TRAVELING SALESMAN PROBLEM

MODULE: CSI_5_ADP_2223

DATE OF SUBMISSION: 31/03/2023

STUDENT ID: 4008609

Table of Contents

Part 1 2
Part 2 4
2.1 Searching process fix 4
2.2 Replay button fix 7
2.3 Cancel button fix 12
2.4 Further improvements 18
2.5 Window close fix 20

Part 1

1) After running the program provided, it clearly does not function as intended. The first behaviour observed is that upon selecting the number of cities and pressing go, the

animation does not show the process of searching all combinations of cities (brute force). It instead only shows the final result. This, in turn, makes it impossible to test that the cancel button works as intended. However, upon evaluating the code, the cancel method does not seem to have any functionality that would interrupt the animation or replay processes/methods. Furthermore, the code appears to be struggling to update the paint call values in line with the required number of iterations. The replay button also doesn't show the process of finding the longest to the shortest route, meaning paint calls increase by only one upon triggering the replay button listener.

2) The main reason the animation does not seem to be functioning as intended is that all updates appear to be happening on the same (main) thread. This means that when the displayRouteUpdate(), displayBestRote and displayOneRoute methods (and others) are called, the event dispatcher thread is blocked until all updates have been completed. There is also no synchronisation with the routes tree set, which could cause the program to struggle when handling concurrent processes (accessing the treeset concurrently) and can lead to race conditions.

Part 2

2.1 Searching process fix

```
71  private void runAnimation() {
72    new  Thread(() -> {
73      final int numberOfCities = this.numberOfCitiesCombo.getSelectedIndex() + MINIMUM_NUMBER_OF_CITIES;
74    this.goButton.setEnabled(false);
75    this.replayButton.setEnabled(false);
76    this.cancelButton.setEnabled(true);
77    this.allRoutes.clear();
78    this.imagePanel.resetPaintCallCounter();
79
80    final TSP tsp = new TSP(numberOfCities, TSPUi.this.image.getWidth(), TSPUi.this.image.getHeight());
81    tsp.setListener(new UIListener(TSPUi.this));
82    tsp.findShortestRoute();
83
84  }).start();
85  }
```

Figure 1: Implementing runAnimation() within anonymous thread

Initially, I attempted to create an anonymous thread within the animation method using a lambda expression to define the runnable object. It can be considered that this is creating a new worker thread. However, it is not explicit as Swingworker is not being utilised. This partially fixed the issue of the program not showing the process/animation of determining the shortest route. Because of this, I also confirmed that the cancel button was not functioning as intended (instead of having to use Thread.sleep in class TSP). However, the animation seemed to be flickering (with almost every iteration). The cause for the flicker could be since the repaint() method in the displayRouteUpdate() method executes asynchronously. This also resulted in paint calls still being asynchronous. A solution to this could involve setting a swing timer to control the animation loop where the repaint method would be called at regular intervals. We could also render graphics to an independent ‘Buffered image’, which could then be copied to the ‘image’. This would, however, require significant changes to the code.

A more straightforward solution, though not entirely synchronous (but more superficial than the buffered image solution suggested above), would be if the displayRouteUpdate() method was called inside a runnable and used invokelater to ensure it was executed on the EDT, however after testing and further reflection, it did not solve the problem as invokelater is not asynchronous as it returns immediately and does not wait for ‘Runnable’ to complete. This would not allow the EDT to keep up with incoming updates, leading to skipping updates. The solution would be to use invoke and wait instead, which would also require catching exceptions and storing the tree in some variable (unlike invokelate).

```java
public void NewDisplayRouteUpdateQueue(final TSPRoute route, final TSPRoute bestRoute) {
  try {
    SwingUtilities.invokeAndWait(new Runnable() {
      @Override
      public void run() {
        displayRouteUpdate(route, bestRoute);
      }
    });
  } catch (InterruptedException | InvocationTargetException e) {
  }
}
```

Figure 2: Wrapping displayRouteUpdate() in invokeAndWait

To test and achieve this, a new method was created named 'NewDisplayRouteUpdateQueue(), which, by calling the original displayRouteUpdate() method and ensuring that UI updates are wrapped inside invokeAndWait, while taking runnable as an argument. This ensures that this method, which is responsible for UI updates, is executed on only the EDT. Another reason this is a good and straightforward solution is that the method will now wait for the completion of Runnable before continuing.

It is worth noting that the method above is called from a different worker thread, allowing the event dispatch thread to handle graphical updates without being blocked by other processes executing on the same worker thread. This, in turn, results in a more responsive user interface, especially when frequency or loads are significant.

NOTE: There was a cleaner way of utilising invokeAndWait without having to create a new method (by using lambda expressions). This was, however, not realised until section 2.4.

```java
@Override
public void displayUpdate(final TSPRoute testRoute, final TSPRoute bestRoute) {
  this.ui.NewDisplayRouteUpdateQueue(testRoute, bestRoute);
  this.updateCalls++;
}
```

Figure 3: Replacing old method with new

To make use of this new method, the IDE was used to check where the original displayRouteUpdate() method was called, in this case within the displayUpdate() method in the UIlistner class, and was replaced with the new method which calls the original method but within runnable and using SwingUtilities.invokeAndWait()

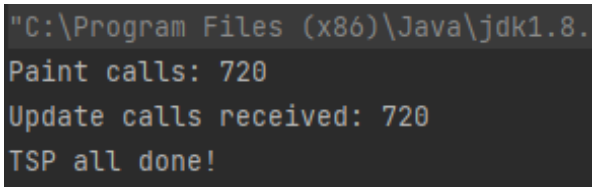

Figure 4: Testing paint calls

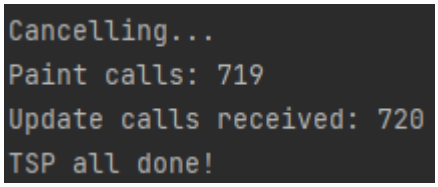

Figure 5: Paint calls error detected

Upon testing, the graphical animation was no longer flickering (and was smooth). The paint and update calls received were also not in sync (see the output above). However, upon triggering the cancel listener (while the animation was still ongoing) and waiting for the process to complete, it was observed that it caused paint calls to be affected with every iteration of interacting with the cancel button. This could be due to the fact that the cancel() does not currently have any functionality or because displayBestRoute is asynchronous and not executing on a separate worker thread.

2.2 Replay button fix

As outlined in part 1, the replay button does not seem to be working as intended. However, after fixing the paint calls synchronisation, it can be tested and observed that clicking the replay button after the search process has completed causes the number of paint calls to increment by one. This suggests the problem could be related to the fact that the EDT is blocking the method, meaning it cannot be run concurrently. The solution to this could be to put the showLongestToShortest() method in a separate thread (similar to the runAnimation() method).

```
private void showLongestToShortest() {
  new Thread(() -> {
    this.replayButton.setEnabled(false);
    this.cancelButton.setEnabled(true);
    this.goButton.setEnabled(false);

    for (final TSPRoute route : TSPUi.this.allRoutes) {
      displayOneRoute(route, Color.WHITE);
    }
    displayBestRoute(TSPUi.this.allRoutes.last());

  }).start();
}
```

The figure to the left shows the implementation of putting the showLongestToShortest() method in an anonymous thread and calling runnable using a lambda expression

Figure 6: Adding replay function to anonymous thread

After implementing this change, the test showed that the replay button was functional and demonstrated the process of determining the shortest path. However, it did seem that the paint calls were occurring relatively fast (possibly because some were being missed). Furthermore, when the number of cities/points exceeded 8, the process flickered similarly to the search after triggering the go listener (section 2.1).

In an attempt to fix this flicker, and upon seeing that the showLongestToShortest was calling two other methods, displayOneRoute and displayBestRoute, I decided I would, as with the displayUpdate() method in section 2.1, I would call these in a new method which would force the original displayBestRoute() method to execute on the EDT and an individual worker thread.

```
2 usages
public void NewDisplayBestRouteQueue(final TSPRoute bestRoute) {
  try {
    SwingUtilities.invokeAndWait(new Runnable() {
      @Override
      public void run() { displayBestRoute(bestRoute); }
    });
  } catch (InterruptedException | InvocationTargetException e) {
  }

}
```

Figure 7: Wrapping displayBestRoute in invokeAndWait()

The implementation above shows the original displayBestRoute method wrapped in invokeAndWait while calling runnable as an argument.

```
1 usage
@Override
public void displayBest(final TSPRoute bestRoute) {
  this.ui NewDisplayBestRouteQueue(bestRoute);
  System.out.println( "Update calls received: " + this.updateCalls);
}

}
```

Figure 8: Changing method call in displayBest()

```
private void showLongestToShortest() {
  new Thread(() -> {
    this.replayButton.setEnabled(false);
    this.cancelButton.setEnabled(true);
    this.goButton.setEnabled(false);

    for (final TSPRoute route : TSPUi.this.allRoutes) {
      displayOneRoute(route, Color.WHITE);
    }
    //displayBestRoute(TSPUi.this.allRoutes.last());
    NewDisplayBestRouteQueue(TSPUi.this.allRoutes.last());

  }).start();
}
```

Figure 9: Changing method call in showLongestToShortest()

The showLongestToShortest() method and displayBest() method in the UiListner class, where the original displayBestRoute methods were called, were replaced with the new method which wrapped invokeAndWait, and the same parameters were passed.

After implementing this and testing the replay button, it did not seem to have fixed the issue. Upon reflection, this was clearly because this method was responsible for displaying the final (shortest) route rather than looping through the entire sorted tree set, which contained all the routes determined in the TSP class.

The flickering problem would be solved if the displayOneRoute() method was called to the EDT (by using invokeAndWait).

```
private void NewDisplayOneRouteQueue(final TSPRoute bestRoute, final Color color) {
  try {
    SwingUtilities.invokeAndWait(new Runnable() {
      @Override
      public void run() { displayOneRoute(bestRoute, color); }
    });
  } catch (InterruptedException | InvocationTargetException z) {
  }
}
```

Figure 10: Wrapping displayOneRoute in invokeAndWait()

The implementation of calling the original displayOneRoute() method is shown above and wraps invokeAndWait while calling runnable as an argument (using lambda expression).

```
private void showLongestToShortest() {
  new Thread(() -> {
    this.replayButton.setEnabled(false);
    this.cancelButton.setEnabled(true);
    this.goButton.setEnabled(false);

    for (final TSPRoute route : TSPUi.this.allRoutes) {
      //displayOneRoute(route, Color.WHITE);
      NewDisplayOneRouteQueue(route, Color.WHITE);
    }
    //displayBestRoute(TSPUi.this.allRoutes.last());
    NewDisplayBestRouteQueue(TSPUi.this.allRoutes.last());

  }).start();
}
```

Figure 11: Changing method call in showLongestToShortest()

The new method then replaced the old in every place it was called, in the figure above, within the showLongestToShortest() method.

```
1 usage
public void displayBestRoute(final TSPRoute bestRoute) {
  //displayOneRoute(bestRoute, Color.GREEN);
  NewDisplayOneRouteQueue(bestRoute, Color.GREEN);
  System.out.println("Paint calls: " + this.imagePanel.paintCalls());
  this.goButton.setEnabled(true);
  this.cancelButton.setEnabled(false);
  this.replayButton.setEnabled(true);
}
```

Figure 12: Changing method call in displayBestRoute()

It was also changed within the displayBestRoute method in the same class. All the same parameters are being passed.

After implementing the changes outlined above and running the program, it seemed that the replay button was no longer clickable (set to false somewhere). The program also acted up if the cancel button (still not functional) was clicked, followed by the ‘Go’ button. Paint calls were also no longer being outputted. Upon reflection and further analysis of the changes made, it became clear that this was due to a deadlock. The deadlock was caused because when calling the NewDisplayOneRouteQueue() method from the displayBestRoute()

method, the EDT was waiting for Runnable to finish executing while Runnable was waiting for the EDT within the same method. A solution to avoiding this cyclic cycle would be to separate the code responsible for the Ui buttons and paint calls within the displayBestRoute() method. This could be done by creating a new method which would be called into the displayBestRoute() method.

```
public void displayBestRoute(final TSPRoute bestRoute) {  //CHANGES
  updateUIComponents();
  NewDisplayOneRouteQueue(bestRoute, Color.GREEN);

  //displayOneRoute(bestRoute, Color.GREEN);
  //    System.out.println("Paint calls: " + this.imagePanel.paintCalls());
  //    this.goButton.setEnabled(true);
  //    this.cancelButton.setEnabled(false);
  //    this.replayButton.setEnabled(true);
}

1 usage
private void updateUIComponents() {     //CHANGES
  System.out.println("Paint calls: " + this.imagePanel.paintCalls());
  this.goButton.setEnabled(true);
  this.cancelButton.setEnabled(false);
  this.replayButton.setEnabled(true);
}
```

Figure 13: Splitting displayBestRoute() to avoid deadlock

The code snippet above shows the separation of the code responsible for setting the state of the button as well as outputting the number of paint calls made to avoid deadlock. Upon retesting the program, it now functions as intended (apart from the cancel button), and the screen also no longer flickers when under excessive load during the replay process. The replay also now occurs at a much more reasonable pace, further suggesting paint calls are not being missed.

2.3 Cancel button fix

As a means to add functionality to the cancel method(), which currently does nothing to stop the processes, I thought it would be a good idea to first initialise and declare two separate variables, one for the searching process and another for replay.

```
3 usages
62  public volatile Thread animation = null;  //CHANGES
3 usages
63  public volatile Thread replay = null; //CHANGES
```

Figure 14: Thread variable declaration

The volatile keyword is used in case the value of these variables needs to be changed by multiple threads at once, these changes would also be visible to other threads. It also provides some synchronisation and memory consistency. They are initialised as null since they do not currently have reference to any thread object yet.

```
1 usage
private void runAnimation() {
  new Thread(() -> {
    animation = Thread.currentThread();
    final int numberOfCities = this.numberOfCitiesCombo.getSelectedIndex() + MINIMUM_NUMBER_OF_CITIES;
    this.goButton.setEnabled(false);
    this.replayButton.setEnabled(false);
    this.cancelButton.setEnabled(true);
    this.allRoutes.clear();
    this.imagePanel.resetPaintCallCounter();

    final TSP tsp = new TSP(numberOfCities, TSPUi.this.image.getWidth(), TSPUi.this.image.getHeight());
    tsp.setListener(new UIListener(TSPUi.this));
    tsp.findShortestRoute();

  }).start();
}
```

Figure 15: Assigning thread to animation variable

The current thread running within the runAnimation() method is assigned to the animation variable. This allows this thread to be manipulated within the cancel method using its reference. This includes checking its state, for example, System.our.println (animation.interrupted).

```
1 usage
private void showLongestToShortest() {
  new Thread(() -> {
    replay = Thread.currentThread();
    this.replayButton.setEnabled(false);
    this.cancelButton.setEnabled(true);
    this.goButton.setEnabled(false);

    for (final TSPRoute route : TSPUi.this.allRoutes) {
      //displayOneRoute(route, Color.WHITE);
      NewDisplayOneRouteQueue(route, Color.WHITE);
    }
    //displayBestRoute(TSPUi.this.allRoutes.last());
    NewDisplayBestRouteQueue(TSPUi.this.allRoutes.last());

  }).start();
}
```

Figure 16: Assigning thread to replay variable

Again, the same assignment was carried out in the showLongestToShortest() method for easier thread manipulation.

```
1 usage
private void cancel() {
    System.out.println("Cancelling...");
    this.goButton.setEnabled(true);
    this.cancelButton.setEnabled(false);

    if (animation != null) {
        animation.stop();
    }

    if (replay != null) {
        replay.stop();
    }
}
```

Figure 17: abruptly stopping thread execution

The code snipped above shows the implementation of the cancel button, which is now technically functional by abruptly stopping the thread. There are a few issues with this method. Abruptly terminating a thread could potentially leave shared resources (in this case, routes stored in set) in an inconsistent state. It could also lead to possible deadlocks. It is worth noting that the Thread.stop() method throws a ThreadDeath exception which can cause unpredictable behaviour.

A better method of achieving this could be to set a boolean (volatile) flag, which changes state when the cancel method is used. The loop within the findShortestRoute in the tsp class can then be broken when the state changes (during cancellation). This has been attempted in the code snipped below.

```
7 usages
public static volatile boolean cancelled = false;
```

Figure 18: Declaring volatile boolean flag

I first initalised the boolean flag named ‘cancelled’ (as false state). Though this can pass through to the findShortestRoute method in the TSP class, it is not ideal as the variable is public static. This can be improved through the instantiation of an object.

```
private void cancel() {
    cancelled = true;
    System.out.println("Cancelling...");
    this.goButton.setEnabled(true);
    this.cancelButton.setEnabled(false);

}
```

Figure 19: Setting cancelled flag to true

When the user interacts with the cancel button and the execution is directed to the cancel method, the flag will change state to true. This can then be used as a conditional.

```
2 usages
public void findShortestRoute() {

  final int[] indexes = new int[this.locations.length];
  for (int i = 0; i < indexes.length; i++) {
    indexes[i] = i;
  }

  final int[] c = new int[indexes.length];

  processRoute(indexes);

  int i = 1;
  while(i < indexes.length && cancelled == false) {
    if (c[i] < i) {
      if ((i%2)==0) { // is even then
        swap(indexes,  p: 0, i);
      } else {
        swap(indexes, c[i], i);
      }
      processRoute(indexes);
      c[i] += 1;
      i = 1;
    } else {
      c[i] = 0;
      i += 1;
    }
  }
  if (cancelled == true){
    cancelled = false;
  }
  this.listener.displayBest( this.bestRoute);
  System.out.println( "TSP all done!");
}
```

The code snippet on the left shows the while loop using the AND opperater allowing the loop to break once the canceled flag changes state to true (when user clicks cancel)

The method then checks if the flag is true, in which case it will return it to its original state (false), allowing to use to carry out another search by interacting with the ‘Go’ button.

Upon testing, this worked as intended, however was not functioning (cancellable) post using the replay button. This is because the shortestToLongest() method does not check for the flag.

Figure 20: Setting condition in while loop for search

```
1 usage
private void showLongestToShortest() {
    new Thread(() -> {
        replay = Thread.currentThread();
        this.replayButton.setEnabled(false);
        this.cancelButton.setEnabled(true);
        this.goButton.setEnabled(false);

        for (final TSPRoute route : TSPUi.this.allRoutes) {
            if (cancelled == true){
                cancelled = false;
                break;
            }
            NewDisplayOneRouteQueue(route, Color.WHITE);
        }
        NewDisplayBestRouteQueue(TSPUi.this.allRoutes.last());

    }).start();
}
```

Figure 21: Implementing break condition in replay method()

The above shows the changes made to the showLongestToShortest() method. The loop which searches through the sorted set containing all the routes includes an if statement where if the flag state changes to true, the loop is broken. The code above also resets the state of the flag back to false, allowing the user to carry out another replay. Upon testing, all functionality of the program works as intended.

There is an even better way of implementing this function though. The ideal way is to interrupt the thread (named animation and replay) within the cancel method, which will set the interrupt flag of said thread to true. This flag can then be checked in the findShortestRoute loop, where the loop would be interrupted once the thread changes state. An attempt at implementing this is shown below.

```
1 usage
public void cancel() {
    System.out.println("Cancelling...");
    animation.interrupt();
    replay.interrupt();
    Thread.currentThread().interrupt();

    try{
        Thread.sleep( millis: 50);
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
        e.printStackTrace();
    }
```

Figure 22: Attempt to interrupt thread

The figure on the left shows the attempt at interrupting all threads, this is because it was not clear which thread would need to be interrupted. Upon reflection this could have been tested and checked using the .getName method.

```
public void findShortestRoute() {
  final int[] indexes = new int[this.locations.length];
  for (int i = 0; i < indexes.length; i++) {
    indexes[i] = i;
  }
  final int[] c = new int[indexes.length];
  processRoute(indexes);
  int i = 1;

  while (i < indexes.length && !Thread.currentThread().isInterrupted())  {
    try {
      sleep( millis: 1);
    } catch (InterruptedException e) {

    }

    if (c[i] < i) {
      if ((i % 2) == 0) { // is even then
        swap(indexes,  p: 0, i);
      } else {
```

Figure 23: Attempt to check for interrupt in while loop

The main reason I was unable to successfully implement this method of cancellation was because the animation and replay threads were unable to be passed through to the TSP class without the use of public static, which would cause serious vulnerabilities within the code. I was also unable to call .interupted, but only able to call .isInterupted, this could be due to the fact that the the/catch statements were not implemented properly.

I was also receiving null pointer exceptions which could have been solved through ‘catch’ (which would not solve the underlying problem), or through not having the thread variables declared as null in the first place

2.4 Further improvements

Upon reflecting on the changes made to the code (both section 2.1 and 2.2), the method calls were getting quite complicated, resulting in spaghetti code. A better way of calling invokeAndWait() on all the methods above would be to simply use the lambda expression rather than creating a new method that must be replaced in every place it is called within the code.

```
public void displayUpdate(final TSPRoute testRoute, final TSPRoute bestRoute) {
  try {
    SwingUtilities.invokeAndWait(() ->this.ui.displayRouteUpdate(testRoute, bestRoute));
  } catch (InterruptedException | InvocationTargetException e) {

  }
  this.updateCalls++;
}
```

Figure 24: Example of code improvement to avoid spaghetti code

An example of how this could be implemented is shown above. This can be replace the changes made in figures 1 and 2 and would significantly simplify the code. However, this method of calling invokeAndWait() in the case of the replay function is extremely risky as it may not address the deadlock caused in the EDT.

```
1 usage
public final Object lock = new Object();

4 usages
private final SortedSet<TSPRoute> allRoutes = Collections.synchronizedSortedSet(new TreeSet<>()); //CHANGES
```

Figure 25: declaring lock and changing set to synchronized

A lock object was initalised for future use. Furthermore the treeset was set to synchronized which will insure that only one thread can access it at a time. This further ensures memory consistency.

```
1 usage
private void runAnimation() {
    new Thread(() -> {
        animation = Thread.currentThread();
        final int numberOfCities = this.numberOfCitiesCombo.getSelectedIndex() + MINIMUM_NUMBER_OF_CITIES;
        SwingUtilities.invokeLater(() -> {
            this.goButton.setEnabled(false);
            this.replayButton.setEnabled(false);
            this.cancelButton.setEnabled(true);
        });
        synchronized (lock) {
            this.allRoutes.clear();           //CHANGES
        }
        this.imagePanel.resetPaintCallCounter();

        final TSP tsp = new TSP(numberOfCities, TSPUi.this.image.getWidth(), TSPUi.this.image.getHeight());
        tsp.setListener(new UIListener(TSPUi.this));
        tsp.findShortestRoute();

    }).start();
}
```

Figure 26: wrapping buttons in invokeLater()

```
1 usage
private void cancel() {
    cancelled = true;
    System.out.println("Cancelling...");

    SwingUtilities.invokeLater(() -> {
        this.goButton.setEnabled(true);
        this.cancelButton.setEnabled(false);
        this.replayButton.setEnabled(true);
    });

}
```

Figure 28: wrapping buttons in invokeLater()

```
1 usage
private void showLongestToShortest() {
    new Thread(() -> {
        replay = Thread.currentThread();
        SwingUtilities.invokeLater(() -> {
            this.replayButton.setEnabled(false);
            this.cancelButton.setEnabled(true);
            this.goButton.setEnabled(false);
        });

        for (final TSPRoute route : TSPUi.this.allRoutes) {
            if (cancelled == true){
                cancelled = false;
                break;
            }
            NewDisplayOneRouteQueue(route, Color.WHITE);
        }
        NewDisplayBestRouteQueue(TSPUi.this.allRoutes.last());

    }).start();
}
```

Figure 29: wrapping buttons in invokeLater()

2.5 Window close fix

As a mean to ensure that all background threads close upon clicking ‘X’ on the window, it was clear that threads must be stopped/interrupted

```
1 usage
public TSPUi(final int width, final int height) {
    setDefaultCloseOperation(EXIT_ON_CLOSE);
```

Figure 30: Ensuring threads ‘Exit on close’

Changing the close operation to EXIT_ON_CLOSE ensures that when the user closes the window, all running and background threads are closed. This is because this operation invokes the System.exit() method which terminated the JVM and all of its threads.

It is however worth noting the abruptly stopping threads without allowing them to clean up their date can cause problems resulting to an inconsistent state.

A better method for this could be to use the window listner, however it would be even better if all threads were interrupted and their condition was checked before terminating the window and all threads.