Appendix A

Genetic Algorithms

A genetic algorithm (GA) is a computational technique that simulates evolution in the search for solutions to complex problems. These are problems with large spaces of potential solutions, characterized by non-linearities. This means there is no linear relation between the performance of solutions as classified along the dimensions used for characterizing them. If there were linear relations, the optimal solution could easily be found by a simple hill-climbing algorithm. A GA assesses the performance of different solutions in parallel and uses the information about how different solutions perform to direct the search towards promising areas of the search space. Adapting the search is done by simulating evolutionary processes.

In particular, a GA works as follows. A population of potential solutions is created randomly and the algorithm then proceeds in a sequence of generations. In each generation, each of the solutions in the population is tested on the problem and given a fitness-score that expresses how well the solution performed in terms of solving the problem. When all solutions have been tested, poorly performing solutions are thrown away and a new generation is created by allowing highly successful solutions from the previous generation to reproduce. Two genetic operators are often used for this purpose: crossover and mutation. Crossover means that two solutions, often represented as strings of bits, are laid side by side, after which a point on the strings is chosen randomly and the substrings
on the left (or right) of that point are swapped between the two strings. Mutation means that the offspring are changed slightly, usually by flipping bits (from 0 to 1 or vice versa) with a very low probability. Two offspring are created in this way that inherit features from each parent which ensures exploitation of features that made parents successful, but features of different parents are recombined and mutated for the purpose of exploration of new possibilities.

The original publication on genetic algorithms is (Holland 1975). More accessible introductions and handbooks are (Goldberg 1989) and (Davis 1991). Classic examples of applications of genetic algorithms to problems in social science include (Axelrod 1987) and (Miller 1996).
Appendix B

The Simulation Program

B.1 Agent-Based, Object-Oriented Programming

The simulation was developed in the general-purpose, object-oriented programming language SIMULA (Birtwistle et al. 1973). The object-oriented paradigm is very well suited for agent-based modeling (see, e.g. McFadzean and Tesfatsion 1999, Epstein and Axtell 1996), and for real-world modeling in general, which was the philosophy underlying the development of SIMULA as the first object-oriented language. Although the original language (SIMULA I) was a SIMUlation LAnguage, the second and final version, SIMULA 67 (nowadays just called SIMULA), is a general-purpose language, and the acronym now stands for SIMple Universal LAnguage. Object-oriented technology ‘simulates’ the real-world, which gives it several desirable properties. Object-oriented programs are modular; the modules are described in classes. These serve as ‘templates’ for the creation (instantiation) of objects, which represent actual objects in the real world. Classes consist of declarations of data (properties, attributes) and methods (behavior) that operate on those data. Subclasses may be defined that inherit the data and methods of the superclass, and may be re-defined or supplemented with data and methods specific for the subclass. Objects may also send messages to other objects. Object-
oriented programming thus consists of specifying classes. If a program is run, the objects interact with each other by sending messages.

B.2 Specification

The program consists of blocks at different levels. A block begins with 'Begin' and ends with 'End;'. Within a block, declarations go before statements. The global structure of the program is as follows (comments start with "!" and end with ";").

1. Begin Program
2. Class agent;
3. Begin
4. !declaration of variables;
5. !declaration of procedures;
6. End Class agent;

The program starts with the declaration of the class agent (line 2). It contains a number of data (variables) and methods (called procedures in SIMULA).

setAlpha This is the procedure that chooses a value to be used for $\alpha$, using a roulette wheel selection process. The wheel is divided like a pie into as many parts as there are possible values for $\alpha$, with the size of each part proportional to the relative strength of the associated value for $\alpha$, i.e. the appropriate value’s strength divided by the total of all strengths. The roulette wheel is spun like a wheel of fortune and the value for $\alpha$ at which the wheel stops is the one that is chosen.

setTau This is a procedure equivalent to the previous one that chooses a value to be used for $\tau$.

updateAlphaWeights(alphaUsed,payoff) This procedure is called for updating the strength of the value used for $\alpha$ (alphaUsed). This
is done by adding the value of payoff to the appropriate value's current strength.

updateTauWeights(tauUsed, payoff) This is a procedure equivalent to the previous one that is used for updating the strength of the value used for \( \tau \).

7. agent Class buyerAgent;
8. Begin
9. ...
10. End agent Class buyerAgent;

11. agent Class supplierAgent;
12. Begin
13. ...
14. End agent Class supplierAgent;

The class agent is used as a prefix in the declaration of two more classes buyerAgent (l. 7) and supplierAgent (l. 11), so that those two classes `inherit` the class agent's data and procedures. These may be supplemented with data and methods declared within the subclass, to further specify subclasses and differentiate them from each other and from the superclass. In addition to the data and methods inherit from the class agent, the class buyerAgent contains the following procedures:

calculateSupplierScores This procedure calculate scores of suppliers and of the buyer himself as Cobb-Douglas functions of profitability and trust, as described above in Section 3.2.

increaseTrust(subject) This increases the buyer's trust in subject on the basis of the number of times they have been matched.

decreaseTrust(subject) This decreases the buyer's trust in subject.

buyerProcess If the buyer is not matched then he makes, and he sells his product in any case. The suppliers' equivalent supplierProcess is executed before the buyers', so if a buyer is matched to a supplier, that supplier will already have produced for him.
Next to the procedures inherited from the class agent, the class supplierAgent contains the following procedures.

- **calculateBuyerScores** This procedure calculates the scores the supplier assigns to each buyer.

- **determineScaleEfficiency** Depending on the number of buyers he is matched to and the differentiation of their products, the supplier determines the scale-efficiency in using general-purpose assets.

- **climbLearningCurve(subject)** On the basis of the number of times they have been matched before, the supplier calculates his efficiency due to learning-by-doing in using subject-specific assets.

- **increaseTrust(subject)** This procedure increases the supplier's trust in subject on the basis of the number of previous times they have been matched.

- **decreaseTrust(subject)** This procedure decreases the supplier's trust in subject.

- **produceFor(subject)** Based on the differentiation of buyer subject's products and the supplier's scale- and subject-specific efficiency, the supplier produces for subject.

- **supplierProcess** Looking at each buyer in turn, if a supplier is matched to that buyer, it produces for that buyer (see the previous procedure produceFor(subject)).

15. Procedure matchAgents;
16. Begin
17. ...
18. End procedure matchAgents;

Next is the declaration of the procedure matchAgents (l. 15), which implements the DCR algorithm.
Now the main loop of the program is entered, after some more variables are declared, such as experiment, run and timeStep (l. 19) and the simulation is initialized (l. 20). The number of buyers and suppliers in the simulation has to be supplied by the user, as well as the duration of the simulation (in timesteps) and the number of repetitions of each simulation (in runs), among others.

```
21. For experiment:=0.25 Step 0.1 Until 0.70 Do Begin
22. For run:=1 Step 1 Until totalRuns Do Begin
23. !initialization of buyers;
24. !initialization of suppliers;
25. For timeStep:=1 Step 1 Until totalTimeSteps Do Begin

The actual simulation now begins. In this case, the experiments are hard-coded in the simulation (l. 21), in the form of different values for the variable experiment, which are then given to the variable differentiation of each buyerAgent. Within each experiment (the for-loop in line 21 is executed for each value of the experiment specified, i.e. 0.25, 0.35, ..., 0.65), a number of runs are executed (l. 22), in each of which, a number of timesteps is cycled through (l. 25). At the beginning of each run, the buyers and suppliers are re-initialized (l. 23–24), because their variables may still contain values from the previous run.

```
26. For count:=1 Step 1 Until totalBuyers Do Begin
27. For count2:=1 Step 1 Until totalSuppliers Do Begin
28. If isSupplier(count2,count) Then Begin
29.   supplier(count2).climbLearningCurve(count);
30.   supplier(count2).increaseTrust(count);
31.   buyer(count).increaseTrust(count2);
32.   End;
33.   End;
34.   End;
All possible combinations of agents are now examined (l. 26–27), to see whether they were matched to each other in the previous run. If they were, which is the case if the corresponding entry in the 2-dimensional boolean array, i.e. matrix isSupplier(·,·) is True, then the supplier identified by the value of count2, climbs the buyer-specific learning curve for the buyer identified by count, (l. 29)) and they increase their trust in each other (l. 30–31).

35. For count:=1 Step 1 Until totalSuppliers Do Begin
36. Inspect supplier(count) Do Begin
37. setAlpha;
38. setTau;
39. calculateBuyerScores;
40. End;
41. End;

42. For count:=1 Step 1 Until totalBuyers Do Begin
43. Inspect buyer(count) Do Begin
44. setAlpha;
45. setTau;
46. calculateSupplierScores;
47. End;
48. End;

All suppliers now assign scores to all the buyers (l. 39) and all buyers assign scores to all suppliers plus themselves (l. 46). Before they do, however, they choose a value for α (l. 37 and 44) and for τ (l. 38 and 45).

49. For count:=1 Step 1 Until totalBuyers Do Begin
50. For count2:=1 Step 1 Until totalSuppliers Do Begin
51. wasSupplier(count2,count):=isSupplier(count2,count);
52. isSupplier(count2,count):=False;
53. End;
54. End;

All connections from the previous timestep, stored in the boolean matrix isSupplier(·,·), are now copied to the boolean matrix wasSupplier(·,·)
B.2. SPECIFICATION

(1.51), after which the first is emptied, i.e. set to False (1.52). It will be re-filled on the basis of what happens during the matching.

55. matchAgents;

56. For count:=1 Step 1 Until totalBuyers Do Begin
57. For count2:=1 Step 1 Until totalSuppliers Do Begin
58. If requestSend(count,count2)
59. And Not hasRejected(count2,count)
60. Then Begin
61. isSupplier(count2,count):=True;
62. End Else Begin
63. If wasSupplier(count2,count) Then Begin
64. supplier(count2).efficiency(count):=0;
65. If requestSend(count,count2) Then Begin
66. buyer(count).decreaseTrust(count2);
67. End Else Begin
68. supplier(count2).decreaseTrust(count);
69. End;
70. End;
71. End;
72. End;
73. End;

The procedure matchAgents is called (1.55), which makes the agents establish strict preference-rankings on the basis of the scores they have just assigned, and matches them on the basis of those preference-rankings. All possible connections between buyers and suppliers are examined again (1.56-57); if a request was send from buyer count to supplier count2 (1.58), then they are matched (isSupplier(count2,count) is set to True; 1.61) if the supplier did not reject that request (1.59). Otherwise, the supplier drops off of his buyer-specific learning curve for buyer count—if he was on it (1.63)—and the buyer decreases his trust in the supplier if he did send a request (1.65-66), while the supplier decreases his trust in the buyer if the buyer did not send a request (1.67-68).
APPENDIX B. THE SIMULATION PROGRAM

74. For count:=1 Step 1 Until totalSuppliers Do Begin
75.  Inspect supplier(count) Do Begin
76.  If nrBuyers(count)>0 Then determineScaleEfficiency;
77.  End;
78.  End;

The buyers assigned scores to suppliers on the basis of their economies of scale in the previous timestep. Only after the matching, however, does it become clear to how many and to which buyers each suppliers is really matched, and what each supplier’s real scale-efficiency is (l. 76).

79. For count:=1 Step 1 Until totalSuppliers Do Begin
80.  Inspect supplier(count) Do Begin
81.  supplierProcess;
82.  End;
83.  End;
84. For count:=1 Step 1 Until totalSuppliers Do Begin
85.  Inspect supplier(count) Do Begin
86.  updateAlphaWeights(alpha,actualProfit);
87.  updateTauWeights(tau,actualProfit);
88.  End;
89.  End;

Each supplier now executes his main procedure, supplierProcess (l. 81). This procedure consists of producing for and supplying to all buyers the supplier is matched to. The fact that all suppliers have to execute their supplierProcess before each of them can update the strengths of the values used for $\alpha$ and for $\tau$, is a remnant from an earlier version of the simulation model that did not employ a matching algorithm as such (see Klos and Nootboon 1998). In that model, what each supplier did affected the other suppliers’ outlook for that timestep. The order in which the suppliers were activated was randomized for that reason (see Huberman and Glance’s (1993) comments about synchronization of agent-interactions in relation to Nowak and May’s (1992) computational prisoner’s dilemma experiments).
Finally, the buyers execute their buyerProcess (l. 92), after which they update the strengths of the values they used for $\alpha$ (l. 93) and $\tau$ (l. 94). During the course of the simulation, enormous amounts of data are written to files, and processed to produce graphs such as those in Chapter 4.
Appendix C

Parameters and Variables

<table>
<thead>
<tr>
<th>parameter/variable</th>
<th>value range</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>general</strong></td>
<td></td>
</tr>
<tr>
<td>number of buyers, $I$</td>
<td>${1, 2, \ldots}$</td>
</tr>
<tr>
<td>number of suppliers, $J$</td>
<td>${1, 2, \ldots}$</td>
</tr>
<tr>
<td>number of runs, $R$</td>
<td>${1, 2, \ldots}$</td>
</tr>
<tr>
<td>number of timesteps, $T$</td>
<td>${1, 2, \ldots}$</td>
</tr>
<tr>
<td>number of values for $\alpha$</td>
<td>${2, 3, \ldots}$</td>
</tr>
<tr>
<td>number of values for $\tau$</td>
<td>${2, 3, \ldots}$</td>
</tr>
<tr>
<td>renormalization constant $C_\alpha$</td>
<td>$\langle 0, \ldots \rangle$</td>
</tr>
<tr>
<td>renormalization constant $C_\tau$</td>
<td>$\langle 0, \ldots \rangle$</td>
</tr>
<tr>
<td>baseTrust, $b$</td>
<td>$\langle 0, 1 \rangle$</td>
</tr>
<tr>
<td>initTrust(subject)</td>
<td>$\langle 0, 1 \rangle$</td>
</tr>
<tr>
<td>trustFactor</td>
<td>${0, 1}$</td>
</tr>
<tr>
<td><strong>per</strong></td>
<td></td>
</tr>
<tr>
<td>differentiation, $d_i$</td>
<td>${0, 1}$</td>
</tr>
<tr>
<td><strong>buyer</strong> $i$</td>
<td></td>
</tr>
<tr>
<td>offer quota, $o_i$</td>
<td>${1, 2, \ldots, J}$</td>
</tr>
<tr>
<td><strong>per</strong></td>
<td></td>
</tr>
<tr>
<td>acceptance quota, $a_j$</td>
<td>${1, 2, \ldots, I}$</td>
</tr>
<tr>
<td><strong>supplier</strong> $j$</td>
<td></td>
</tr>
<tr>
<td>scaleFactor</td>
<td>${0, 1}$</td>
</tr>
<tr>
<td>learnFactor</td>
<td>${0, 1}$</td>
</tr>
</tbody>
</table>

Table C.1: Parameters and variables allowed in the simulation.