Watts-Strogatz Small World Simulation

User Guide

Spring 2015

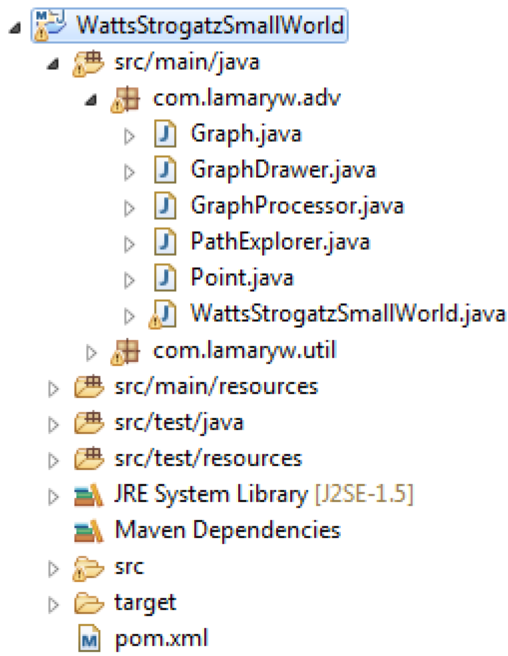
Table of Contents

1. Project Requirements and Use Cases

This project is to simulate a small-world graph according to Watts-Strogatz Model. The small-world graph consists of a ring lattice with 5000 nodes (N = 5000) and each node has 20 short range neighbors (k = 20). The goal is to understand the properties of small-world graphs through simulation.

2. Create project in Eclipse

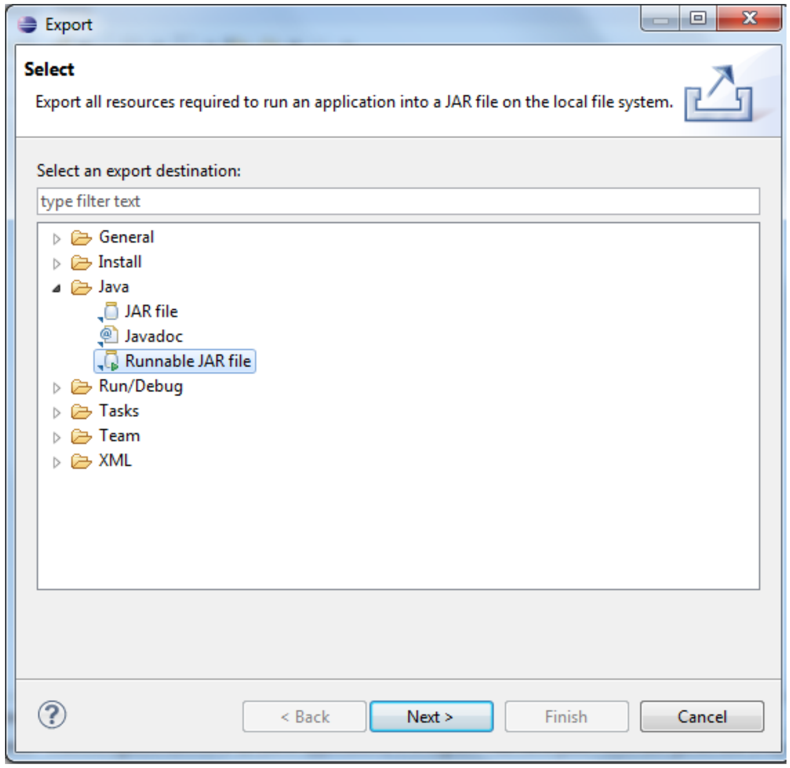
Import the project in to Eclipse



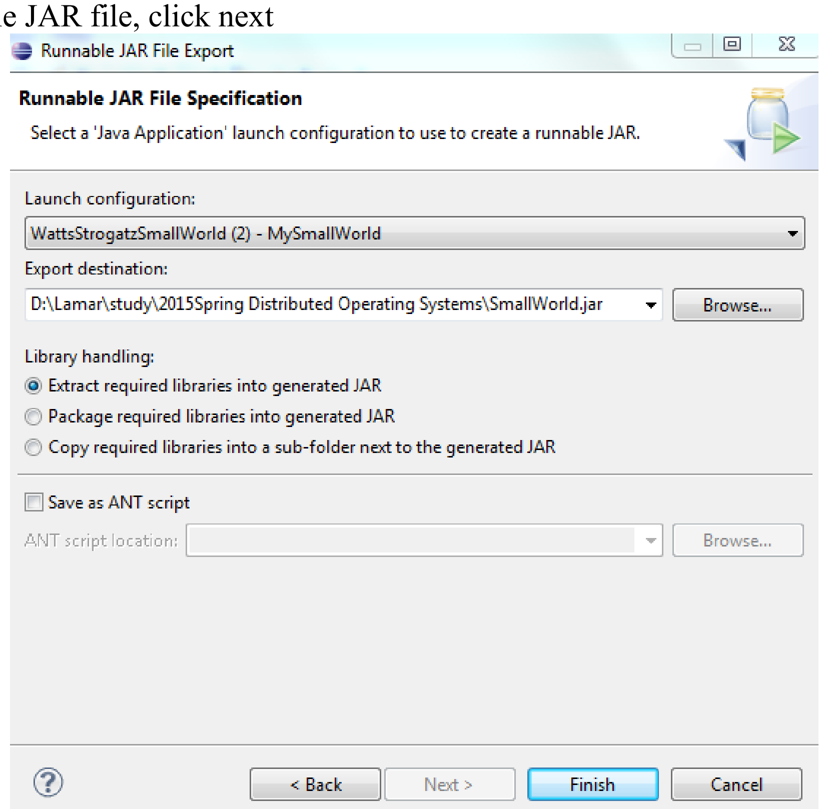
Then you can run the project from file WattsStrogatzSmallWorld.java.

3. Export project as executable jar from Eclipse

Right click the project MySmallWorld, then click Export.



Choose Runnable JAR file, click next

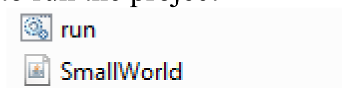


Choose Launch configuration, Export destination and Library handling, then click finish.

Create a run.bat file

```
start java -Xmx256m -Xms128m -jar SmallWorld.jar
```

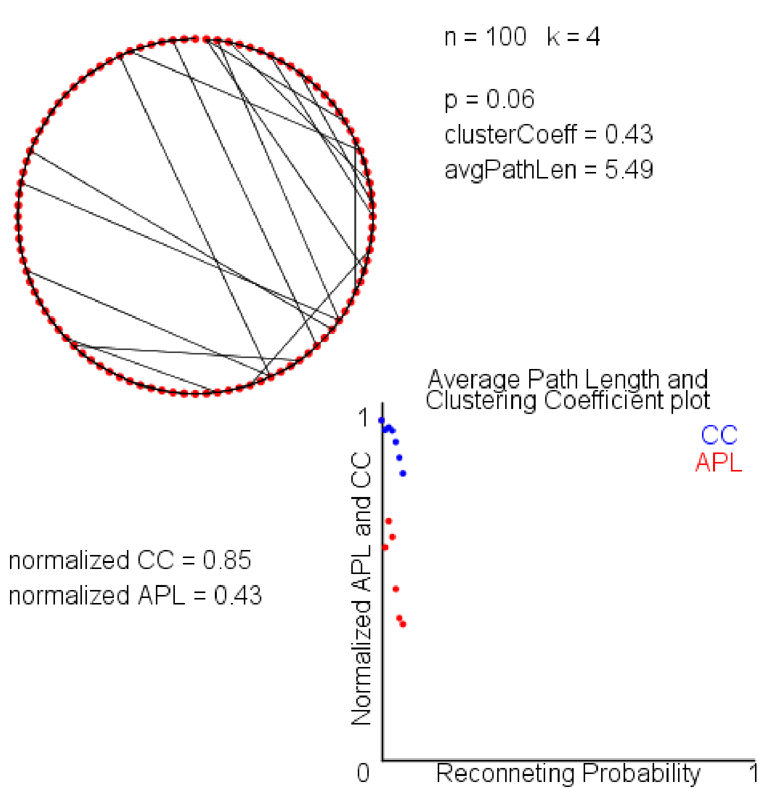
Then you can double click run.bat to run the project



4. Expected Output

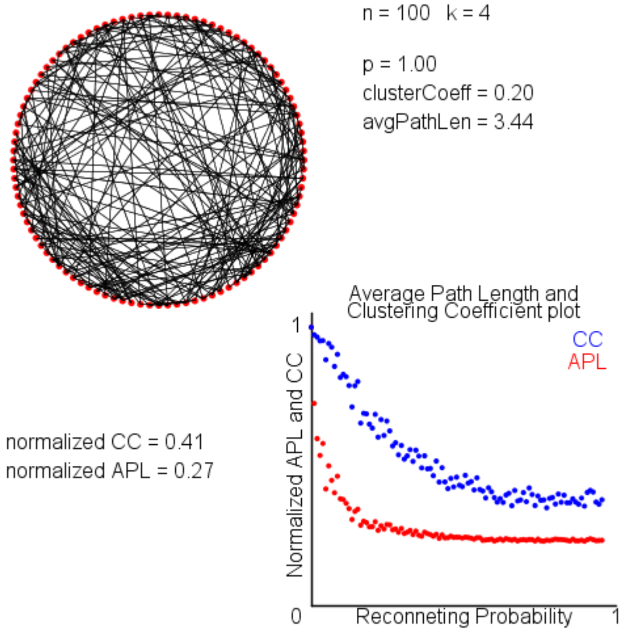
Sample Output 1:

Random reconnecting simulation with p = 0.06 for n = 100, k = 4



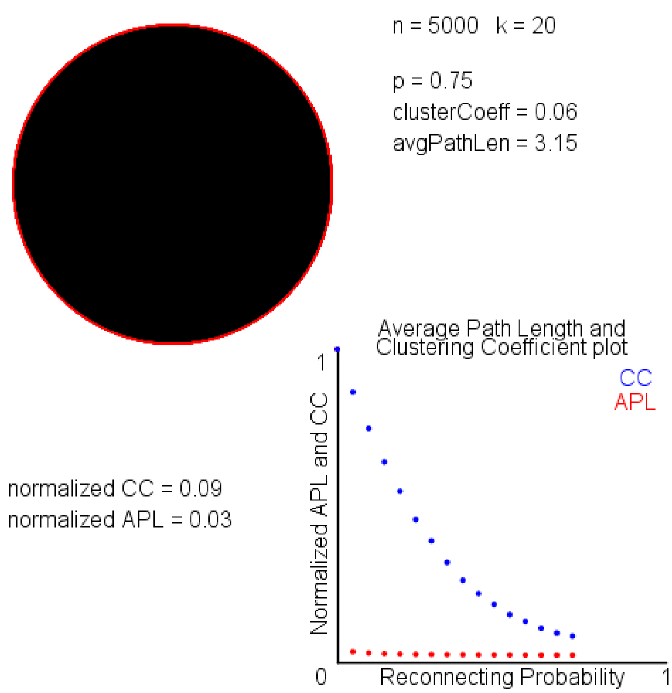
Sample Output 2:

Random reconnecting simulation with p = 1 for n = 100, k = 4



Sample Output 3:

Random reconnecting simulation with p = 0.75 for n = 5000, k = 20.



5. How the Project is Designed and Implemented

To accomplish the simulation, the system is designed to consist of five main parts: Graph, GraphDrawer, GraphProcessor, PathExplorer and a client WattsStrogatzSmallWorld.

The Graph is to implement a graph abstraction data type. This is used represent a graph for processing by the GraphProcessor. The API of the Graph is designed as Table 5.1.

Table 5.1: Graph API

public class Graph		
	Graph()	*create an empty graph*
void	addEdge(Integer v, Integer w)	*add edge v-w*
void	deleteEdge(Integer v, Integer w)	*delete edge v-w*
Iterable<Integer>	getNodes()	*get all nodes in the graph*
Iterable<Integer>	adjacentTo(Integer v)	*neighbors of v*
boolean	hasNode(Integer v)	*is v a node in the graph?*
boolean	hasEdge(Integer v, Integer w)	*is v-w an edge in the graph?*

The GraphDrawer is to draw a graph and display it on a window. This is serve to facilitate us observing and analyzing the transformation process of a graph, from regular to random. The API of the GraphDrawer is designed as Table 5.2.

Table 5.2: GraphDrawer API

public class GraphDrawer		
	GraphDrawer(Graph G)	*create a graphDrawer for a graph G*
void	drawGraph(Graph G)	*draw graph G and display*

The GraphProcessor is to process a graph and compute its clustering coefficient and average path length. The API of the GraphProcessor is designed as Table 5.3.

Table 5.3: GraphProcessor API

public class GraphProcessor		
Graph	createRegularGraph(int n, int k)	*create a regular graph*
void	reconnectGraph(Graph G, double p, int k)	*reconnect graph G*
double	avgPathLen(Graph G)	*compute average path lengh*
double	clusterCoeff(Graph G)	*compute lustering coefficient*

The PathExplorer is to find the length of the shortest path from a source node to any other nodes in a graph. The API of the PathExplorer is designed as Table 5.4.

Table 5.4: PathExplorer API

public class PathExplorer		
	PathExplorer(Graph G, Integer s)	*create a path explorer for source s*
boolean	hasPathTo(Integer v)	*Check if v is reachable from source s*
double	pathLenTo(Integer v)	*Get the shortest path length from s to v*

We use a Map of sets to implementing the Graph abstract data type. The key of the Map is a node's ID and the value is its set of neighbors. We use the breadth-first search algorithm to search the path length in our project. By applying this algorithm, we are able to compute the path length from a source to a destination in linearithmic time. To correctly simulate the Watts-Strogatz small-world model, reconnecting is important and is a key manipulation to transform a regular graph into a small-world graph. The algorithm we use to reconnecting is: For each node i and each edge (i, j) with i < j; with probability p, replace (i, j) with (i, k) where k is chosen uniformly from vertices not equal to or adjacent to i.