

6 Artificial Intelligence

6.1 Aims

This module looks at some of the varied approaches to artificial intelligence that have been applied over the past twenty five years. Traditionally, artificial intelligence has been viewed as two interacting operations - high level planning and low level control. This view is in contrast to behaviour based systems that have many parallel integrated planning and control units. We will consider both views of artificial intelligence, and explore where they are appropriate and how they can be integrated. The module aims to provide enough information about intelligence for games to form the basis of the design for an autonomous game agent.

6.2 Sense and Behave Models

The traditional approach to agent intelligence involves the creation of plans based on state space representations of the world. The plans are then executed, with the planner being reinvoked if anything goes wrong. This approach has been said to have a functional decomposition as shown in the following diagram.

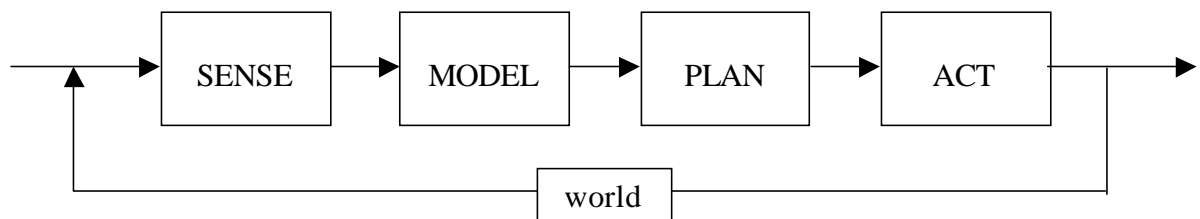


Figure 6.1: Traditional approach to agent intelligence.

The Behaviour-based approach removes the need for modelling and planning and links the sensors to the actuators as shown in the following diagram. This involves creating many task achieving links between sensors and actuators and then merging them. The interaction of task-achieving modules (or Behaviours with a capital B) produces *emergent* intelligence.

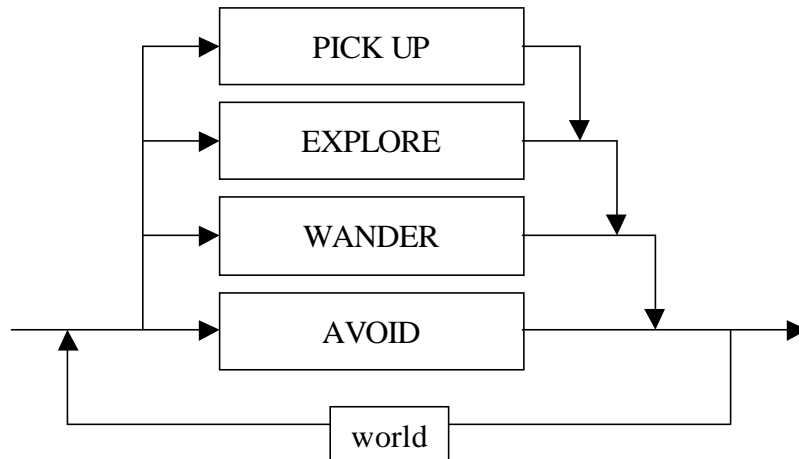


Figure 6.2: Task Decomposition for a Behaviour-based approach to agent intelligence.

The reason for the success of these types of implementations is that they do not have to generate anything but the simplest of symbolic representations of the world. Complex representations of the world take a lot of information to build, which implies long computation times. In Behaviour-based AI, the representation of the world that is used for action selection is inherent in simple crisp values. By sensing and actuating often, the agent can react quickly to change, and will not be sensitive to incorrect decisions.

Traditional systems have a problem when conflicting data comes in. Which data should be used to build the world model? Should the world model account for both sets of readings? Behaviour-based systems do not have this problem, as typically a behaviour will work with a single set of sensors. However there will still be conflicts between behaviours, not only from disparate sensor models, but also from conflicting opinions from competing behaviours. Behaviour-based systems all have an arbitration mechanism to cope with this competition, but do not view it as a problem. If the arbitration system is any good, appropriate behaviour will still emerge from the system. This provides the robustness that is characteristic of these systems.

6.2.1 Subsumption Architecture

Brook's subsumption architecture provides a way of combining distributed real-time control from sensor-driven behaviours. Behaviours are simply layers of control systems that all run in parallel whenever appropriate sensors fire. Sometimes an agent will receive contradictory sensor data. The subsumption architecture resolves this conflict by selecting appropriate behaviour rather than trying to work out which sensor is correct. This process is known as *fusion*; in the case of conflicting sensor data it is called *sensor fusion*. Sensor fusion is a difficult problem that often produces poor results. Brook's technique shifts the emphasis from *sensor fusion* to *behaviour fusion*. This ensures that the agent is always following some form of sensible behaviour, even if the sensor data is contradictory.

Behaviours are quite different to subroutines or functions. Higher level behaviours do not call lower level behaviours, but do have the power to temporarily suppress lower level behaviours. When the higher level behaviour is inactive, due to lack of sensory input, they cease their suppressive behaviour and the lower levels resume control. In

order to understand this idea more vividly, let us imagine some behaviours we could create on an agent.

6.2.2 Behaviour Networks

Let's consider a game agent that must try to seek a goal location, while simultaneously destroying as many enemy agents as possible. It keeps a range and bearing to the nearest enemy, and also to the goal location. We can create a simple subsumption program to achieve the desired behaviour.

Seek is a software process that checks the bearing to the goal. If the bearing is to the right, it turns to the right; if to the left, it turns to the left; if aligned it moves forward. The instructions to do this are very simple. The behaviour that emerges is one that causes the agent to move towards the goal. **Seek** is an example of a *task-achieving* behaviour. Useful in itself, it provides a minimum level of competence for our agent. Next we can add more sophisticated behaviours without redesigning lower-level behaviours already in place. The simple behaviour network is shown in Figure 6.3.

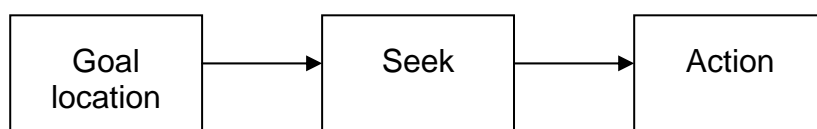


Figure 6.3: Simple behaviour network for the Seek behaviour.

Suppose we wrote a second behaviour, called **Destroy**, whose purpose is to track down enemy agents within range. The code for is quite simple, if the agent is to the left, turn left; if the agent is to the right, turn right; if lined up, chase! However, the code for **Destroy** is only run when an enemy agent is in range.

We now have two task achieving behaviours, **Seek** and **Destroy**, which, under some circumstances, will contend with each other for control of the agent's motion. Figure 6.4 shows one way to resolve this conflict. We have broken the wire connecting **Seek** with **Motors** and inserted a *suppressor node*, which is represented by an "S" in the circle.

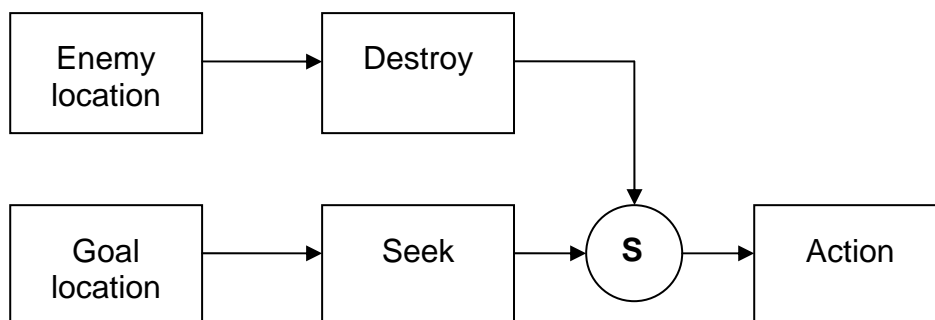


Figure 6.4: Subsumption network with two behaviours arbitrated by a suppressor node.

A suppressor node allows messages from the original wire to pass through to the output, unless a message arrives at the same time from the new connection, the arrowhead. Figure 6.5 shows the series of messages that might be produced by **Seek** and **Destroy** as the agent moves towards the goal, occasionally distracted by enemy

agents wandering past. Note that messages sent to the inferior connection that are suppressed are not saved up and transmitted later. They are simply lost.

Seek:	L	L	L	F	F	F	F	F	F	F	F	F	F	L	L	L	L	L	L	F	F	F	F	F	F	F	L	
Destroy:		R		R			L		L			R			R	R			L		R			L		L		L
Agent:	L	R	L	F	F	F	L	F	L	F	F	R	F	L	R	R	L	L	L	L	R	F	F	L	F	L	F	L

Figure 6.5: Messages passed from the two behaviours and how they are received by the agent as action.

With **Seek** and **Destroy** connected as shown the agent behaves in the desired way. When there are no enemy agents about it will move to its goal location. When an enemy agent comes into range it will charge after it using **Destroy**. If it does manage to lose the agent, for whatever reason, it will head towards the goal again using the low level behaviour, **Seek**. If it locates an enemy agent again the higher level routine will take over.

Thus it can be seen that **Destroy** *subsumes* the function of **Seek** in order to produce a higher level of competence; hence the *subsumption architecture*. This style of programming where the agent’s AI system is decomposed into a network of task-achieving behaviours is the essence of the subsumption architecture.

The subsumption architecture has a number of significant implications for programming agents. The tight coupling of sensing and actuation means that most behaviour modules can be thought of as simple reflexes. This means that the agent requires very little memory, that the code is uncomplicated and that there is plenty of processing power available.

Another powerful feature of a subsumption organisation of an agent’s intelligence system is that it can be improved incrementally. New layers of competence, in the form of additional behaviours, can be written and then simply wired into the existing structure. Basic capabilities are never lost as new ones are added.

6.2.3 Implementing Behaviours in C

In this approach to implementing AI, the AI code seeks to replace the normal user input for an agent. This means that the agent will still be subject to the normal physics and processing associated with other game agents, and will tend to look more natural, and will be difficult to force to “cheat”. Nobody likes a game agent that can arbitrarily change the laws of physics in order to win.

The behaviour modules are easily implemented. Let's look at the code for the **Destroy** module.

```
#define NONE 0
#define LEFT 1
#define RIGHT 2
#define FORWARD 3

int destroy_msg;

void destroy () {
    int range, bearing;
```

```

FindNearestEnemy (&range, &bearing);
if (range < ENEMY_RANGE_THRESHHOLD) {
    if (bearing > LEFT_THRESHHOLD)
        destroy_msg = LEFT;
    else if (bearing < RIGHT_THRESHHOLD)
        destroy_msg = RIGHT;
    else
        destroy_msg = FORWARD;
} else {
    destroy_msg = NONE;
}
}

```

This module like all the other modules we will implement declares a global variable for the message wire from the module. Here `destroy_msg` contains the information that we wish to send to other behaviour modules. The values `LEFT` and `RIGHT` are defined globally for passing to the action module, and the value `NONE` is used by all the behaviours to indicate that no message is being sent. The use of the value `NONE` is an essential part of our subsumption implementation.

We are now in a position to implement the **Seek** behaviour:

```

int seek_msg;

void seek () {
    int range, bearing;

    FindGoal (&range, &bearing);
    if (bearing > LEFT_THRESHHOLD)
        seek_msg = LEFT;
    else if (bearing < RIGHT_THRESHHOLD)
        seek_msg = RIGHT;
    else
        seek_msg = FORWARD;
}

```

The action module remains to be implemented.

```

void action() {
    if (action_command == FORWARD) {
        /* Simulate the user input for making
           the agent move forward */
    } else if (action_command == LEFT) {
        /* Simulate the user input for making
           the agent move left */
    } else if (action_command == RIGHT) {
        /* Simulate the user input for making
           the agent move right */
    } else {

```

```

        /* Simulate no user input */
    }
}

```

The `arbitrate` function implements message passing between the other behaviours. After the behaviours have been designed, a wiring diagram specifies how they are to be connected. Here `arbitrate` implements wiring instructions with an ordered list of statements. When multiple outputs are directed to the same input, those occurring later in the list of connections subsume (actually overwrite) earlier ones.

```

task arbitrate() {
    while (1 == 1) {
        action_command = seek_msg;
        if (destroy_msg != NONE)
            action_command = destroy_msg;
    }
}

```

It may seem pointless to go through the exercise performed by `arbitrate` when the behaviours could have passed the messages to the motor driver themselves. There are two reasons.

First we wish to maintain modularity. Suppose an agent control system has been written and debugged and contains no formal message passing. All the behaviours write directly to the action inputs. If we now want to add a new layer of complexity, it is not possible to simply write new modules and add them into the connection diagram. Instead, we must have a detailed knowledge of which behaviours pass what and to where and in which order. This might be OK for a very small system, but it quickly becomes intractable for a larger system.

Secondly, the relationship between the subsumption diagram and the code becomes difficult to understand. The connections, rather than being explicitly represented in the connection list, are now hidden in the code. If the order in which the behaviours are executed changes or new behaviours are added, the overall behaviour may change in unexpected ways.

Now all that remains is to activate each behaviour in turn from the main game loop. The process would usually be activated from the same place as you would gather user input in the user's game loop:

```

seek();
destroy();
arbitrate();
action();

```



Modify DEMO12_2 to use the subsumption architecture for the bat AI code. Use the same principles for the ghost code, too. Now modify the bat AI to fly to the centre of the screen when the ghost is out of range. Modify the ghost behaviour to randomly float about when there is no user input.

6.2.4 Modifying an Agent's Behaviour

In the behaviours described above, the key factors in the behaviours were the #defined thresholds on acceptable values between actions. If these values are stored as a property of the agent, they can be modified on the fly to give the agent different reactions in different situations. Similarly, agents that use the same behaviour network can have individual personalities depending on the threshold values. In the example above, the agent can be given a far less aggressive personality by reducing the range threshold at which the agent will go chasing enemy agents.

6.3 Listening to Plans

Often an agent will need advice from a higher level of intelligence, associated with longer term plans for the agent. Plans are evaluated less often than behaviours, and are expected to be valid for some period into the future. Consequently, plans are often based on the predicted future state of the game, rather than the here and now used by behaviours. These plans may be derived from:

- pre-defined lists that the agent must work through,
- using classical AI search techniques to derive a non-reactive solution to a problem,
- a broader situation evaluation function that accounts all of the game factors,

or any of a number of other techniques.

Plans in this sense should not be thought of as producing actions for the agent, rather they should produce goals. The goals can then be achieved in robust fashion by the behaviours. If the goal is not, or cannot, be achieved the agent will still be operational, as the low level behaviours will still operate effectively. The plans become a resource that the agents can use, rather than a pattern that the agent must follow.

In the example above, one might change the goal of the agents **Seek** behaviour so that it moved to different goal locations depending on the broader game state, or according to some pre-defined time table.



Further modify DEMO12_2 to cause the bat to use the crypt entrances as its home locations, swapping between the crypts' entrances at random times. Ensure that the rest of the behaviours remain intact.

6.4 Path Planning

One of the most common tasks of game AI is to get an agent from A to B along an intelligent path without running into anything. The path from A to B forms a plan that should be integrated with behaviour based code that forms the rest of the intelligence of the agent. This section describes some robust and quick to compute techniques for path planning.

This scheme generates a 2d map of the environment which it then segments into navigable and non-navigable segments. This forms the state of the environment. The

idea then is to calculate the shortest path by a form of tree search. Figure 6.6 shows the distance transform algorithm applied to a segmented environment.

								7	6	5	4
								6	5	4	3
						19		7	6		2
					S	18		8	7		1
				19		17		9	8		G
				18	17	16		10	9		1
		19				15		11	10		2
	19	18	17	16	15	14	13	12	11		3
		19	18	17	16	15	14	13	12		4

Figure 6.6: Using a segmented approach to path planning. The black squares are not navigable. The algorithm starts at the goal square and fills the surrounding squares incrementally until the current position is reached.

The distance transform method starts at the goal with a value of 0. It then fills all adjacent navigable squares with an incremented value. It repeats this cycle until the square that contains the agent is reached. It never overwrites a previously filled square. The path is chosen by moving the agent to the adjacent square that contains lowest value. This method of navigation relies on an accurate map of the environment, which must include an updated position of other agents that can also form obstacles. The grid nature of the spatial representation can also lead to "chunky" movement where the agent does not take smooth paths across open regions. Techniques such as path relaxation can make for smoother operation.

The other problem is missed paths. With a coarse grid, a small obstacle near a grid boundary may take up two grid cells instead of one. This means that a cell that an agent may have been actually able to navigate through will appear blocked. The obvious solution to this problem is to have a finer grid. While the increased memory and computation load may be dealt with using improved computing machinery, there is the additional problem that now the agent may find paths that are too narrow for it to navigate. This problem may be solved by increasing the boundary of all obstacles by the radius of the agent, and then treating the agent as a single point.

The path can also be improved by filling the squares with values that reflect the Cartesian distance from the goal rather than the Manhattan distance used above. For example, the algorithm could fill all eight surrounding squares, using an increment of distance of 1 for the adjacent cells, and $\sqrt{2}$ for the diagonal cells. A more complex algorithm is one that calculates the time between cells, allowing for the existing motion of the agent. This means that as a path progresses down a straight line the time between cells gets less, showing the effect of acceleration of the agent.