Team 8

SoftEng 370 University Of Auckland

Innov8 data solutions

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

Contents

[Algorithm 1](#_Toc459720834)

[What algorithm(s), give pseudo-code 1](#_Toc459720835)

[Bound, heuristic function 1](#_Toc459720836)

[Important data structures 1](#_Toc459720837)

[Pruning techniques 1](#_Toc459720838)

[Used libraries 1](#_Toc459720839)

[Parallelisation 2](#_Toc459720840)

[Parallelisation approach ∗ How is work split? How is work synchronised? ∗ Change in data structures/additional data structures ∗ Pseudo-code – Parallelisation technology ∗ Threads, ExecutorService, Pyjama, Parallel Task, ....? – How was it implemented? 2](#_Toc459720841)

[Visualisation 3](#_Toc459720842)

[– Concept – What is displayed? Why? – How was that implemented? – Difference sequential and parallel visualization 3](#_Toc459720843)

[Testing 4](#_Toc459720844)

[What was tested? 1 – How was it tested? 4](#_Toc459720845)

[Development Process 5](#_Toc459720846)

[Describe your development process – Communication and decision making – Conflict resolution – Used tools and technologies – Comment on team cohesion and spirit 5](#_Toc459720847)

[Task Contribution 6](#_Toc459720848)

# Algorithm

### Algorithms Used (What algorithm(s), give pseudo-code)

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:

// Initialise

GRAPH: JGraphT graph of all vertices

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

//Calculate new possible partial solutions based on ps

Get all child vertices cv with all parent vertices in ps

Create new partial solution with cv that expands on ps

Add to UNEXPLORED

Add ps to EXPLORED

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

global unsigned best-bound = ∞

global Node best-solution = null

% Branch and bound

Node branch-and-bound():

Node root = make-root-node(init())

depth-first-branch-and-bound(root)

**return** best-solution

% Depth-first visit for branch and bound

**void** depth-first-branch-and-bound(Node n):

% base cases

f = n.g + h(n.state)

if f > best-bound **return**

if n.state.is-goal()

best-bound = n.g

best-solution = n

**return**

% depth-first recursion

**foreach** <s,a> in n.state.successors()

depth-first-branch-and-bound(n.make-node(s,a))

### Bound and Heuristic Function

A crude upper bound was calculated along with the two heuristic functions discussed in lectures. 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.

The first heuristic function used the next vertex to be added into the current partial solution. It retrieved the earliest time the vertex could start on any processor and added this to its bottom level. The second heuristic function added the time of the crude upper bound 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 would result in the minimum finish time for the new partial solution and it was then used to sort the partial solution 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 stored in a specific order 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 AllocatorInfo. AllocatorInfo which holds its start time and allocated processor for a vertex is then accessed during output file creation.

### Pruning Techniques

To remove unpromising subtrees during the search, two pruning techniques were used. The first technique detected duplicate partial solutions through accessing 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 previous been examined. If so, it would not be added, otherwise it would.

The second technique pruned equivalent partial solutions when there was more than one empty processor. For example, if adding one task to the first empty processor was equivalent to adding it to the second empty processor, only a new partial solution would be created for the first case and added to the priority queue. Therefore, eliminating a large number of partial solutions at the start of the process.

### 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 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 which contains unexplored solutions reaches 1000 elements. It then divides these partial solutions evenly among N PriorityQueues, one to each thread that is 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 is the variance in execution time of the threads, even through the 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 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

i = 0

#### 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.

### Synchronising 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. This version is much faster as it is not crucial if pruning is occasionally skipped rather than using a thread-safe HashSet which has contention between threads. Other than the addition of more PriorityQueues, there was no other changes to data structures.

### Pseudo-code

//Initialise

### 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-size graph known to run for 5-10 seconds 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. <Pseudocode>
* 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.

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.

The reason the alternatives failed is that there – blocking queue failure?

# Visualisation

### Concept

### Components Displayed (What is displayed? Why)

### Implementation (How was that implemented?)

### Sequential and Parallel visualization (differences)

# Testing

### 

### Components tested

### Method of Testing (How was it tested?)

# Development Process

### The development process

The development process followed in this project was similar to the waterfall model, however, it did not restrict the developers from going back to previous stages if there were any implementation errors that needed to be fixed.

This meant that there was still a requirements stage where the client’s needs were analysed and a proceeding implementation stage where code was created to fulfil these needs. However, if a change occurred in these needs during a team presentation or interview, the code could still be altered.

### Communication and decision making

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 was the mode of communication to each team member on a group chat.

Most decisions were made during these team meetings and online messaging.

### Conflict resolution

The majority of conflicts that arose were based on a lack of communication about code functionality. This was addressed and

Not enough comunicatio n about what has been coded and how they worked,

Sat and explained

### Used tools and technologies

### Comment on team cohesion and spirit

# Task Contribution