1 Agents

An **autonomous agent** (or simply an **agent**) is a largely self-contained computational entity which manipulates its environment in response to feedback received from that environment. Engineers refer to this notion as having **closed loop control**. Two of these parts of the definition are particularly important:

> **Self-Contained** If you're controlling your agent with a joystick, it's not autonomous. It's **teleoperated control**.
>
> **Manipulation In Response to Feedback** If the agent doesn't use sensor information about its environment to do its environmental manipulation, but instead just blindly blunders through the environment doing things, it's not autonomous. It's **open loop control**.

There are gradients between teleoperated and autonomous control of course. For example, NASA has only partial teleoperated control over its rovers on Mars. This is because the time to send a command from Earth and get a response back from the robot on Mars is about an hour given the speed of light. So NASA sends high-level commands ("go to that rock") and lets the robot figure out how to handle the low-level details on its own. This raises the teleoperation up one level of abstraction. Approaches like these, along abstraction gradients between full teleoperation and full autonomy, are known as **semi-autonomy**.

An agent can be anything: a robot, a game agent, a web spider, an autonomous stock market bidding agent. The "environment" of an agent doesn't have to be the real world per se: it could be the internet, or a game, or a simulated environment, or the stock market. Furthermore, AI algorithms hardly have to be shoehorned into the agent mold, but it's often common for AI systems, as a whole, to look like agents: after all, the things they're imitating (people) are agents too.

The basic top-level agent architecture is pretty trivial. An agent is just a loop which does three things. First, it gathers information about the environment. We call this information the **external state** of the agent. Second, it uses this information to decide on an **action** to perform. We call this **action selection**. Finally, the agent performs the chosen action:

Algorithm 0 *An Agent*

```
1: loop
2:     externalState ← GetCurrentExternalState()
3:     action ← SelectAction(externalState)
4:     Perform(action)
```

...or if you like,

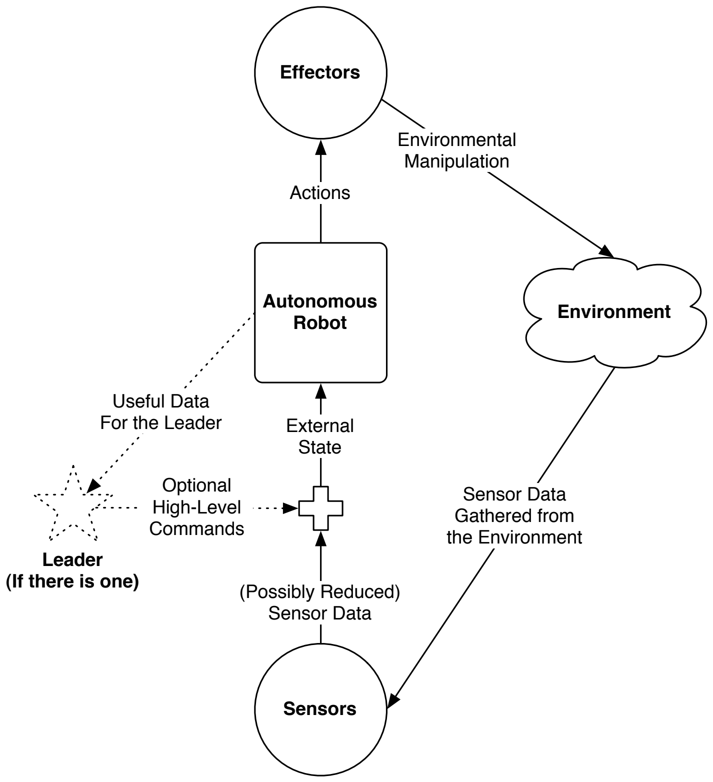
```
1: loop
2:     Perform(SelectAction(GetCurrentExternalState()))
```

Not hard. But these three functions may be arbitrarily complex. Some examples of complexity: GetCurrentExternalState() function might involve a lot of processing (perhaps computer vision routines). And the Perform() function could be recursive: it might fire off a high-level directive to *another* agent to perform in a more detailed fashion. But perhaps most interesting are the variety of forms that the SelectAction() function can take:

- A simple function which produces an action for the given internal state. This function may be stateless, or it may require some small amount of **internal state** (memory) which it uses to decide on the right state at the moment. Such methods are often called **reactive** or **behavior-based** architectures, and we'll discuss them more at length below.
- A function which builds up and retains a significant amount of internal state (memory) about the world which explains how the world works: for example, a map of the environment. Internal state of this kind is commonly known as a **model** of the environment.
- An inductive or **learning** function: a function which develops a model which attempts to predict scenarios the agent has not seen based on previous experience so far.
- A deductive or **planning** function: a function which makes, carries out, and updates sophisticated plans to manipulate the world in order to achieve its ends.

An agent could have any number of combinations of the above too. And agents may have different kinds of goals. Some agents may be seeking a particular ends, such as "move box A from location X to location Y". Other agents may simply be trying to optimize the amount of happiness they receive over their lifetime. Still other (less interesting) agents may be trivially hard-coded to perform operation with no goal or optimization in mind.

1.1 Autonomous Robots


Effectors
Environmental Manipulation
Actions
Autonomous Robot
Environment
Useful Data For the Leader
External State
Optional High-Level Commands
Sensor Data Gathered from the Environment
Leader (If there is one)
(Possibly Reduced) Sensor Data
Sensors

AI has a long and storied relationship with robotics, and much of AI's agent architecture tradition is directly descended from robot architecture history. For this reason, and because robot architectures are rich examples of both complex and very simple agent architectures, it's illustrative to cover them here.

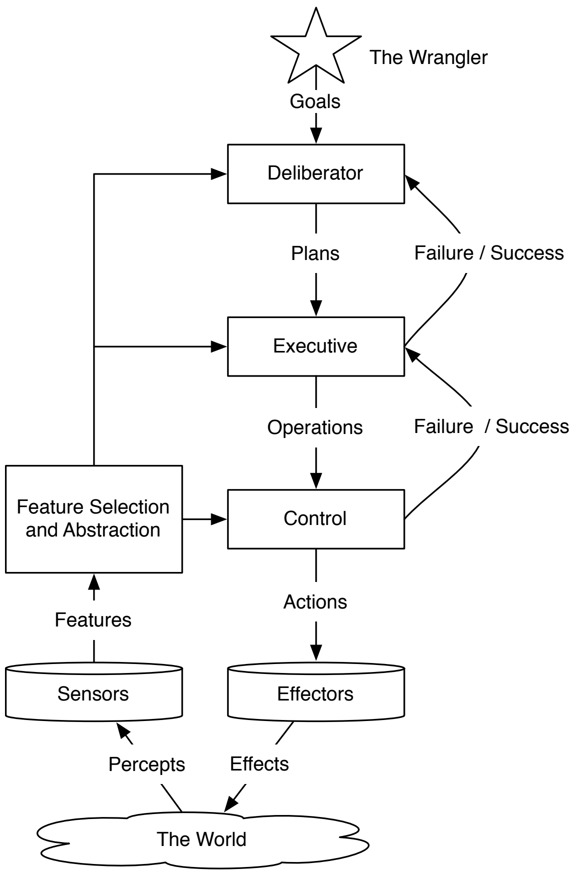
An **autonomous robot** is an autonomous agent whose environment is the real world. A robot manipulates its world by sending **actions** for its **effectors** (wheel motors, gripper servo, arm servos, etc.) to perform. In turn it receives feedback from the environment in the form of **sensor data** gathered by its **sensors**. Sensors may provide this data to the robot in a raw form, or in some abstracted, simplified form designed to be more easily understood and used by the robot. It's common for sensor modules (or sensor-extraction functions in the robot) to reduce the sensory data to a small number of essential **features** which the robot actually cares about. This collection of features is often known as a **feature vector**.

The robot might have a *leader* of some sort: perhaps another agent higher up in the agent hierarchy. Or perhaps the leader might be a human controlling the robot. The leader likely doesn't get his hands dirty with the specifics of what the agent is doing — that'd be more along the lines of teleoperation — but rather directs the agent with fairly high-level directives and receives abstract information from the agent in turn. I like to refer to a human leader as the agent or robot's **wrangler**.

The sensor data provided to the robot, combined with higher-level commands from the robot's leader (if any) collectively form the **external state** of the world as far as the robot is concerned. This is not comprehensive: for example, the external state might include things like signals from other sibling agents, or from subordinates, etc.

1.1.1 The Traditional Robot Architecture and its Discontents

The **three-tiered architecture** is a classic architecture long used in robotics for much of the previous century. The architecture consists of a hierarchy of three modules:



- The **deliberator** constructed high-level **plans** intended to enable the robot to achieve **goals** provided to it by the wrangler.

- The **executive** receives plans from the deliberator and converts them into **sequences** of **operations** for the **controller** to perform The executive also **monitors** the progress of the robot in performing these plans. Monitoring might take the form of **action monitoring**, where the executive determines if a control operation cannot be achieved, or **plan monitoring**, where the executive determines that an entire plan, as presently constructed, cannot be achieved.[22]

  If a plan or operation cannot be achieved as constructed, the executive may attempt to perform a quick-and-dirty **replan** to patch the plan and continue operating. If no obvious patch is possible, the executive will inform the deliberator of its failure, forcing the deliberator to replan from scratch.

[22]The term "action" is very heavily over-used to mean different things. Sorry about that.

- The **controller** attempts to achieve its operation by sending actions to the robot's effectors. Controllers rely on immediate low-level feedback from the environment to achieve consistency and stability in their controlling operation. Controllers are often mixtures of traditional PID **linear controllers** and (more often than not) more complex nonlinear controllers.

Though typically only the controller would actually directly manipulate the effectors of the robot, all three modules might receive sensor information (external state from the world). This sensor information typically takes the form of **sensor features**: sensor information that's directly related to the task of the robot, and are gathered together to form the feature vector. Such features often undergo **feature selection**, to remove certain features from the vector because they're not of importance to the module, or **feature abstraction**, where various features are combined and abstracted into simpler features of interest to the module.

If you squint your eyes you can see how this baroque structure roughly maps into the basic loop that defines an agent architecture. You could view the three-tiered Deliberator, Executive, and Control modules as all part of the classical agent ActionSelection() function.

Or you could view them as three nested agents. The outer agent, the "deliberator agent", receives external state in the form of sensor features and goals from the Wrangler, and its Perform() method issues plans to the second agent. This second agent (the "executive" agent) receives external state in the form of sensor features and plans from the "deliberator" agent, then has its Perform() method issue operations to the third agent. The third agent (the "control") agent receives external state in the form of sensor features and operations from the "executive" agent, then has its Perform() method manipulate the effectors directly.

1.1.2 Example

Let's say our robot's job is to pick up students and take them to the airport. The deliberator has been told goals such as at(student, airport) and may know certain initial conditions such as at(student, GMU). It has a large set of high-level operations with which it can construct plans, such as pickUp(student) or driveTo(airport).

Some of these high-level operations may be **decomposed** into pre-made plans consisting of simpler operations. For example, driveTo(airport) might contain operations like driveTo(BraddockRoad) and turn(fromBraddockRoad, toFairfaxCountyParkway).

The planner sends a completed plan to the executive. Plans often have unordered operations (you can "drive down Braddock" and "call ahead" in either order). The executive decides on some ordering (or **sequencing**) of the operations and performs them one by one. Its job is to make sure each of those operations is successful, possibly find another fix-up to the problem if (say) Fairfax County Parkway is blocked off, and if if such fix-ups don't solve the problem, ask the deliberator to come up with a new plan to get to the airport.

One by one the executive sends operations to the controller, whose job is to perform the operations stably, smoothly, and efficiently. An operation might be driveDown(BraddockRoad). The controller spends all its time driving straight down the road, and staying in lane, using immediate feedback from sensors to stay in lane properly.

All the time new sensor information comes in. Low-level sensor information ("sonar reports obstacle in front 5 meters") is useful for the controller. Information might be simplified for the executive or deliberator ("Fairfax County Parkway appears to be blocked off"), ("student has jumped from car in fear").

1.1.3 Rebellion

In the 1980s this dominant architecture began to be criticized. Some common criticisms:

- The architecture was typically **brittle**, meaning it was not **robust** to unexpected changes in the environment. This was particularly a problem aimed squarely at the **deliberator**.
- The architecture was typically **computationally expensive** at all levels, making it hard to deploy to very simple robots (like insect robots).
- The architecture was particularly **slow** in deliberation, which could be NP-hard or worse. As such, it could not quickly react when the world changed rapidly. If the world was more or less static, the architecture would work fine. But if the world was fast-changing, by the time the architecture got around to constructing and deploying a new plan, the world could have changed so much that the plan was already worthless. This was essentially a complaint derived from the notion of **bounded rationality**: that it's not enough to have an optimally rational agent... one must have one which is as optimal as possible given the computational resources available to it.

The most famous and divisive figure in attacking this architecture was **Rodney Brooks**, who proposed concentrating instead on simple architectures for simple but highly robust robots. Brooks's architecture was known as the **subsumption architecture**, and was applied to small robots with simple tasks. Brooks's efforts led to a whole field, originally known as **behavior-based architectures** (or sometimes **reactive architectures** because they immediately made decisions rather than performing any deliberation).

1.2 Stateless Behavior-Based Architectures

The simplest possible implementation of SelectAction() is **open loop**: the agent ignores the sensor information and simply does its own hard-coded thing, such as "always go forward". This isn't very interesting.

The next step up is more interesting: the agent performs actions based on sensor information, but retains no memory. It doesn't learn or gather model information of any kind. Surprisingly, you can get pretty far with this simple architecture. And keep in mind that while we're using robots as examples, these and several later kinds of architectures are applicable to a wide range of AI agent purposes.

Imagine for the moment that the action the robot performs is any one of *n* discrete actions which you have hard-coded for the robot, such as RunAwayFromTheObjectClosestToMe or GoToTheLight. We might begin by constructing a **table** of if/then statements like this:

If External State is thus...	*...then perform this Action*
Closest Object to me is within 1m	RunAwayFromTheObjectClosestToMe
Closest Object to me is over 2m from me	GoToTheLight
Closest Object to me is between 1 and 2m	SitStill

Our SelectAction function is then just a big case statement of each of these possibilities. We call this kind of robot architecture a **stateless table-driven** architecture. This is the first of several very simple architectures which are commonly known by the collective name of **reactive behavior architectures**.

Assume that Closest Object is the only feature we care about. Then notice two important items about this table. First, every possible value of ClosestObject is handled by an action. If this was not true, we'd describe the sensor situation as **under-specified**. Second, no single value of ClosestObject is handled by *more than one* action. If this was not true, we'd describe the sensor situation as **over-specified**. Note that it's entirely possible for the sensor situation to be *both* over- and under-specified.

1.2.1 Under-Specification

It's easy to deal with under-specification: just have a default value. For example, we could have had a default like this:

If External State is thus...	*...then perform this Action*
Closest Object to me is within 1m	RunAwayFromTheObjectClosestToMe
Closest Object to me is over 2m from me	GoToTheLight
In All Other Situations	SitStill

Of course, under-specification suggests a possible flaw in your approach to solving the problem. What happens when that default occurs? You might want to verify that things are looking good there.

1.2.2 Over-Specification

Over-specification is more complex. Here's an over-specified program:

If External State is thus...	*...then perform this Action*
Closest Object to me is within 0.5m	Yell "Get out of the way!"
Closest Object to me is within 1m	RunAwayFromTheObjectClosestToMe
Closest Object to me is over 2m from me	GoToTheLight
In All Other Situations	SitStill

So, if we're within 0.5m do we yell or do we run away? Both actions would be valid according to these rules. There's no reason, of course, that we can't do both simultaneously. If all of our actions are **independent** of one another, then we could deal with over-specification simply by doing, in parallel, every action that got triggered by our current external state.

Action independence is fairly rare though. For example, what if we with our robot to perform different actions based on different sensor values?

If External State is thus...	*...then perform this Action*
Closest Object to me is within 1m	RunAwayFromTheObjectClosestToMe
I See The Light!	GoToTheLight
In All Other Situations	SitStill

Here, our robot runs away from close-by obstacles, but also tries to follow the light. What happens if the robot can see the light *and* there's an obstacle within 1m? Which action does it do?

We need some form of **arbitration** to determine what action(s) to perform.

1.2.3 Arbitration

When several rules are triggered by the same external state situation, and they differ in the action they propose, our robot will have to decide on which one to do. This is known as arbitration. There are a variety of approaches to doing arbitration:

Simultaneous Actions If rules suggest two different actions, and the actions do not conflict with one another, we could plausibly just do both actions at the same time.

Voting If several rules are triggered, the action pushed by the most rules wins.

Weighted Voting Some rules are more important than others, and their vote counts more. Perhaps this is because they're an important rule. Or perhaps it's because their need is urgent ("da plane! It's da plane, boss!") and they have a higher temporary strength as a result.

Most Specific Rule if a rule's range is smaller than another, it might take precedent because it is considered to be a more accurate description of what should happen in a situation. For example, Yell was triggered at 0–0.5m, but RunAwayFromTheObjectClosestToMe was triggered at 0–1m. Here, Yell would be selected.

Precedence Some rules take precedence over others. The simplest approach here is to just order the rules, and pick the first one which matches the situation.

Subsumption Like Precedence, in Subsumption rules may "turn off" or "subsume" other rules. But when a rule is turned off, it *loses the ability to turn off other rules in turn*. For example, consider three rules A, B, and C, where A turns off B and B turns off C. B and C have both fired. Since B turns off C, B wins and its action occurs. But imagine if A also fired. A turns off B, making B incapable of turning of C. Thus A's and C's actions occur. Perhaps A doesn't have an action at all — it only exists to turn things off. Then C happens even when B is trying to turn it off. This notion was originally proposed by Rodney Brooks, a roboticist who made reactive behaviors popular. He saw this as a way to have high-level abstract goals override specific, low-level immediate needs among *parallel behaviors* (more on that later). But I've mostly found it helpful for basic action arbitration.

Averaging If the actions can be described mathematically, they might be averaged. For example, if GoToTheLight defined a vector of magnitude 1 pointing north (where the light has appeared), and RunAwayFromTheObjectClosestToMe defined a vector of magnitude 1 pointing east (because the Bad Guy was to the west at the moment), we might add the two together into an action vector of magnitude $\sqrt{2}$ pointing northeast. If some rules are stronger than others (see Weighted Voting earlier), we might weight the vectors before adding them. There is a specific behavior technique used in mobile robot trajectory planning called **potential field guidance** which relies on adding weighted vectors and then moving in the direction of their sum. More on this approach later.

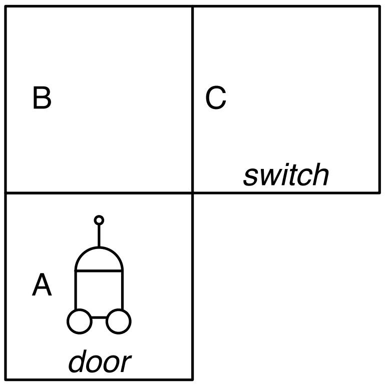
Random Choice Pick an action at random. You might choose from a uniform distribution over the actions, or you might weight the action choice-probabilitiess according to votes for those actions or pre-defined rule importance.

These arbitration mechanisms are not the only ones, and they can be used in a variety of combinations: you could for example, use weighted voting to pick an action, and break ties using the most specific rule, and break further ties using a hard-coded precedence.

1.3 Stateful Behavior-Based Architectures

The problem with the simple architectures described earlier is that they do not contain internal state: the robot does not maintain a history or memory of what it has done. This is important because sometimes the robot will find itself in two different situations where its *external state* is the same, but where the action to perform must be different. How is the robot to know what to do?

For example, consider the example at right. The robot is presently in the cell with a locked door. To open and go out the door (the robot's goal), the robot must go to the cell with the switch, and flick the switch, then return to the cell with the door.



It sounds like an easy problem, but there's a catch. The robot can tell what room he's in from his sensor data — the wall configuration, for example — and in room **A** he can tell if the door's open, and in room **C** he can tell if the switch is flicked. But in room **B** he can't tell whether the switch has been flicked or if the door is open. The previous architectures would have the robot decide what the robot must do in any given room entirely based on this external state. So in the room marked **A**, should the robot go down or right? It depends on whether the switch has been flicked. But the robot can't see the switch!

We call this situation an **aliased state**: a situation (in this case, cell **B**) whose external state information is insufficient for the robot to rely on to determine what to do. The simplest solution to this problem is to add some degree of **randomness** to the robot's behaviors, so that eventually it'll make the right decision to go from **B** to **A**. This will work in theory but may take a very long time; additionally randomness may not be desirable if certain random decisions imperil the robot.

Here's a different approach. In this situation, the robot needs one bit of internal memory. It's initially 0. When in cell **B**, at internal state 0, the action is to move *right*. When the robot flicks the switch, the robot also changes its internal state to 1. Thus when the robot reaches **B** again, with internal state 1, it knows to go *down* instead, in order to ultimately reach the door.

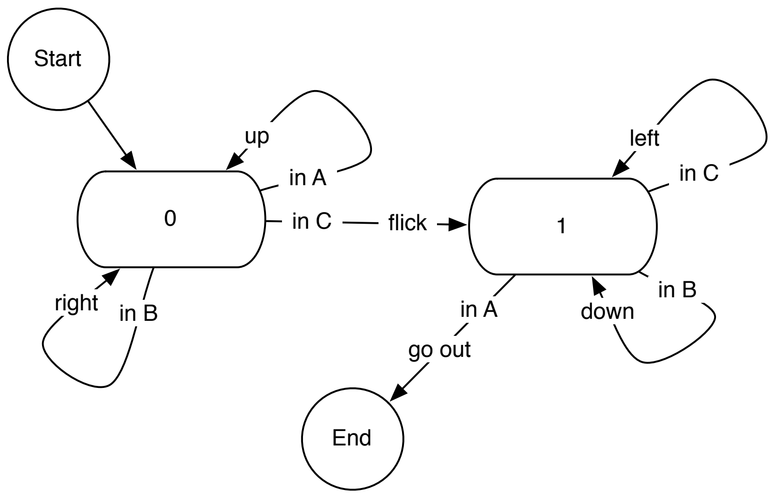
So instead of a table with two columns (external state; action), we need a full finite state machine. A full state machine has four columns (external state; internal state; action; change internal state). Each row now says: "if the external state is thus, and the internal state is thus, then perform the following action, and change the internal state thusly".

Here's a simple one-state program which does this:[23]

[23]Note that this program assumes that there's no randomness or malicious entity in your world which causes the switch to get flicked back occasionally when you leave room C. What would you change in this table to be a little more robust?

If External State is ...	*And Internal State is ...*	*...then do Action...*	*...and change Internal State to...*
Room A	0	Up	0
Room B	0	Right	0
Room C	0	Flick	1
Room C	1	Left	1
Room B	1	Down	1
Room A	1	Go Out	1

Here's the table expressed as a **Mealy machine**, a version of a finite-state automaton where nodes are internal states and edges are labelled both with transition functions and with actions to perform.



Note how we've written this: states are nodes in the graph, and edges are rules (rows in the table above). An edge's relationship with a rule results in it being written like this:

$$\begin{tikzcd}\boxed{\text{IfInternalStateIs}} \arrow[r, "\text{andExternalStateIs}\text{——}\text{thenDoAction}"] & \boxed{\text{andChangeToThisInternalState}}\end{tikzcd}$$

1.4 Continuous Behavior

Sometimes modeling a robot behavior as a finite state machine is easier even if the behavior can be done entirely with a simple, stateless system. This is often the case because robots often do the same basic thing while in a given state; because they often run forever in a loop; and because accidents happen and they have to deal with that.

Imagine a mobile robot which is capable of seeing cans out of a camera on its front, and also capable of grabbing cans once it's close enough to see them. Its job is to collect cans in a big pile, and its approach to doing this is to find a can, grab it, find another can, and move the first can next to the second. It repeats this over and over until, eventually, all the cans are together in the pile. It's possible for the robot to lose grip, and also possible for it to lose sight of a can. After the robot has grabbed a can, and after it has released a can, it spins for a large random angle. It does this to prevent it from grabbing a can out of a pile and then suddenly deciding that this pile is the one it's going to put the can back in.

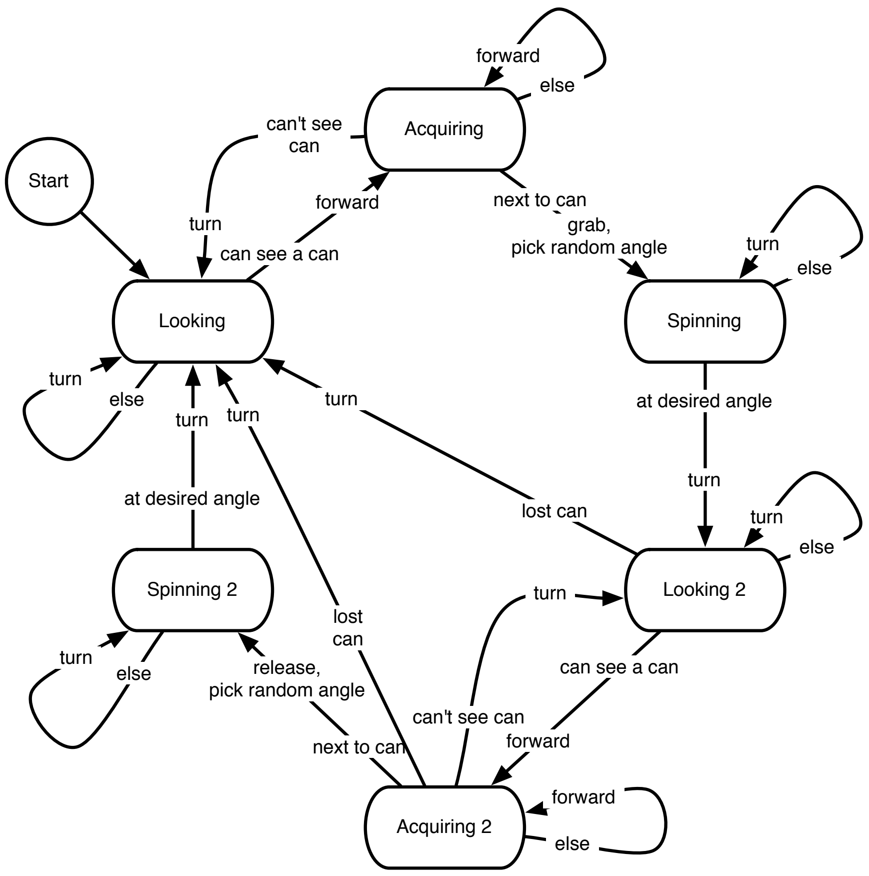


Figure 1 Mealy Machine for can collecting.

If the robot had sensors to tell it that it was holding a can, *and* the robot had a built-in timer which would send an even to it when it had ″spun enough″, we could in fact do this entirely as a stateless system:

1. If I'm not holding a can and I can't see a can, turn until I see a can.
2. Else if I'm not holding a can and can see a can but I'm not close to it, move towards the can.
3. Else if I'm not holding a can and I can see a can and I'm close to it, grab it and start the timer.
4. Else if the timer is running, turn.
5. Else if I'm holding a can and can't see another can, turn until I see a can.
6. Else if I'm holding a can and can see another can but I'm not close to it, move towards the can.
7. Else if I'm holding a can and can see another can and I'm close to it, release my can and start the timer.

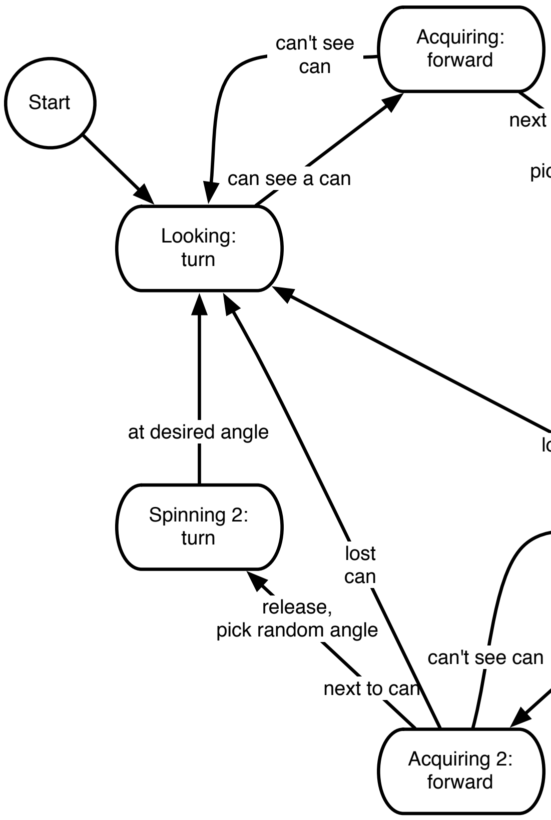


Figure 2 Moore Machine for can collecting.

But it's a bit easier to understand the behavior if we model it with some internal state. And if we don't have a way to tell if we're holding a can or if a timer is set, we'd have to do it with internal state nonetheless. Figure 1 shows a possible six-state finite-state Mealy Machine for this behavior:

Formally, internal state is used to distinguish between aliased external state situations. Here we are using partly for that, but mostly to distinguish between "categories of things the robot is working on". Note that while staying in a given state the robot is basically doing one particular action; though while transitioning from state to state it may occasionally do a different action. This means we may often simplify the state diagram by putting the basic action inside the node with the state, and get rid of self-edges. As it turns out, many remaining edges are simply doing the action of the state they're pointing to, so we can simplify them to just:

$$\boxed{\text{IfInternalStateIs}}\text{—andExternalStateIs}\longrightarrow\boxed{\text{andChangeToThisInternalStateAndDoThisAction}}$$

This state-machine formalism is known as a **Moore machine**. We'll temper it a bit: some edges still need to do some custom action on transition. Thus they may still be labeled with that action and are written as usual. All this allows us to simplify this state machine diagram as shown in Figure 2:

Just because the previous example could, in theory, be done with internal state if you had the right sensors, doesn't mean that internal state isn't all that useful. Consider, for example, if your

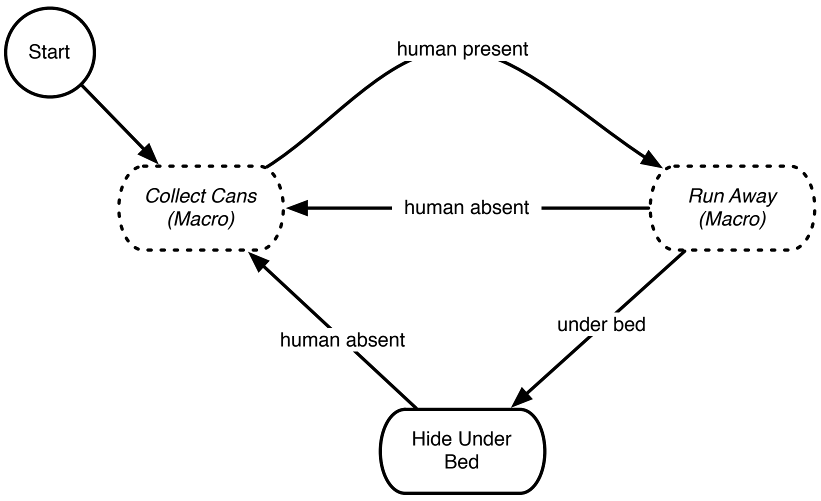


Figure 3 Hierarchical Finite-State Automaton (HFA) for can collecting and other tasks.

job is to collect cans of *two different colors* and sort them into piles by color. Now the particular can you're going after will depend on the can color you're holding. If you don't remember what can color you're holding, you're in trouble unless you have some sensor which can tell you. Two cameras can be expensive. Memory is cheap!

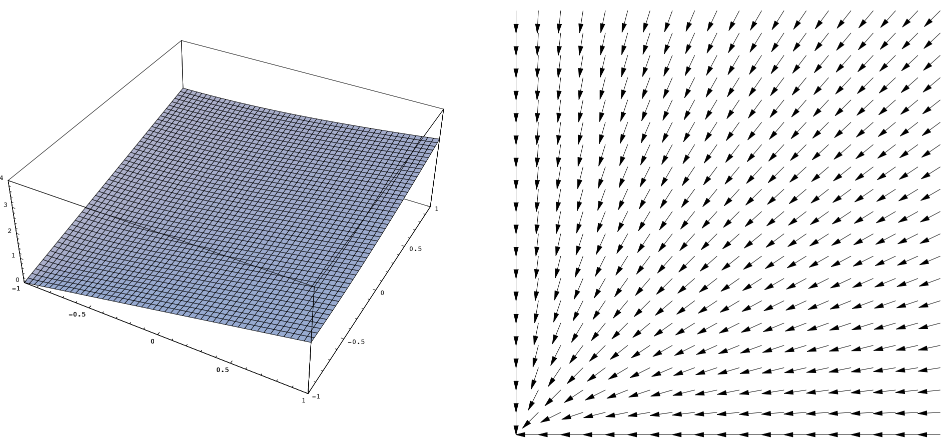
1.5 Finite-State Automata Macros

Building a large FSA like this can quickly get tedious: it's like coding everything in one giant function. One way to alleviate this is to encapsulate common behaviors into *finite-state machine macros* which can be triggered by an FSA at a higher level. Consider, for example, a cockroach can-collecting robot. This robot's job is to collect cans when people are not around, but when a person is around, it runs and hides under the bed to get out of the person's way. When the person leaves, can collecting resumes. We might encapsulate our previous can-collecting behavior into the macro Collect Cans. In the diagram shown in Figure 3, two macro behaviors (Collect Cans and Run Away) are used along with a basic behavior, Hide Under Bed.

This higher-level FSA hands off control to lower-level macro finite state automata when it enters the states representing them. When a lower-level FSA is triggered, the higher level FSA is pushed on a stack and paused, and the lower-level FSA begins at its Start state. But how do we *exit* the lower-level FSA? This is done when an event occurs for which the higher-level FSA has defined a transition edge away from its macro node. In this case, if humans present is triggered, then the Collect Cans macro FSA is immediately quit, and control is passed back up to the higher-level FSA, which then transitions to the Run Away state. This state in turn happens to be a macro FSA node, and so its underlying FSA is fired up in a similar fashion.

A finite-state automaton which uses other a finite-state automata in a recursive macro fashion like this is commonly known as a **hierarchical finite-state automaton** or **HFA**. Such beasts are particularly common in the game industry. Some things to notice:

- The human present even in some sense *overrides* the Collect Cans finite-state automata events.



(a) Motor Schema Function Surface (b) Potential field (gradient)

Figure 4 Motor schema behavior for a robot heading from top-right to bottom-left in the absence of obstacles. Subfigure (a) describes the function as essentially a tilted floor: wherever the robot is, it wishes to go down that floor. Subfigure (b) shows the same function as a potential field or gradient.

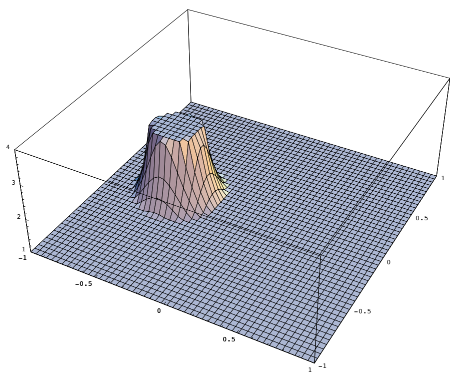
This provides a form of subsumption[24] among behaviors, allowing escape hatches from behaviors when more pressing needs arise.

- The high-level FSA contains a mix of basic states and encapsulated macro FSA nodes. Indeed, there could be any number of levels to the hierarchy. But FSAs don't play well with recursion: at the very bottom of any hierarchy, at any point in time, there ought to be a basic state performing its basic action.
- The Collect Cans and run Away macro nodes do not have an "action" associated with them so to speak, whereas the Hiding node has the lurk action associated with it. This is natural as the macro nodes are performing actions hidden in their underlying FSAs, whereas the Hiding node requires an action (it's a basic behavior). Generally macro behaviors are more easily expressed when using the "simplified" state machine diagram form discussed earlier.
- There's no reason that we can't build an exit mechanism into a macro-level FSA when it's done with its task. At the very least, we could create a special behavior called EmitEvent which emits an event which might trigger a transition at a higher level. Or we could pre-define an exit state which higher-level FSAs might recognize as a cue to exit the FSA in one or more ways.

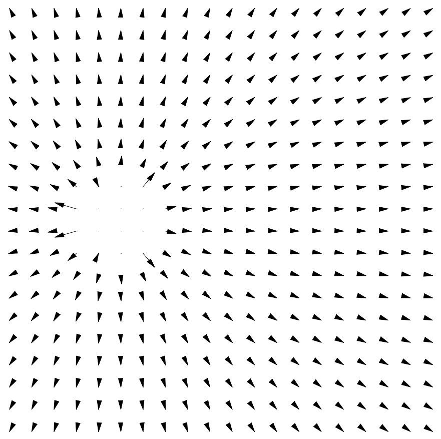
1.6 Motor Schema Behaviors

A **motor schema** is a simple arbitration model where multiple behaviors each output a desired *vector* to direct the robot, and the arbiter then simply **adds up the vectors** and the robot goes in the

[24]One important difference is that Subsumption "pauses" the preempted behavior—when the higher-level behavior is no longer active, the preempted behavior might resume automatically. But if we exit the lower-level FSA, it's not resumed automatically. How might we modify this whole procedure to allow situations like this? Certainly it'd be nice to pause a behavior to handle something else for the moment, and then return to it as necessary. Perhaps FSAs which remember their previous state prior to being exited?



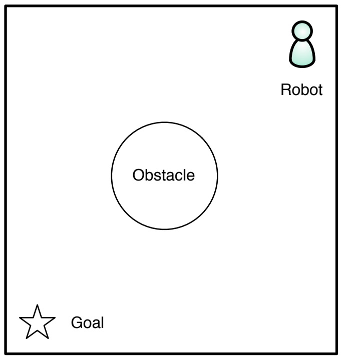
(a) Motor Schema Function Surface



(b) Potential field (gradient)

Figure 5 Motor schema behavior for a robot heading away from a circular obstacle (more force is applied if the robot is closer to the obstacle surface).

resulting direction. The magnitude of the vector may be viewed as how badly the behavior feels the robot should go in a given direction.



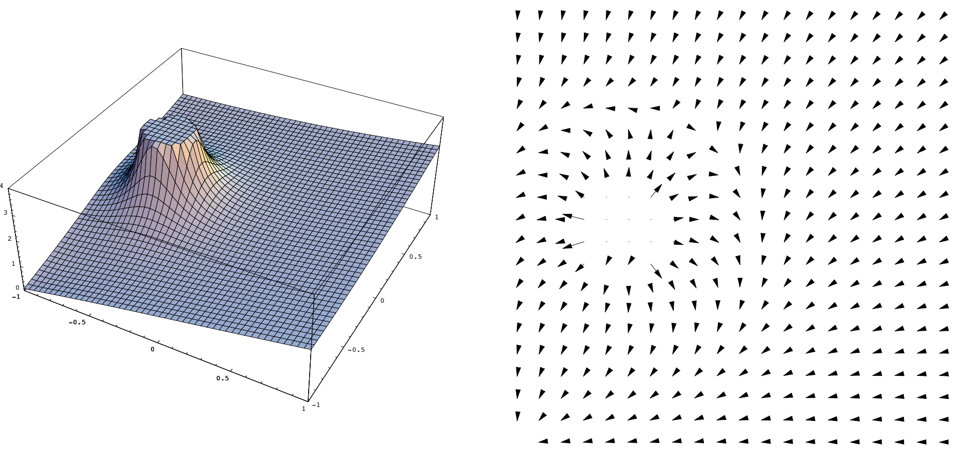
Imagine that a robot starts at the top-right corner of a room and wishes to reach a goal location in the bottom-left corner. If the robot hits the walls of the room it's no big deal. But there also exists a circular *obstacle* in the robot's way. We wish to craft a behavior to direct the robot to go directly to the goal, but route around the obstacle if necessary.

Let's begin with a function which directs the robot to the goal in the absence of an obstacle. This direction takes the form of a **force** on the robot to urge it to go in a given heading. The force takes the form of a function $f(\vec{x}) \rightarrow \vec{v}$, which takes a location $\vec{x}$ and returns a vector $\vec{v}$ indicating the direction the robot should head and (if possible) the speed with which the robot should go. Such a function might be described with the Figure 4. In this Figure, the arrows in Subfigure (b) show the direction the function will guide the robot: and the magnitude of the arrows is how strongly they will urge the robot to move in that direction.

Though the plots show the function plotted for all points in the room, it is important to note that **this function does not in any way require global knowledge about the room**. For example, all it might require is knowing where the goal is, so it can orient the robot in that direction. This could be done with a simple beacon located at the goal, for example.

Now consider a different function $g(\vec{x}) \rightarrow \vec{v}$, which acts to push the robot away from the obstacle. The closer the robot is to the obstacle, the stronger the force. Figure 5 shows such a function.

One could imagine functions to direct the robot down hallways, through doorways, away from walls, and so on. Though we can't show it easily in the figures here, it's perfectly plausible to have a dynamic function as well, where the arrows aren't fixed in stone. For example, consider a function which avoids the nearest obstacle detectable by the robot. If that obstacle is a person approaching,



(a) Motor Schema Function Surface (b) Potential field (gradient)

Figure 6 Motor schema behavior for a robot heading away from a circular obstacle and towards the goal.

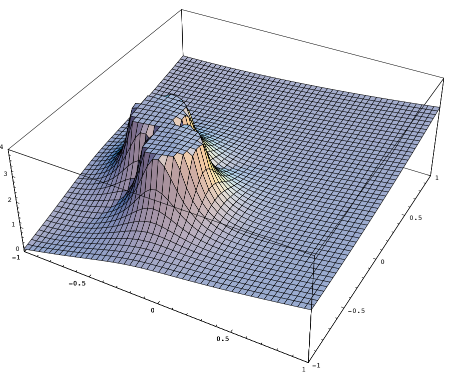
the robot would shy away from the person as he walks by.

Motor schemata become interesting when you add the functions together. We create a new function $h(\vec{x}) = f(\vec{x}) + g(\vec{x}) \rightarrow \vec{v}$ which simultaneously directs the robot to the goal while avoiding the obstacle. Figure 6 shows this function. As can be seen by the potential field version of the function, the robot will veer away from the obstacle and "slide" down towards the goal.

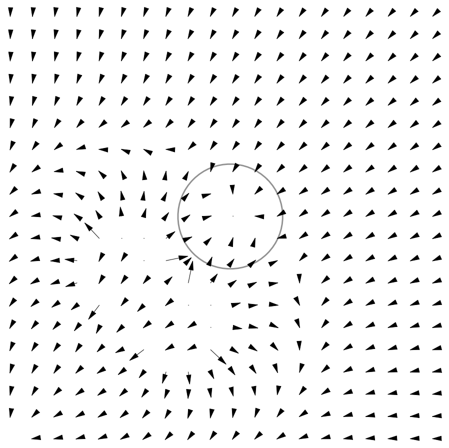
The problem with motor schema functions is that they are typically greedy. The function only considers the robot's current location and the situation in that region, and not the entire environment, in telling the robot where to go: no global path-planning is going on. Of course we could *construct* a motor schema which represents globally optimal decision making at all positions, but this is rarely the case: usually it's just local decision-making based on the robot's sensors.

How does this get us into trouble? Consider Figure 7, with a figure-eight-shaped obstacle in the way. Obviously the robot could slide around the obstacle, but it doesn't know that. The robot will initially head straight towards the goal (and the obstacle) and then get caught in a little suboptimal zero-gradient valley directly before the obstacle, where all around the valley forces are recommending that the robot go down to the bottom of the little valley. The robot will get stuck here.

The easiest way to get out of such suboptima (assuming we have a greedy motor schema function) is again to use a bit of randomness in our actions. If with some very small **epsilon** of probability the robot performs a random action rather than the one asked of it, it'll eventually shake itself out of that local suboptimum. But again it may be quite a while. If the suboptimum is a broad bowl, the robot will require a large number N of lucky random actions to make it out of the bowl; else it'll get sucked right back in. This can be remedied by making epsilon fairly large: but then we have a robot making crazy random actions all the time. This is a version of the classic optimization trade-off known as **exploitation versus exploration**: do you want your robot wasting time *exploring* (lots of randomness) to get out possible local suboptima — if ones even exist! — or would you like its time making locally optimal actions (*exploiting* the gradient)? And again, it's possible that randomness may not be particularly desirable if some robot actions are detrimental to its well-being.



(a) Motor Schema Function Surface



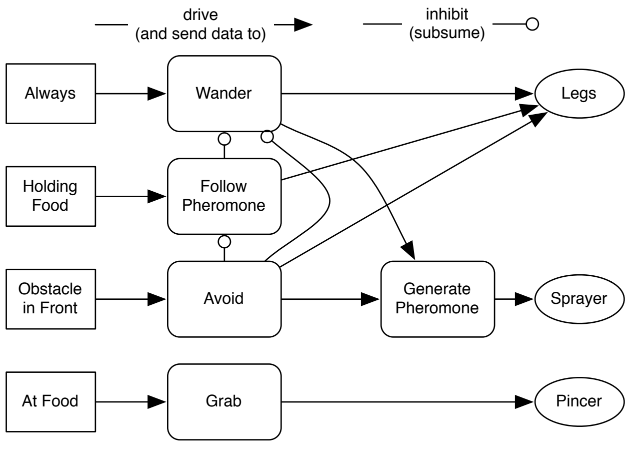
(b) Potential field (gradient)

Figure 7 Motor schemata behavior for a robot heading away from a two-circle obstacle and towards the goal. The gray circle in Subfigure (b) indicates a **local suboptimum with zero gradient** in which the robot may get stuck.

1.7 Subsumption

Rodney Brooks's **subsumption architecture** was the original simple robot behavior. Unlike the finite-state automata discussed earlier, but somewhat like the motor schema architecture, subsumption is a **parallel** architecture. All behaviors are simultaneously running. Though not all of them may be simultaneously having an effect.

Subsumption behaviors are wired up to one another in various ways. Behaviors can **drive** other behaviors, or send **signals** to one another, telling other behaviors to start working, or to change how they do their task. But a behavior can also explicitly **inhibit** or **subsume** other behaviors — make them unable to have an effect — while the behavior is doing its task. This way only certain behaviors are having an effect on the motors at a given time.



The diagram at right shows a simple subsumption architecture for a pheromone-spraying ant-like robot whose task is to find food and take it back home. The robot has four simple sensors ("I am holding food", "there is an obstacle in front of me", "I am at the food", and "this sensor always reports true"). These sensors in turn drive various behaviors ("Wander", "Follow Pheromone", "Avoid", "Grab") which inhibit one another and which may trigger other behaviors ("Generate Pheromone") and ultimately drive the legs, pheromone sprayer, and pincer of the robot.

These behaviors can operate in parallel. For example, Grab can operate in parallel with Wander, Follow Pheromone, or Avoid, and also with Generate Pheromone. Other behaviors have been set up to explicitly exclude others ("Avoid" turns off "Wander" and "Follow Pheromone").

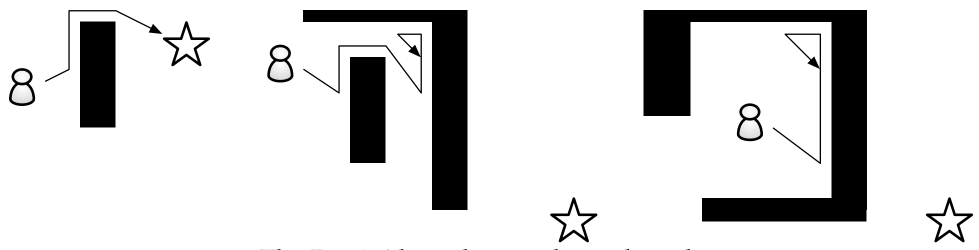


Figure 8 The Bug0 Algorithm on three obstacle scenarios.

Subsumption can be stateful in theory and show a fair degree of complexity in the kinds of signals, inhibitory mechanisms, and behaviors available. And subsumption obviously can be parallel. But in my experience many subsumption architectures tend to reduce to hierarchies of behaviors with explicit ordering. For example: if there's a bad guy, hide; else if there's food, eat it; else if you're hungry, look for food; else sleep. This isn't a comment on subsumption per se — perhaps many common behaviors and tasks fall into explicit hierarchies.

Ultimately subsumption architectures must deal with the same issues of arbitration as other behaviors. When two behaviors contend for control of the legs, and one doesn't explicitly inhibit the other, how are the legs controlled? What if one behavior inhibits another, but a third behavior insists that the inhibition be removed? Such arbitration issues might be handled inside the behaviors themselves, deciding from among various signals what to do.

1.8 Bug Algorithms

Bug algorithms are examples of greedy path-planning behaviors consisting of simple rules. They get their name because they look like the path a bug might take to reach its goal in the presence of obstacles: hugging the obstacle tightly as it rounds about it. Most Bug Algorithms have two basic parts:

1. When to head straight towards the obstacle and how to do that.
2. When to start wall-following the obstacle.

There are quite a number of bug algorithms. Here we just show a few.

1.8.1 The Bug0 Algorithm

The Bug0 algorithm is a suboptimal algorithm which is usually the first one a person might think of when working on the problem. Bug0 works like this:

1. Head straight towards the goal.
2. When you encounter an obstacle, turn left and start wall-following it.
3. When you can start going towards the goal without hitting an obstacle, go to #1.

Figure 8 shows the Bug0 algorithm in action on three obstacle scenarios. As can be seen, on the far left (simplest) scenario, the Bug0 algorithm works just fine. Indeed, on any scenario where the obstacles are simple convex shapes, Bug0 will work without a hitch. However on scenarios with concave shapes, things become problematic. Note that the agent is caught in an infinite loop in the center and right scenarios.

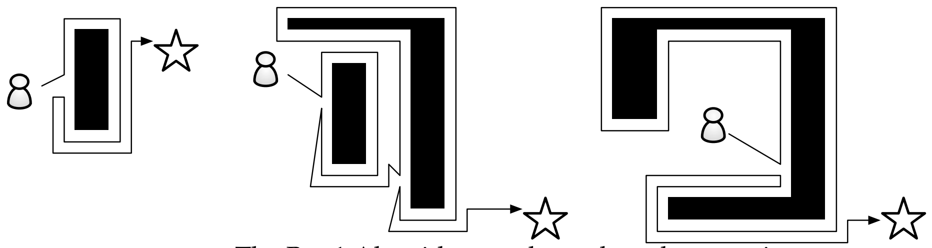


Figure 9 The Bug1 Algorithm on three obstacle scenarios.

1.8.2 The Bug1 Algorithm

The difficulties with the Bug0 algorithm stems from the fact that it has no memory, and obstacle avoidance would benefit from it. The Bug1 algorithm adds a bit of memory, and works as follows:

1. Head straight towards the goal.
2. When you encounter an obstacle, turn left and wall follow *all the way around it*. As you are going, memorize the closest point to the goal that you discovered on the border of the obstacle.
3. When you have completed wall-following around the entire perimeter of the obstacle, back-track to the closest point to the goal that you discovered on the border of the obstacle.
4. Go to #1.

Figure 9 shows the Bug1 algorithm in action. The Bug1 algorithm is guaranteed to reach the goal unless the robot is trapped inside an O-shaped obstacle. In this case, Bug1 can detect this inescapable situation as it repeatedly attempts to head to the goal from the same "closest" location.

The problem with the Bug1 algorithm should be obvious from the diagrams: it can be quite expensive. The robot will wander clear around an obstacle even when it can obviously short-cut this path and head to the goal.

1.8.3 The Bug2 Algorithm

The Bug2 algorithm is a variant of the Bug0 algorithm which attempts to overcome some of its deficiencies. The algorithm begins by defining a line L from the goal to the robot's starting point. The robot only heads to the goal at points which intersect this line. The algorithm works as follows:

1. Determine L.
2. Head along L straight towards the goal.
3. When you encounter an obstacle, turn left and start wall following.
4. When you reach a point on L where you can continue going towards the goal, go to #2. Note that this point may be on the other side of the goal.

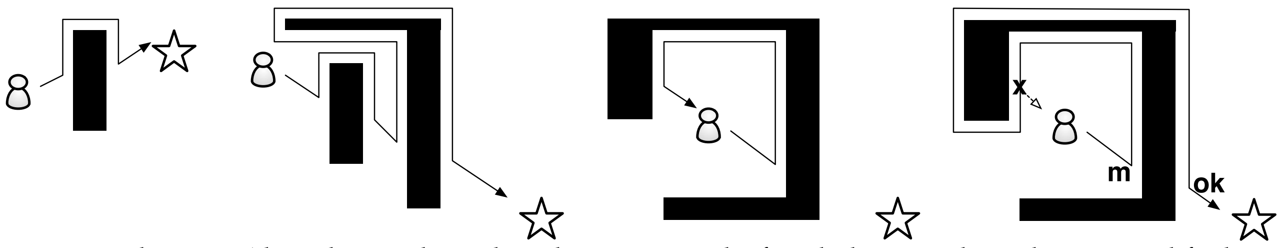


Figure 10 The Bug2 Algorithm on three obstacle scenarios. The fourth diagram shows how to modify the Bug2 Algorithm to solve the third scenario. The algorithm notes the point **m** on the line passing through the robot's starting point and the goal. The point **x** denotes a place on this line which is ignored because it is further form the goal than **m**. Finally the point marked **ok** denotes a valid point on the line because it is closer to the goal than **m**.

Figure 10 shows the Bug2 algorithm in action. Note that though the Bug2 algorithm can now solve the second obstacle group, it is still stuck in an infinite loop in the third obstacle. Again, the basic issue is memory: the algorithm doesn't realize that when it stops wall-following, it is doing so at a point which has made *less progress* than the point where it *started* wall-following.

The solution to this is to have the robot keep track of the point *m* it has encountered along *L* which has been closest to the goal. Armed with this, we can modify the Bug2 algorithm as follows:

1. Determine *L* and set *m* to the robot's start location.
2. Head along *L* straight towards the goal.
3. When you encounter an obstacle, First, determine if the current location *c* is closer to the goal than *m* was. If so, set *m* to *c*. Regardless, turn left and start wall following.
4. When you reach a point *d* on *L* where you can continue going towards the goal, and *d* is closer to the goal than *m* was, set *m* to *d* and go to #2. Note that this point may be on the other side of the goal.

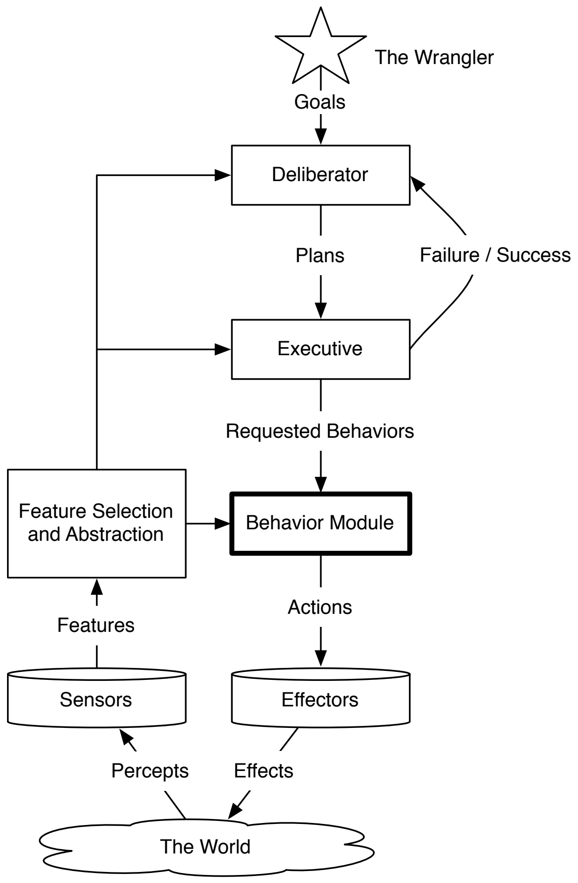
The far-right diagram in Figure 9 shows this revised algorithm in action, solving the third obstacle group. Like Bug1, Bug2 is guaranteed, but often is much faster. There do exist pathological scenarios where Bug1 is faster though.

1.9 Hybrid Architectures

There's absolutely no reason why you can't have hybrids of behaviors. You could have a hierarchical finite-state automaton with behaviors which take the form of subsumption architectures, which in turn have behaviors which take the form of motor schema, or other kinds of behavior architectures in the mix.

More importantly, nowadays many modern systems employ some kind of **hybrid of behaviors and deliberative systems**. There are many different architecture choices which merge the two. But many of them essentially replace the control subsystem with some kind of behavior-based mechanism, as shown at right.

It's not the only possibility: you could have behaviors in the executive, sending requests to the control module. You could have both a behavior-based system and a deliberative system side-by-side, and arbitrate between the two when their guidance to the controls or effectors differ. In fact, you could have a behavior-based system where one of the behaviors is a deliberation procedure. There are a lot of possibilities.


The Wrangler
Goals
Deliberator
Plans
Failure / Success
Executive
Requested Behaviors
Feature Selection and Abstraction
Behavior Module
Actions
Features
Sensors
Effectors
Percepts
Effects
The World