the genetic algorithm outperformed recursive backtracking when N was large (> 16) in terms of both absolute and consistency of runtime performance. For N's that were not large (<= 16), recursive backtracking performed slightly better; however, there did not appear to be much difference in performance.

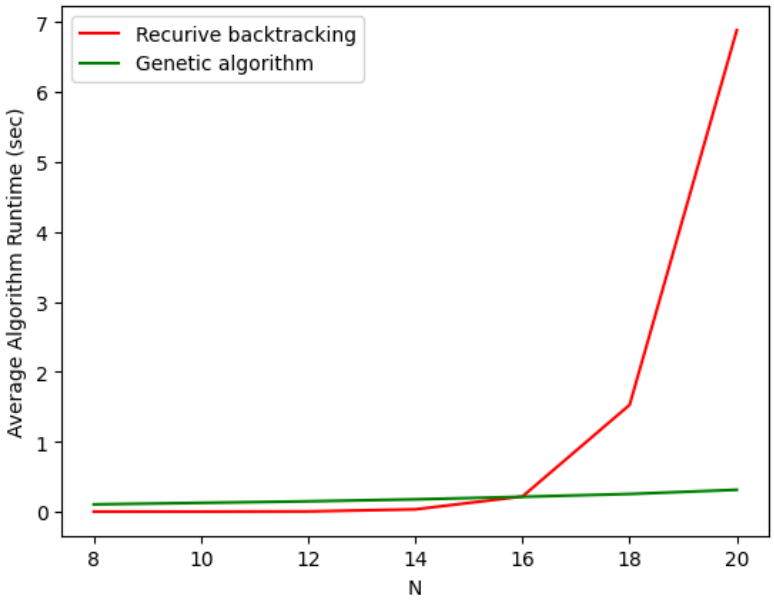

Figure 4. Plot of Average Algorithm Runtime vs. [Board Size] N

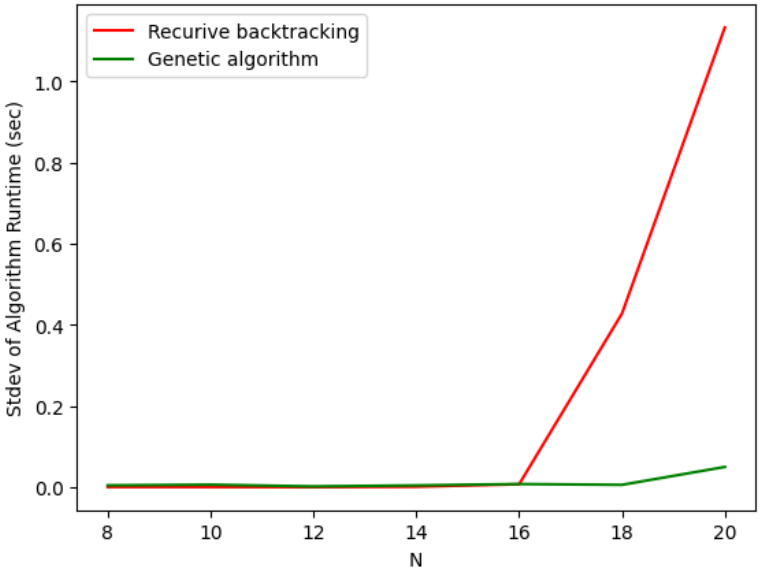

Figure 5. Plot of Stdev of Average Algorithm Runtime vs. [Board Size] N

Recursive backtracking was slightly faster for small to medium board sizes (N's) (<= 16); however, the difference was marginal. For large N's (> 16), the runtime of recursive backtracking grew very quickly. On the other hand, the runtime of the genetic algorithm remained fairly constant, even for large N's. As such, we concluded that a recursive backtracking approach was superior in terms of runtime when N was not large while a genetic algorithm approach was better for larger N's. Furthermore, the variation in runtimes between trials for a given N remained fairly

constant for all N's with the genetic algorithm. In contrast–while low when N was not large–the variation in runtime increased quickly for the recursive backtracking approach with larger N's. This suggests that–while there did not appear to be much difference in runtime consistency between the approaches when N was not large–the genetic algorithm was much less volatile in terms of runtime when N was large, making it a safer choice for large inputs.

2.3 Class Scheduling

2.3.1 Methods.

For the class scheduling problem, our initial goal was to match the capability of the traditional topological sort, which effectively schedules classes without violating prerequisites. We successfully achieved this by implementing a genetic algorithm that we crafted on our own, effectively demonstrating its ability to handle the basic scheduling requirements.

```
def schedule_fitness(schedule, verbose=False):
    utility = 0

    # we should now check for prereqs
    taken_names = set()
    for semester in schedule.semesters:
        temp_taken = []
        for course in semester.courses:
            if course.name in prereqs:
                for prereq in prereqs[course.name]:
                    if prereq not in taken_names:
                        if verbose:
                            print(f"Prereq not taken: {prereq} for {course.name}")
                        utility -= 1000
            temp_taken.append(course.name)
        taken_names.update(temp_taken)

    # now we should check for duplicates
    taken_names = set()
    for semester in schedule.semesters:
        for course in semester.courses:
            if course.name in taken_names:
                utility -= 1000
            taken_names.add(course.name)

    return utility
```

Figure 6. Genetic algorithm implementation of class scheduling problem

However, we didn't stop there. Our team extended the project by tweaking the fitness function of our genetic algorithm. We added a simple yet practical criterion: a reward for scheduling physics classes earlier in the curriculum (to simulate preparing for a physics internship that our student wants to be ready for in the summer). This adaptation showcased the versatility of genetic algorithms, as it allowed for personalization and the incorporation of student preferences into the scheduling process. It was a clear demonstration of how even

students like us could use genetic algorithms in real-life situations, easily modifying rewards to see various potential outcomes.

```
# lets reward physics classes in the first two semesters
if "PHYS 1600" in [course.name for course in schedule.semesters[0].courses]:
    utility += 300
if "PHYS 1601" in [course.name for course in schedule.semesters[1].courses]:
    utility += 300
```

Figure 7. Genetic algorithm reward function modification

However, the class scheduling problem wasn't without its challenges. When we initially tried the standard two-parent crossover method for our genetic algorithm, we encountered a significant issue. This approach, while effective in many genetic algorithm applications, proved impractical for our scheduling problem due to a high likelihood of producing duplicate classes in schedules. This hindered the exploration of a diverse range of scheduling options. To overcome this, we devised a novel “mitosis” approach. Instead of combining two parent solutions, we allowed the best solutions to undergo a mutation process, where classes were swapped within a single schedule. As a result, this method eliminated the occurrence of duplicates, enabling a more effective exploration of the solution space.

```
def schedule_mutate(schedule):
    # make a copy of the schedule
    schedule = deepcopy(schedule)
    # swap two courses
    semester1 = random.choice(schedule.semesters)
    semester2 = random.choice(schedule.semesters)
    while semester1 == semester2:
        semester2 = random.choice(schedule.semesters)
    course1 = random.choice(semester1.courses)
    course2 = random.choice(semester2.courses)
    if course1 == course2:
        return schedule
    semester1.courses.remove(course1)
    semester2.courses.remove(course2)
    semester2.courses.append(course1)
    semester1.courses.append(course2)

    return schedule
```

Figure 8. Genetic algorithm “mitosis” mutation function

2.3.2 Results

While it was exciting to see our solution work, the difficulties we encountered also highlighted how sometimes complex thinking is needed to properly craft an algorithm that works

for specific, abnormal situations. In some cases, traditional genetic approaches may create too many “stillborn” nodes or fail to properly maintain genetic diversity.

Our experiment with the class scheduling problem illustrated not only the flexibility and adaptability of genetic algorithms but also their practicality in real-world applications. For this problem, we felt that a quantitative evaluation was not necessary. Rather, our qualitative experience developing a novel implementation of the genetic algorithm to solve our unique problem statement served as evidence of the superior flexibility of generic algorithms over a traditional topological sort for a class scheduling problem with added constraints.

```
Best fitness: 600
CS 1101 PHYS 1600 RELG 1101 ENGL 1101 MATH 1300
CS 2201 PHYS 1601 PHIL 1101 CS 2212 MATH 1301
PHIL 1102 HIST 1101 CS 3270 ENGL 1102 MATH 2300
CS 3281 RELG 1102 CS 3250 HIST 1102 CS 3251
```

Figure 9. Results of Class Scheduling run of genetic algorithm

Our experience with this problem was a testament to the power of genetic algorithms in solving not just theoretical problems but also those with direct relevance to our everyday lives.

3. Conclusion

3.1 Lessons Learned

Several key lessons and insights have emerged from our exploration of genetic algorithms. We've learned that while genetic algorithms offer a versatile and powerful tool for solving a wide range of problems, they are not a one-size-fits-all solution. Their effectiveness can vary greatly depending on the nature and scale of the problem at hand. For instance, their ability to provide quick, approximate solutions in large-scale scenarios is offset by occasional limitations in finding the optimal solutions, particularly in complex problems like the large 0/1 Knapsack.

3.2 Limitations and Future Work

Our project also faced certain limitations, primarily in the scope of problem sizes and the depth of algorithm customization. The genetic algorithms we developed were tailored for specific problems, but further testing is needed to enhance their efficiency and applicability to a

broader range of scenarios. Furthermore, it may be easier, depending on the problem at hand, to employ an out-of-the-box approach rather than turning to a genetic algorithm (such as in a simple 0/1 knapsack situation or small-sized N-Queens problem). This leads us to the discussion of future work. If we were to continue going forward, we would aim to explore more sophisticated genetic algorithm techniques, such as hybrid models that combine the strengths of genetic algorithms with other optimization methods. Additionally, we would plan to apply our algorithms to a wider array of problems, further testing their scalability and adaptability. This continuous exploration will not only contribute to our understanding of genetic algorithms but also to the broader field of computational problem-solving, demonstrating the ever-evolving nature of technology and its application in real-world scenarios.

3.3 Team Member Contributions

We felt our team struck a very nice balance of even contributions. Reed focused mainly on creating the 0/1 Knapsack and class scheduling problem notebooks. Matt focused on the N-Queens notebook as well as collecting all of the programs into a core program that is easy to run for other students evaluating the code. Lastly, we contributed evenly in splitting up the sections for our report and presentation. Our notebooks and peer evaluation executable can be found in our GitHub repository[1].

[1] https://github.com/mjschwarz/genetic-algorithms

References

Bondar, S. (2023, October 11). *Python algorithms: The N-queen problem*. Reintech Media. https://reintech.io/blog/python-algorithms-solving-n-queen-problem

GeeksforGeeks. (2023, October 11). *0/1 Knapsack problem*. GeeksforGeeks. https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/

Kazmi, H. (n.d.). *Educative answers - trusted answers to developer questions*. Educative. https://www.educative.io/answers/solving-the-8-queen-problem-using-genetic-algorithm

MathWorks. (n.d.). *What Is the Genetic Algorithm?*. MathWorks. https://www.mathworks.com/help/gads/what-is-the-genetic-algorithm.html