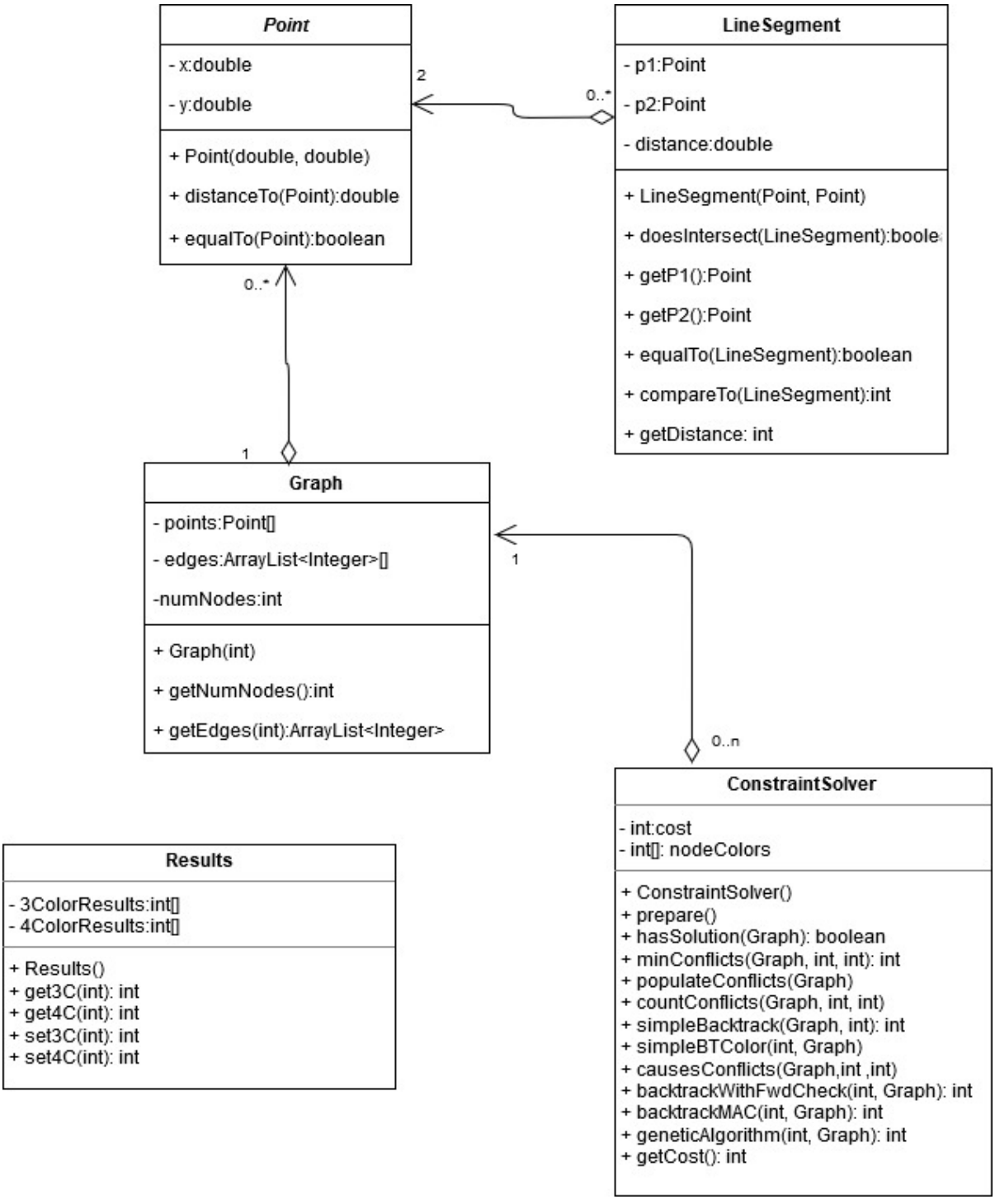
Ben Polk
Ben Barnett
CSCI 446 – Project 1 Design Document

Major Classes:

UML:



Point:

- Used during the construction of the graph; will have double values to represent the x and y attributes of the point
- Will have a method, distanceTo(**Point**), that takes another point object as a parameter and returns the distance between them
- Will have a method to test if it's equal to another point
    - Considered equal if their x values are the same and their y values are the same

LineSegment:

- Used during the construction of the graph. Will have two **Point** objects, point1 and point2
- Will have a distance attribute, equal to point1.distanceTo(point2); the constructor will automatically populate the distance property using the distanceTo function of one of the points
- Will have a method, doesIntersect(**LineSegment**), that returns a true/false value indicating if two line segments intersect
- Will implement the comparable interface in Java based on the distance property; lineSegment1 < lineSegment2 if lineSegment1.distance < lineSegment2.distance; this will allow us to use it with a PriorityQueue during the graph construction
- Will have a method to test if it's equal to another line segment
    - Considered equal if the two line segments use the same points

Graph:

- Will use an adjacency-list representation to store the graph nodes and edges
    - Number of nodes will be represented as an integer
    - Array of arrayList objects will represent a node's edges; an element of a node's arrayList, represents the index of a different node that is connected
- Will have array of **Point** objects to store graph nodes during graph construction
- Constructor will take *n* as parameter for number of nodes in the graph and then generate the graph according to the specifications in the assignment
    - This will require the use of some extra data structures
    - Will have an array of priority queues of type **LineSegment** - PriorityQueue<LineSegment>[]
        - Each priority queue is for a specific point (one of the elements in the array of graph nodes); we'll call that point *p*
        - The priority queue will store the line segments from *p* to all the other points (all the other nodes in the graph)
        - When it comes time to create the graph, and we need to get a point's closest neighbor, we'll be able to just get the next item off that priority queue
    - Will have an ArrayList of **LineSegment** objects to represent the segments we've already created

ConstraintSolver:

- This will be the class that houses all the coloring algorithms
- Will have a cost attribute in order to track the computational cost of an algorithm

- Will have an array tracking the color of each node in the graph, in order to check a coloring for consistency
- Will have one method for each algorithm that takes a graph, an integer for the number of colors, and possibly other arguments
    - minConflicts – needs an extra argument for the maximum number of tries, to prevent infinite (or functionally infinite) execution
    - simpleBacktrack
    - backtrackWithFwdCheck
    - backtrackMAC
    - geneticAlgorithm
- Each algorithm method will use a prepare method to reset the cost, the node colors, and the possible color values
- Each algorithm method will, upon termination, check whether a coloring solution was found for the graph; if not, then the associated cost for that algorithm and number of colors will be negated, in order to still track the cost but also represent that it resulted in no solution
- Will have a method, hasSolution, that returns whether a solution was found for the graph with the given number of colors

Results:

- Simple data structure for grouping the cost of each algorithm to color a given graph
- Has two integer arrays that store the number of major decisions made by an algorithm for 3 coloring and 4 coloring a given graph
- There will be one results object per algorithm, which will allow us to store and later analyze the results for each algorithm

General Architecture:

Part 1 – Graph generation (see pseudocode):

1. Create a point object with random x and y values between 0 and 1
2. Check to make sure that it doesn't equal any already created point objects
3. If it is genuinely new, then add it to the array of graph nodes
4. Repeat steps 1-3 *n* times
5. For each point, fill a priority queue with LineSegment objects; there will be one LineSegment for every combination of the point with all other points in the graph
6. Generate a graph edge by picking a random point and getting the first LineSegment off its priority queue; if that LineSegment is valid then use it to add an edge between the random point and the other point in the LineSegment, if it is not valid then get the next LineSegment off the priority queue
7. Repeat step 6 until all the point priority queues are empty

Pseudocode for graph generation:

```
//************************************************
//*********Code for Graph*****************************

graph(int size) //graph constructor
{
	numNodes = size
	points = new Point[size]
	edges = new ArrayList<Integer>[size]

	for i = 0 to size - 1 //initialize points
		curPoint = new Point(random x, random y)
		if not(alreadyExists(curPoints, point) then points[i] = curPoint
	end loop

	//priority queues to determine closest neighbor
	neighborQueues = new PriorityQueue<LineSegment>[size]

	//populate the queue for i with all the possible line segments from point i
	to other points
	for i = 0 to size - 1
		for j = 0 to size - 1
			if not(i == j) then
				//LineSegment implements comparable based on distance
				property
				neighborQueues[i].add(new LineSegment(points[i],
				points[j]))
			end if
		end loop
	end loop

	breakLoop = false
	allSegments = new ArrayList<LineSegment>

	do while breakLoop = false
		index = random between 0 and size - 1
		oldIndex = index

		//if we picked a point that's gone through all it's neighbors, try
		the next one
		while neighborQueues[index] is empty and breakLoop = false
			index = index + 1
			if index = size then index = 0

			//if we've gone through all the points and they're all out of
			neighbors then break the loops
			if index = oldIndex then breakLoop = true
		end loop
```

Pseudocode for graph generation (cont.):

```
            //get the next eligible line segment
            getNextSegment = true
            while not(neighborQueues[index] is empty) and getNextSegment = true
                  curSegment = neighborQueues[index].dequeue

                  //test for eligibilty - scan allSegments to see if it's
                  already been used or intersects with any other segments
                  if isValidSegment(allSegments, curSegment) then getNextSegment
                  = false
            end loop

            if getNextSegment = false //means the current segment is valid
                  //determine the index of the line segment points in the
                  points[] array
                  i = getIndexOfPoint(curSegment.getPoint1)
                  j = getIndexOfPoint(curSegment.getPoint2)

                  //Add the edges
                  edges[i].Add(j)
                  edges[j].add(i)
            end if
      end loop
}

//helper function to test if a segment is valid
isValidSegment(ArrayList<LineSegment> allSegments, LineSegment testSegment)
{
      for each curSeg in allSegments
            if curSeg.equalTo(testSegment) or curSeg.doesIntersect(testSegment)
            then return false
      end loop

      return true
}

//*********End Code for Graph***********************
//************************************************
```

Part 2 – Graph Coloring (Constraint Solving):

1. Create data structures to hold the results for each algorithm for each number of colors
2. Generate 10 graphs consisting of 10 – 100 nodes (in steps of 10) according to the specification above
3. For each graph, perform each algorithm with 3 colors and 4 colors
    a. minConflicts: This will be a local search algorithm that uses minimum number of conflicts as the heuristic function to determine which value (color) to assign to a variable. It will also have a limit on the number of attempts it makes to repair the graph in order to prevent infinite execution.
        i. Randomly initialize the node colors of the graph (variable assignment)
        ii. Pick a node randomly from the list of nodes that violate constraints (have conflicts)
        iii. Assign that node the value that minimizes its number of conflicts (constraint violations); if there are multiple value assignments that tie for minimum, choose one randomly
        iv. Regenerate the list of nodes with conflicts
        v. Increment the number of attempts
        vi. While the list of conflicted nodes is not empty and the number of attempts is less than the limit, go back to step ii and repeat
    b. simpleBacktrack: This will be a simple implementation of a depth-first search through the possible states (colorings) of the graph in order to find a state that satisfies the constraints.
        i. Begin with node 0
        ii. Assign the first possible valid color (value assignment) to a given node N, valid here meaning a color that doesn't cause N to have any conflicts
        iii. Recursively attempt to find a valid color for the next node, N + 1, assuming N + 1 is a valid node (i.e. recursively go back to step ii above, using node N + 1 instead)
        iv. If node N + 1 was unable to be assigned a valid color, attempt to color node N with the next color
        v. Repeat steps iii-iv for each possible color value
        vi. If no valid color can be found, assign node N the value corresponding to "uncolored" (the backtracking step)
    c. backtrackWithFwdCheck: This will be another implementation of a depth-first search through the possible colorings of the graph. However, instead of assigning the first possible valid color to a node, this algorithm will also perform a "look-ahead", to make sure that that new color does not render any other nodes uncolorable.
        i. The process will essentially be the same as outline above in b. However, in step ii, valid will mean first a color that doesn't cause N to have any conflicts, and second that the assignment passes the look-ahead function.
    d. backtrackMAC: This will be another implementation of a depth-first search through the possible colorings of the graph. However, instead of assigning the first possible valid color to a node, this algorithm will also perform constraint propagation to ensure that the proposed coloring maintains arc-consistency (MAC).

i. The process will essentially be the same as outline above in b. However, in step ii, valid will mean first a color that doesn't cause N to have any conflicts, and second that the assignment maintains arc-consistency with the other nodes.

e. geneticAlgorithm: This algorithm will be based on the Local Search method that looks at different possible states for the graph, and changes the state of the graph without keeping track of where it came from. It will then apply a selection process to this method, where we try various different start states of the graph, change them in small ways, get rid of ones that don't perform well, and focus on the ones that do.

4. Store the cost of each coloring for each algorithm in the results data structures. If no solution was found, make the cost negative in order to represent that.

Experimental Design:

The general experimental design is to run a coloring algorithm on a variety of graphs and track the number of major computational steps taken for each. This is a version of a constraint satisfaction problem, where an agent is attempting to solve the problem by searching through possible states to find one that satisfies the constraints. The relevant constraints are that every node is assigned a color, and that no two neighbors on the graph may share the same color (i.e. for any edge E in the graph defined as connecting nodes P1 and P2, the color assigned to P1 cannot match the color assigned to P2).

The graphs will be generated in such a way that they are guaranteed to be planar and suitable to coloring; also, they are therefore guaranteed to have a solution with 4 colors (although the 3-coloring attempts may fail). The number of nodes in each graph will range from 10 to 100, in steps of 10. While the backtracking algorithms are guaranteed to find a solution if one exists, the ones based on local search are not, so it will be interesting to compare those outcomes.

Varying both the number of nodes in the graph and the number of colors will allow for a robust analysis of the performance of each algorithm. This will be done with a counter that is incremented at each major computational step or decision; the incrementation will mainly occur in loop bodies and/or at the beginning of methods. The "result" of using an algorithm to color a certain graph with a certain number of colors will consist of two parts: the computational cost and whether a solution was found. Once we have obtained all the results, we can present them as charts and graphs. We should then be able to roughly estimate the growth rate of each algorithm in terms of the input parameters, which would allow us to compare our implementations in Big O notation to find the most efficient one.

Ideally, we would run all tests for all algorithm, number of color, and graph size combinations in one procedure in the program. However, due to time constraints, it may become necessary to run the algorithms on smaller sets of graphs. The important factor here will be to ensure that, for any given graph of a specified size, we run all the algorithms for that graph in one iteration of the program. Because the graphs are randomly generated, it would affect the results if we were to generate one graph of *N* nodes, use that for one algorithm, and then generate a completely different graph, still of *N* nodes, and use that for another algorithm. In other words, all algorithms must be tested with the same graph in order to maintain an experimental control and get accurate, meaningful results.

Pseudocode for testing procedure:

```
//************************************************
//*********Code for Main Testing Proc***************

main
{
	//build problem sets
	graphs = new Graph[10]
	for i = 1 to graphs.size
		graphs[i - 1] = new Graph(i * 10)
	end loop

	resultsDict = new Dictionary<String, Result>

	resultsDict.add("minConflicts", new Result(graphs.size))
	resultsDict.add("simpleBacktrack", new Result(graphs.size))
	resultsDict.add("backtrackWithFwdCheck", new Result(graphs.size))
	resultsDict.add("backtrackMAC", new Result(graphs.size))
	resultsDict.add("geneticAlgorithm", new Result(graphs.size))

	solver = new ConstraintSolver

	for i = 0 to graphs.size - 1
		solver.minConflicts(graphs[i], 3, max tries)
		results["minConflicts"].set3C(i, solver.getCost)
		solver.minConflicts(graphs[i], 4, max tries)
		results["minConflicts"].set4C(i, solver.getCost)

		solver.simpleBacktrack(graphs[i], 3)
		results["simpleBacktrack"].set3C(i, solver.getCost)
		solver.simpleBacktrack(graphs[i], 4)
		results["simpleBacktrack"].set4C(i, solver.getCost)

		and so on for each coloring algorithm for 3 and 4 colors
	end loop

	print results as output to screen
}

//*********End Code for Main Testing Proc************
//************************************************
```

References for Program Design:

Backtracking. (2020, January 16). Retrieved January 23, 2020, from https://en.wikipedia.org/wiki/Backtracking

Constraint. (n.d.). Retrieved January 23, 2020, from https://cs.fit.edu/~dmitra/ArtInt/lectures/constraint.pdf

Local consistency. (2019, December 29). Retrieved January 23, 2020, from https://en.wikipedia.org/wiki/Local_consistency#Arc_consistency

Look-ahead (backtracking). (2019, December 31). Retrieved January 23, 2020, from https://en.wikipedia.org/wiki/Look-ahead_(backtracking)

Min-conflicts algorithm. (2019, December 24). Retrieved January 23, 2020, from https://en.wikipedia.org/wiki/Min-conflicts_algorithm