A Genetic Algorithm for Task Scheduling

Strickland, Sean and Sanders, Theisen
Iowa State University

Abstract

Task scheduling is an optimization problem where a set of tasks are allocated to a finite number of processors. This has been a prevalent concept in computing for a long time, and has proven difficult because the problem is NP-Complete for more than two processors. Due to this complexity, a heuristic approach is required to effectively tackle the general task-scheduling problem. In this paper we explore using a genetic algorithm for this purpose. Genetic algorithms are local search algorithms in which a population of solutions is evolved over generation to produce more optimal solutions. We initialize this population using random mutations of the solution acquired from a simple minimum completion time heuristic. The population is then evolved over generations; with each solution's fitness measured by the total time (time for schedule to complete) and prioritized flow time (sum of task completion times weighted by their priority) measures. Our results show that this process can be effective at producing good solutions to the general task-scheduling problem.

Introduction

We began this project with three goals. The first was to apply a genetic algorithm to a well-known NP-Complete problem. The second was to learn more about genetic algorithms ourselves. Lastly, we wanted to provide a way for others to easily understand what genetic algorithms are, why they are useful, and how they work. Task Scheduling became our problem when we realized how applicable it is to the computing world and that since an optimal solution cannot be found in polynomial time, a genetic algorithm could provide a good solution compared to other common heuristics.

Problem

The problem of task scheduling is something that can take a very different form depending on the application. Here we will define our task-scheduling problem.

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.

The efficiency of a schedule is determined by two measures. The first is total time, which is the time it takes for the longest running processor to finish executing tasks. The second is the prioritized flow time, which is the sum of all the completion times of the tasks, weighted by priority. By minimizing total time we know that a schedule finishes as quick as possible; by minimizing prioritized flow time we know that all task are being executed as quickly as the can in the schedule with higher priority tasks being executed first.

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 constraints are constraints on tasks such that they cannot be executed until other tasks complete. Temporal constraints are time constraints that can be placed on the schedule.

The complexity of this task scheduling problem has been shown to be NP-Complete (Garey, Johnson, and Sethi 1976) for problems with three or more processors. This means that no optimal solution can be found in polynomial time. So we implemented a genetic algorithm to try to find a suboptimal solution such that that solution is good and the run time is less than that of most other heuristics.

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 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 some number of new solutions for the next generation. In addition, there is a random chance that an individual solution may mutate (be modified slightly) before the next generation. A fitness function is then used to determine which solutions survive to the next generation.

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 survive to the next generation. A good solution 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 fitness 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, likely rewarding the solution it is a part of with the gift of surviving and reproducing. Through multiple iterations of reproduction, these independently formed good features can come together in children to form more optimal solutions.

Given an appropriate schedule representation (one where the positions of tasks in the representation 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 effectively in task scheduling problems described in the previous section.

The 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 tasks. Each inner list represents a processor, and the tasks in the list represent the tasks (in order) that will run on that processor. An example representation for an eight task schedule is shown in figure 1.

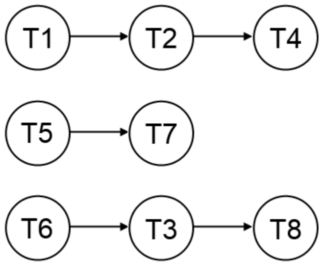


Figure 1 - Representation of solution

Initialization

Initialization involves building up the initial population. The first solution in our initial population is created using a simple minimum completion time heuristic. The minimum completion time for a task is recursively defined as the duration of the task plus the maximum of the minimum completion times of the task's dependencies, or just the task's duration if the task has no dependencies. The tasks are then sorted by their minimum completion times in ascending order and assigned to processors in a round-robin manner. This prevents any direct dependency violations and hopefully produces an initial solution with some optimal qualities. The rest of the population is then produced from random mutations of the initial solution.

Fitness Function

The fitness function evaluates the optimality of a schedule by computing a weighted sum of the total time and prioritized flow time measures mentioned above. 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 flow time measures can be easily computed, as well as any violations to temporal constraints. These factors are then subtracted from a total time bound and prioritized flow time bound computed during initialization, weighted, and added to determine a fitness score for the schedule.

Selection

The selection process is responsible for selecting the solutions in the population that get to survive to the next generation. This is done by using a roulette selection method in which each solution has a random chance, proportional to its fitness value, of surviving to the next generation. A fitness value of zero is reserved for invalid solutions, such as solutions that have obtained multiple of the same task through the reproduction process. Such solutions have zero chance of surviving to the next generation.

Reproduction

Each generation, each solution has a random chance of being involved in reproduction. Reproduction between two solutions involves first randomly choosing a crossover index, as represented by the vertical dotted line in figure 2. This process splits the two schedules into four pieces. Diagonally adjacent pieces are then put together to produce exactly two child solutions, which are immediately added to the population.

Mutations

Each generation, each solution has a random chance of being mutated. Mutation of a solution involves choosing a random task in the schedule and moving it to a random position in the schedule. One constraint we put on the mutation process is that we only randomly choose from

Table 1 - Results

Test #	Tasks	Dependencies	Processors	MCT Total Time	MCT Flow Time	GEN* Total Time	GEN* Flow Time
1	3	0	2	5	25	4	23
2	6	3	2	16	211	15	201
3	10	5	3	16	433	14	417
4	15	8	3	17	495	15	455

dependency obeying positions, meaning for example we will not place a task T2 before task T1 if T2 depends on T1. The diagonal dotted arrows in figure 2 represent the process of mutation.

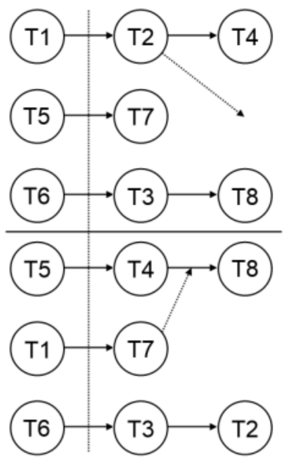


Figure 2 – Selection of crossover index

Implementation

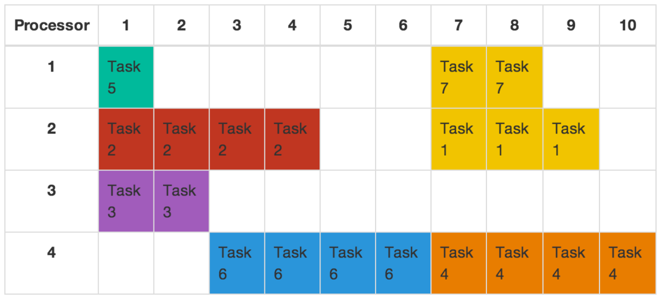
Our algorithm was built with the Python programming language. This was chosen because we were both somewhat familiar with they language, and it has been shown that the Python language promotes quick development and fast turn around time. Additionally we were familiar with several web frameworks written in Python that would be able to easily support our single page web application. It made sense to choose a single language for the backend process.

Backend

The backend is written in Python and utilizes a web microframework for serving up the static 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 upon the algorithm to generate an optimal schedule. Upon completion of the genetic algorithm, it returns a JSON formatted representation of the optimal schedule back to the client.

Frontend

The frontend provides two important things to the application. The first is a way to input tasks and define the algorithm constraints. This has been accomplished through the use of simple web forms, with the schedule updating on every constraint change. The second thing the frontend provides is a way to display a task schedule. This has been accomplished using the AngularJS MVC framework and setting up a table element that automatically mirrors a JavaScript object representing the best schedule that was found by the algorithm.



Processor	1	2	3	4	5	6	7	8	9	10
1	Task 5						Task 7	Task 7		
2	Task 2	Task 2	Task 2	Task 2			Task 1	Task 1	Task 1	
3	Task 3	Task 3								
4			Task 6	Task 6	Task 6	Task 6	Task 4	Task 4	Task 4	Task 4

Figure 3 - An example of a task schedule in the UI

Results

During testing we wanted to find out whether our genetic algorithm was able to produce more optimal solutions than the minimum completion time heuristic used during initialization of the algorithm. This would provide a good measure of whether genetic algorithms can be used effectively to improve the optimality of solutions in task scheduling. We arbitrarily chose test inputs of varying complexities and used the total time and prioritized flow time measures to grade the results of each algorithm.

The results, along with relevant details of each test are recorded in the Table 1 on the previous page. In general, the genetic algorithm was able to improve on the solution produced by the minimum completion time heuristic, especially in the prioritized flow time measure.

This improvement is more dramatic for more complex test cases with a greater number of dependencies and more processors for allocation.

Conclusion

As with all genetic algorithms, the performance and optimality of solutions is in the fine-tuning. This is something we encountered as we tweaked the weights used in the fitness function and decided on the constraints enforced during reproduction and mutations. However, given an appropriate solution encoding and accurate fitness function, it is reasonable to believe they could be used to effectively find optimal solutions in very large solution spaces, such as the general task scheduling solution space. Our results support the idea that genetic algorithms could be valuable local search tools for producing good solutions to task scheduling.

References

Garey, M. R., D. S. Johnson, and R. Sethi. "The Complexity of Flowshop and Jobshop Scheduling." Mathematics of Operations Research 1.2 (1976): 117-29. Print.

Kaur, Kamaljit, Amit Chhabra, and Gurvinder Singh. "Heuristics Based Genetic Algorithm for Scheduling Static Tasks in Homogeneous Parallel System." International Journal of Computer Science and Security 4.2 (2010): n. pag. Print.