GitLab Links:
src/graphs/shortestpaths:
https://gitlab.cs.washington.edu/cse373-root/24au/students/ktran000/-/tree/main/src/main/java/graphs/shortestpaths?ref_type=heads
src/seamfinding:
https://gitlab.cs.washington.edu/cse373-root/24au/students/ktran000/-/tree/main/src/main/java/seamfinding?ref_type=heads

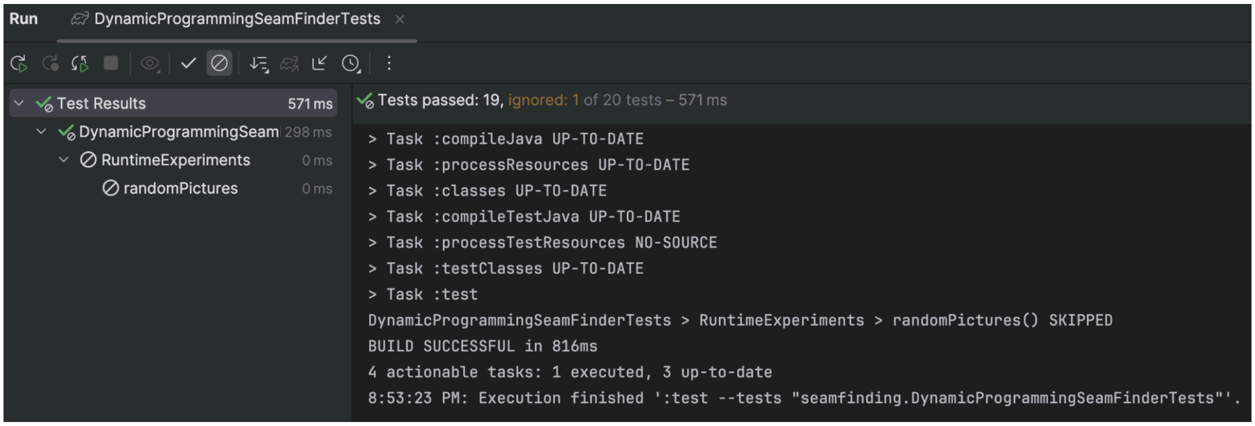
GenerativeSeamFinder

The part of the GenerativeSeamFinder class that I'm most proud of programming is the neighbors method in `GenerativeSeamFinder.PixelGraph.Pixel.neighbors`, particularly the loop at line 147 because I used a single lop to handle all three directions (up, straight, and down), which eliminated the need for three separate conditional blocks. The way I implemented this makes the code more concise and maintainable, while the range [-1, 0, 1] represents vertical movement. Another part of the for loop that I believe was a good implementation choice was after the conditional check at line 154 where I made sure that the pixels were in bounds, I wrote the line "Pixel neighbor = new Pixel(x + 1, newY);" which is a very efficient way of creating neighbors only when they're needed. Rather than pre-computing and storing all possible neighbors for all pixels upfront, this implementation choice only creates neighbors when they're needed, only generates valid neighbors (within bounds due to the conditional check), and reduces memory by not storing the entire graph structure. Next, at line 155 I wrote: "double energy = f.apply(picture, x + 1, newY);" which is a dynamic way to calculate the energy at the exact moment it's needed, only for valid neighbors, and using the current state of the picture. This is particularly efficient because there is no need to store energy values, no precomputation of unused energies, and the energy values are always current.

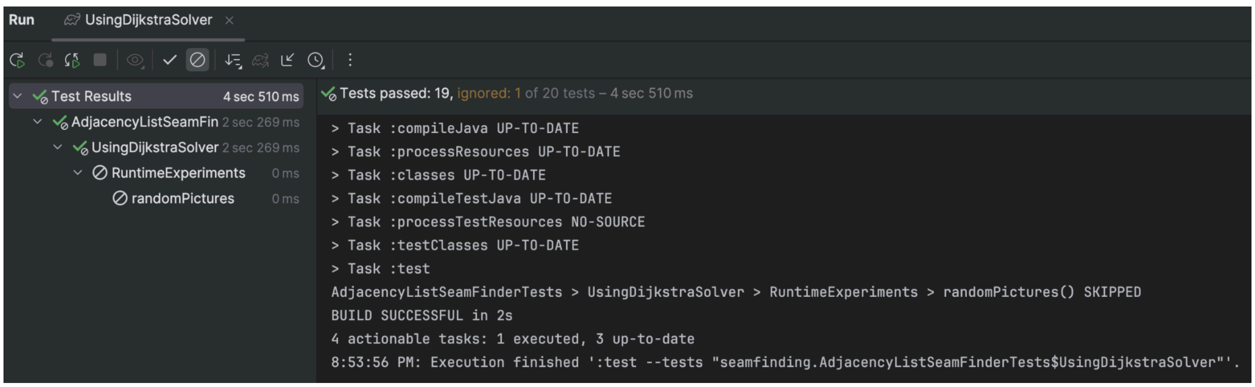
DynamicProgrammingSeamFinder

The part of the DynamicProgrammingSeamFinder class that I'm most proud of programming is the nested for loop at line 29 in the findHorizontal method which fills the cost table. I am proud of this implementation because the nested for loop naturally processes the pixels in topological order so there is no need for explicit topological sorting or graph traversal. The left-to-right processing guarantees that all dependencies (left neighbors) are computed before they're needed. This is because the outer loop iterates over columns from left to right, and the inner loop iterates over rows. This order ensures that when processing a pixel at position '(x,y)', all potential predecessor pixels '(x-1, y-1)', '(x-1, y)', '(x-1, y+1)' have already been processed. By processing from left to right, the algorithm guarantees that these dependencies are satisfied before calculating the current pixel's cost. Therefore, there is no need for explicit topological sorting. IN graph algorithms, topological sorting is often required to ensure that each node is processed only after all its dependencies. Here, the natural order of the loops inherently provides this guarantee, simplifying the implementation.

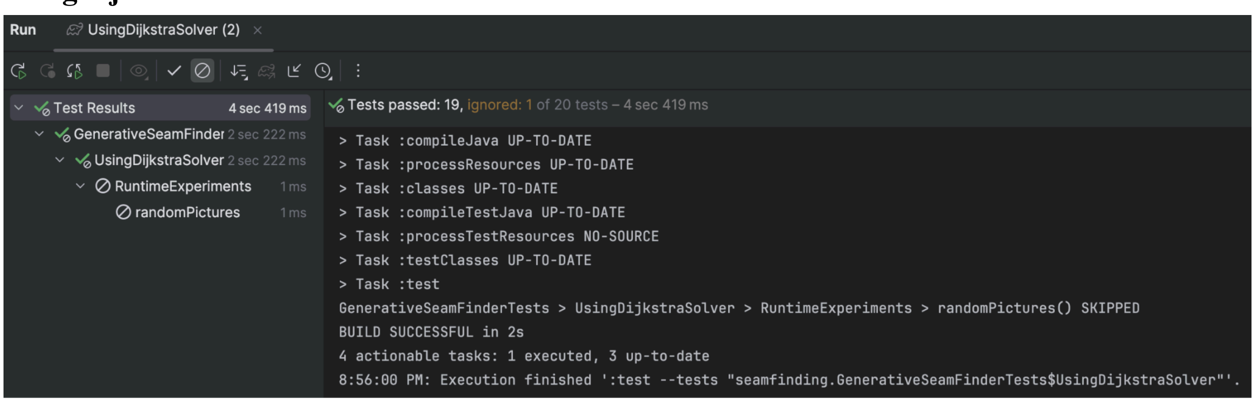
DynamicProgrammingSeamFinderTests:



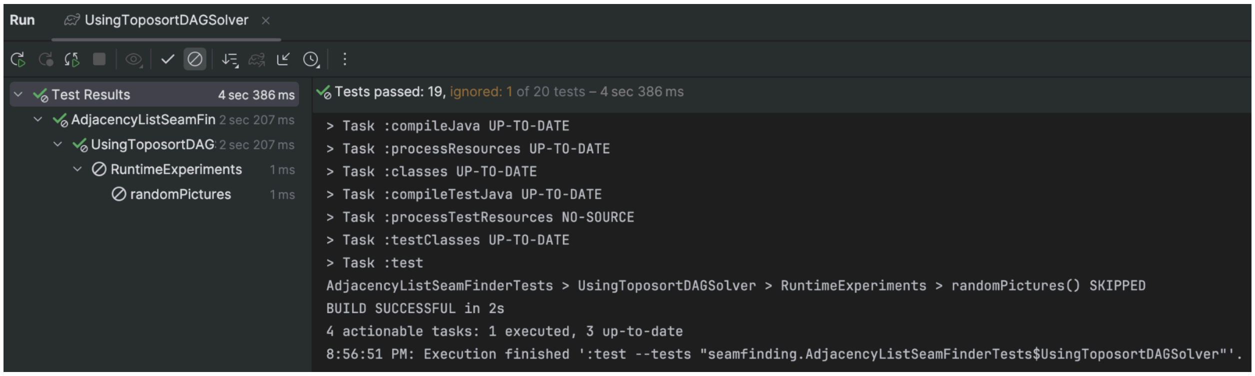
UsingDijkstraSolver in AdjacencyListSeamFinderTests:



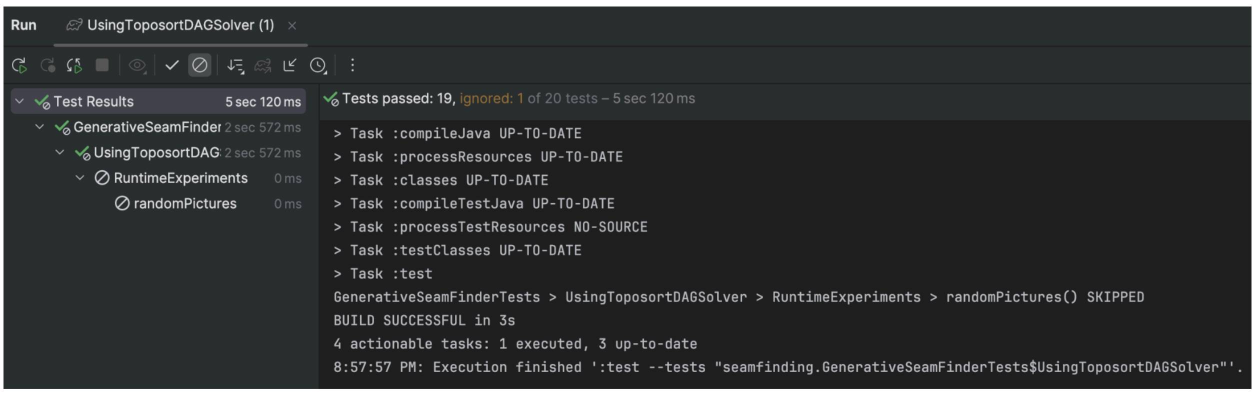
UsingDijkstraSolver in GenerativeSeamFinderTests:



UsingToposortDAGSolver in AdjacencyListSeamFinderTests:



UsingToposortDAGSolver in GenerativeSeamFinderTests:



An interesting bug that I encountered was while implementing my DynamicProgrammingSeamFinder class. Originally, I updated ‘cost[x][y]’ before finding the minimum cost from the predecessors. In the code, this was written as “cost[x][y] = f.apply(picture, x, y);” immediately inside the nested for loop at line 29 before finding min predecessor. So originally, my code calculated and set the current pixel’s energy cost first, found the minimum cost path from predecessors, then added the minimum cost to the already set cost. This causes a bug because it breaks the dynamic programming principle of building solutions from previously computed optimal subproblems. After realizing this, I corrected my implementation to find the minimum cost path from predecessors first, then add the current pixel’s energy to that minimum cost, then store this total in ‘cost[x][y]’. This order is crucial because it ensures that each ‘cost[x][y]’ represents the total minimum energy path from the left edge to that pixel, makes sure that each entry in the cost table represents a complete path cost (not just partial calculation), and prevents double-counting or incorrect accumulation of costs.

Citing Sources

https://stackoverflow.com/questions/2505431/breadth-first-search-and-depth-first-search