Chapter 14 Creating the Two Slightly More Complex Benchmarks

Abstract To test the performance of a neuroevolutionary system after adding a new feature, or in general when trying to assess its abilities, it is important to have some standardized benchmarking problems. In this chapter we create two such benchmarking problems, the Pole Balancing Benchmarks (Single, Double, and With and Without dampening), and the T-Maze navigation benchmark, which is one of the problems used to assess the performance of recurrent and plasticity enabled neural network based systems.

Though we have created an extendible and already rather advanced TWEANN platform, how can we prove it to be so when we only have the basic XOR benchmark to test it on? As we continue to improve and advance our system, we will need to test it on more advanced benchmarks. In this chapter we develop and add two such benchmarking problems, the pole balancing benchmark, and the T-Maze navigation benchmark. Both of these benchmarks are standard within the computational intelligence field, and our neuroevolutionary system's ability to solve them is the minimum requirement to be considered functional.

To allow our TWEANN to use these benchmarks, we need to create a simulation/scape of the said problems, and create the agent morphology that contains the sensors/actuators that the NN based agents can use to interface with these new scapes. In the following sections we will first build the pole balancing simulation. Afterwards, we will develop the T-Maze simulation, a problem which can be much better solved by a NN system which can learn and adapt as it interacts with the environment, by a NN which has plasticity (a feature we will add to our neuroevolutionary system in Chapter-15).

Once these two types of new simulations are created, we will briefly test them, and then move on to the next chapter, where we will begin advancing and expanding our neuroevolutionary system.

14.1 Pole Balancing Simulation

The pole balancing benchmark consists of the NN based agent having to push a cart on a track, such that the pole standing on the cart is balanced and does not tip over and fall. Defined more specifically, the pole balancing problem is posed as follows: Given a two dimensional simulation of a cart on a 4.8 meter track, with a pole of length L on the top of a cart, attached to the cart by a hinge, and thus free to swing, the NN based controller must apply a force to the cart, pushing it back

G.I. Sher, *Handbook of Neuroevolution Through Erlang*,
DOI 10.1007/978-1-4614-4463-3_14,

and forth on the track, such that the pole stays balanced on the cart and within 36 degrees of the cart's vertical. For sensory inputs, the NN based agent is provided with the cart's position and velocity, and the pole's angular position (from the vertical) and angular velocity. The output of the NN based agent is the force value *F* in newtons (N), saturated at 10N of magnitude. Positive F pushes the cart to the left, and negative pushes it to the right. Given these conditions, the problem is to balance the pole on the cart for 30 simulated minutes, or as long as possible, where the fitness is the amount of time the NN can keep the pole balanced by pushing the cart back and forth.

The temporal granularity of the simulation is 0.01 seconds, which means that every 0.01 seconds we perform all the physics based calculations, to determine the position of the cart and the pole. The Agent requests sensory signals and acts every 0.02 seconds. The simulation termination conditions are as follows: the cart must stay on the 4.8 meter track or the simulation ends, the simulation also ends if the pole falls outside the 36 degrees of the vertical.

There are multiple versions of this problem, each one differs in its difficulty:

1. The simple single pole balancing problem, as shown in Fig-14.1a. In this simulation the NN based agent pushes the cart to balance the single 1 meter pole on it. This problem is further broken down into two different versions.
   - The NN receives as a sensory signal the cart's position on the track (CPos), the cart's velocity (CVel), the pole's angular position (PAngle), and the pole's angular velocity (PVel). Sensory_Signal = [CPos, CVel, PAngle, PVel].
   - The NN receives as a sensory signal only the CPos and PAngle values. To figure out how to solve the problem, how to push the cart and in which direction, the NN will need to figure out how to calculate the CVel and PVel values on its own, which requires recurrent connections. Sensory_Signal = [Cpos,PAngle].

It is possible to very rapidly move the cart back and forth, which keeps the pole balanced. To prevent this type of a solution, the problem is sometimes further modified with the fitness of the NN based agent not only being dependent on the amount of time it has balanced the pole, but on how smoothly it has pushed the cart. One type of fitness function simply rewards the NN based on the length of time it has balanced the pole, while the other rewards the NN based on the length of time it has balanced the pole, and penalizes it for very high velocities and rapid velocity changes. The first is the standard fitness function, while the other is called the damping fitness function.

2. A more difficult version of the pole balancing problem is the double pole balancing version, as shown in Fig-14.1b. In this problem we try to balance two poles of differing lengths at the same time. The closer the lengths of the two poles are, the more difficult the problem becomes. Usually, the length of one pole is set to 0.1 meters, and the length of the second is set to 1 meter. As with the single pole balancing problem, there are two versions of this, and again for each version we can use either of the two types of fitness functions:

- The sensory signal gathered by the NN is composed of the cart's position and velocity (CPos,CVel), the first pole's angle and velocity (P1_Angle, P1_Vel), and the second pole's angle and velocity (P2_Angle, P2_Vel). Sensory_Signal = [CPos,CVel,P1_Angle,P1_Vel,P2_Angle,P2_Vel].
- The second more complex version of the problem, just as with the single pole balancing problem, only provides the NN with partial state information, the cart's position, and the first and second pole's angular position. Sensory_Signal = [CPos,P1_Angle,P2_Angle]. This requires the NN based agent to derive the velocities on its own, which can be done by evolving a recurrent NN topology.

As with the single pole balancing problem, the fitness can be based on simply the amount of time the poles have been balanced, or also on the manner in which the agent pushes the cart, using the damping fitness function.

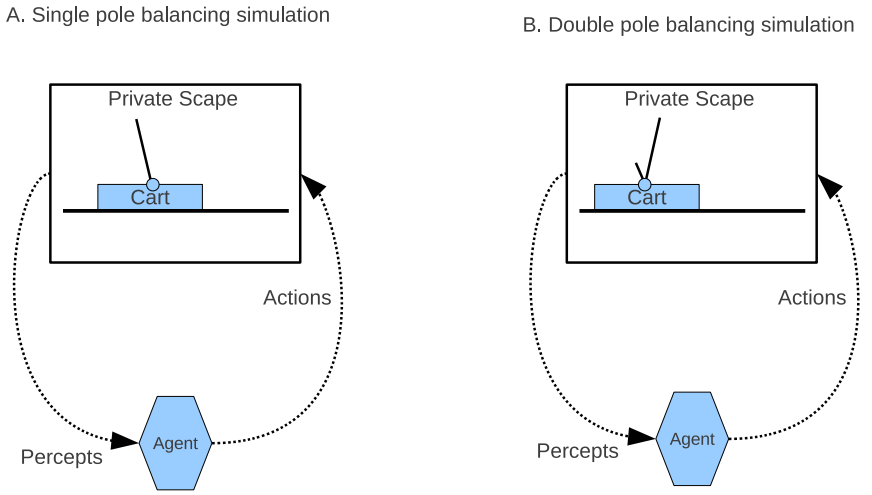

Fig. 14.1 The architecture of single (A.) and double (B.) pole balancing simulations, represented as private scapes with which the agents can interface with, to push the cart and balance the pole/s.

As with the XOR simulator, we will set the pole balancing simulation to be self contained in a private scape process, which will accept *sense* and *push* messages from the agent to whom it belongs. Since the simulation of the track/cart/pole is independent of the types of sense signals the agent wishes to use, we will only need to implement a single version of such private scape. We will implement the system using a realistic physical model of the system, and fourth order Runge-Kutta integration, as is specified and done in [1].

Because the two-pole balancing problem is simply an extension of the single pole balancing problem, and because the two poles are independent of each other, we can create a single double pole balancing simulator, which can then be used for either benchmark. It will be the *sense* and *force* messages that determine what in-

formation is sent to the sensors of the NN based agent. Furthermore, depending on the parameters sent by the actuator of the agent, the scape will calculate the fitness and decide on whether to use both poles or only a single pole with regards to the termination conditions.

Thus, the scape will always be simulating two poles. But if the agent is being applied to the single pole balancing problem, and this fact will be specified by the actuator and sensor pair used by the agent, the scape which receives the messages from the sensor and actuator of that agent, will simply not take into account the second pole. In this manner, if the second pole falls, deviates more than 36 degrees from the vertical... it will not trigger the termination condition or affect the fitness in any way. The parameter sent by the actuator will notify the scape that the agent is only concerned with the single pole being balanced.

We will set up the functionality of each such pole balancing simulation, contained and wrapped in a private scape, represented as a single process, to use the following steps:

1. PB (pole balancing) private scape is spawned.
2. The PB scape initializes the physical simulation, with the first pole's initial angle from the vertical randomly selected to be between -3.6 and 3.6 degrees, and the second pole's angle set to 0 degrees. Furthermore, the first pole's length will be set to 1 meter, and 0.1 meter for the second one.
3. The PB process drops into its main loop, and awaits for sense and push messages.
4. **DO:**
    5. **If** {From_PId, *sense*, Parameters} message is received: The Parameters value specifies what type of sensory information should be returned to the caller. If Parameters is set to 2, then the scape will return the cart position and the pole position information. If the Parameters value is set to 3, then the scape will return the cart, pole_1, and pole_2 positions. If 4, then cart position and velocity, plus pole_1 angular position and velocity, will be returned. Finally, if Parameters is set to 6, then the scape will return the cart position and velocity, and the pole_1 and pole_2 angular positions and velocities.
    6. **If** {From_PId, *push, Force,* Parameters} message is received: The PB scape applies the force specified in the message to the cart, and calculates the results of the physical simulation. The response to the push are calculated for two 0.01s time steps, taking the simulation 0.02 seconds forward, and then returning the scape back to waiting for the sense/push messages again. Furthermore, the *Parameters* value will have the form: {Damping_Flag, PB_Type}, where the Damping_Flag parameter specifies whether the fitness function will be calculated with damping features to prevent the rapid shaking of the cart, and where the PB_Type parameter specifies whether the private scape should be used as a single pole or double pole balancing simulator. If it is used as a single pole balancing

simulator, then the condition of the second pole will not affect the fitness value, and its reaching the termination condition (falling beyond 36 degrees from the vertical) will not end the simulation.

UNTIL: Termination condition is reached (goal number of time steps, or one of the boundary condition breaches).

The termination condition is considered to be any one of the following:

- The simulation has run for 30 simulated minutes, which is composed of 90000 0.02 second time steps.
- The pole has deviated 36 or more degrees from the cart's vertical.
- The cart has left the track. The track itself is 4.8 meters long, and the cart will start at the center, and thus be 2.4 meters away from either side. If it goes beyond -2.4 or 2.4 point on the axis of the track, the termination condition is reached.

Based on this architecture, we will in the following subsection create the private scape process, and its main loop which after receiving the *push* message calls the function which does the physical simulation of the track/cart/pole system. Afterwards, we will create the sensors/actuators and the new morphology specification entry in the morphology module. These will be the sensors and actuators used by the agents to interface with this type of private scape. Finally, we will then compile and run a quick test of this new problem, to see how well our system performs.

14.1.1 Implementing the Pole Balancing Scape

For the pole balancing simulation, the process will need to keep track of the position of the cart on the track, its velocity, the angular position and velocity of both poles, the time step the simulation is currently in, the goal time steps, and finally the fitness accumulated by the interfacing agent. To keep track of all these values, we will use a state record. Listing-14.1 shows the implementation of the pb_sim/2, the pole balancing simulation scape. We will add the source code of this listing to the scape module. The comments after every function in Listing-14.1 elaborate on the details of its implementation.

Listing-14.1 The complete implementation of the pole balancing simulation scape.

```
-record(pb_state,{cpos=0,cvel=0,p1_angle=3.6*(2*math:pi()/360),p1_vel=0, p2_angle=0,
p2_vel=0, time_step=0, goal_steps=90000,fitness_acc=0}).

pb_sim(ExoSelf_PId)->
    random:seed(now()),
    pb_sim(ExoSelf_PId,#pb_state{}).
```

```
%pb_sim/1 is executed to initialize and startup the pole balancing simulation scape. Once exe-
cuted it creates initial #pb_state{}, and drops into the main simulation loop.

pb_sim(ExoSelf_PId,S)->
    receive
        {From_PId,sense, [Parameter]}->
            SenseSignal=case Parameter of
                cpos -> [S#pb_state.cpos];
                cvel -> [S#pb_state.cvel];
                p1_angle -> [S#pb_state.p1_angle];
                p1_vel -> [S#pb_state.p1_vel];
                p2_angle -> [S#pb_state.p2_angle];
                p2_vel -> [S#pb_state.p2_vel];
                2 -> [S#pb_state.cpos,S#pb_state.p1_angle];
                3 -> [S#pb_state.cpos,S#pb_state.p1_angle,S#pb_state.p2_angle];
                4 -> [S#pb_state.cpos, S#pb_state.cvel, S#pb_state.p1_angle,
S#pb_state.p1_vel];
                6 -> [S#pb_state.cpos, S#pb_state.cvel, S#pb_state.p1_angle,
S#pb_state.p1_vel, S#pb_state.p2_angle, S#pb_state.p2_vel]
            end,
            From_PId ! {self(),SenseSignal},
            pb_sim(ExoSelf_PId,S);
        {From_PId,push,[Damping_Flag,DPB_Flag], [F]}->
            AL = 2*math:pi()*(36/360),
            U_S=sm_DoublePole(F,S,2),
            TimeStep=U_S#pb_state.time_step,
            CPos=U_S#pb_state.cpos,
            CVel=U_S#pb_state.cvel,
            PAngle1=U_S#pb_state.p1_angle,
            PVel1=U_S#pb_state.p1_vel,
            case (abs(PAngle1) > AL) or (abs(U_S#pb_state.p2_angle)*DPB_Flag > AL)
or (abs(CPos) > 2.4) or (TimeStep >= U_S#pb_state.goal_steps) of
                true ->
                    From_PId ! {self(),0,1},
                    pb_sim(ExoSelf_PId,#pb_state{});
                false ->
                    Fitness = case Damping_Flag of
                        without_damping ->
                            1;
                        with_damping ->
                            Fitness1 = TimeStep/1000,
                            Fitness2 = case TimeStep < 100 of
                                true ->
                                    0;
                                false ->
```

```
                                                    0.75/(abs(CPos) +abs(CVel) +
abs(PAngle1) + abs(PVel1))
                                                end,
                                                Fitness1*0.1 + Fitness2*0.9
                                    end,
                                    From_PId ! {self(),Fitness,0},
                                    pb_sim(ExoSelf_PId, U_S#pb_state{fitness_acc
=U_S#pb_state.fitness_acc+Fitness})
                        end;
                {ExoSelf_PId,terminate} ->
                        ok
    end.
```

%The pole balancing simulation scape can accept 3 types of messages, *push, sense*, and *terminate*. When a sense message is received, the scape checks the Parameter value, and based on whether the Parameters == 2, 3,4, or 6, it returns a sensory list with an appropriate number of elements. 2 and 4 specify that the NN based agent wants a sensory signal associated with the single pole balancing problem, with partial or full system information, respectively. 4 and 6 implies that the NN wants the scape to send it sensory information associated with double pole balancing, with partial or full system information respectively. When the scape receives the push message, based on the message it decides on what fitness function is used (with or without damping), the actual force to be applied to the cart, and whether the termination condition should be based on the single pole balancing problem (DPB_Flag=0) or double pole balancing problem (DPB_Flag=1). When the angle of the second pole is multiplied by DPB_Flag which is set to 0, the value will always be 0, and thus it cannot trigger the termination condition of being over 36 degrees from the vertical. When it is multiplied by DPB_Flag=1, then its actual angle is used in the calculation of whether the termination condition is triggered or not. Once the message is received, the scape calculates the new position of the poles and the cart after force F is applied to it. The state of the poles/cart/track system is updated by executing the sm_DoublePole/3 function, which performs the physical simulation calculations.

```
sm_DoublePole(_F,S,0)->
    S#pb_state{time_step=S#pb_state.time_step+1};
sm_DoublePole(F,S,SimStepIndex)->
    CPos=S#pb_state.cpos,
    CVel=S#pb_state.cvel,
    PAngle1=S#pb_state.p1_angle,
    PAngle2=S#pb_state.p2_angle,
    PVel1=S#pb_state.p1_vel,
    PVel2=S#pb_state.p2_vel,
    X = CPos, %EdgePositions = [-2.4,2.4],
    PHalfLength1 = 0.5, %Half-length of pole 1
    PHalfLength2 = 0.05, %Half-length of pole 2
    M = 1, %CartMass
    PMass1 = 0.1, %Pole1 mass
    PMass2 = 0.01, %Pole2 mass
```

```
    MUc = 0.0005, %Cart-Track Friction Coefficient
    MUp = 0.000002, %Pole-Hinge Friction Coefficient
    G = -9.81, %Gravity
    Delta = 0.01, %Timestep
    EM1 = PMass1*(1-(3/4)*math:pow(math:cos(PAngle1),2)),
    EM2 = PMass2*(1-(3/4)*math:pow(math:cos(PAngle2),2)),
    EF1 = Pmass1*PHalfLength1*math:pow(PVel1,2)*math:sin(PAngle1)+(3/4)*PMass1
*math:cos(PAngle1)*(((MUp*PVel1)/(PMass1*PHalfLength1))+G*math:sin(PAngle1)),
    EF2 = Pmass2*PHalfLength2*math:pow(PVel2,2)*math:sin(PAngle2)+(3/4)*PMass2
*math:cos(PAngle2)*(((MUp*PVel2)/(PMass1*PHalfLength2))+G*math:sin(PAngle2)),
    NextCAccel = (F - MUc*functions:sgn(CVel)+EF1+EF2)/(M+EM1+EM2),
    NextPAccel1 = -(3/(4*PHalfLength1))*((NextCAccel*math:cos(PAngle1))
+(G*math:sin(PAngle1))+((MUp *PVel1)/(PMass1*PHalfLength1))),
    NextPAccel2 = -(3/(4*PHalfLength2))*((NextCAccel*math:cos(PAngle2))
+(G*math:sin(PAngle2))+((MUp *PVel2)/(PMass2*PHalfLength2))),
    NextCVel = CVel+(Delta*NextCAccel),
    NextCPos = CPos+(Delta*CVel),
    NextPVel1 = PVel1+(Delta*NextPAccel1),
    NextPAngle1 = PAngle1+(Delta*NextPVel1),
    NextPVel2 = PVel2+(Delta*NextPAccel2),
    NextPAngle2 = PAngle2+(Delta*NextPVel2),
    U_S=S#pb_state{
            cpos=NextCPos,
            cvel=NextCVel,
            p1_angle=NextPAngle1,
            p1_vel=NextPVel1,
            p2_angle=NextPAngle2,
            p2_vel=NextPVel2
    },
    sm_DoublePole(0,U_S,SimStepIndex-1).
%sm_DoublePole/3 performs the calculations needed to keep track of the two poles and the
cart, it simulates the physical properties of the track/cart/pole system. The granularity of the
physical simulation is 0.1s, and so to get a state at the end of 0.2s, the calculation of the state is
performed twice at the 0.1s granularity. During the first execution of the physical simulation we
have the force set to the appropriate force sent by the neurocontroller. But during the second,
F=0. Thus the agent actually only applies the force F for 0.1 seconds. This can be changed to
have the agent apply the force F for the entire 0.2 seconds.
```

With the simulation completed, we now need a way for our agents to spawn and interface with it. This will be done through the agent's morphology, its sensors and actuators, which we will create next.

14.1.2 Implementing the Pole Balancing morphology

Like the case with the xor_mimic morphology function, which when called returns the available sensors or actuators for that particular morphology, we will in this subsection develop the pole_balancing/1 morphology function which does the same. Unlike the xor_mimic though, here we will also populate the *parameters* element of the sensor and actuator records.

For both sensors and actuators we will again specify the scape element to be of type private: *scape = {private, pb_sim}*. For the sensor, we will set the parameters to: [2], this parameter can then be modified to 3, 4, or 6, dependent on what test we wish to apply the population of agents to. After every such *parameters value* change, the morphology module would then have to be recompiled before use. We could simply create multiple morphologies, for example: *pole_balancing2, pole_balancing3, pole_balancing4,* and *pole_balancing6*, but that would not add an advantage over changing the parameters and recompiling, since it would still require us to use our neuroevolutionary system on different problems and thus to change the constraints in either population_monitor or benchmarker modules, and then recompile them still...

Similarly, the actuator record's *parameters* element is set to: [no_damping,0]. The *no_damping* tag specifies that the fitness function used should be the simple one that does not take damping into account. The *0* element of the list specifies, based on our implementation of the *pb_sim*, that the second pole should not be taken into account when calculating the fitness and whether the termination condition is reached. This is achieved in: *(abs(U_S#pb_state.p2_angle)*DPB_Flag > AL)* , where *DPB_Flag* is either *0* or *1*. When set to 1, the second pole's condition/angle is taken into account, and when 0, it is not. This is so because *0 = 0*P2_Angle*, and 0 is never greater than *AL* which is set to 36 degrees. Listing-14.2 shows the implementation of this new addition to the morphology module.

Listing-14.2 The pole_balancing morphology; adding the new pb_GetInput sensor and pb_Push actuator to the morphology module.

```
pole_balancing(sensors)->
    [
        #sensor{name=pb_GetInput,scape={private,pb_sim},vl=2,parameters=[2]}
    ];
pole_balancing(actuators)->
    [
        #actuator{name=pb_SendOutput,scape={private,pb_sim},vl=1, parameters
=[no_damping,0]}
    ].
%Both, the pole balancing sensor and actuator, interface with the pole balancing simulation.
The type of benchmark the pole balancing simulation is used as (whether it is used as a double
```

pole or a single pole balancing benchmark) depends on the sensor and actuator parameters. The sensor's vl and parameters specify that the sensor will request the private scape for the cart's position and pole's angular position. The actuator's parameters specify that the scape should use no_damping type of fitness, and that since only a single pole is being used, that the termination condition associated with the second pole is zeroed out, by being multiplied by 0. When instead of using 0 we use 1, the private scape will use the angular position of the second pole as an element in calculating whether the termination condition has been reached or not.

Having specified the sensor and the actuator used by the pole_balancing morphology, we now need to implement them both. The pb_GetInput sensor will be similar to the xor_GetInput, only it will use its Parameters value in its message to the private scape it is associated with, as shown in Listing-14.3. We add this new sensor function to the sensor module, placing it after the xor_GetInput/3 function.

Listing-14.3 The implementation of the pb_GetInput sensor.

```
pb_GetInput(VL,Parameters,Scape)->
    Scape ! {self(),sense,Parameters},
    receive
        {Scape,percept,SensoryVector}->
            case length(SensoryVector)==VL of
                true ->
                    SensoryVector;
                false ->
                    io:format("Error in sensor:pb_GetInput/2, VL:~p
SensoryVector:~p~n", [VL,SensoryVector]),
                    lists:duplicate(VL,0)
            end
    end.
```

Similarly, Listing-14.4 shows the implementation of the actuator pb_SendOutput/3 function, added to the actuator module. It too is similar to the xor_SendOutput/3 function, but unlike its neighbor, it sends its Parameters value as an element of the message that it forwards to the scape. Because we usually implement the morphologies and the scapes together, we can set up any type of interfacing, and thus be able to implement complex scapes and messaging schemes with ease.

Listing-14.4 The implementation of the pb_SendOutput actuator.

```
pb_SendOutput([Output],Parameters,Scape)->
    Scape ! {self(),push,Parameters,[10*functions:sat(Output,1,-1)]},
    receive
        {Scape,Fitness,HaltFlag}->
            {Fitness,HaltFlag}
```

```
end.
```

Though simple to implement, this new problem allows us to test the ability of our neuroevolutionary system to evolve neurocontrollers on problems which require a greater level of complexity than the simple XOR mimicry problem. The benchmarking of our system on this problem also allows us to compare its results to those of other neuroevolutionary systems. Having implemented this new simulation, we now move forward in running a quick test on it in the next subsection.

14.1.3 Benchmark Results

In the previous chapter we have developed the benchmarking and reporting tools specifically to improve our ability to test new additions to the system. Thus all we must do now is to decide which variation of the pole balancing test to apply our system to, and then execute the benchmarker:start/1 function with the appropriate *constraint, pmp,* and *experiment* parameters.

Our benchmarker, on top of generating graphable data, also calculates the simple average number of evaluations from all the evolutionary runs within the experiment, which is exactly the number we seek because the benchmark here is how quickly a solution can be evolved on average using our system. Let us run 3 experiments, which will only entail us to execute the benchmarker:start/1 function 3 times, each time with a different sensor and actuator specification. Thus we next run three experiments, each with its own morphological setup:

1. The single pole, partial information, standard fitness function (without damping) benchmark:

```
pole_balancing(sensors)->
    [ #sensor{name=pb_GetInput,scape={private,pb_sim},vl=2,parameters=[2]} ];
pole_balancing(actuators)->
    [#actuator{name=pb_SendOutput,scape={private,pb_sim},vl=1, parameters
=[without_damping,0]}].
```

2. The double pole, partial information, standard fitness function (without damping) benchmark:

```
pole_balancing(sensors)->
    [ #sensor{name=pb_GetInput,scape={private,pb_sim},vl=3,parameters=[3]} ];
pole_balancing(actuators)->
    [#actuator{name=pb_SendOutput,scape={private,pb_sim},vl=1, parameters
=[without_damping,1]}].
```

3. The double pole, partial information, with damping fitness function benchmark:

```
pole_balancing(sensors)->
    [ #sensor{name=pb_GetInput,scape={private,pb_sim},vl=3,parameters=[3]} ];
pole_balancing(actuators)->
    [ #actuator{name=pb_SendOutput,scape={private,pb_sim},vl=1, parameters
=[with_damping,1]} ].
```

We must also set the *pmp*'s fitness goal to 90000, since with the standard, without_damping fitness function, the 90000 fitness score represents the NN based agent's ability to balance a pole for 30 minutes. But what about the *with_damping* simulation? In that event a neurocontroller will have different fitness scores for the same number of time steps that it has balanced the pole/s, since the fitness will be based on its effectiveness of balancing the poles as well. In the same manner, different number of time steps of balancing the pole/s might map to the same fitness score... This situation arises due to the fact that the more complicated problems will not have a one-to-one mapping with regards to fitness scores reached, and progress towards solving a given problem or achieving some goal. Different such simulations and problems will have different types of fitness scores, and using a termination condition based on a fitness goal value set in the population_monitor, will not work. On the other hand, each simulation/problem itself, will have all the necessary information about the agent's performance to decide whether a goal has been reached or not.

Furthermore, sometimes we wish to see just how quickly on average the neuroevolutionary system can generate a result for a problem, at those times we only care about the minimum number of evaluations needed to reach the solution. In our system no matter when the termination condition is reached, it is not until all the agents of the current generation, or all the currently active agents, have terminated, that the evolutionary run is complete. This means that the total number of evaluations keeps incrementing even after the goal has already been reached, simply because the currently-still-running agents are continuing being tuned.

To solve both problems, we can allow each scape to inform the agent that it has reached the particular goal of the problem/scape when it has done so. At this point the agent would forward that message to the population_monitor, which could then stop counting the evaluations by freezing the tot_evaluations value. In this one move we allow each scape to use the extra feature of *goal_reached* notification ability to be able to, on its own terms, use any fitness function, and at the same time be able to stop and notify the agent that it has reached the particular fitness goal, or solved the problem, and thus stop the evaluations accumulator from incrementing. This will allow us to no longer need to calculate fitness goals for every problem by pre-calculating various values (fitness goals) and setting them in the population_monitor. This method will also allow us to deal with problems where the fitness score is not directly related to the completion of the problem or to the reaching of the goal, and thus cannot be used as the termination condition in the first place. Thus, before we run the benchmarks, let's make this small program modification.

Currently when the agent has triggered the scape's stopping condition, the scape sends back to the agent the message: {Scape_PId,0,1}, where 0 means that it has received 0 fitness points for this last event, and 1 means that this particular scape has reached its termination condition. The actuator does nothing with this value but pass it to the cortex, thus if we retain the same message structure, we can piggyback it with new functionality. We will allow each scape to also have, on top of the standard termination conditions, the ability to check for its own goal reaching condition. When that goal condition is reached, instead of sending to the actuator the original message, the scape will send it: {Scape_PId,goal_reached,1}. The actuator does not have to be changed, its job is simply to forward this message to the cortex.

In the cortex we modify its *receive* clause to check whether the *Fitness* score sent to it is actually an atom *goal_reached*. The new *receive* clause is implemented as follows:

```
{APId,sync,Fitness,EndFlag} ->
    case Fitness == goal_reached of
        true ->
            put(goal_reached,true),
            loop(Id,ExoSelf_PId,SPIds,{APIds,MAPIds},NPIds,CycleAcc,FitnessAcc,
EFAcc +EndFlag, active);
        false ->
            loop(Id,ExoSelf_PId,SPIds,{APIds,MAPIds},NPIds,CycleAcc,FitnessAcc
+Fitness, EFAcc +EndFlag, active)
    end;
```

We also modify the cortex's message to the exoself when its evaluation termination condition has been triggered by the EndFlag, when the actuator sends it the message of the form: {APId, sync, Fitness, EndFlag}. The new message the cortex sends to the exoself is extended to include the note on whether *goal_reached* is set to true or not. The new message format will be: {self(), evaluation_completed, FitnessAcc, CycleAcc, TimeDif, get(goal_reached)}.

Reflectively, the exoself's receive pattern is extended to receive the *GoalReachedFlag* message, and to then forward it to the population_monitor, as shown by the boldfaced source code in the following listing:

Listing-14.5 The updated exoself's receive pattern.

```
loop(S)->
    receive
        {Cx_PId,evaluation_completed,Fitness,Cycles,Time,GoalReachedFlag}->
            case (U_Attempt >= S#state.max_attempts) or (GoalReachedFlag==true) of
                true ->%End training
                    A=genotype:dirty_read({agent,S#state.agent_id}),
```

```
                    genotype:write(A#agent{fitness=U_HighestFitness}),
                    backup_genotype(S#state.idsNpids,S#state.npids),
                    terminate_phenotype(S#state.cx_pid,S#state.spids,S#state.npids,
S#state.apids, S#state.scape_pids),
                    io:format("Agent:~p terminating. Genotype has been backed
up.~n Fitness:~p~n TotEvaluations:~p~n TotCycles:~p~n TimeAcc:~p~n", [self(),
U_HighestFitness, U_EvalAcc,U_CycleAcc, U_TimeAcc]),
                    case GoalReachedFlag of
                        true ->
                            gen_server:cast(S#state.pm_pid,
{S#state.agent_id, goal_reached);
                        _ ->
                            ok
                    end,
                    gen_server:cast(S#state.pm_pid,{S#state.agent_id,terminated,
U_HighestFitness});
…
```

Next, we update the *population_monitor* by first adding to its *state* record the goal_reached element, which is set to *false* by default, and then by adding to it a new handle_cast clause:

```
handle_cast({_From,goal_reached},S)->
    U_S=S#state{goal_reached=true},
    {noreply,U_S};
```

This cast clause sets the *goal_reached* parameter to true when triggered. Finally, we add to all population_monitor's termination condition recognition cases the additional operator: "**or S#state.goal_reached**", and modify the *evaluations* message receiving handle_cast clause to:

```
handle_cast({From,evaluations,Specie_Id,AEA,AgentCycleAcc,AgentTimeAcc},S)->
    AgentEvalAcc=case S#state.goal_reached of
            true ->
                0;
            _ ->
                AEA
    end,
```

This ensures that the population_monitor stops counting evaluations when the goal_reached flag is set to true. These changes effectively modify our system, giving it the ability to use the goal_reached parameter. This entire modification is succinctly shown in Fig-14.2.

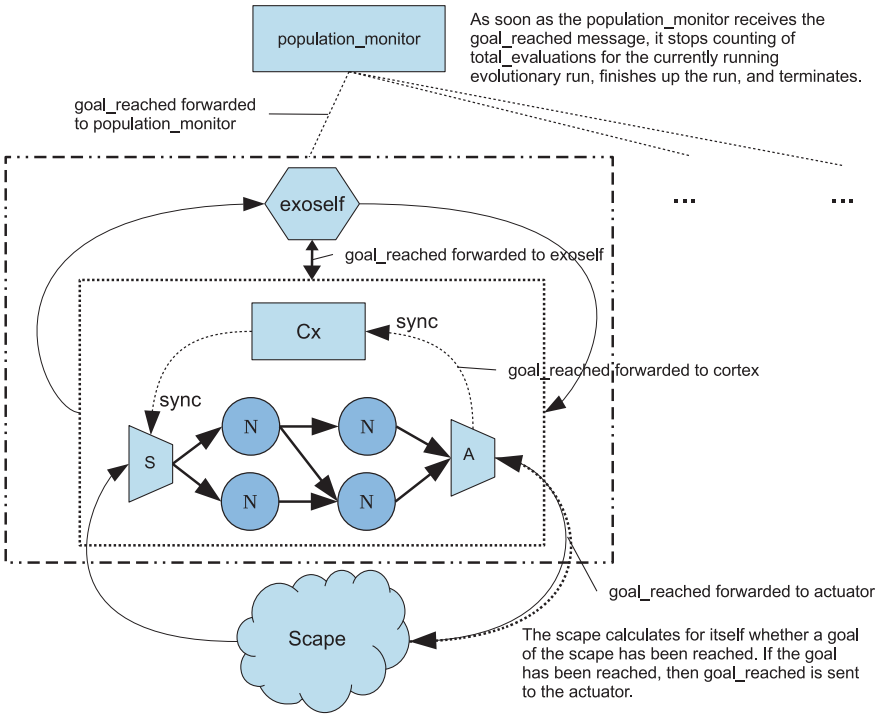

Fig. 14.2 The updated goal_reached message processing capable scape, and the goal_reached signal's travel path: scape to actuator to cortex to exoself to population_monitor.

This small change allows us to continue with our pole_balancing benchmarking test. And thus we finally set the experiment's *tot_runs* parameter to 50, which makes the benchmarker run 50 evolutionary runs in total, which means that the calculated average is based on 50 runs, which is a standard for this type of problem.

To run the first benchmark, we simply use the morphology setup listed earlier, set the fitness_goal parameter of the pmp record to 90000, the tot_runs to 50, and leave everything else as default. We then compile and reload everything by running polis:sync(), and execute the benchmarker:start(spb_without_damping) function, where *spb_without_damping* is the Id we give to this experiment, which stands for Single Pole Balancing Without Damping.

With this setup, the benchmarker will spawn the population_monitor process, wait for the evolutionary run to complete, add the resulting trace to the experiment's stats list, and then perform another evolutionary run. In total 50 evolutionary runs will comprise the benchmark. The result we are after is not the graphable data, but the report's average evaluations value (the average number of evaluations taken to reach the goal), and its standard deviation. The results of the first benchmark are shown in the following listing.

Listing-14.6 The results of the single pole balancing, partial information, without_damping, benchmark.

```
3> benchmarker:start(spb_without_damping).
...
******** Traces_Acc written to file:"benchmarks/report_Trace_Acc"
Graph:{graph,pole_balancing,
        [1.1782424242424248],
        [0.16452932686308724],
        [60910.254989899],
        [24190.827695700948],
        [75696.42],
        [32275.24],
        [6.04],
        [1.232233744059949],
        [457.72],
        []}
Tot Evaluations Avg:646.78 Std:325.8772339394086
```

It works! The results are also rather excellent, on average taking only 646 evaluations (though as can be seen from the standard deviation, there were times when it was much faster). We achieved this high performance (as compared to the results of other neuroevolutionary systems) without even having taken the time to optimize or tune our neuroevolutionary system yet. If we compare the resulting evaluations average that we received from our benchmark (your results might differ slightly), to those done by others, for example compared to the list put together in paper [1], we see that our system is the most efficient of the topology and weight evolving artificial neural network systems on this benchmark. The two faster neuroevolutionary systems ESP [2], and CoSyNE [3], do not evolve topology. The ESP and CoSyNE systems solved the problem in 589 and 127 evaluations respectively, while the CNE [4] and SANE [5] and NEAT [6] solved it in 724, 1212, and 1523 evaluations on average, respectively.

When using the non topology and weight evolving neuroevolutionary systems (ESP, CMA-ES, and CoSyNE), the researcher must first create a topology he knows works (or have the neuroevolutionary system generate random topologies, rather than evolving one from another), and then the neuroevolutionary system simply optimizes the synaptic weights to a working combination of values. But such systems cannot be applied to previously unknown problems, or problems for which we do not know the topology, nor its complexity and size, beforehand. For complex problems, topology cannot be predicted, in fact this is why we use a topology and weight evolving artificial neural network system, because we cannot predict and create the topology for *non-toy* problems on our own, we require the help of evolution.

Next we benchmark our system on the second problem, the more complex double pole balancing problem which uses a standard fitness function without damping. Listing-14.7 shows the results of the experiment.

Listing-14.7 The double pole balancing benchmark, using the *without damping* fitness function.

```
3> benchmarker:start(spb_without_damping).
...
Graph:{graph,pole_balancing,
        [2.4315606060606063],
        [0.8808311444164436],
        [22194.480560606058],
        [15614.417335306674],
        [34476.74],
        [6285.78],
        [7.34],
        [1.4779715829473847],
        [500.0],
        []}
Tot Evaluations Avg:5184.0 Std:3595.622677645695
```

Our system was able to solve the given problem in **5184** evaluations, whereas again based on the table provided in [1], the next closest TWEANN in that table is ESP [2], which solved it in 7374 evaluations on average. But, the DXNN system we discussed earlier was able to solve the same problem in 2359 evaluations on average. As we continue advancing and improving the system we're developing together, it too will improve to such numbers.

Finally, we run the third benchmark, the double pole balancing with partial state information and with damping. Because we have added the goal_reached messaging by the scapes, we can deal with the non one-to-one mapping between the number of time steps the agent can balance the cart, and the fitness calculated for this balancing act. Thus, we modify the pmp's fitness_goal back to inf, letting the scape terminate when the goal has been reached, and thus when the evaluation run should stop (we could have done the same thing during the previous experiment, rather than using the fitness goal of 90000, which was possible due to the goal and fitness having a one-to-one mapping). The results of this experiment are shown in Listing-14.8.

Listing-14.8 The results of running the double pole balancing with damping benchmark.

```
Graph:{graph,pole_balancing,
        [3.056909090909092],
        [1.3611906067001034],
```

```
        [67318.29389102172],
        [84335.29879824212],
        [102347.17542007213],
        [11861.325171196118],
        [7.32],
        [1.5157836257197137],
        [500.0],
        []}
Tot Evaluations Avg:4792.38 Std:3834.866761127432
```

It works! The goal_reached feature has worked, and the average number of evaluations our neuroevolutionary system needed to produce a result is highly competitive to other state of the art systems as shown in Table-14.1 which quotes the benchmark results from [1]. The DXNN system's benchmark results are also added to the table for comparison, with the results of our system added at the bottom. Note that neither CMA-ES nor CoSyNE evolves neural topologies. These two systems only optimize the synaptic weights of the already provided NN.

Table 14.1 Benchmark results for the pole balancing problem.

Method	Single-Pole/Incomplete state Information	Double-Pole/Partial Information W/O Damping	Double-Pole W/ Damping
RWG	8557	415209	1232296
SANE	1212	262700	451612
CNE*	724	76906*	87623*
ESP	589	7374	26342
NEAT	-	-	6929
CMA-ES*	-	3521*	6061*
CoSyNE*	127*	1249*	3416*
DXNN	**Not Performed**	**2359**	**2313**
OurSystem	647	**5184**	**4792**

* These do not evolve topologies, but only optimize the synaptic weights

Having completed developing these two benchmarks, and having finished testing our TWEANN system on the pole and double pole balancing benchmark, we move forward and begin developing the more complex T-Maze problem.

14.2 T-Maze Simulation

The T-Maze problem is another standard problem that is used to test the ability of a NN based system to learn and change its strategy while existing in, and interacting with, a maze environment. In this problem an agent navigates a T shaped maze as shown in Fig-14.3. At one horizontal end of the maze is a low reward,

and at another a high reward. The agent is a simulated robot which navigates the maze. Every time the robot crashes into a wall or reaches one of the maze's ends, its position is reset to the start of the maze. The whole simulation run (agent is allowed to navigate the maze until it either finds the reward and its position resets to base, or crashes into a wall and its position is reset to base) lasts X number of maze runs, which is usually set to 100. At some random time during those 100 maze runs, the high and low reward positions are swapped. The goal is for the agent to gather as many reward points as possible. Thus, if the agent has been reaching the high reward end of the maze, and suddenly there was a switch, the best strategy is for the agent when it has reached the location of where previously there was a high reward, is to realize that it now needs to change its strategy and always go to the other side of the maze, for the remainder of the simulation. To do this, the agent must remember what reward it has picked up and on what side, and change its traveling path after noticing that the rewards have been switched, which is most easily done when some of the agent's neurons are plastic.

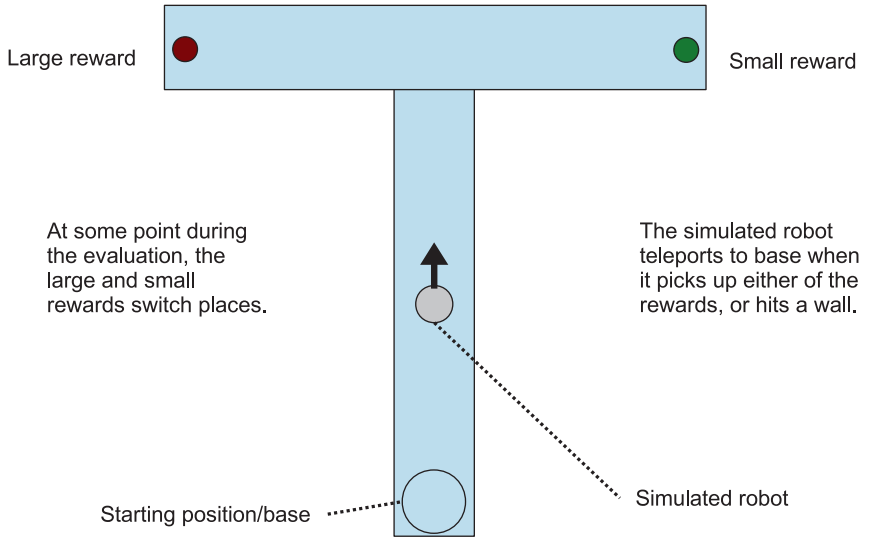

Fig. 14.3 The T-Maze setup.

We will create a simplified version of the T-Maze problem. It is used widely [6,7], and it does not require us to develop an entire 2d environment and robot simulation (which we will do in Chapter-18, when we create an Artificial Life simulation). Our T-Maze will have all the important features of the problem, but will not require true navigation in 2d space. We will create a discrete version of the T-Maze, as shown in Fig-14.4.

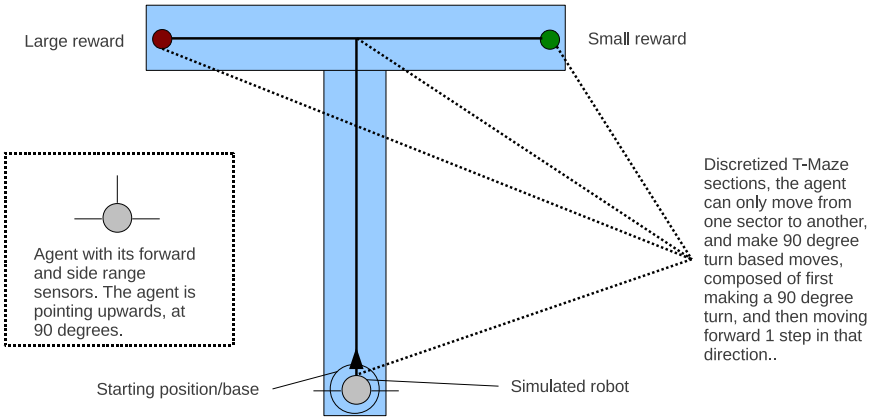

Fig. 14.4 A discrete version of the T-Maze simulation.

The agents traveling through the maze will be able to move forward, and turn left or right, but there will be no width to the corridors. The corridors will have a certain discrete length, and the agent will see forward in a sense that its range sensor will measure the distance to the wall ahead, and its side sensors will measure a distance to the sides of the "corridor" it is in, which when traveling down a single dimensional corridor will be 0, yet when reaching the T intersection, will show that it can turn left or right. The turns themselves will be discrete 90 degree turns, thus allowing the agent to turn left or right, and continue forward to gather the reward at the end of the corridor. This version of the T-Maze though simple, still requires the agent to solve the same problem as the non discrete Maze. In the discrete version, the agent must still remember where the reward is, evolve an ability to move down the corridors and turn and move in the turned direction where there is space to move forward, and finally, remember on which side of the maze it last found the highest reward.

The T-Maze will be contained in a private scape, and the movement and senses will, as in the previous simulation, be done through the sending and receiving of messages. Because we will create a discrete version of the maze, we can simulate the whole maze by simply deciding on the discrete length of each section of the corridor, and what the agent will receive as its sensory signals when in a particular section of the maze. The agent will use a combination of the following two sensors:

1. distance_sensor: A laser distance sensor pointing forward, to the left side, and to the right side, with respect to the simulated robot's direction. Since the maze is self contained and closed, the sensors will always return a distance. When traveling down the single dimensional corridor, the forward sensor will return the distance to the wall ahead, and the side distance sensors will return 0, since there is no place to move sideways. When the agent reaches an intersection, the side range sensors will return the distances to the walls on the side, thus the

agent can decide which way to turn. If the agent has reached a dead end, then both the forward facing, and the side facing range sensors will return 0, which will require the agent to turn, at which point it can start traveling in the other direction.

2. reward_consumed: The agent needs to know not only where the reward is, but how large it is, since the agent must explore the two rewards, and then for the remainder of the evaluation go towards the larger reward. To do this, the agent must have a sensory signal which tells it how large the reward it just consumed is. This sensor forwards to the NN a vector of length one: [RewardMagnitude], where RewardMagnitude is the magnitude of the actual reward.

The agent must also be able to move around this simplified, discrete labyrinth. There are different ways that we could allow the NN based agent to control the simulated robot within the maze. We could create an actuator that uses a vector of length one, where this single value is then used to decide whether the agent is to turn left (if the value is < -0.33), or turn right (if the value is > 0.33) or continue moving forward (if the value is between -0.33 and 0.33). Another type of actuator could be based on the differential drive, similar to one used by the Khepera [5] robot (a small puck shaped robot). The differential_drive actuator would have as input a vector of length 2: [Val1,Val2], where Val1 would control the rotation speed of the left wheel, and Val2 would control the rotation speed of the right wheel. In this manner if both wheels are spinning backwards (Val1 < 0, and Val2 < 0), the simulated robot moves backwards, if both spin forward with the same speed, then the robot moves forward. If they spin at different speeds, the robot either turns left or right depending on the angular velocities of the two wheels. Finally, we could create an actuator that accepts an input vector of length 2: [Val1,Val2], where Val1 maps directly to the simulated robot's velocity on the Y axis, and Val2 maps to the robot's velocity on the X axis. This would be a simple *translation_drive* actuator, and the simulated robot in this scenario would not be able to rotate. The inability to rotate could be alleviated if we add a third element to the vector, which we could than map to the angular velocity value, which would dictate the robot's rotation clockwise or counterclockwise, dependent on that value's sign. Or Val1 could dictate the robot's movement forward/backward, and Val2 could dictate whether the robot should turn left, right, or not at all. There are many ways in which we could let the NN control the movement of the simulated robot. For our discrete version of the T-Maze problem, we will use the same movement control method that was used in paper [7] which tested another NN system on the discrete T-Maze problem. This actuator accepts an input from a single neuron, and uses this accumulated vector: [Val], to then calculate whether to move forward, turn counterclockwise and move forward in that direction, or turn clockwise and then move forward in that direction. If Val is between -0.33 and 0.33, the agent moves one step forward, if it is less than -0.33, the agent turns counterclockwise and then moves one step forward, and if Val is greater than 0.33, the agent turns clockwise and moves one step forward in the new direction.

Due to this being a discrete version of the maze, it can easily be represented as a state machine, or simply as a list of discrete sections. Looking back at Fig-14.4, we can use a list to keep track of all the sensor responses for every position and orientation within the maze. In the standard discrete T-Maze implementation used in [7], there are in total 4 sectors. The agent starts at the bottom of the T-Maze located at {X=0,Y=0}, it can then move up to {0,1}, which is an intersection. At this point the agent can turn left and move a step forward to {-1,1}, or turn right and move a step forward to {1,1}.

If we are to draw the maze on a Cartesian plane, the agent can be turned to face towards the positive X axis, at 0 degrees, the positive Y axis at 90 degrees, the negative X axis at 180 degrees, and finally the negative Y axis, at 270 degrees. And if the maze is drawn on the Cartesian plane, then each sector's Id can be its coordinate on that plane. With the simulated robot in this maze being in one of the sectors (on one of the coordinates {0,0},{0,1},{1,1},or {-1,1}), and looking in one particular direction (at 0, 90, 180, or 270 degrees), we can then perfectly define what the sensory signals returned to the simulated robot should be. But before we can do that, we need a format for how to store the simulated robot's location, viewing direction, and how it should perceive whether it is looking at a wall, or at a reward located at one of the maze's ends. The superposition of the T-Maze on a Cartesian plane, with a few examples of the agent's position/orientation, and what sensory signals it receives there, is shown in Fig-14.5.

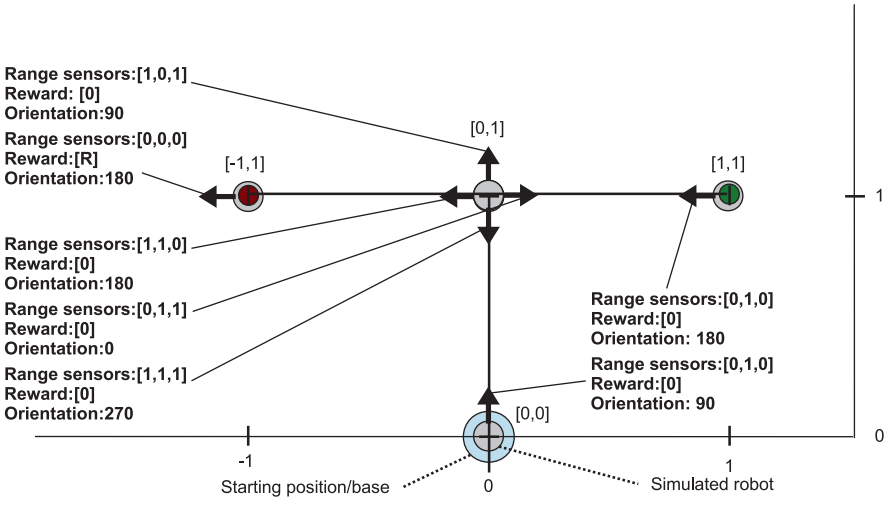

Fig. 14.5 Discrete T-Maze, and the sensory signals the simulated robot receives at various locations and orientations. The agent is shown as a gray circle, with the arrow pointing in the direction the simulated robot is looking, its orientation.

We will let each discrete sector keep track of the following:

- **id**: It's own id, its Cartesian coordinate.

- **r**: The reward the agent gets for being in that sector. There will be only two sectors that give reward, the two horizontal endings of the "*T*". This reward will be sensed by the *reward_sensor*.
- **description**: This will be the list that contains all the sensory information available when the agent is in that particular sector. In this simulation it will contain the range sensory signals. This means that each section will contain 4 sets of range sensory signals, one each for when the simulated robot is turned and is looking at 0, at 90, at 180, and at 270 degrees in that sector. Each of the range signals appropriate for the agent's particular orientation can then be extracted through a key, where the key is the agent's orientation in degrees (one of the four: 0, 90, 180, or 270). The complete form of the *description* list is as follows: *[{0, NextSector, RangeSense}, {90, NextSector, RangeSense}, {180, NextSector, RangeSense}, {270, NextSector, RangeSense}]*. The *NextSector* parameter specifies what is the coordinate of the next sector that is reachable from the current sector, given that the agent will move forward while in the current orientation. Thus, if for example the agent's forward is at 90 degrees, looking toward the positive Y axis on the Cartesian coordinate, and its actuator specifies that it should move forward, then we look at the 90 degree based tuple, and move the agent to the NextSector of that tuple.

We will call the record containing all the sector information of a single sector: *dtm_sector*, which stands for *Discrete T-Maze Sector*. An example of the sector located at coordinate [0,0], and part of the maze shown in the above figure, is as follows:

```
#dtm_sector{id=[0,0],description=[{0,[],[1,0,0]},{90,[0,1],[0,1,0]},{180,[],[0,0,1]},{270,[],[0,0,
0]}],r=0}
```

Let's take a closer look at this sector, located at [0,0], and on which the agent is for example turned at 90 degrees, and thus looking towards the positive Y axis. For this particular orientation when the agent requests sensory signals, they will come from the following tuple: **{90,[0,1],[0,1,0]}**, also highlighted in the above record. The first value, 90, is the orientation for which the follow-up sensory information is listed. The [0,1] is the coordinate of the sector to which the agent will move if it decides to move forward at this orientation. The vector [0,1,0] is the range sensory signal, and is fed to the agent's range sensor when requested. It states that on both sides, the agent's left and right, there are walls right next to it, and the distance to them is 0, and that straight ahead the wall does not come up for 1 sector. The value r=0 states that the current sector has no reward, and this is the value fed to the agent's reward sensor.

Thus this allows the agent to move around the discrete maze, travel from one sector to another, where each sector has all the information needed when the agent's sensors send a request for percepts. These sectors will all be contained in a single record's list, used by the private scape which represents the entire maze.

We will call the record for this private scape: *dtm_state*, and it will have the following default format:

```
-record(dtm_state,{agent_position=[0,0],agent_direction=90,sectors=[],tot_runs=60,
run_index=0, switch_event, fitness_acc=0}).
```

Let's go through each of this record's elements and discuss its meaning:

- **agent_position**: Keeps track of the agent's current position, the default is [0,0], the agent's starting position in the maze.
- **agent_direction**: Keeps track of the agent's current orientation, the default is 90 degrees, where the agent is looking down the maze, towards the positive Y axis.
- **sectors**: This is a list of all the sectors: [SectorRecord1...SectorRecordN], each of which is represented by the dtm_sector record, and a list of which will represent the entire T-Maze.
- **tot_runs**: Sets the total number of maze runs (trials) the agent performs per evaluation.
- **run_index**: This parameter keeps track of the current maze run index.
- **switch_event**: Is the run index during which the large and small reward locations are switched. This will require the agent, if it wants to continue collecting the larger reward, to first go to the large reward's original position, at which it will now find the smaller reward, figure out that the location of the large reward has changed, and during the following maze run go to the other side of the maze to collect the larger reward.
- **switched**: Since the switch of the reward locations needs to take place only once during the entire tot_runs of maze runs, we will set this parameter to *false* by default, and then to true once the switch is made, so that this parameter can then be used as a flag to ensure that no other switch is performed for the remainder of the maze runs.
- **step_index**: If we let the agents travel through the maze for as long as they want, there might be certain phenotypes that simply spin around in one place, although not possible with our current type of actuator, which requires the agent to take a step every time, either forward, to the right, or to the left. To prevent such infinite spins when we decide to use another type of actuator, we will give each agent only a limited number of steps. It takes a minimum of 2 steps to get from the base of the maze to one of the rewards, 1 step up the main vertical hall, and 1 turn/move step to the left or right. With an eye to the future, we will give the agents a maximum of 50 steps, after which the maze run ends as if the agent crashed into a wall. Though not useful in this implementation, it might become useful when you extend this maze and start exploring other actuators, sensors...

As with the pole balancing, this private scape will allow the agent to send it messages requesting sensory signals, either all signals (range sense, and the just

acquired reward size sense) merged into a single vector, or one sensory signal vector at a time. And it will allow the agent to send it signals from its actuators, dictating whether it should move or rotate/move the simulated robot.

Thus, putting all of this together: The scape will keep track of the agent's position and orientation, and be able to act on the messages sent from its sensor and actuator, and based on them control the agent's avatar. The T-Maze will start with the large and small rewards at the two opposite sides of the T-Maze, and then at some random maze run to which the switch_event is set (different for each evaluation), the large and small reward locations will flip, and require for the agent to figure this out and go to the new location if it wants to continue collecting the larger of the two rewards. As per the standard T-Maze implementation, the large reward is worth 1 point, and the small reward is worth 0.2 points. If at any time the agent hits a wall, by for example turn/moving when located at the base of the maze, and thus hitting the wall, the maze run ends and the agent is penalized with -0.4 fitness points, is then re-spawned at the base of the maze, and the *run_index* is incremented. If the agent collects the reward, the maze run ends and the agent is re-spawned at the base of the maze, with the *run_index* incremented. Finally, once the agent has finished *tot_runs* number of maze runs, the evaluation of the agent's fitness ends, at which point the exoself might perturb the NN's synaptic weights, or end the tuning run... To ensure that the agents do not end up with negative fitness scores when setting the tot_runs to 100, we will start the agents off with 50 fitness points. Thus an agent that always crashes will have a minimum fitness score of 50 – 100*0.4 = 10.

Finally, though we will implement the T-Maze scenario where the agent gets to the reward at one of the maze's ends, and is then teleported back to the base of the maze for another maze-run, there are other possible implementations and scenarios. For example, as is demonstrated in Fig-14.6, we could also extend the maze to have teleportation portals located at {-2,1} and {2,1}, through which the agent has to go after gathering the food, so that it is teleported back to the base to reset the rewards. Or we could require it to have to travel all the way back to the base manually, though we would need to change the simple actuator so that it can rotate in place without crashing into walls. Finally, we could also create the T-Maze which allows for both options, teleportation and manual travel. All, the 3 extended T-Mazes, and 1 default T-Maze which we will implement, are shown in the following figure.

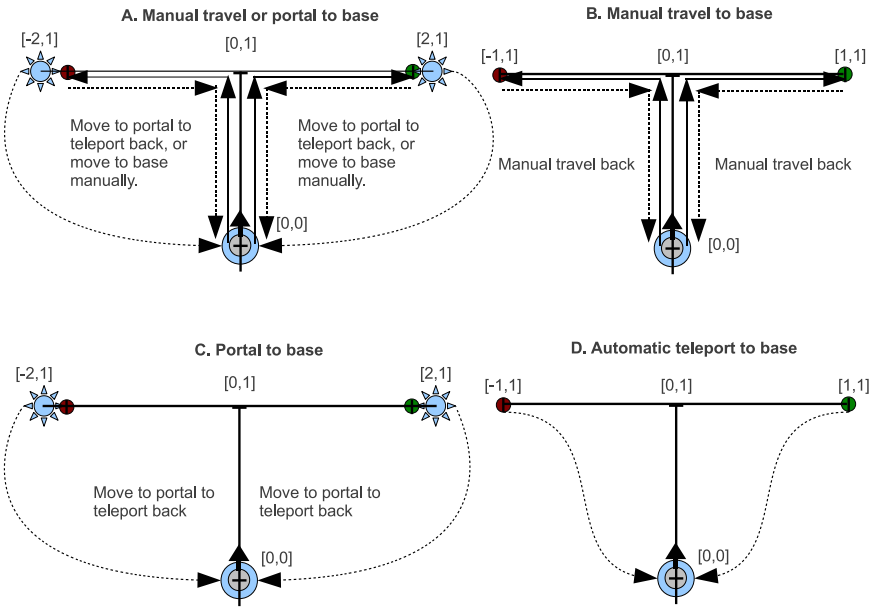

Fig. 14.6 The various possible scenarios for the T-Maze after the agent has acquired the reward.

Having decided on the architecture, and having created Fig-14.5 and Fig-14.6d to guide us in the designing and setting the T-Maze system and each of its sectors, we can now move forward to the next subsection and implement this private T-Maze scape, and the needed sensors and actuators to interface with it.

14.2.1 T-Maze Implementation

Through Fig-14.5 we can immediately map the maze's architecture to its implementation shown in Listing-14.9. For the implementation we first define the two new records needed by this new scape: the dtm_sector and dtm_state records. The function *dtm_sim/1* prepares and starts up the maze, dropping into the process's main loop. In this main loop the scape process can accept requests for sensory signals, and accept signals from the actuators and return to them a message containing the fitness points acquired. The sensors we will use will poll the private scape for an extended range sensor, which is a vector of length 4, and contains the signals from the agent's range sensor, appended with the reward value in the current maze sector: *[Reward,L,F,R]*, where Reward is the value of the actual reward, L is the range to the left wall, F is the range to the wall in front, and R is the range to the wall on the right.

Listing-14.9 The implementation of the Discrete T-Maze scape.

```
-record(dtm_sector,{
    id,
    description=[],
    r
}).

-record(dtm_state,{
    agent_position=[0,0],
    agent_direction=90,
    sectors=set_tmaze_sectors(),
    tot_runs=100,
    run_index=0,
    switch_event=35+random:uniform(30),
    switched=false,
    step_index=0,
    fitness_acc=50
}).

dtm_sim(ExoSelf_PId)->
    io:format("Starting dtm_sim~n"),
    random:seed(now()),
    dtm_sim(ExoSelf_PId,#dtm_state{}).

dtm_sim(ExoSelf_PId,S) when (S#dtm_state.run_index == S#dtm_state.switch_event) and
(S#dtm_state.switched==false)->
    Sectors=S#dtm_state.sectors,
    SectorA=lists:keyfind([1,1],2,Sectors),
    SectorB=lists:keyfind([-1,1],2,Sectors),
    U_SectorA=SectorA#dtm_sector{r=SectorB#dtm_sector.r},
    U_SectorB=SectorB#dtm_sector{r=SectorA#dtm_sector.r},
    U_Sectors=lists:keyreplace([-1,1],2,lists:keyreplace([1,1],2,Sectors, U_SectorA),
U_SectorB),
    scape:dtm_sim(ExoSelf_PId,S#dtm_state{sectors=U_Sectors, switched=true});
dtm_sim(ExoSelf_PId,S)->
    receive
        {From_PId,sense,Parameters}->
            APos = S#dtm_state.agent_position,
            ADir = S#dtm_state.agent_direction,
            Sector=lists:keyfind(APos,2,S#dtm_state.sectors),
            {ADir,NextSec,RangeSense} = lists:keyfind(ADir,1, Sec-
tor#dtm_sector.description),
            SenseSignal=case Parameters of
                [all] ->
```

```
                        RangeSense++[Sector#dtm_sector.r];
                [range_sense]->
                        RangeSense;
                [reward] ->
                        [Sector#dtm_sector.r]
        end,
        From_PId ! {self(),percept,SenseSignal},
        scape:dtm_sim(ExoSelf_PId,S);
    {From_PId,move,_Parameters,[Move]}->
        APos = S#dtm_state.agent_position,
        ADir = S#dtm_state.agent_direction,
        Sector=lists:keyfind(APos,2,S#dtm_state.sectors),
        U_StepIndex = S#dtm_state.step_index+1,
        {ADir,NextSec,RangeSense} = lists:keyfind(ADir,1,
Sector#dtm_sector.description),
        if
                (APos == [1,1]) or (APos == [-1,1]) ->
                        Updated_RunIndex=S#dtm_state.run_index+1,
                        case Updated_RunIndex >= S#dtm_state.tot_runs of
                                true ->
                                        From_PId ! {self(), S#dtm_state.fitness_acc
+Sector#dtm_sector.r, 1},
                                        dtm_sim(ExoSelf_PId,#dtm_state{});
                                false ->
                                        From_PId ! {self(),0,0},
                                        U_S = S#dtm_state{
                                                agent_position=[0,0],
                                                agent_direction=90,
                                                run_index=Updated_RunIndex,
                                                step_index = 0,
                                                fitness_acc = S#dtm_state.fitness_acc
+Sector#dtm_sector.r
                                        },
                                        dtm_sim(ExoSelf_PId,U_S)
                        end;
                Move > 0.33 -> %clockwise
                        NewDir=(S#dtm_state.agent_direction + 270) rem 360,
                        {NewDir,NewNextSec,NewRangeSense} =
lists:keyfind(NewDir, 1, Sector#dtm_sector.description),
                        U_S = move(ExoSelf_PId,From_PId,S#dtm_state{
agent_direction =NewDir},NewNextSec,U_StepIndex),
                        dtm_sim(ExoSelf_PId,U_S);
                Move < -0.33 -> %counterclockwise
                        NewDir=(S#dtm_state.agent_direction + 90) rem 360,
```

```
                        {NewDir,NewNextSec,NewRangeSense} =
lists:keyfind(NewDir, 1, Sector#dtm_sector.description),
                        U_S = move(ExoSelf_PId,From_PId,S#dtm_state{
agent_direction=NewDir},NewNextSec,U_StepIndex),
                        dtm_sim(ExoSelf_PId,U_S);
                true -> %forward
                        move(ExoSelf_PId,From_PId,S,NextSec,U_StepIndex)
        end;
    {ExoSelf_PId,terminate} ->
        ok
end.
% The dtm_sim/2 function generates a simulated discrete T-Maze scape, with all the sensory
information and the maze architecture specified through a list of sector records. The scape can
receive signals from the agent's sensor, to which it then replies with the sensory information,
and it can receive the messages from the agent's actuator, which it uses to move the agent's av-
atar around the maze.

move(ExoSelf_PId,From_PId,S,NextSec,U_StepIndex)->
    case NextSec of
        [] -> %wall crash/restart_state
            Updated_RunIndex = S#dtm_state.run_index+1,
            case Updated_RunIndex >= S#dtm_state.tot_runs of
                true ->
                    From_PId ! {self(),S#dtm_state.fitness_acc-0.4,1},
                    dtm_sim(ExoSelf_PId,#dtm_state{});
                false ->
                    From_PId ! {self(),0,0},
                    U_S = S#dtm_state{
                        agent_position=[0,0],
                        agent_direction=90,
                        run_index=Updated_RunIndex,
                        step_index = 0,
                        fitness_acc = S#dtm_state.fitness_acc-0.4
                    },
                    dtm_sim(ExoSelf_PId,U_S)
                end;
        _ -> %move
            From_PId ! {self(),0,0},
            U_S = S#dtm_state{
                agent_position=NextSec,
                step_index = U_StepIndex
            },
            dtm_sim(ExoSelf_PId,U_S)
    end.
%The move/5 function accepts as input the State S of the scape, and the specification of where
```

```
the agent wants to move its avatar next, NextSec. The function then determines whether that
next sector exists, or whether the agent will hit a wall if it moves in its currently chosen direc-
tion.

set_tmaze_sectors()->
    Sectors = [
            #dtm_sector{id=[0,0],description=[{0,[],[1,0,0]},{90,[0,1],[0,1,0]},{180,[],[0,0,1]},
{270,[], [0,0,0]}],r=0},
            #dtm_sector{id=[0,1],description=[{0,[1,1],[0,1,1]},{90,[],[1,0,1]},{180,[-1,1],
[1,1,0]}, {270, [0,0], [1,1,1]}],r=0},
            #dtm_sector{id=[1,1],description=[{0,[],[0,0,0]},{90,[],[2,0,0]},{180,[0,1],[0,2,0]},
{270,[], [0,0,2]}],r=0.2},
            #dtm_sector{id=[-1,1],description=[{0,[0,1],[0,2,0]},{90,[],[0,0,2]},{180,[],[0,0,0]},
{270,[],[2,0,0]}],r=1}
    ].
% The set_tmaze_sectors/0 function returns to the caller a list of sectors representing the T-
Maze. In this case, there are 4 such sectors, the vertical sector, the two horizontal sectors,
and the cross section sector.
```

With the T-Maze implemented, we now need to develop the complementary sensor and the actuator. For the sensor, since the agent needs all the information appended: sensory vectors from the *range_sensor*, and the *reward* sensor, combined into a single vector, we will create a single sensor which will contain the information from both of these sensors. What sensory signal the scape sends back to the agent's sensor will be defined by the sensor's parameter message. The actuator will simply forward the NN based agent's output to the discrete T-Maze process, which will then interpret the signal as turning left and moving forward 1 step, turning right and moving forward 1 step, or just moving forward 1 step. We first create the morphology, which follows the same format as the one we created for the pole_balancing morphology. This morphology we will call *discrete_tmaze*, with its implementation shown in Listing-14.10, and which we add to the morphology module.

Listing-14.10 The discrete_tmaze morphology specification.

```
discrete_tmaze(sensors)->
    [
            #sensor{name=dtm_GetInput,scape={private,dtm_sim},vl=4,parameters=[all]}
    ];
discrete_tmaze(actuators)->
    [
            #actuator{name=dtm_SendOutput,scape={private,dtm_sim},vl=1,parameters=[]}
    ].
```

Similarly, the sensor's implementation is shown in Listing-14.11, which we add to the sensor module.

Listing-14.11 The *dtm_GetInput* sensor implementation.

```
dtm_GetInput(VL,Parameters,Scape)->
    Scape ! {self(),sense,Parameters},
    receive
        {Scape,percept,SensoryVector}->
            case length(SensoryVector)==VL of
                true ->
                    SensoryVector;
                false ->
                    io:format("Error in sensor:dtm_GetInput/3, VL:~p
SensoryVector:~p~n", [VL,SensoryVector]),
                    lists:duplicate(VL,0)
            end
    end.
```

Finally, the actuator implementation is shown in Listing-14.12, which we add it to the actuator module.

Listing 14.12 The *dtm_SendOutput* actuator implementation.

```
dtm_SendOutput(Output,Parameters,Scape)->
    Scape ! {self(),move,Parameters,Output},
    receive
        {Scape,Fitness,HaltFlag}->
            {Fitness,HaltFlag}
    end.
```

And with that we've completely developed all the parts of the discrete T-Maze benchmark. We've created the actual private scape that represents the maze and in which an agent can travel. And we created the complementary morphology, with its own sensor and actuator set, used to interface with the T-Maze scape. With this particular problem/benchmark, we will now be able to test whether our topology and weight evolving artificial neural network system is able to evolve NN based agents which can perform complex navigational tasks, evolve agents which have memory and can make choices based on it, and even learn when the neurons within the tested NN have plasticity.

14.2.2 Benchmark Results

Let's run a quick test of our system by applying it to our newly developed problem. Though I do not expect our neuroevolutionary system to evolve an agent capable of effectively solving the problem at this stage, we still need to test whether the new scape, morphology, sensor, and actuator, are functional. Before we run the benchmark, let us figure out what fitness score value represents that the problem has been solved.

An evaluation is composed of 100 total maze runs, and sometime during the midpoint, between run 35 and 65, the high and low rewards are flipped. In this implementation, we set the switch_event to occur on the run number: *35+random:uniform(30)*. It will take at least one wrong trip to the reward to figure out that its position has been changed. Also, we should expect that eventually, evolution will create NNs that always first go to the maze corner located at [1,1], which holds the high reward before it is flipped.

So then, the maximum possible score achievable in this problem, a score representing that the problem has been solved, is: 99*1 + 1*0.2 + 50 = **149.2**, which represents an agent that first always goes to the right corner, at some point it goes there and notices that the reward is now small (0.2 instead of 1), and thus starts going to the [-1,1] corner. This allows the agent to achieve 99 high rewards, and 1 low reward. A score which represents that the agent evolved to always go to {1,1}, is at most: 65*1 + 35*0.2 + 50 = **122**, which is achieved during the best case scenario, when the reward is flipped on the 65th count, thus allowing the agent to gather high reward for 65 maze runs, and low reward for the remaining 35 maze runs. The agent will perform multiple evaluations, during some evaluations the reward switch event will occur early, and every once in a while it will occur on the 65th maze run, which is the latest time possible. During that lucky evaluation, the agent can reach 122 fitness points by simply not crashing and always going to the {1,1} side. The agent can accomplish this by first having: *0.33> Output >-0.33*, which will make the avatar move forward, and during the second step have *Output > 0.33*, which will make the avatar turn right and move forward to get the reward. Finally, the smallest possible fitness is achieved when the agent always crashes into the wall: *50 – 100*0.4 = **10***.

With this out of the way, we now set the Morphology element in the benchmarker module within the *?INIT_CONSTRAINTS* macro, to *discrete_tmaze*. We then set generation limit to inf, and evaluations_limit to 5000, in the *pmp* record. Finally, we run *polis:sync()* to recompile and load everything, then start the polis, and then finally execute benchmarker:start(dtm_test), as shown in Listing-14.3.

Listing-14.3 The results of running the T-Maze benchmark.

```
Graph:{graph,discrete_tmaze,
```

```
        [1.1300000000000001,1.12,1.195,1.1816666666666666,
         1.1633333333333333,1.156111111111111,1.2322222222222223,
         1.1400000000000001,1.1766666666666665,1.1800000000000002],
        [0.10535653752852737,0.11661903789690603,0.10234744745229357,
         0.10026354161796684,0.10214368964029706,0.08123088569087163,
         0.13765675688483067,0.11575836902790224,0.1238726945070803,
         0.092736184954957],
        [111.38000000000011,115.31900000000012,112.4590000000001,
         114.4511111111112,112.8790000000001,112.6335555555556,
         112.13066666666677,111.12500000000009,110.68722222222232,
         114.57700000000014],
        [9.305813236896594,6.245812917467183,6.864250796700242,
         8.069048898318606,8.136815662374111,9.383282426018074,
         7.888934134455533,9.98991266228088,9.41834002503416,
         8.867148978110151],
        [122.00000000000001,122.00000000000001,122.00000000000001,
         122.00000000000001,122.00000000000001,122.00000000000001,
         122.00000000000001,122.00000000000001,122.00000000000001,
         122.00000000000001],
        [10.000000000000115,10.000000000000115,10.000000000000115,
         10.000000000000115,10.000000000000115,10.000000000000115,
         10.000000000000115,10.000000000000115,10.000000000000115,
         10.000000000000115],
        [8.1,8.8,9.1,8.9,8.0,7.75,8.1,7.65,7.9,7.8],
        [0.8888194417315588,1.2884098726725124,0.8306623862918073,
         0.7681145747868607,0.8366600265340756,0.8874119674649424,
         0.9433981132056604,0.7262919523166975,1.57797338380595,
         1.3638181696985856],
        [500.0,500.0,500.0,500.0,500.0,500.0,500.0,475.0,500.0,500.0],
        []}
Tot Evaluations Avg:5083.75 Std:53.78835840588556
```

We are not interested in the "Tot Evaluations Avg" value, since the benchmark was not set up to use the goal_reached feature. But from the graph printout we do see the score **122.00**, boldfaced. I've boldfaced the list showing the highest fitness scores achieved amongst all the evolutionary runs. Though as we guessed, the system did not produce a solution (which requires plasticity as we will see in the next chapter), it has rapidly (within the first 500 evaluations), produced the score of 122, which means that agents learned to always navigate to the right corner.

It is always a good idea to at least once double check and printout all the information produced within the scape, following it in the console, and manually analyzing it to check for bugs. We will do that just this once, following a single extracted agent, and the signals its sensors acquire and its actuators produce. First, we run the function population_monitor:test() with the same parameters we started the benchmarker until a fit agent is evolved. We then add the line:

```
io:format("Position:~p SenseSignal:~p ",[Apos,SenseSignal]),
```

And lines:

```
timer:sleep(1000),
io:format("Move:~p StepIndex:~p RunIndex:~p~n", [Move,U_StepIndex,
S#dtm_state.run_index]),
```

To the receive *sense* and *move* pattern matchers, respectively. We then extract the evolved fit agent, and execute the function: *exoself:start(AgentId,void)* to observe the path the agent takes. A short console printout I saw when performing these steps is shown in Listing-14.4. The console printout shows the agent's starting moves, up to the point when the position of the rewards was switched, and a few steps afterwards.

Listing-14.4 Console printout of a champion agent's maze navigation.

```
exoself:start({7.513656492058022e-10,agent},void).

Starting dtm_sim
Position:[0,0] SenseSignal:[0,1,0,0] <0.5846.1>
Move:4.18876787545547e-15 StepIndex:1 RunIndex:0
Position:[0,1] SenseSignal:[1,0,1,0] Move:0.7692260090076106 StepIndex:2 RunIndex:0
Position:[1,1] SenseSignal:[0,0,0,1] Move:0.011886120521166272 StepIndex:3 RunIndex:0
Position:[0,0] SenseSignal:[0,1,0,0] Move:4.18876787545547e-15 StepIndex:1 RunIndex:1
Position:[0,1] SenseSignal:[1,0,1,0] Move:0.7692260090076106 StepIndex:2 RunIndex:1
Position:[1,1] SenseSignal:[0,0,0,1] Move:0.011886120521166272 StepIndex:3 RunIndex:1
Position:[0,0] SenseSignal:[0,1,0,0] Move:4.18876787545547e-15 StepIndex:1 RunIndex:2
Position:[0,1] SenseSignal:[1,0,1,0] Move:0.7692260090076106 StepIndex:2 RunIndex:2
Position:[1,1] SenseSignal:[0,0,0,1] Move:0.011886120521166272 StepIndex:3 RunIndex:2
Position:[0,0] SenseSignal:[0,1,0,0] Move:4.18876787545547e-15 StepIndex:1 RunIndex:3
...
Position:[0,0] SenseSignal:[0,1,0,0] Move:4.18876787545547e-15 StepIndex:1 RunIndex:38
Position:[0,1] SenseSignal:[1,0,1,0] Move:0.7692260090076106 StepIndex:2 RunIndex:38
Position:[1,1] SenseSignal:[0,0,0,1] Move:0.011886120521166272 StepIndex:3 RunIndex:38
Position:[0,0] SenseSignal:[0,1,0,0] Move:4.1887678754555e-15 StepIndex:1 RunIndex:39
Position:[0,1] SenseSignal:[1,0,1,0] Move:0.7692260090076106 StepIndex:2 RunIndex:39
Position:[1,1] SenseSignal:[0,0,0,0.2] Move:0.837532377697202 StepIndex:3 RunIndex:39
Position:[0,0] SenseSignal:[0,1,0,0] Move:4.1887678754555e-15 StepIndex:1 RunIndex:40
Position:[0,1] SenseSignal:[1,0,1,0] Move:0.7692260090076106 StepIndex:2 RunIndex:40
Position:[1,1] SenseSignal:[0,0,0,0.2] Move:0.837532377697202 StepIndex:3 RunIndex:40
Position:[0,0] SenseSignal:[0,1,0,0] Move:4.18876787545547e-15 StepIndex:1 RunIndex:41
Position:[0,1] SenseSignal:[1,0,1,0] Move:0.7692260090076106 StepIndex:2 RunIndex:41
Position:[1,1] SenseSignal:[0,0,0,0.2] Move:0.837532377697202 StepIndex:3 RunIndex:41
...
```

I've boldfaced the very first maze run, where we see the agent taking the steps from [0,0] to [0,1] to [1,1], and receiving the reward 1. Then we fast-forward and see that during the **RunIndex:39**, the reward has been switched. We know this because when the agent gets to [1,1] on that run, the reward is a mere 0.2 now. On the RunIndex: 40, the agent still goes to this same location, indicating it has not learned, and it has not evolved the ability to change its strategy.

14.3 Summary & Discussion

In this chapter we built two new problems to benchmark and test our neuroevolutionary system on. We built the Double Pole Balancing (DPB) simulation, and the Discrete T-Maze (DTM) simulation. We created different versions of the pole balancing problem, the single pole balancing with and without damping, and with and without full system state information, and the double pole balancing with and without damping, and with and without full system state information. The complexity of solving the pole balancing problem grows when we increase the number of poles to balance simultaneously, when we remove the velocity information and thus require the NN based agent to derive it on its own, and when we use the damping based fitness function instead of the standard one. We also created a discrete version of the T-Maze navigation problem, where an agent must navigate a T shaped maze to collect a reward located at one of the horizontal maze ends. In this maze there are two rewards, located at the opposite ends of the maze, one large and one small, and their location is switched at a random point during the 100 maze runs in total. This requires the agent to remember where the large reward was last time, explore that position, find that the reward is now small, and during the remaining maze runs navigate to the other side of the maze to continue collecting the large reward. This problem can be further expanded by changing the fitness function used, and by requiring the agent to collect the reward and then return to the base of the maze, rather than being automatically teleported back as is the case with our current implementation. Furthermore, we could expand the T-Maze into a Double T-Maze, with 4 corners where the reward can be collected, and thus requiring the agent to remember more navigational patterns and reward locations.

Based on our benchmark, the system we've built thus far has performed very well on the DPB problem, with its results being higher than those of other Topology and Weight Evolving Artificial Neural Networks (TWEANN), as was seen when the results we achieved were compared to the results of such systems referenced from paper [1]. Yet still the performance was not higher than that of DXNN, because we have yet to tune our system. When we applied our TWEANN to the T-Maze Navigation problem, it evolved NNs that were not yet able to change their strategy based on their experience. Adding plasticity in the next chapter will further expand the capabilities of the evolved NNs, giving us a chance to

again apply our system to this problem, and see that the performance improves, and allows the agents to achieve perfect scores.

Having a good set of problems in our benchmark suit will allow us to add and create features that we can demonstrate to improve the system's generalization abilities and general performance. The two new problems we added in this chapter will allow us to better test our system, and the performance of new features we add to it in the future. Finally, the T-Maze problem will allow us to test the important feature that we will add in the next chapter: *neural plasticity*.

14.4 References

[1] Gomez F, Schmidhuber J, Miikkulainen R (2008) Accelerated Neural Evolution through Cooperatively Coevolved Synapses. Journal of Machine Learning Research 9, 937-965.

[2] Sher GI (2010) DXNN Platform: The Shedding of Biological Inefficiencies. Neuron, 1-36. Available at: http://arxiv.org/abs/1011.6022.

[3] Durr P, Mattiussi C, Soltoggio A, Floreano D (2008) Evolvability of Neuromodulated Learning for Robots. 2008 ECSIS Symposium on Learning and Adaptive Behaviors for Robotic Systems LABRS, 41-46.

[4] Blynel J, Floreano D (2003) Exploring the T-maze: Evolving Learning-Like Robot Behaviors using CTRNNs. Applications of evolutionary computing 2611, 173-176.

[5] Khepera robots: www.k-team.com

[6] Risi S, Stanley KO (2010) Indirectly Encoding Neural Plasticity as a Pattern of Local Rules. Neural Plasticity 6226, 1-11.

[7] Soltoggio A, Bullinaria JA, Mattiussi C, Durr P, Floreano D (2008) Evolutionary Advantages of Neuromodulated Plasticity in Dynamic, Reward-based Scenarios. Artificial Life 2, 569-576.