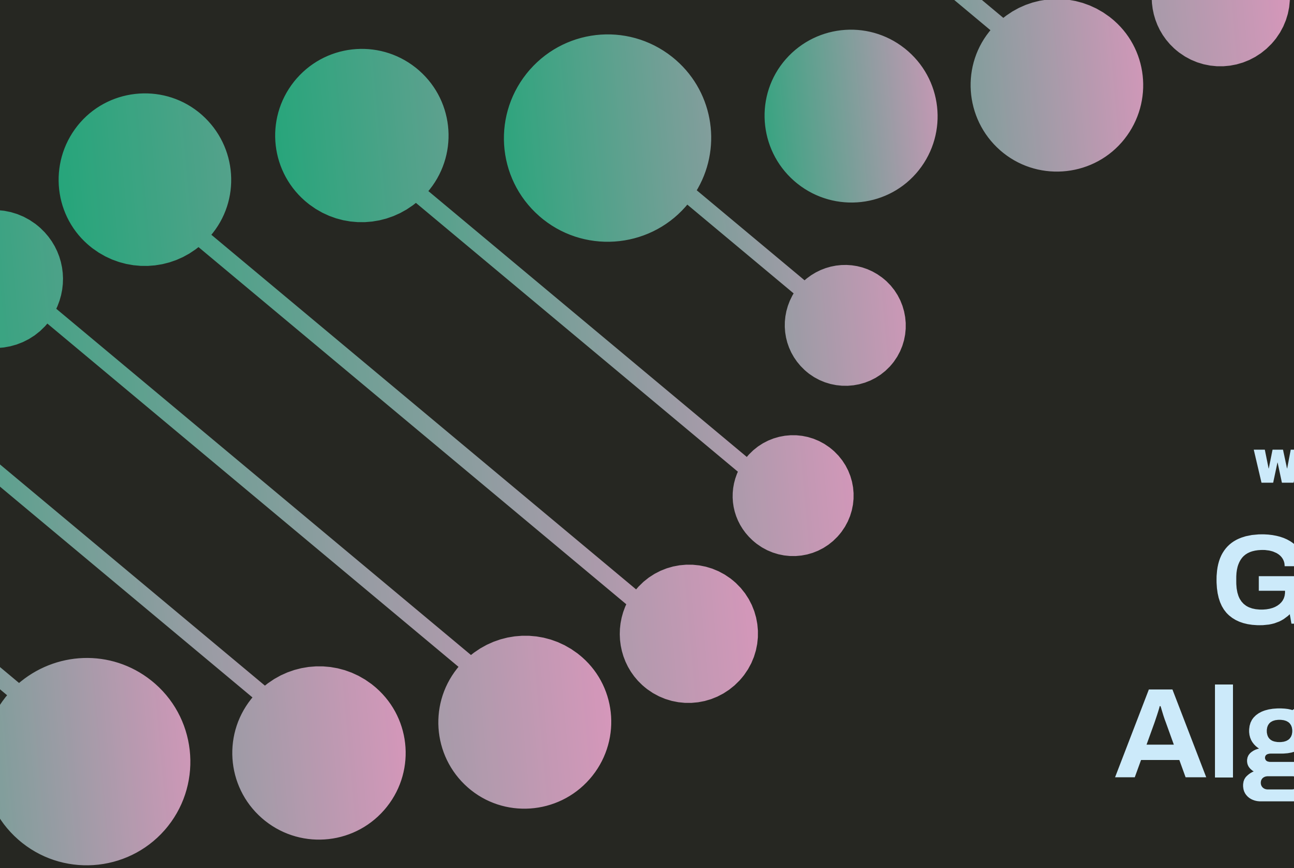

Strengths & Weaknesses of

Genetic Algorithms

Reed Andreas and Matt Schwarz

OBJECTIVE

Compare the performance of **genetic algorithms** versus **traditional approaches** for three different types of problems.

GENETIC ALGORITHMS (REVIEW)

- Create random versions of a solution (population) that are almost certainly suboptimal
- Rank them by a utility (fitness) function
- Select versions to be parents
    - (Unfortunately) Kill non-parents
- Breed parents (can take many forms) with mutation
- Repeat over many generations

PROBLEM TYPES

01

0/1 Knapsack

Find the optimal packing of a knapsack with given weight capacity and a set of items with individual values and weights such that the value of the knapsack is maximized

02

N-Queens

Find a valid placement of N Queens on an NxN chessboard such that no two queens threaten each other by lying in the same row, column, or diagonal

03

Class Scheduling

Find a valid class ordering given a list of courses with prerequisites (and possibly some other constraints)

01.

0/1 KNAPSACK

Dynamic programming

METHODS & RESULTS

- DP solution effectively handled smaller-capacity examples
- Limitations of DP in terms of scalability and computation time
- Genetic algorithm more flexible and scalable
- BUT genetic algorithm did not always guarantee an optimal solution for large inputs

```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 for 0/1 Knapsack

02.

N-QUEENS

Recursive backtracking

METHODS & RESULTS

- Tried N's of {8, 10, 12, 14, 16, 18, 20}
- Recursive backtracking runtime grows quickly
- Genetic algorithm superior for large inputs

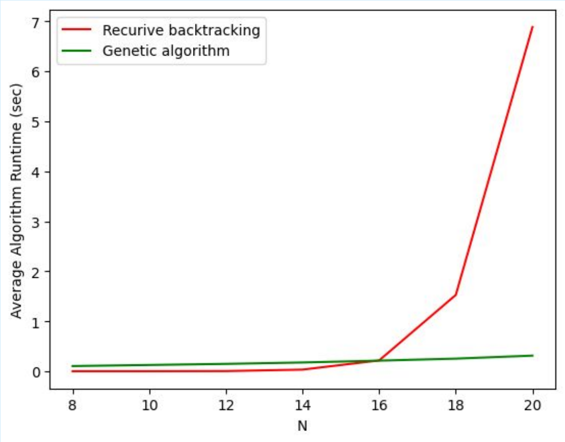

Figure 3. Plot of Average Algorithm Runtime vs. N

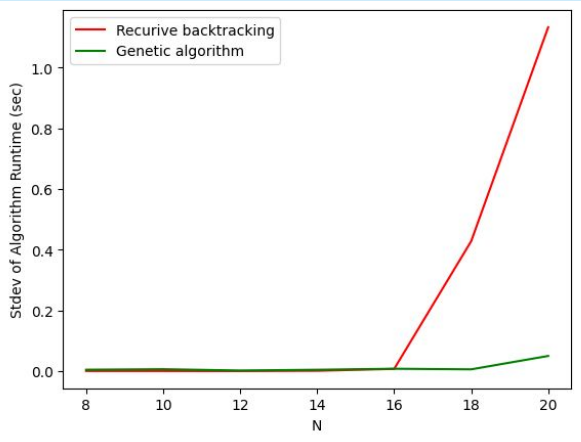

Figure 4. Plot of Stdev of Average Algorithm Runtime vs. N

```python
'''
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 5. Genetic algorithm implementation of N-Queens problem

03.

CLASS SCHEDULING

Topological sort

METHODS & RESULTS

- Topological sort is limited to solving prerequisite problem
- Genetic algorithm allows additional constraints to be optimized through reward function
- “Mitosis” mutation function

```python
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 7. Genetic algorithm implementation of class scheduling problem

```python
# 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 6. Genetic algorithm reward function modification

```python
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

DEMO

Peer Evaluation Executable

```
"""
-- Executable for peer evaluation --
You will need a relatively recent version of Python installed (likely 3.7+)

You may need to install a few packages if you have not already.
To do so, run the following commands in the terminal:

pip3 install numpy
pip3 install matplotlib

Using the terminal, enter into the directory containing this file
and run the following command:

python3 peer_eval.py

Now, wait for the code to execute
and you should see the output printed in the console.
"""
```

Video
(5x Speed)

CONCLUSION

Genetic algorithms are:

- versatile and powerful
- able to solve a wide range of problems
- good for quick, approximate solutions

Genetic algorithms are **not**:

- a one-size-fits all solution
- always able to find an optimal solution

ANY QUESTIONS?