Test Results: S1

Performed tasks 1 and 4, skipped 2 and 3 due to time constraints.
Programming skills: java expertise

Introduction

During the explanation of SSE he was largely familiar with the concepts of graphs and those terms, the concepts of SSE were quickly understood.

Challenge 1

Task 1 - Run goat game and draw the state space

When drawing the state space diagram the occurrence of transitions to a known state was sometimes confusing because then no new state needed to be drawn. In the end without much help he was able to draw the correct state space diagram.

He did not run the exploration to termination, but stopped once the 'solution' state was drawn, because of this the final few states do not have all their outgoing transitions drawn. The transition leading from the solution to the state prior to it is missing.

He was able to deduce that the algorithm executed was BFS.

Task 2 - Change the algorithm to DFS

He searched through the code to understand the framework.
He used the inheritance of the classes and interfaces to figure out how the framework was built up, and did not pay much attention to javadoc. The naming scheme of classes was the most important method of discovery.

He ended up finding the correct frontier and static method.

Task 3 - Implement early goal termination

He first inspected the state representation class (GameState) and found the GameState::isGoal method. Now he wanted to know how he could use that to end exploration.
Solution 1: Change the next-function to not provide states if the given state is the goal state. This did not work since the exploration still had states left in the frontier once the goal was found.

Solution 2: Use the heap in some way
He wanted to test the heap for presence of a goal state and use that to figure out if the goal state had been found. This attempt was not fully implemented and did not work

Solution 3: Wipe the frontier once goal is found
(I had hinted at this point that the frontier would still contain states once the goal has been identified)
The next-function would test states to be the goal state and once found then the frontier would be cleared and the function would not return any new states.

(I hinted now that he should have used a behaviour for this)
Solution 4: Using a behaviour
He looked at the query::addBehaviour method and started to implement an anonymous implementation of the interface (ExplorationBehaviour) to do this.
I had to explain to him that he needed to use the tappable system and helped him setup the tap. He understood taps (the interceptors) as decorators from python.

Behaviours were difficult to grasp: the link between query events and behaviours was not obvious. The purpose of the AbstractBehaviour class also wasn't clear and only the base interface was used.

Challenge 2

Skipped

Challenge 3

Skipped

Challenge 4

First attempt was a manually crafted next function that optimizes the exploration. His next-function would only return the stepping-back option when near the top of the staircase. When the additional requirements were introduced his approach changed.
He looked at the implementation of the goatgame state and then implemented a similar next-function where all options are generated.

When wanting to optimize the order of the frontier he implemented his own frontier. He did not register TreeMapFrontier as a suitable frontier type. I think he may have read over it because the words did not trigger relevant to him.
His custom frontier implemented the base Frontier interface and only tracked a single state.

Unfortunately we ran out of time at this point.

Test Results: S2

Background: non computer science background
Performed challenges 1, 2 and 3. Challenge 4 was skipped due to being too difficult.

Programming skills: programming basics, java unfamiliar.

Introduction

Challenge 1

Task 1 - Run goat game and draw the state space

He was able to draw the state space without much difficulty.
He had drawn the transitions with lines instead of arrows.
He numbered the state discovery order.
He was able to figure out that the exploration was BFS.

Task 2 - Change the algorithm to DFS

He used autocomplete on the QueueFrontier to find the lifo frontier fairly quickly.
He knew that the frontier would have to be a lifo frontier because I hinted to this during the introduction.

Task 3 - Implement early goal termination

I hinted that he would need to use a behaviour to implement this.
I hinted that the framework had a set of standard behaviours but he did not end up looking through them.
He looked through the LogEventsBehaviour (already added to the query) source code to figure out what a behaviour could do.

Solution 1: Loop through the frontier to find the goal state
He wanted to loop through the frontier before or after the query execution. I clarified that this would not work because the frontier would not contain any states yet.

Solution 2: Use some query events to do something with states.
I started providing guidance with step-by-step building of a solution. Breaking the solution down into some steps:

- Perform some logic when we find new states
- Identify a goal state
- Stop the query somehow

He looked through the query class but skipped past the tappable events because he did not understand them. Once I explained that the tappable system was an event system that he could use to interact with the pipeline he picked the beforeStateEvaluation.
While he was trying to write the callback for the evaluation event he was looking through the documentation of the class, he did not see the link to the base class in the source-code and because of this missed the method to terminate exploration manually.
Once I hinted to this he was able to find the correct method and finish the custom behaviour.

Challenge 2

He understood the link between positions in the maze and states in the state space.

Task 1 - Determine the exploration method

He deduced that the method of exploration was BFS

Task 2 - Terminate the exploration when reaching the goal state

He did not spot the TerminateOnGoalBehaviour again when looking through the list of framework provided behaviours. I think directed him to the correct behaviour to see how usage of the behaviour would go.
He had difficulty building the predicate, he did not fully understand that it was a callback and how it was supposed to function.
Although the lack of documentation on the PlayerState class did not help in this case.
It became clear that java was a major obstacle for this participant in terms of completing the tasks.

Task 3 - Optimize the query

He came up with the idea to explore states nearest to the goal state instead of using the BFS order.
I hinted that there is a frontier that is able to do that and while he was looking through the frontiers he found the TreeMapFrontier.
This class was lacking in documentation on its function and how to use it.
He first tried to create an instance of it with the constructor, he ignored the error thinking it was because the invoke was not correct in arguments yet. Once he had fully setup the comparator then he realized that the constructor was private and could not be used.
He looked at the constructor and noticed the static methods below it and then tried to use the first one of them (withExactOrdering).
When executing the query the flaw with the exact ordering came up where cells that are transposed from each other in the grid would be considered equivalent.
He thought this was because he used the wrong frontier.
I had to explain this quirk of the exact ordering frontier and helped him correct it with the hashcode fix.
After that his solution worked fine.

Challenge 3

Task 2

The task was already completed in the default setup of the testset

Task 4

Implementing the sweep-line behaviour was very difficult.
The concept itself was not fully understood.
The documentation was used by the subject and explained what he had to do to get the SweepLineBehaviour to work to some extent.
When displaying the effect of the saw-tooth the subject noticed that the effect was not very notable in this use-case.
In hind-sight this test should have been re-designed to make it easier to implement SL, and highlight its benefits more.

Challenge 4

Skipped due to complexity

Test Results: S3

Studies master in computer science (ST track)
Had sufficient programming experience, not specifically java but was well capable.

Introduction

No notes

Challenge 1

Task 1 - Run goat game and draw the state space

He drew the state space correctly with directional arrows in both directions.
At the second step he started drawing the initial state again but realized that it was simply a transition back instead.
He was able to deduce that the algorithm was BFS.

Task 2 - Change the algorithm to DFS

He changed the lettering on the fifo function call and had then immediately implemented the required change.
I asked him to verify the change by running the simulation again and he ran it again. When running again the discovery of both states at the main branch in the state space threw him off for a second but then realized that the exploration only actually continued on one path.

Task 3 - Implement early goal termination

When looking through the standard behaviours he found the termination behaviour and was also able to implement the lambda fairly easily.

He verified that the solution worked, he figured out that manual termination meant that the exploration was over.

Challenge 2

Task 1 - Determine the exploration method

During the explanation of the interface he already noted that the exploration occurs as BFS.

Task 2 - Terminate the exploration when reaching the goal state

He applied the same behaviour again and was able to have the query terminate at 2219 steps.

Task 3 - Optimize the query with an ordered frontier

The purpose of TreeMapFrontier was not clear at first, he noted that the naming scheme only clarified the implementation details of the frontier and not its actual behaviour.
The private constructor of the TreeMapFrontier confused the subject, he eventually found one of the static factory methods.
He was able to implement a comparator easily enough.
His solution also suffered the equivalence-bug, once I explained why this was the case he was able to implement a solution to this.

Task 4 - Freely optimize the query further

He wanted to only let the actual solution trace be part of the frontier.
Because the maze was the same every time the program was run he wanted to try and only let the true solution enter the frontier.

- He identified the FrontierFilterBehaviour as a behaviour to do this in.
- Unclear how Query::addBehaviour and ExplorationBehaviour::attach() cooperate, the AbstractBehaviour was also not clear in purpose and utility.

I explained that the TerminateOnGoalBehaviour had an event that you could use to retrieve the solution of the maze after a run completed.

Feedback: Would like to be able to extend a generic behaviour for which you can override some of the methods. He got this idea because he wanted to do the same as the LogEventsBehaviour has with a bunch of methods for the events.

The event system was not clear when and how to use it, the terminology was not understood and getting the info needed about events was not clear to the subject from the documentation alone.

Challenge 3

Skipped

Challenge 4

He implemented

Task 3 - Log number of evaluations

No idea how to create their own behaviour.
Was able to use the events of the query directly to implement the counter.

Solution 1: Count the number of states in the heap
He wanted to get the heap from the query with Query::getHeap but was disappointed he only got a Heap instance back which did not implement the functionality he desired.
He did not figure out that casting it to ManagedHeap would let him perform the solution anyway.

Solution 2: Increment a counter
He used a tappable event to increment a global counter and after exploration he prints the counter value.

Task 4 - Avoid broken steps

From the previous challenge he wanted to use the FrontierFilterBehaviour again and implement a predicate that identified the broken steps based on a list.

(task interrupted as we ran out of time)

Afterwards

After wrapping up the tests he noted that he saw a potential to use the framework to easily manage a larger number of sets of tests and implementations, that also switching out queries was pretty easy for a given problem.

This inspired me to later ask more participants about this.

Test Results: S4

A Computer science master student
Programming experience: java expertise

Introduction

We went very quickly through the introduction.
The explanation about the pipeline was a lot of new information, perhaps not all details were fully absorbed.

At this point I started to explain explicitly the functions of frontier and heaps more clearly, as in previous explanations their purpose may have been left blank. This means that now the subject also knew about the next-selection responsibility the frontier has.

Challenge 1

Task 1 - Run goat game and draw the state space

He only drew transitions for discoveries, not for undo-transitions.
He established that the algorithm was BFS.
He did not draw the full state space.

Task 2 - Change the algorithm to DFS

He looked through the documentation of functions and classes
He noted that he would have liked to have had the documentation separate from the source-code.
He identified that the exploration order should change and found the lifo function on the QueueFrontier without difficulty.
He verified that the frontier had the correct behaviour.

Task 3 - Implement early goal termination

He looked through the list of default behaviours and identified the TerminateOnGoalBehaviour as a suitable candidate to implement this.
He copied the example of applying the LogEventsBehaviour and adapted it to the termination behaviour.
Implementing the prediate was easy due to the autocompletion that intelliJ provided.

Summary

He found the documentation sufficient to figure out what to do

Challenge 2

Task 1 - Determine the exploration method

BFS was obvious to the subject, he noticed it already during the introduction of the UI.

Task 2 - Terminate the exploration when reaching the goal state

He implemented the termination behaviour again.
Exploration terminates after 2219 steps.

Task 3 - Optimize the query with an ordered frontier

The TreeMapFrontier was found but the documentation made it difficult to use.
I had to explain the usage of the api here for the subject to be able to use it.

After using the new frontier the termination occurs after 669 steps.

Task 4 - Freely optimize the query further

Idea: add direction to the position class to track where you came from.
The subject thinks of the states as mutable entities, perhaps it should be clearer that they are supposed to be immutable.

Challenge 3

While it was thought to be broken we did perform the challenge.
After pointing out that he had to use the SweepLineBehaviour to implement the required changes at task 3 the subject was able to use the documentation on how to implement the required pieces and components to make the algorithm function correctly.

Challenge 4

For the state representation the subject decided to make the class hold two integer variables: one for the current position of the robot and one for the height of the staircase. What was not clear during the programming was that the subject had to create a hashcode and equals implementation also, however intelliJ is able to generate them based on the class fields so it was not much of a problem.

The subject copied the static method to build a query object from the previous examples and adjusted the generic types to his needs.

For the next function, he created one using the NextFunction.wrap() utility together with a lambda.
He implemented two methods, climb and descend which each created a new state with the proper modifications.
He first wanted to do bounds checking himself but after a short while decided to use a filter on the output stream instead. He used the same technique that the other challenges used where a filter removes all illegal states from the next function output.

Task ? - Avoid broken steps

He implemented this quickly as the time was almost over, but managed to do it in time by using the FrontierFilterBehaviour.

Afterwards

I asked him what he thought of the experience afterwards and he gave the following feedback:

Found the filter feature very useful
Also sees that variations on queries are easy to manage.

Test Results: S5

Technical Computer Science student
Programming experience: Mostly with python, no java expertise

Introduction

I had explained the project to Joanne prior, so she may have been more comfortable with the materials.

Challenge 1

Task 1 - Run goat game and draw the state space

She drew the state space without any flaws
She drew them with directional arrows
She started drawing the initial state a second time before realizing that it was a backlink.
She deduced the exploration was BFS.

Task 2 - Change the algorithm to DFS

What do you want to do first?

- “Make the frontier order different”
- She inspected the auto complete of the QueueFrontier and found the lifoFrontier method
- She was looking for “filo” and lifo was identified as a suitable candidate

Task 3 - Implement early goal termination

I hinted that there are standard behaviours.
She looked through them and found the right behaviour from the names alone.
She saw from the constructor that she needed a predicate.
She did not know what a predicate was, I had to explain this and how it interacts with java lambda expressions.
Unclear what the predicate needed or how to implement it.
I helped her implement the lambda expression, the difficulties may have been lack of knowledge on how java lambda’s function. Once she understood that a state was input it was pretty clear.
Understood the manual termination signal in the simulation.

Summary

When asked how she would have compared the implementation to writing the entire algorithm herself she noted that she prefers using lifo-fifo frontiers over implementing DFS with recursive implementation, but the comparison was not very realistic since she did not know how to implement the algorithm herself in the first place.

Challenge 2

Task 1 - Determine the exploration method

Identified BFS by playing with the interface

Task 2 - Terminate the exploration when reaching the goal state

Started adding the behaviour, was able to figure out how to use the Maze state api, being able to apply this displays an understanding of the effect the predicate has on execution of the algorithm.
She tested the solution with the simulator and verified it worked.

Task 3 - Optimize the query with an ordered frontier

Looking through the frontiers, the TreeMapFrontier did not stick out as a suitable candidate.
She noticed OrderedFrontier and used the IDE to figure out what classes implement it.
She followed through the DynamicallyOrderedFrontier interface to TreeMapFrontier.
She wanted to instantiate it directly with the constructor. The constructor was difficult to use as she did not understand how to use it, the documentation was lacking for the TreeMapFrontier class.

The comparator was relatively easy to build, some maze-specific details she needed help with to implement the comparator but otherwise the purpose was clear.
I preemptively fixed the withExactOrdering bug, as she could not have known of its existence and the implemented solution ended up working.

Challenge 3

Task 1 - Add the sweep-line behaviour
This task was difficult to complete, and needed hints to use the Helpers functions.

She noted that this style of building algorithms is like “drag-and-drop” programming.
“This is useful for problems where using different algorithms is key.”
“It is easy to try different approaches on a problem.”

Challenge 4

Build the query and execute it

When building the main function, she wanted to pass the initial states to the explore method and did not know what to try otherwise.
When met with the empty console termination, she thought that the next-function was wrong.

It was not clear that an initial state had to be provided.

(although up until that point that was not required anyway)

Count the number of states

Solution: To count the states she wants the heap to count the number of states.
When counting for a staircase of 1000, then 997 states were in the heap.

Afterwards

- Java heavy
- Educational purpose is good(/clear) and also very generalized
- The framework is very generalized and not geared to any specific exploration algorithm
- She thinks that once you understand the framework it can save time.

Analysis - How did things go

Changing the frontier from BFS to DFS

In C1T2 the subjects needed to change the algorithm from BFS to DFS and received hints that led them to conclude they needed a different frontier.
What led them to find the correct frontier?

S1

He had a lot of prior experience with java programming and was familiar with the IDE (intellij) this led him to explore the framework as soon as possible, looking at the class hierarchy and interface implementations. Once he knew what he was looking for he managed to find the method that creates the correct frontier in the source-code of the QueueFrontier class.

S2

He received the hint that the frontier order would need to be different and checked for alternate methods on the QueueFrontier class and found the method quickly.

S3

He figured out that the exploration order would have to be “filo” and used the autocomplete function to list the alternate methods on the QueueFrontier.

S4

He looked into the source code of the QueueFrontier and found the method defined there

S5

She received the hint that the frontier would need to be ordered differently, from there she looked into alternate methods on the QueueFrontier class using auto completion and found the method there.

Subject	Looked at source-code	Used javadoc	Main method
S1	Yes	Yes	Source code
S2	No	No	Autocompletion
S3	No	No	Autocompletion
S4	Yes	No	Source code
S5	No	No	Autocompletion

This leads to believe that for java-experts the framework is fairly self-explanatory and from the initial example they are able to make a basic modification.

Implement on goal early termination

In each challenge one of the first tasks was to change the query to make it terminate the exploration when a goal state would be found. Here we analyse how that was achieved

S1

They received no guidance and made several attempts before achieving a solution. They first focussed on the next-function, perhaps in an understanding it was the most controllable source of data. They did look through the list of default behaviours before this attempt but did not register the default termination behaviour in the list. Their successful attempt consisted of clearing the frontier using the afterStateEvaluation event. They needed guidance on how to implement an abstract-behaviour.

S2

When looking through the list of default behaviours they did not register the suitable default behaviour. He only saw the list of filenames in the folder view of the source code, he had not taken the time to inspect the documentation thoroughly. When tasked with using the behaviour system or event system he could not figure out how to use them.

S3, S4 and S5

They all had a similar experience in applying the termination behaviour: They found the suitable default behaviour by the class name in the project explorer. They had varying difficulty using the predicate, mostly due to it being a relatively new feature they had not used before. But the purpose was clear from context and/or the documentation.

Observations & conclusions

The experience of S1 and S2 points out that for newcomers the concepts of the events, tappables and behaviours is left unexplained and that the documentation of the functions which expose and consume them are not sufficient by themselves. A possible solution to this is to provide small guides/tutorials on these systems which explain their usages, how to implement them and how to find/recognize them in the code base.

The experiences of S3, S4 and S5 have shown that for the TerminateOnGoalBehaviour is relatively easy to use without much help.

A specific experience S3 had with this behaviour was in Challenge 2, Task 4 where he received the advice that the behaviour exposed events that signal when a goal state is found. By that description alone he could not figure out how to use the behaviour in such a way. The major obstacles there were again lacking knowledge in the events system. So one of the guides may have to be how to use events exposed by non-query systems (such as another behaviour).

Using a ordered frontier

In Challenge 2, Task 3 the subjects were asked to change the frontier to one which is sorted by a user-specified metric.

S1

He had difficulty finding the correct frontier, he used the project explorer to find out what code resides in the “abeona.frontiers” folder but did not think (at first) that TreeMapFrontier would be the required class for the solution. After some exploration of the class/interface hierarchy of the Frontier interface it became clear for him that he had to use that class.

S2

When applying the TreeMapFrontier he looked at the source code and documentation to figure out what it exposed. The first view he had was of the top of the class and this included the constructor. As such he immediately attempted to call it, even though it was private. This sequence of events also occurred with several other subjects.

S3

Similarly to S1 there was some difficulty spotting TreeMapFrontier as the correct solution, he specifically remarked that the naming of this class was more a hint towards the implementation details than it was a descriptive name of its purpose. First attempts to use it focussed on the constructor instead of the factory methods.

S4

Once the TreeMapFrontier was found the usage was difficult to deduce because of lacking documentation.

S5

She did not notice the TreeMapFrontier by name as a suitable solution but found it ultimately by checking the class hierarchy for the Frontier interface. The implementation details causing problems for comparators which report states as equivalent caused the subject to think the frontier was wrong.

Observations & conclusions

Based on S1, S3 and S5 we can see that the name of the TreeMapFrontier is not very eye-catching and threw off the subjects early on. The OrderedFrontier was more often the focus of the subjects, and usually lead to them back to the TreeMapFrontier. Perhaps a naming scheme where TreeMapFrontier contains a word closely related to “ordered” or “sorted” would be better suited.

Besides that, the usage of a Comparator was also not entirely clear to users immediately. The name comparator is not java-specific itself but it application in java and the framework is. The documentation for the TreeMapFrontier did not adequately explain what was required of the comparator and only java-experts knew to use the Comparator factory functions. Those factory functions were kept in mind when designing the TreeMapFrontier API and as has become clear the documentation may need to hint towards those functions to help new users implement comparators easily.

The TreeMapFrontier instantiation was confusing for several subjects, when they first viewed the class they saw the documentation and source code of the top of the class. Since this included the constructor they immediately tried to call the constructor. The documentation of the class may need to specifically mention that it can only be constructed with one of the two factory methods.

Finally, none of the subjects had any information about the comparator edge-case/bug where some states would be considered equivalent based on the equivalence. Perhaps the framework should implement a workaround so they are never exposed to it. Alternatively, the subjects each picked the first occurring factory method they saw from the source code view, simply putting the factory method which implements the workaround first could also nip this problem in the bud.

Executing their own query

Not every subject worked on challenge 4, where they had to build their own state representation as well as use the query object themselves.

S1

He looked through the Query class to find the methods available and noticed the Query::explore method. I had to remind him that the frontier needs an initial state when on the first run the query terminated immediately.

When trying to optimize the query to use a greedy algorithm he tried implementing his own frontier. Because he did not do Challenge 2 the existence of the TreeMapFrontier was not known to him. From the documentation alone he was able to figure out how to implement the Frontier interface. The fact that there were other more specialized interfaces of Frontier (e.g. ManagedFrontier) was lost on him, perhaps the documentation on the Frontier interface could further clarify this, however he did not carefully read the documentation in the first place so perhaps exploring sub-interfaces of Frontier might come naturally.

S2

Did not perform this challenge

S3

He copied the structure of a “createQuery” method from the previous examples to built the query in a similar fashion as in prior challenges. When it came to executing the query he was confused why it terminated immediately, he expected the framework to use some initial state, though he did not specify one. He used the source code of one of the examples to figure out how the query object built by him was being used. There he discovered that the initial state has to be inserted into the frontier before exploration.

S4

He first wrote his state representation and then built the query code.
His state representation contained methods to create alternate versions of the current state with some mutation applied, such as moving forward or backward.
When building the query code he also copied the createQuery method from a previous example and adjusted it to his needs. The next-function was made with the NextFunction.wrap helper. When explaining the next-function interface he wondered what the wrap function was for and if he needed to implement it. The signature of the function was too confusing (due to the large generics declarations) for him to figure out how the wrap function relates to the next-function immediately.

S5

She did not have enough java experience to complete this task on her own, I had to assist in writing the code in a timely manner. I also hinted at reusing code from previous exercises to speed up building the query. Her implementation of the state representation used a in-class next-function which was something the other subjects had not done. She decided to do this after I explained how the next-function works and could be implemented with either a lamda or on a class directly.

In her approach she used if-statements in the next-function to determine whether some actions are possible or not.

Observations & conclusions

On every first run of the query the subjects did not specify an initial state. This is because none of the queries prior were built with an initial state specified. They all assumed that the framework somehow built an instance automatically. S5 specifically asked whether the Query::explore method could take a state as input and this might be an easy to implement solution to provide a simple API that solves this. Additionally, an error-signal or special termination type may be created for starting queries without an initial state.

A small observation I made is that in the framework interfaces used for the majority of laying the foundation of the class hierarchy, however this was hardly ever noticed or obvious to the users. Perhaps the reliance on interfaces and composition is not quite clear enough on first use of the framework and it might need a specific mention in one of the first introductions/guides for the framework. Understanding that most features and components are found in an interface and may have multiple sub-interfaces which further extend it would encourage more thorough looks at the api from users when implementing their own components.

In the implementations of the subjects, their states sometimes included both state and problem level details in the state representation. A generic model for encoding states, problem properties and state mutations might help make the creation of state representations more modular also. In the current case often the state representation and next-function held all details of the problem context and possible mutations of the state. For the problem context this means that it is difficult to extend or modify the problem context if the code defining it is not under your control. Also for possible mutations a friendlier interface that allows adding mutations later on would fit well with the modular design of the rest of the framework. Perhaps these could even be behaviours in some way. These last suggestions are more dependent on state representation of the user and was out of scope for the project but might be good future work to extend the framework.

Influence of expertise

For programmers with java expertise (not to be confused with programming expertise), the framework was noticeably easier to use than for those without java-specific knowledge. The usage of modern java features (e.g. lamba’s and functional interfaces) has led to a higher barrier-to-entry because the users need to understand how these features work. Also, to make maximum use of for example the comparator interface the usage of helper methods such as Comparator.comparing() is very useful to easily create comparators, however this may be somewhat hidden for those that do not know about it.

The java experts also had a much easier time exploring the structure and hierarchy of the framework.

Regardless of expertise, none of the subjects were able to quickly understand the events system. This was different from the behaviour system and I think this is because the behaviour system was introduced to them through sample code in the challenges, each challenge's query included the LogEventsBehaviour. There was no code demonstrating the tappable events system to them.

What was also noticeable was that many subjects preferred to modify the query's behaviour by modifying the next-function. The concept and advantage of behaviours is something that needs to be more front-and-center in the introduction of the framework.

Conclusions

The framework in its current state is sufficiently usable for those with experience with java programming. For those without java programming the framework is usable but the lack of java specific knowledge does put up a barrier and makes the framework API more difficult to use due to its minimalist design.

For new users the sample code provides good examples of how to build queries but lacks in demonstrating the advanced use cases of behaviours and completely lacks to highlight the usage of the tappable events system. Once explained these systems are not incredibly complex to use but understanding them is not easily done through the documentation alone.

Creation of custom behaviours also has a bit of a barrier to entry, since there is no simple base to quickly start with. The usage of AbstractBehaviour is too complex and difficult to understand from the current documentation.

The used naming scheme for classes is good and logical for the users but in cases where the name reflects the implementation details rather than the purpose of the class/interface this falls short.

The documentation is sufficient all round, but the TreeMapFrontier documentation lacks severely. The TreeMapFrontier implementation has hidden requirements when instantiating it.

The API is very modular and generalized for exploration algorithms, its usage case in educational purposes and management of sets of exploration methods is clear.

Suggestions

- Create a guides/tutorials on how to use the framework:
    - Creating your own state class
    - Creating a next-function
    - Creating and using a query
    - Using query events

    - Using query behaviours
    - Creating custom behaviours
    - Using behaviour events
- Create a generic abstract behaviour which hooks all events and exposes these as overridable methods
- Rename the TreeMapFrontier to make it more obvious for its use-case
- Improve the TreeMapFrontier documentation
- Provide better out-of-the-box workaround for the TreeMap Comparator equivalence edge-case
- Make it clearer that the TreeMapFrontier needs to be instantiated through the factory methods instead the constructor
- Allow the Query::explore API to ingest initial states
- Warn the user of starting explorations without initial states
- Create a modular system for defining/composing next-functions
- Improve documentation on what interfaces and implementations exist for the base interfaces of components (frontier, heap, behaviour)