A Genetic Algorithm for Task Scheduling

Strickland, Sean and Sanders, Theisen

Abstract

Coming soon

Introduction

There are many different approaches taken to solving the task scheduling problem. Our problem specifically concerns task scheduling that involves multiple processors and any number of constraints applied to the schedule. Obviously, a brute force scheduler that tries every possible schedule and choses the best is not a reasonable solution. A heuristic, however, can get us close. Although a genetic algorithm will provide a suboptimal solution, depending on its implementation, it can provide an approximate solution to the problem.

Problem

In this section we will dig deeper into the problem of task scheduling and explain why a genetic algorithm could be a good solution.

Task Scheduling

In general, task scheduling is the problem of allocating time and resources for the running of a set of tasks in an efficient manner. In this paper, we assume some finite number of processors to execute tasks, which is consistent with the very common problem of multitasking. We also assume that the system is non-preemptive, which means no task can be interrupted once its execution begins. Thus, the efficiency of a schedule is determined by two measures:

- Total Time - The time it takes for the longest running processor to finish executing tasks.
- Prioritized Throughput - The sum of all the waiting times (in our case end times) of each task, weighted by its priority.

In the real world there are often further constraints on these tasks, which determine whether a schedule is valid or acceptable. In this paper we assume two possible constraints:

- Procedural - A constraint on two tasks T1 and T2 such that T1 must be completed before T2 can start execution.
- Temporal - A constraint on a task T1 such that T1 must be completed before a specified time.

These constraints and efficiency measures often conflict with each other. This makes it no surprise that the problem is NP-Complete, which means a heuristic will have to be used to determine a good schedule. A genetic algorithm is one such heuristic method that could be used.

Genetic Algorithms

Genetic algorithms are local search algorithms in which a population of solutions is maintained and evolved over generations to produce better solutions. A solution is usually represented by a string of characters from a finite alphabet. A population is a set of these solutions (or individuals), a portion of which (determined by a fitness function) are used to reproduce and create the next generation population. The reproduction process usually involves randomly selecting a crossover portion of the solution string, which two solutions will swap to form two new solutions for the next generation. In addition, there is a random chance that a child solution can mutate (be modified slightly after the crossover process).

The key to a genetic algorithm's performance lies in the choice of string representation for the solution, as well as the choice of fitness function used to influence which solutions get to reproduce. A good solution string representation is one where characters next to each other in the string are related to each other in a meaningful way that has an impact on the utility of the solution. A good solution, along with a fitness function that accurately measures the utility of a solution, allows small good features of a solution to form independently of each other, rewarding the solution it is a part of with the gift of reproduction. Through multiple iterations of reproduction, these independently formed good features can be combined in children to form better and better solutions.

Given an appropriate schedule string representation (one where the positions of characters in the string relate to the order and processor in which tasks are run), and fitness function that accurately evaluates the efficiency of a schedule, it is reasonable to suspect that a genetic algorithm could be used to effectively solve task scheduling problems as described in the previous section.

Algorithm

In this section we describe in detail the genetic algorithm developed and used during our research. To accomplish this, we will describe each of the components key to genetic algorithms.

Solution Representation

A schedule is represented as a list of lists of integers. Each inner list represents a processor, and the integers in the list represent the tasks (in order) that will run on that processor. The representation for the four task schedule in the figure below would be [[1,4], [2], [3]].

Fitness Function

The schedule is first inflated (the process of setting the start times of the tasks) to satisfy procedural constraints. The optimal inflation (each task starts as early as possible) can be computed efficiently by considering the DAG (directed acyclic graph) formed by the procedural constraints on the tasks. After the schedule is inflated, the total time and prioritized throughput measures can be easily computed, as well as any violations to temporal constraints. These factors are then weighted and used to determine a fitness score for the schedule.

Reproduction

Reproduction between two schedules S1 and S2 involves first randomly choosing a crossover index. Then, for each of the processor lists in S1 swap the sublist starting at the crossover index with the sublist starting at the crossover index of the corresponding processor list of S2.

Mutations

A mutation of a schedule involves choosing two tasks from any processors and swapping them.

Implementation

Once the algorithm was built, it was time to start building a web framework to consume it. This allows for easy visualization of the algorithm and sharing of the functionality without requiring users to install anything on their machines.

Backend

The backend is written in Python and utilizes a web microframework for serving up the resources as well as providing endpoints to tie into the algorithm. The web framework accepts a JSON dictionary of the tasks and constraints as specified by the frontend. It converts this dictionary into a format recognized by the algorithm and calls the algorithm with the given input. Upon completion of the algorithm, it returns a JSON formatted representation of the most optimal genome to send back to the frontend.

Frontend

Besides the actual algorithm, the frontend is the most complicated part of this project. The frontend will need to provide two things, a way to provide input to the algorithm and a way to visualize the output of the algorithm. The input will be fairly straightforward with global constraints being input with checkboxes and text inputs. An example of the output formatting is shown in Figure 1 below.

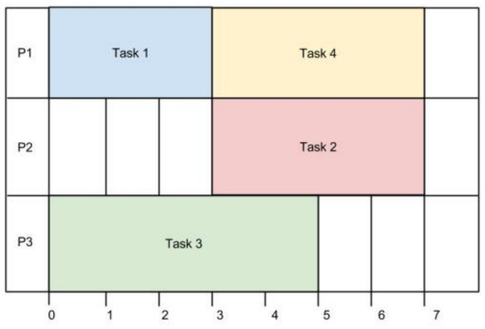

Figure 1

Results

Coming soon

Conclusion

Coming soon

References

Coming soon