SOFTENG 306 Project 1

Group 13 - Notice me Sinnenpai

Peter Lindsay, Yuno Oh, Sheldon Rodricks, Tushar Thakur, Elisa Yansun

1 Algorithm

We created a DFS branch-and-bound backtracking algorithm to solve the optimal task scheduling problem. DFS was chosen over other algorithms as it avoids storing many partial schedules in memory, unlike an algorithm such as A*.

For every free candidate task t, we try to schedule t on all processors p at the earliest possible time. This is reflected in the new state. This includes updating the set of free nodes, the time the processors are available, the in-degree of t's children and a few other pieces of information that is used for optimisation. We then recurse to explore the next state. The base case is when there are no more tasks to schedule on any processor. At this stage, we update the best schedule if the current schedule has a lower finish time. After this, we revert the changes made to the state. The pseudocode is shown below:

```
Optimal(Set of candidate tasks Tasks, Set of processors Processors):
If there are no candidate tasks:
      If current schedule is better than best schedule:
            Update best schedule
      Return

For all candidate tasks, t:
      For all processors, p:
            Schedule t on the earliest possible time on p
            Update free tasks, and processor state
            Optimal(Tasks, Processors)
            Revert state to original state in this iteration
End
```

1.1 Backtracking DFS

Each partial schedule is represented by a ‘state’ consisting of multiple data structures. Duplicating and modifying the current state at each iteration is memory intensive, and slow. Instead, we transform the existing state into the next state, which we recursively explore. Once the recursive call finishes we revert the changes, transforming the state back into the original state. This allows us to explore all partial schedules while only maintaining one instance of the state.

1.2 Heuristics

1.2.1 Task Exploration Order

Free tasks are scheduled in increasing order of their b-level value (Section 1.3.4). Tasks with smaller b-levels will be scheduled before larger ones. This provides us with a heuristic to pick tasks, raising the probability of creating more-optimal schedules early. This improves the average best schedule allowing us to prune more schedules, reducing the search space. We chose this order because it provided the best average run time across many different types of task graphs compared to other orders.

1.2.2 Greedy

We reused our heuristic algorithm from milestone one as a greedy task scheduler. The schedule it produces is used as an upper bound for our optimal scheduling algorithm. This allows the algorithm to begin pruning partial schedules using the upper bound from the beginning of the algorithm, and reduces the number of states it has to explore.

The greedy solution schedules a free task on the processor with the earliest finishing time. We run this algorithm five times, with different task exploration orders, and take the best schedule produced. This improves the initial greedy solution, and ensures that we consistently get decent greedy solutions across many different types of task graphs. The task explorations are: 1. The order tasks become free, 2. Decreasing b-level, 3. Increasing b-level, 4. Decreasing task length, 5. increasing task length.

1.3 Data Structures

1.3.1 HashSets

HashSets provide constant time lookup for a given key. We use HashSets to store nodes that we have seen before to check for equivalence, and to store the hashcodes of partial schedules to avoid duplication.

1.3.2 Linked Lists

We used a linked list to store the free tasks for a given partial schedule. A free task is one that has no in-coming edges. We use a linked list because it can behave like a dequeue. We can remove a task from the head of the queue and add it to the tail when we have finished trying to schedule it on all processors.

1.3.3 Arrays

We use arrays to store the state of tasks and processors. Two arrays represent the state of the tasks; one array stores the finish times for all tasks, and another stores the processor the tasks are run on. To represent the state of the processors, an array stores the earliest available time for each processor. We also use arrays to store information about the edges, weights, and transfer costs.

1.3.4 Classes

We modularised our code a lot with many different classes. Some of these classes, such as `Schedule` and `Task`, acted like data structures. They helped encapsulate the information for a schedule and a task respectively. The classes also provided more utility than typical data structures because we could implement `Comparable` with its methods such as `compareTo` in `Task`, which helps us sort `List<Task>` easily.

1.4 Pre-Processing

1.4.1 Node Duplication

Nodes that are equivalent from the start of the algorithm will stay equivalent throughout the run time of the algorithm. Therefore, it is safe to pre-calculate for each node, the nodes that are equivalent to it. An array list of equivalent nodes to node `n` is kept in index `n` in an array and used within the algorithm for pruning purposes.

1.4.2 Bottom Levels (B-Levels)

The b-level of a node, i.e. its max length to an exit node, stays constant throughout the running of the algorithm. It is safe to pre-calculate these and store them in an array.

1.5 Pruning

Since our search space is exponential, we need to find pruning methods such that it becomes manageable for us to find the optimal schedule in a reasonable amount of time.

1.5.1 Partial equivalent solutions

The same order of tasks on processes may reoccur from the exploration of different states. As such, we only need to explore these states once.

Given a set of free tasks `{a, b, c}`, suppose that we want to schedule task `a` on the processor `i`. If `a` has no children, then scheduling `b` on any processor less than `i` in the next recursive call will result in the algorithm exploring the same partial state twice. This is because scheduling `a` on processor `i`, then `b` on processor `i+1`, is the same as scheduling `b` on processor `i+1` first, and then `a` on processor `i` (in the next recursive call).

1.5.2 Fixed Task Order (FTO)

Suppose we are in the process of scheduling our tasks. Let us call the list of tasks where there are either no dependencies, or their dependencies have been completed, our list of `candidateTasks`. We can fix the order of tasks in our task graph if the `candidateTasks` fulfil several conditions:

1) All the `candidateTasks` must have either no child or the same child, and at most one child.
2) All the `candidateTasks` must have either no parent or their parents are scheduled on the same processor, and at most one parent.
3) The `candidateTasks` can be sorted such that the list fulfils the following two conditions. This will be the fixed task order.
    a) The tasks in the list are in non-decreasing data ready time. `Data ready time = finish time of parent + communication cost of parent to the task`.
    b) The tasks in the list are in non-increasing out-edge costs. `Out-edge cost = communication cost of task to its child`, or 0 if the task does not have a child.

The fixed task order means that among the tasks in `candidateTasks`, an optimal solution can contain these tasks scheduled in this order. By fixing the task order, we are able to prune our tree by a factor of the number of tasks in `candidateTasks`, as we no longer need to check every single ordering. Scheduling tasks in non-decreasing data ready time ensures the minimisation of idle time of processors, and the scheduling of tasks in non-increasing out-edge costs ensures that the start time of any tasks that depend on our set of `candidateTasks` can be minimised.

Once we get an FTO, we know that we can schedule the first task in our FTO safely. However, once the first task is scheduled, and if it has a child, this child joins `candidateTasks`, and our `candidateTasks` may no longer form a valid FTO. FTOs incur extra computation costs that generally max out at a couple hundred milliseconds for the average graph. However, they can vastly decrease the computation time of special graphs which were previously unsolvable within the 30-minute time frame of the client.

1.5.3 Load Balancing

The load balanced time (LBT) is the remaining time if all the remaining unscheduled tasks are spread evenly amongst the processors, not including communication costs. `LBT = ceil(sum(unscheduled task durations) / number of processors)`. Since the LBT is a minimum bound on the finish time of the current schedule, if `LBT + earliest start time > current best time`, we know that the current schedule can't possibly be an optimal schedule.

1.5.4 B-Levels

A B-level of a node is the sum of its run time plus the maximum path from itself to an exit node. We can use the B-level of a node to underestimate the finishing time of the optimal schedule. The underestimate for the finishing time of the optimal schedule if we want to schedule task `i` on the processor `j` is: `earliest start time of task i on processor j + B-Level of task i`. We can guarantee that this estimate is an underestimate because all the descendants of node `i` must be scheduled strictly after the finish time of `i`.

1.5.5 Latest Processor Finishing Time

We stop considering a state if the largest processor finishing time is greater than the current best schedule end time. The initial best schedule is our greedy one discussed above, and it gets updated if we find better schedules in our algorithm.

1.5.6 Processor Normalization

Two processors are isomorphic if they have no tasks scheduled on them. Scheduling a task on multiple isomorphic processors produces equivalent states. In our algorithm, within a given state, we check if a task has previously been scheduled on a processor with a finish time at time 0. If it has, and the current processor we are considering to schedule it on is isomorphic we continue to the next processor.

1.5.7 State Duplication Avoidance

Our algorithm can create duplicate schedules at different times when it's running. Duplicate schedules are valid schedules that have exactly the same tasks that are scheduled at the same time on processors. We use a `Stack` to store the list of task IDs and their starting time on a processor in the current schedule. We then use a `HashSet` to store the `Stack`s. `HashSet`s are unordered, so if we had tasks that were scheduled exactly the same but just on different processors, that would be considered a duplicate schedule.

We use the default hashing algorithm on Java to store the information on schedules that we have seen so far. If we have a duplicate schedule, then we can prune the rest of the branches from this schedule because all possibilities should have already been covered by a previous schedule.

Hash collisions is a problem which arises with this approach. The `hashcode()` method on a set returns an integer which represents the hash code value for that set. The hash code of a set is the sum of the hash codes of the elements in the set. This ensures that `s1.equals(s2)` implies that `s1.hashCode()==s2.hashCode()` for any two sets `s1` and `s2`. If two sets are equal, they will always return the same hash code. However, it is also possible that two fundamentally different sets `x1` and `x2` return the same hash code, which is what we call a hash collision.

With a sufficiently sized input graph, it is inevitable that different partial schedules return the same hash code due to the pigeonhole principle, as the size of an integer in Java can be less than the number of states. As such, it is possible that the solution incorrectly prunes sections of the graph where it believes that the schedule has already been seen when it hasn't, and if that section contains the optimal solution, we could return a non-optimal solution.

Despite this, we have decided to use hash codes for duplication detection in our solution for the following reasons:

1. Given the relatively small nodes and processors the solution is expected to run on, the likelihood of a scenario in which all optimal schedules in the graph are incorrectly detected as duplicates is minimal.
2. The solution run time is significantly improved through the use of state duplicate avoidance via hash codes.
3. Without the duplication detection, it is possible that we need to explore many more states, and potentially run out of memory or run overtime. We believed that this option would satisfy our client's needs the most.
4. Our implementation of state duplicate avoidance is easily detachable from the rest of the solution if the client wishes to forego the sizable speed increase.

1.5.8 Node equivalence

Two nodes are equivalent if they have the same duration, the same parents and children and the edge costs between their parents and children are the same. For a given state, there is no point in scheduling two identical tasks on the same processor. In our algorithm, when considering if to schedule a task on a processor, we check to see if we have already scheduled an equivalent task on this processor.

1.5.9 Single processor

If there is only one processor, then all tasks should simply be scheduled sequentially on the processor with no idle time. The finish time of the optimal schedule is the sum of all the durations of the tasks. All we need to do is to find a valid order to schedule the tasks.

1.6 Used Libraries

GraphStream was used for input and output parsing for dot files. This gives us a robust method of dealing with all valid dot graphs, and reduces development time to deal with input and output dot files.

Apache Commons CLI was used to parse command line arguments. This helped deal with user options dynamically.

2 Parallelisation

2.1 Approach

For multiple partial schedules to be explored in parallel, the threads in charge of exploring those states must not interfere with each other. To ensure this, we need to give each thread its own copy of the state which it can modify. To do this, we created a SearchState class which consists of fields that keep track of the free tasks in the state, in-degrees of tasks, task start times, processors that the tasks are scheduled on, processor finish times, and the sum of all the durations of tasks yet to be scheduled; this is similar to the data structures used in the sequential algorithm (section 1.2). This class has a method which returns a deep copy of itself, which is used to give each thread an independent copy.

The parallel algorithm is similar to the sequential one. Given a state, we try to schedule all tasks on all processors. Therefore, in the process of scheduling a task, we create more computational work by modifying the state, deep copying it, recursively exploring it, and finally backtrack to try something else. Finally, we pass the created work of searching a state to a pre-created ThreadPool, which distributes the tasks among the threads in the pool.

Pseudocode (Work refers to the computation done by each thread, including the state it searches):

```
Optimal(State s):
If there are no candidate tasks:
      If current schedule is better than best schedule:
            Update best schedule
      Return

For all candidate tasks, t:
      Initialize a list workList of Work.
      For all processors, p:
            Schedule t on the earliest possible time on p
            Update free tasks, and processor state in s
            Duplicate s to create s1.
            Add Work of searching s1 to workList
            Revert s to original state in this iteration
      Submit workList to thread pool
End
```

2.2 Technology

We used the `ForkJoinPool` and `RecursiveAction` classes. `ForkJoinPool` is a type of `ExecutorService` (Thread pool) which holds queued tasks and distributes them to the threads that it contains/manages. `ForkJoinPool` is capable of scheduling `ForkJoinTask`s on the member threads. `ForkJoinPool`s, unlike general `ExecutorService`s, are optimized for tasks that create their own sub-tasks. This makes them useful when parallelizing our recursive natured algorithm. `RecursiveAction` is a class which implements the `ForkJoinTask`, and encapsulates the work a thread must do.

2.3 Implementation

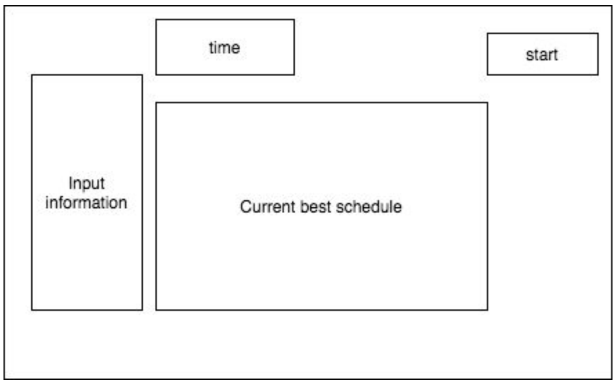
We create an inner class called `RecursiveSearch` which extends `RecursiveAction` and overrode its `compute()` method. This compute method is called by the thread assigned to run it and contains code to try to schedule all free tasks on all processors. For this, two for-loops are necessary. Within the processor assigning loop, we change the state object to reflect the scheduling of a particular task, duplicate the state object, use it to create a recursive search object and add it to the list of tasks to search. Once we have finished this loop, we send the list of tasks to the ForkJoinPool, and backtrack the current state which adds them to its internal queue from which threads that are free can remove and run them.

The outer class essentially sets up the initial state and then hands control over to the inner `RecursiveSearch` class. The fields `bestStartTime`, `bestScheduledOn`, `bestFinishTime` and `seenSchedules` are global variables that are used by all threads. Because of this, they are only used within synchronized blocks to prevent concurrent issues. We use class level locks to ensure all instances of `RecursiveSearch` are synchronized.

3 Visualisation

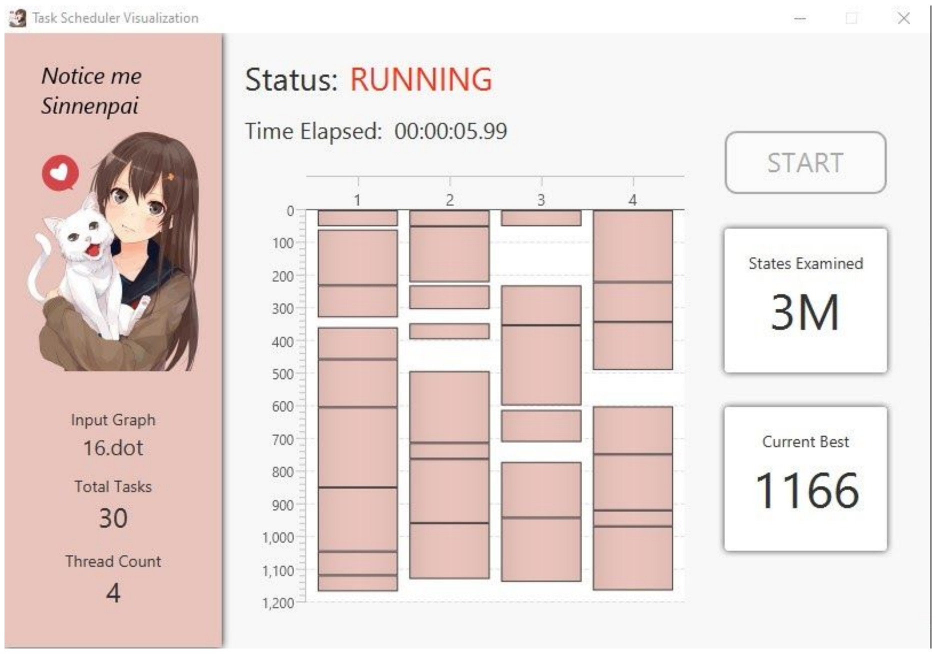
3.1 Concept/Design

The goal of the visualisation was to display a meaningful representation of the algorithm's search for the optimal solution. To achieve this, we wanted to provide the user with live updates of the intermediate results of the algorithm. We designed our GUI to be unique, interesting, and not overload the user with irrelevant information.



3.2 Information Displayed

The graphical user interface (GUI) that we created is shown below:



The left section of the GUI contains static information about the inputs, which confirms to the user that they are correct. Our goals of uniqueness and simplicity are reflected in the logo and colour scheme chosen.

The main section of the GUI displays the current best schedule found by the algorithm. This is also updated in real time to indicate the progress of the algorithm. It also allows the user to visually understand the nature of the search and how different scheduling of tasks improves the finish time.

Other dynamic information we display includes the status, running time, the number of states examined, and the finish time of the displayed best schedule.

3.3 Implementation

We used JavaFX and Scene Builder to easily integrate the GUI with the rest of the project and speed up development. This allowed the logic of the GUI to be separated from the visual components in a controller class and fxml file respectively. To convert the project into a JavaFX application, we created a `Visualiser` class which encapsulates the process of launching the GUI.

Communication between the GUI and the solution was handled with a wrapper class. This class contains an instance of a solution which it runs. It also extends Java's native `Thread` class which was necessary to provide information to the GUI from the solution via polling. While the solution runs, it updates fields which are accessible to the wrapper class. When the thread is polled, the wrapper class provides an interface by which the GUI is passed the necessary information it needs to update.

Another reason for using a wrapper class was due to the polymorphic nature of the solution object. The wrapper class can wrap both the sequential solution and the parallel solution object; the GUI does not need to care about which implementation of the solution it is communicating with. From a developer's perspective, this was beneficial as it reduced code duplication and simplified the interface between the logic of the algorithm and the GUI.

3.4 Sequential vs Parallel

We deemed it important to maintain a consistent user experience, regardless of whether we are running the sequential or parallel version of the algorithm. Thus, we decided to maintain the same GUI for both versions. This achieves consistency, as well as reducing development time as it integrates well with the aforementioned class structure. To distinguish between the two, we included a label in the GUI, which shows how many threads are being utilised in the computation of the algorithm.

4 Testing

Task scheduling is a complex problem, and it is difficult to conclude that our solution is 100% correct, since we can't feasibly test for every single scenario. However, we conducted many tests to ensure that our solution is valid.

4.1 Solution Validator

We created a `SolutionValidator` in Milestone 1 that tests whether the generated solution is a valid one. A valid solution has the same tasks, durations, and weights of the input, and the tasks are arranged on the processors such that dependencies are respected. In Milestone 2, this class was extended to directly test the input and output dot files using the `scheduler.jar`, because we wanted to directly test the files that the client would access.

4.2 Random Graph Generator and Tester

We created a `RandomGraphGenerator` that takes in a set of parameters such as the number of graphs to create, the minimum and maximum number of tasks, and the maximum weights of the tasks and edges in the graph to generate valid input .dot files. Our `RandomGraphTester` takes these input files in and creates output files using `scheduler.jar`, then passes them through the `SolutionValidator` to verify that our solution is valid.

We tested our task scheduler on thousands of random graphs and it has always yielded valid results. This also provides a good indication of the speed of our scheduler on different types of graphs. When 1000 graphs were randomly generated with 20 tasks each and a random number of edges between 0-210 edges, our solution could process them within 20s each.

4.3 Unit Tests

We have created JUnit test cases in `Tester` that cover different aspects of the code. Output from these test cases is run through the `SolutionValidator`, and the output finish time is compared with a human-calculated best output. The test cases cover different types of graphs and is our best attempt at covering all cases of possible graphs we can manage by human calculation. The unit tests provided us a good base test for when we made changes to the code base.

We also found test dot files in an online repository which included their optimal times. We tested the dot files on our solution and compared outputs. Some of the online dot files were very difficult, with some taking over 10 minutes to compute, or not finishing during the time that we ran them for. Our solution yielded non-optimal results for only one of these dot files, with the problem being the hashing problem discussed in Section 1.4.7. The optimal solution had a finishing time of 810, and our solution gave a finishing time of 811.

5 Development Process

Our team followed the waterfall model for our development process. As part of this, we completed the majority of our planning in the initial stages of the project. This included the identification of tasks which could be done in parallel to one another, and the delegation of these tasks along with their deadlines.

5.1 Communication and Decision Making

In these team meetings, we evaluated assigned tasks and their progress. This time was also used to explain changes made to the project since the last meeting and confirm these changes. This ensured that every team member was fully aware of the entire project at all stages of development. Important decisions were made with a unanimous agreement in most cases.

Almost all development was done through pair programming. This helped ensure code quality and reduce the chance of introducing bugs. We used version control (git) as another measure to ensure this. Each feature was developed on a separate branch, and this was only merged to the master branch once the changes had been approved by another team member.

5.2 Conflict Resolution

There were some differing stances on what should be prioritised in the project.
One example of this was when creating the optimisation for equivalent schedules, using hashing (section 1.3.7). Although hashing greatly increased the speed of our algorithm, it may very occasionally produce the wrong solution. Due to this, some team members favoured the slower, deterministic solution. This was resolved through long-winded discussions about what was best for the client, and what was feasible within the time constraints of the project. Exploration through testing and implementation of different options also aided in the decision making. Ultimately it was decided that hashing was the best middle-ground solution between no hashing, which caused timeouts, and more collision resistant hash functions, which caused memory problems.

Another source of conflicting opinions was when considering how many optimisations to add to the algorithm. Due to the time constraints, we were unable to add every optimisation we explored in the planning and research stage of our project. The optimisations we added were also complex in implementation, and often introduced bugs into our code. For some optimisations, our team had different views on whether the benefit of the optimisation was worth the potential introduction of new bugs and increased time pressure. These conflicts were also resolved through discussions, where we were able to reach a mutual agreement. For example, we made the difficult decision to remove FTO pruning (section 1.3.2) from our parallelised solution.

5.3 Used Tools and Technologies

Due to the mandated lockdown, we primarily used online tools for communication. We used Discord and Zoom for voice discussions, with different voice channels emulating different meeting rooms. Google Drive was used for file sharing, and Git was used to manage branches and code. The GitHub issue tracker was also utilised to highlight bugs and requirements needed for the completion of the project. A wiki folder on GitHub also helped keep track of improvements made to the existing code. We used Maven to manage the dependencies of the project, which made the build for each team member's local repository consistent.

5.4 Team Cohesion and Spirit

Team cohesion and spirit was one of our biggest strengths. Each member was very invested in the quality of our project, and thus we were all on the same page about the standard of work we wanted to produce. Conflicts were resolved without any grievances and mostly resulted from each member's passion for the project. The workload was distributed evenly, as was the contribution in discussions.

6 Table of Major Tasks

Task	**Member(s)**	**Percentage Contribution**
Input/Output Parsing	Peter Lindsay	50%
	Sheldon Rodricks	50%
Valid Solution	Yuno Oh	50%
	Tushar Thakur	50%
Optimal Solution	Sheldon Rodricks	30%
	Yuno Oh	25%
	Tushar Thakur	25%
	Peter Lindsay	10%
	Elisa Yansun	10%
Optimisation	Peter Lindsay	33%
	Sheldon Rodricks	33%
	Elisa Yansun	33%
Parallelisation	Yuno Oh	33%
	Sheldon Rodricks	33%
	Tushar Thakur	33%
Visualisation	Yuno Oh	50%
	Tushar Thakur	50%
Testing	Elisa Yansun	75%
	Peter Lindsay	25%
DevOps (Maven)	Elisa Yansun	100%
Documentation	Everyone	Roughly equal amounts