INNOV8 DATA SOLUTIONS (TEAM 8)

Stefan Gorgiovski, Rebecca Lee, Kristy Tsoi, Sabrina Zafarullah, Isabel Zhuang

SoftEng 306 University Of Auckland

Algorithm

Algorithms Used

Two prominent branch-and-bound algorithms, Depth-First Search and A*, could have been chosen to solve this NP-hard scheduling problem. The algorithm chosen by Team 8 was A*, as it is known for being faster than Depth-First Search. However, Depth-First Search was still implemented to aid in testing.

A* Pseudo-code:

```
/GRAPH: JGraphT graph of all vertices and edges
UNEXPLORED: Priority Queue of unexplored partial solutions
EXPLORED: HashSet of explored partial solutions

For all starting vertices v in the GRAPH
    Create a partial solution for v
    Add to UNEXPLORED

Loop:
   Pop best partial solution ps from UNEXPLORED
        If ps is complete (contains all vertices in GRAPH)
            Found optimal solution, return ps
        Else
           Expand all children of ps
           Prune children
           Add remaining children to UNEXPLORED
        Add ps to EXPLORED
```

Depth-First Search branch-and-bound Pseudo-code:

```
GRAPH: JGraphT graph of all vertices and edges
BESTBOUND: Set to infinity
BESTSOL: Empty partial solution
Queue: queue of partial solutions

Create a partial solution PS for each starting vertex in GRAPH and add it to Queue
For every starting solution PS
    Perform DFS on PS
Return BESTSOL

Depth First Search:
    If minimum finish time of PS is greater than BESTBOUND
        Return (do not explore solution further)
    If PS is complete (contains all vertices in GRAPH)
        BESTBOUND = finish time of ps
        BESTSOL = ps
        return
    Else
        For all child partial solutions CPS do Depth first search
```

Heuristic Function

This A* search uses the two heuristics: one was based on the maximum time of all scheduled tasks plus their bottom level, and the other used perfect load balance plus the solution's idle time. The bottom level heuristic only needed to use the next vertex to be added, as the minimum finish time is also based on the parent solution's finish time, meaning that all previously scheduled vertices have already been taken into account. It retrieved the start time of the vertex to be allocated and added this to its bottom level to get a minimum finishing time.

The perfect load balance heuristic added the weight of all nodes to the total idle time for the new partial solution and divided the value by the number of processors that were present. The maximum of these two values and the minimum finish time of the parent solution would result in the minimum finishing time for the new partial solution. This is the value that was used to sort partial solutions in the priority queue.

Important Data structures

The important data structures used in this project was the DefaultDirectedWeightedGraph from JGraphT, PriorityQueue, HashSets and HashMaps.

The DefaultDirectedWeightedGraph was essential for storing the input as a directed graph as it also had supporting methods which allowed easy access to the required information of each vertex and edge. The PriorityQueue allowed unexplored partial solutions to be retrieved best-first while the HashSets were used to store vertices that had and had not been allocated to a processor.

The HashMaps stored a vertex and its information pair, with the information held in a class called AllocationInfo, which holds a vertices start time and allocated processor and is used during output file creation.

Pruning Techniques

To remove subtrees that are guaranteed to not be optimal during the search, two main pruning techniques were used. The first technique detected duplicate partial solutions by checking the closed set of explored partial solutions. Before adding a new partial solution to the priority queue, it would check if exactly the same solution had previously been explored. If so, it would not be added, otherwise it would.

The second technique prevents the creation of equivalent partial solutions when there was more than one empty processor. For example, adding one task to the first empty processor is equivalent to adding it to the second empty processor. As a result, the algorithm would only allocate a vertex to the leftmost empty processor and not to any others. With this pruning, only a new partial solution would be created for the first case and added to the priority queue. This eliminates a large number of partial solutions at the start of the process.

A crude upper bound was also calculated along with the two pruning techniques above. The crude upper bound was a summation of the weight of each vertex, representing the time taken if all tasks were running sequentially on the same processor. A solution would not be considered viable if its minimum finish time was larger than this bound.

Libraries Used

The libraries used in this project were JGraphT, GraphStream and JFreeChart. JGraphT provided a way to store the directed graph as an object while GraphSteam and JfreeChart aided in visualisation of the process. Using GraphStream meant that the graph could be easily displayed and JFreeChart assisted in the creation of a Gantt chart that showed the processor each task was added to, their start and finishing times, and the order they would be run.

Parallelisation

Parallelisation approach

The parallelization approach taken involves running the A* algorithm sequentially until the initial main PriorityQueue contains 1000 unexplored solutions. It then divides these partial solutions evenly among N PriorityQueues, one for each thread that will be created. The queues in each thread are then filled cyclically, that is, one solution is popped from the main queue and allocated to each PriorityQueue at a time, until the initial main queue is empty.

This main issue with this approach was the variance in execution time of the threads, as there was no use of load balancing. This is most likely the main reason for a less-than-ideal speedup as a thread can end up doing almost twice as much work as another thread in the worst case.

Queue Distribution Pseudo-code:

```
//Initialise
Queue[]: Array with N priority queues

i = 0
While mainPriorityQueue is not Empty
    Pop partial solution ps from mainPriorityQueue
    Add ps to Queue[i]
    Increment i
        If i equals length of Queue //Loop back to the first queue
            i = 0
```

Pseudo-code for Parallelisation:

```
runnables[] : runnables that perform an A* search
Threads[] : array of threads
for i = 1 to number of threads
        initialise runnables[i] with its own priority queue
        initialise threads[i] by passing in runnables[i]
        start threads[i]

initialise runnables[0]
run runnables[0]
wait for all threads to return a solution

compare the returned solutions and return the shortest
```

Splitting the Work

An array of runnables and threads are then created, each of which have their own PriorityQueue from the `Queue[i]` array created above. The threads, when started, run A* and generate a solution which is optimal given their starting PartialSolutions. Once all threads are complete, each solution is compared and the true shortest solution is returned by comparing their finishing times.

Synchronisation and Changes in Data Structures

Due to the nature of the design, the only synchronisation needed is at the point where the main thread waits for all of the background threads to finish using `thread.join()`. The closed set is the only structure shared between the threads, but an unsynchronized version is used as the contention between threads results in an increase in runtime despite the possibility that less solutions will be pruned. Other than the addition of more PriorityQueues, there were no other changes to data structures.

Parallelisation Technology

Options and Implementation

ExecutorService, Pyjama, Paratask and Java threading were all possible parallelisation techniques and were trialled in comparison to running A* sequentially. The approach above was the only one to perform faster than the sequential implementation of A* with the use of four threads. This version runs up to 30% faster when using a medium sized graph known to run for 5-10 seconds when run sequentially.

The alternatives trialled were:

- Using Java Threads with a shared PriorityBlockingQueue which resulted in up to 30% slowdown with 4 threads
- A nested for loop was used to expand a given partial solution. Using Pyjama and Paratask to parallelise this execution resulted in a total finish time which was 3 times slower than running A* sequentially.
- Using ExecutorService on the same nested for loop also resulted in a longer finish time. The time taken was around 10 times worse than running A* sequentially.

Parallelised for-loop:

```
expandPartialSolution(partialsolution)
for every free vertex V in partialsolution
    for every processor
        create new solution with V allocated to that processor
```

What was concluded from trialling these options on the nested for loop was that creating a partial solution occurs too quickly to be parallelised efficiently. In a test which took 1085ms, 176,220 partial solutions were created and so the overhead of parallelisation would counteract the benefit of creating the solutions in parallel.

Using a shared PriorityBlockingQueue also seems to simply result in too much resource contention between threads in order to be a viable solution. No input graph resulted in a significant decrease in time taken, and most took longer than the sequential time.

Visualisation

Concept

The intent of the live visualisation was to display the activity of the A* algorithm as it searched for the optimal schedule. Key aspects of the algorithm such as details of the open queue and close set, would need to be conveyed in real time, in which the statistics can be easily understood by the user. As a result, alongside displaying numerical statistics, the decision was made to display non-numeric data in the form of graphs as they can easily convey meaningful information to the user.

Components Displayed

In the A* Graph Visualisation window, the input graph was displayed with vertex and edge weights. As the .dot files are difficult to read, displaying the graph lets the user know exactly what the input looks like.

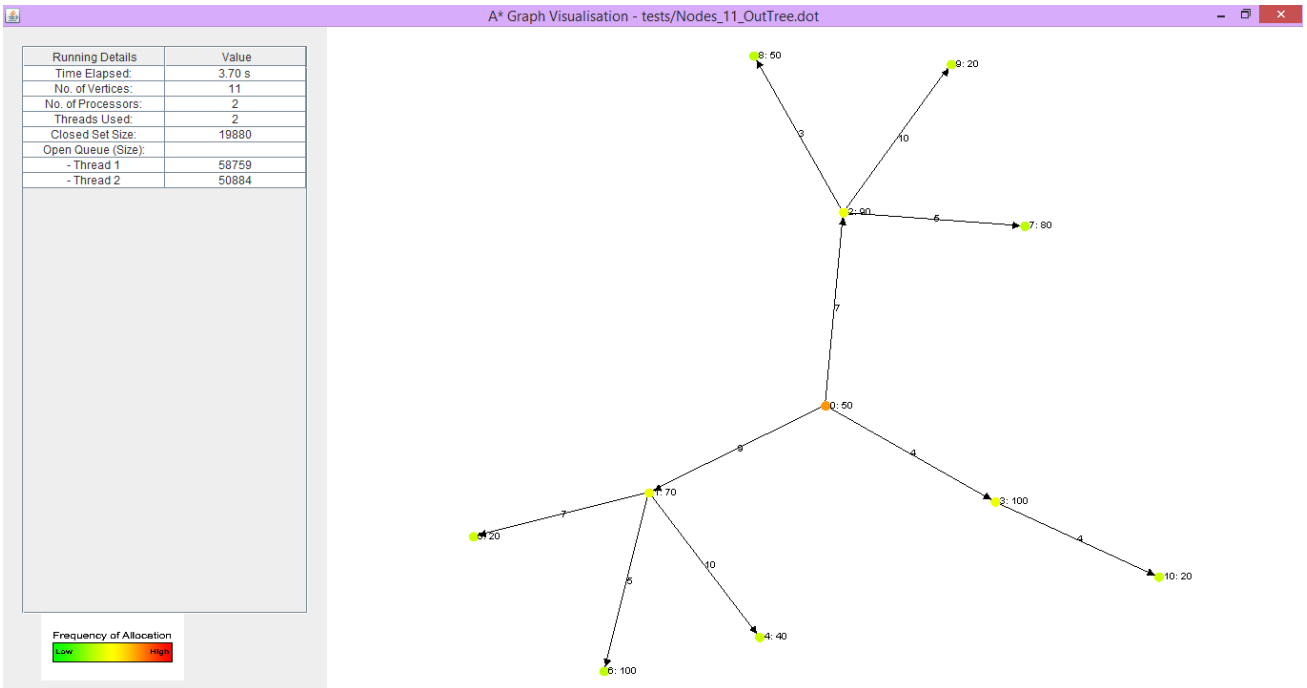
The colour of each vertex changes according to the number of times it has been allocated in a viable partial solution. Black vertices correspond to unused vertices, green represents vertices with a low frequency of allocation and red represents vertices with a high frequency of allocation. Thirty-one intermediate shades were created between green and red to achieve a better transition. The vertices were coded to advance one shade closer to red every 10,000 times they had been allocated in a viable partial solution. The frequency of 10,000 was chosen, as the visualisation was trialled against several inputs and this was the value which gave the smoothest transitions with larger sized graphs.

A key was also provided at the bottom left of the frame, to inform the users of what the colours represent.

Features Displayed:

Live search statistics:

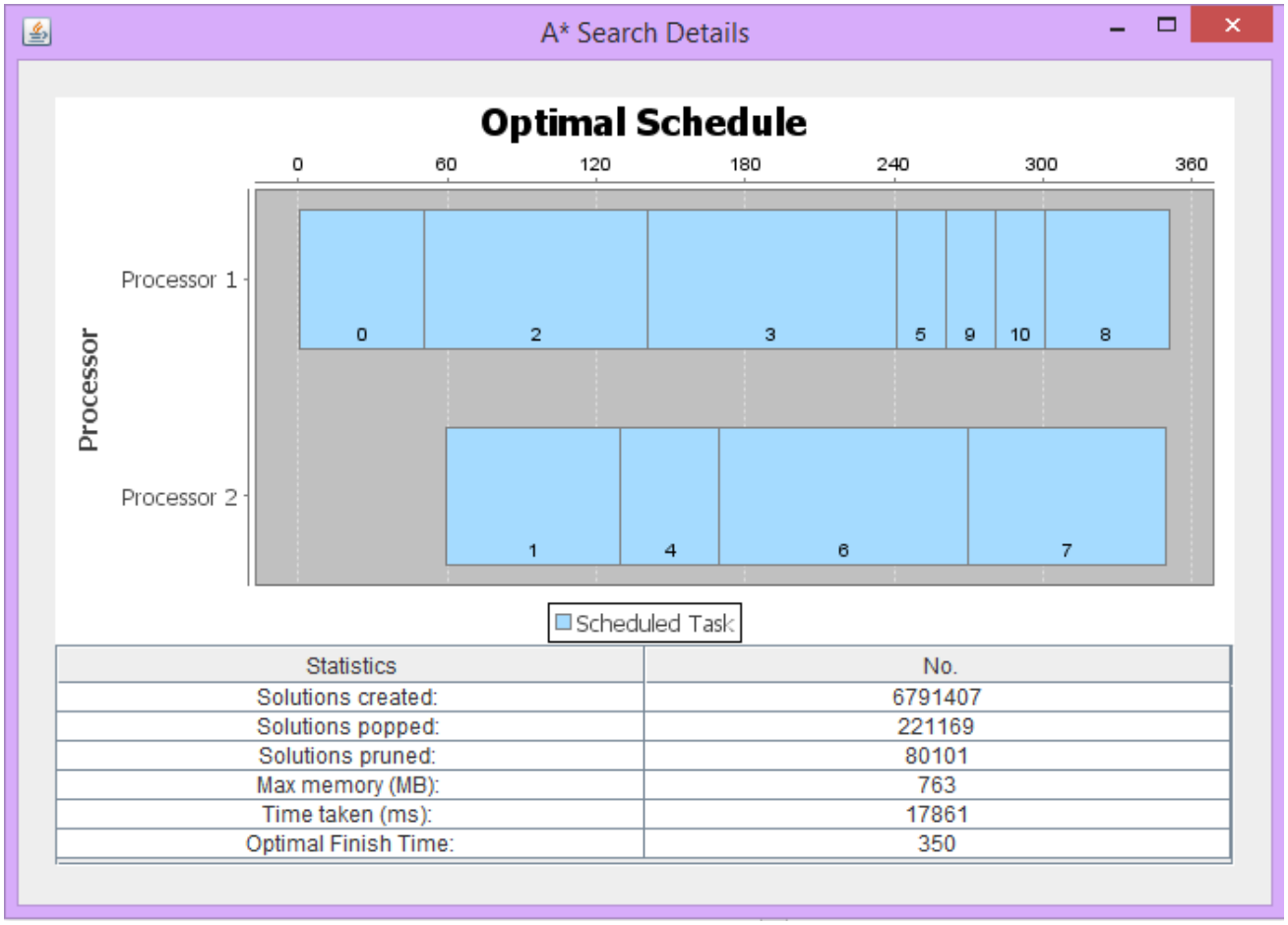
- Time elapsed
- Number of processors
- Number of nodes
- Number of threads
- Open queue size
- Closed set size



At the end of the search, a popup is displayed with details of the optimal solution:

- Total time taken
- Number of solutions pruned
- Maximum memory used in Megabytes
- Number of solutions generated during the search
- Number of solutions explored
- Finish time of the optimal solution

A Gantt chart of the final schedule is also displayed in the centre of the popup. It displays the scheduled tasks where each task's name, start time, finish time and processor allocation can be seen. This is done to improve user experience, so that the user will not have to read the poorly formatted output .dot file in order to understand the schedule.



Statistics	No.
Solutions created:	6791407
Solutions popped:	221169
Solutions pruned:	80101
Max memory (MB):	763
Time taken (ms):	17861
Optimal Finish Time:	350

Implementation

The visualisation frames were created using Java swing components, including JPanels, JTables and JLabels. The input graph was displayed using the GraphStream library and was instantiated during the graph parsing process. A built-in layout function was used for the graph which spaced out the vertices and edges, providing a clear and uncluttered display.

For the rest of the application to update statistics in the frame, an instance of GraphVisualistion was created upon initialisation, if the user had enabled visualisation of the search. Visualisation classes that extended the default algorithm classes were created which would maintain the current search statistics and update the interface so the unvisualised version did not perform unnecessary computation.

The method to update the open queue and closed set sizes were called every time a solution was popped or when a new solution had been added. Similarly, when a vertex was allocated to a partial solution, the value representing the number of times it had been used was incremented. Although in this case, the main algorithm only notified the graph instance every 10,000 uses so that the colours were not updated too frequently, which would cause the visualisation to lag.

The Gantt chart was implemented using the JFreeChart library. The GanttChart class receives the optimal partial solution as an input and creates Task objects for each vertex, storing its start and finish times. The data is then passed into the FinalDetails class. From here, the renderer for the Gantt chart is set to be an instance of GanttChartRenderer and the renderer for the date axis is set to be an instance of TimeAxis. GanttChartRenderer customises labels for each of the subtasks or vertices and TimeAxis renders the time values as integers instead of dates. Both renderers extend default renderers in the JFreeChart library and override some methods to allow customisation.

Sequential and Parallel Visualization

The difference between sequential and parallel visualisation was that in sequential, the size of the open queue of the main thread is displayed, as it is the only thread that is computing the solution. But in parallel, the open queues sizes of all the threads used are displayed. In parallelisation, each thread will maintain their own counters for the final details pane. When the computation is complete, the main thread will get the results from all the threads and add them together to get a total.

Testing

Process

Testing was performed throughout the implementation process of the project and can be separated into three different phases: Basic Milestone, AStar Implementation and Validation. The JUnit4 testing library was used to implement all the test cases.

Method of Testing

In the Basic Milestone phase, tests were implemented for the GraphParser and TopologicalSolution classes. The GraphParser class contains `parse()` that was used to create a weighted directed graph using the values read from the input file. TestGraphParser tests `parse()` to check that the graph created contains the correct number of vertices and edges based on the input file given.

For the basic milestone, a topological sort was used to easily produce a valid solution. TestTopologicalSolver checks that the algorithm correctly calculates a finish time which is equal to the sum of all vertex weights, and that all the vertices are added in the correct topological order.

The AStar, BottomLevelCalculator and PartialSolution classes were implemented during the AStar algorithm development phase. TestBottomLevelCalculator was created to test the BottomLevelCalculator class, which is used to calculate the longest path from a given vertex to any leaf vertex. It was vital that BottomLevelCalculator was working correctly as the bottom levels of vertices were used in a heuristic for the AStar algorithm, aiding in finding the optimal solution.

The PartialSolution class contained information regarding all the allocated vertices in the partial solution, as well as their start times and the processor they are allocated on. PartialSolution has the functionality to get the next available vertices to be added into the partial solution, calculate the start time of a vertex and the finish time of a partial solution. TestPartialSolution has tests for both the constructor functionalities of creating the initial partial solution and adding a vertex into an existing solution. The test class also checks that the class correctly retrieves the next available vertices and calculates the correct finish time of the solution.

Within the PartialSolution class there is a `verify()` method that was created to be used as a test throughout the implementation stage to check that in the final optimal solution, there are no overlapping vertices in a processor and that all vertices meets each of their parent's dependencies.

TestAStar was written to test the initial AStar algorithm implementation, but this test class no longer works as the original code that it tested was refactored so that the algorithm works more efficiently. When TestAStar was working, it was used to test the starting state of the solution and if an added vertex had the correct start time.

The branch and bound algorithm was implemented during the Validation phase. This was so that there was a way of checking that the final solution the AStar algorithm found was truly the optimal solution. TestBranchAndBound was written to test that the Branch and Bound implementation produced a valid solution. As with TestAStar, TestBranchAndBound no longer works due to code being refactored.

Test11NodeValidOptimalSolution, Test5NodeValidOptimalSolution and Test7NodeValidSolution were created as a replacement. They compare the optimal solution finish times of the Branch and Bound Algorithm and AStar Algorithm to validate the AStar algorithm.

Development Process

The development process

Before implementation, the client's requirements were analysed and discussed during the planning stage. If a change had been made in these requirements during a team presentations or interviews, the code could be altered at a later stage.

An iterative development process was used in this project, where testing was done after each milestone that was reached. The developers were allowed to go back to previous stages if there were any implementation errors that needed to be fixed.

A Gantt chart of the timeline was mapped out but this was not followed closely, as the tasks were longer than the estimated duration. The network diagram created during the planning stage was useful in recognising dependencies between tasks and to keep the project on track.

Communication

Communication mainly occurred face-to-face during weekly team meetings set up through online messaging. A team discussion would occur about code implementation and progress on allocated tasks. Whenever this was not possible, online messaging on a group chat was the mode of communication. Most decisions were made during these team meetings.

Conflict Resolution

The majority of conflicts that arose were based on a lack of communication about code functionality. This was addressed by holding a session where the code was explained.

Used Tools and Technologies

GitHub was the chosen method for source control with Eclipse being the IDE for writing Java code. For the final report, Google Drive was used as a means of collaboration.

Team Cohesion

The team worked well together without having many conflicts, however, any that did arise were due to lack of communication about tasks that had been completed and the function of the code that was created.

Task Contribution

Task	Stefan	Rebecca	Kristy	Sabrina	Isabel
Input	20%	10%	10%	20%	30%
Output	10%	10%	15%	55%	10%
A*	40%	10%	20%	10%	20%
Parallelisation	60%	10%	10%	10%	10%
Visualisation	10%	40%	10%	25%	15%
Validation	15%	10%	27.5%	37.5%	10%
Report	20%	20%	20%	20%	20%