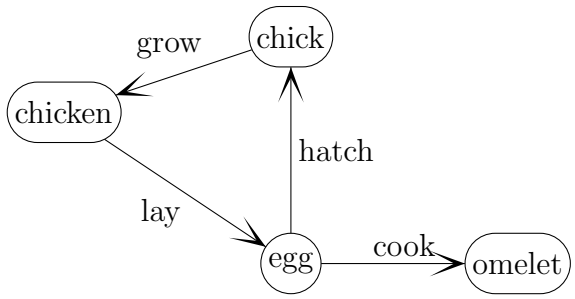
Chapter 19

State Diagrams

In this chapter, we'll see state diagrams, an example of a different way to use directed graphs.

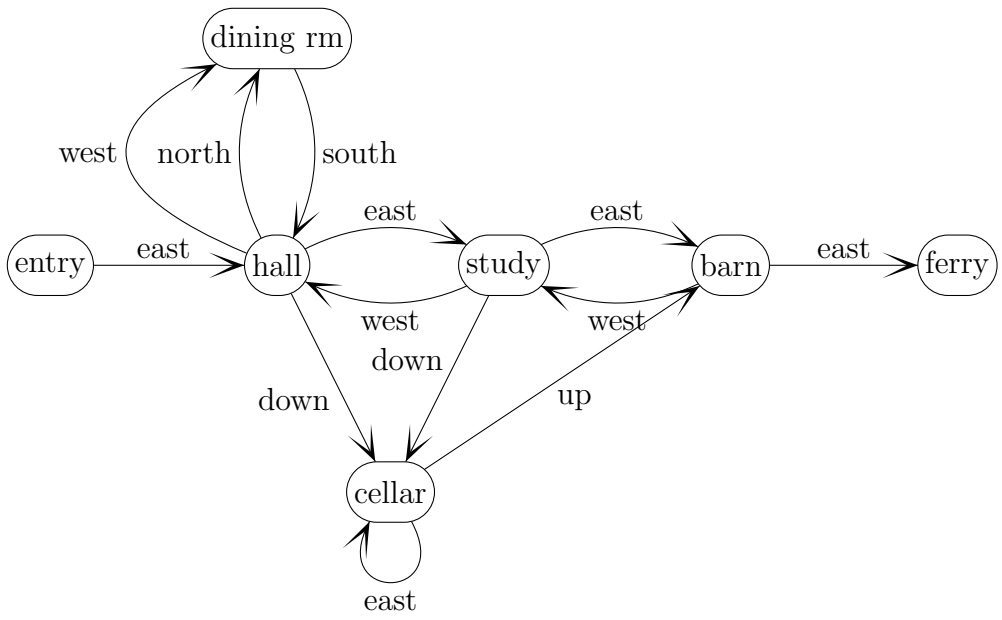
19.1 Introduction

State diagrams are a type of directed graph, in which the graph nodes represent states and labels on the graph edges represent actions. For example, here is a state diagram representing the life cycle of a chicken:



The label on the edge from state A to state B indicates what action happens as the system moves from state A to state B. In many applications, all the transitions involve one basic type of action, such as reading a character

or going though a doorway. In that case, the diagram might simply indicate the details of this action. For example, the following diagram for a multi-room computer game shows only the direction of motion on each edge.



Walks (and therefore paths and cycles) in state diagrams must follow the arrow directions. So, for example, there is no path from the ferry to the study. Second, an action can result in no change of state, e.g. attempting to go east from the cellar. Finally, two different actions may get you to the same new state, e.g. going either west or north from the hall gets you to the dining room.

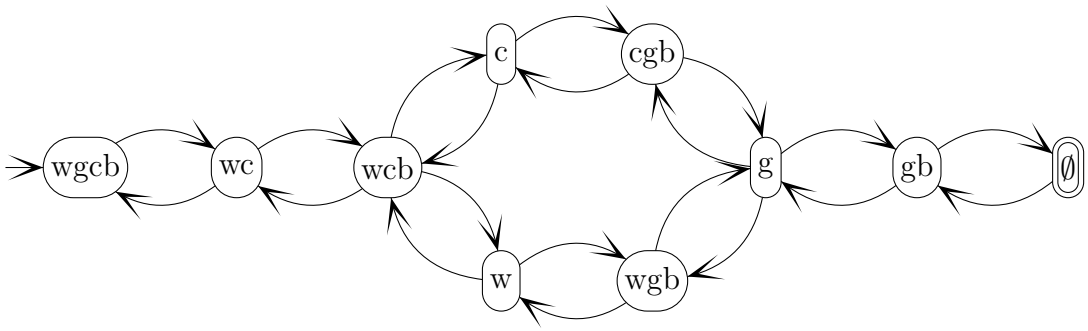
Remember that the full specification of a walk in a graph contains both a sequence of nodes and a sequence of edges. For state diagrams, these correspond to a sequence of states and a sequence of actions. It's often important to include both sequences, both to avoid ambiguity and because the states and actions may be important to the end user. For example, for one walk from the hall to the barn, the full state and action sequences look like:

states:	hall		dining room		hall		cellar		barn
actions:		west		south		down		up	

19.2 Wolf-goat-cabbage puzzle

State diagrams are often used to model puzzles or games. For example, one famous puzzle involves a farmer taking a wolf, a goat, and a cabbage to market. To do this, he must cross from the east to the west side of a river using a boat that can only carry him plus one of his three possessions. He cannot leave the wolf and goat together unsupervised, nor the goat and the cabbage, because one will eat the other.

We can represent each state of this system by listing the objects that are on the east bank: w is the wolf, g is the goat, c is the cabbage, and b is the boat. States like wc and wgb are legal, but wg would not be a legal state. The diagram of legal states then looks as follows:

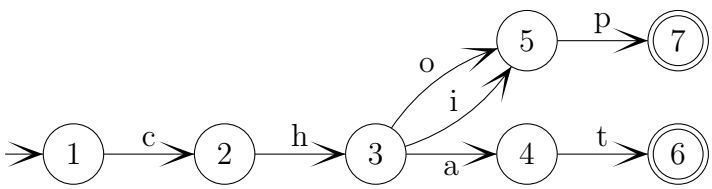


In this diagram, actions aren't marked on the edges: you're left to infer the action from the change in state. The start state (wgcb) where the system begins is marked by showing an arrow leading into it. The end state (∅) where nothing is left on the east bank is marked with a double ring.

In this diagram, it is possible for a (directed) walk to loop back on itself, repeating states. So there are two shortest solutions to this puzzle, but also an infinite number of other solutions that involve undoing and then redoing some piece of work.

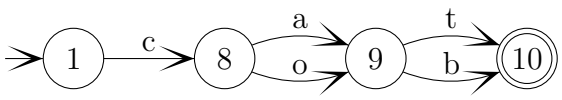
19.3 Phone lattices

Another standard application for state diagrams is in modelling pronunciations of words for speech recognition. In these diagrams, known as *phone lattices*, each edge represents the action of reading a single sound (a *phone*) from the input speech stream. To make our examples easy to read, we'll pretend that English is spelled phonetically, i.e. each letter of English represents exactly one sound/phone.[1] For example, we could model the set of words {`chat`, `chop`, `chip`} using the diagram:



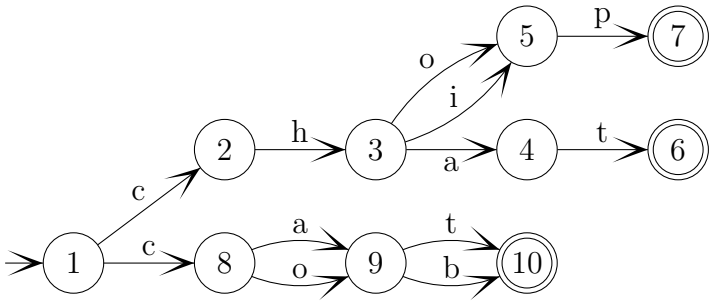
As in the wolf-goat-cabbage puzzle, we have a clearly defined start state (state 1, marked with an incoming arrow). This time there are two end states (6 and 7) marked with double rings. It's conventional that there is only one start state but there may be several end states.

Another phone lattice might represent the set of words {`cat`, `cot`, `cab`, `cob`}.



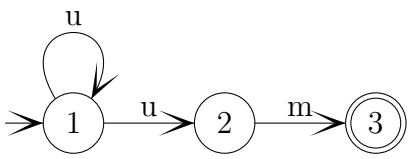
We can combine these two phone lattices into one large diagram, representing the union of these two sets of words:

[1] This is, of course, totally wrong. But the essential ideas stay the same when you switch to a phonetic spelling of English.



Notice that there are two edges leading from state 1, both marked with the phone c. This indicates that the user (e.g. a speech understanding program) has two options for how to handle a c on the input stream. If this was inconvenient for our application, it could be eliminated by merging states 2 and 8.

Occasionally, it's useful for a phone lattice to contain a loop. E.g. people often drag out the pronunciation of the word "um." So the following represents all words of the form `uu*m`, i.e. `um`, `uum`, `uuuuuum`, etc.



Many state diagrams are passive representations of a set of possibilities. For example, a room layout for a computer game merely shows what walks are possible; the player makes the decisions about which walk to take. Phone lattices are often used in a more active manner, as a very simple sort of computer called a *finite automaton*. The automaton reads an input sequence of characters, following the edges in the phone lattice. At the end of the input, it reports whether it has or hasn't successfully reached an end state.

19.4 Representing functions

To understand how state diagrams might be represented mathematically and/or stored in a computer, let's first look in more detail at how functions are represented. A function $f : A \to B$ associates each input value from A with an output value from B. So we can represent f mathematically as a set of input/output pairs (x, y), where x comes from A and y comes from B. For example, one particular function from $\{a, b, c\}$ to $\{1, 2, 3, 4\}$ might be represented as

$$\{(a, 4), (b, 1), (c, 4)\}$$

In a computer, this information might be stored as a linked list of pairs. Each pair might be a short list, or an object or structure. However, finding a pair in such a list requires time proportional to the length of the list. This can be very slow if the list is long i.e. if the domain of the function is large.

Another option is to convert input values to small integers. For example, in C, lowercase integer values (as in the set A above) can be converted to small integers by subtracting 97 (the ASCII code for a). We can then store the function's input/output pairs using an array. Each array position represents an input value. Each array cell contains the corresponding output value.

19.5 Transition functions

Formally, we can define a state diagram to consist of a set of states S, a set of actions A, and a *transition function* δ that maps out the edges of the graph, i.e. shows how actions result in state changes. Each edge of the state diagram shows what happens when you are in a particular state s and execute some action a, i.e. which new state(s) could you be in.

So an input to δ is a pair (s, a) of a state and an action. Each output of δ is a set of states. So the type signature of δ looks like $\delta : S \times A \to \mathbb{P}(S)$.

For many input pairs, δ produces a set containing a single state. So why does δ produce a set of states rather than a single state? This is for two reasons. First, in some state diagrams, it might not be possible to execute

certain actions from certain states, e.g. you can't go up from the study in our room layout example. δ returns $\emptyset$ in such cases. Second, as we saw in the phone lattice example, it is sometimes convenient to allow the user or the computer system several choices. In this case, δ returns a set of possibilities for the new state.

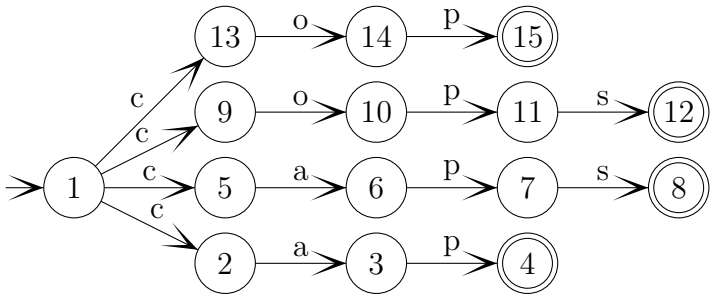
The hard part of implementing state diagrams in a computer program is storing the transition function. We could build a 2D array, whose cells represent all the pairs (s, a). Each cell would then contain a list of output states. This wouldn't be very efficient, however, because state diagrams tend to be *sparse*: most state/action pairs don't produce any new state.

One better approach is to build a 1D array of states. The cell for each state contains a list of actions possible from that state, together with the new states for each action. For example, in our final phone lattice, the entry for state 1 would be $((c, (2, 8)))$ and the entry for state 3 would be $((o, (5)), (i, (5)), (a, (4)))$. This *adjacency list* style of storage is much more compact, because we are no longer wasting space representing the large number of impossible actions.

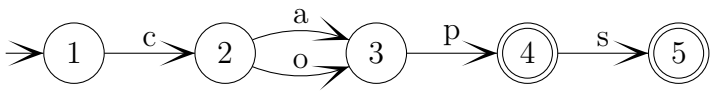
Another approach is to build a function, called a *hash function* that maps each state/action pair to a small integer. We then allocate a 1D array with one position for each state/action pair. Each array cell then contains a list of new states for this state/action pair. The details of hash functions are beyond the scope of this class. However, modern programming languages often include built-in *hash table* or *dictionary* objects that handle the details for you.

19.6 Shared states

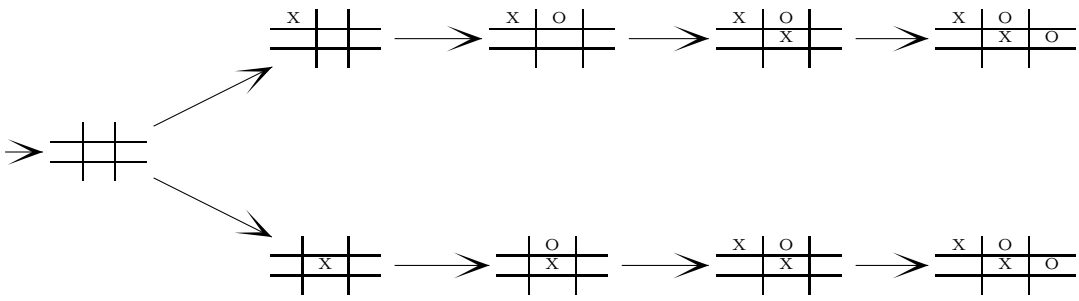
Suppose that each word in our dictionary had its own phone lattice, and we then merge these individual lattices to form lattices representing sets of words. We might get a lattice like the following one, which represents the set of words $\{\texttt{cop}, \texttt{cap}, \texttt{cops}, \texttt{caps}\}$.



Although this lattice encodes the right set of words, it uses a lot more states than necessary. We can represent the same information using the following, much more compact, phone lattice.

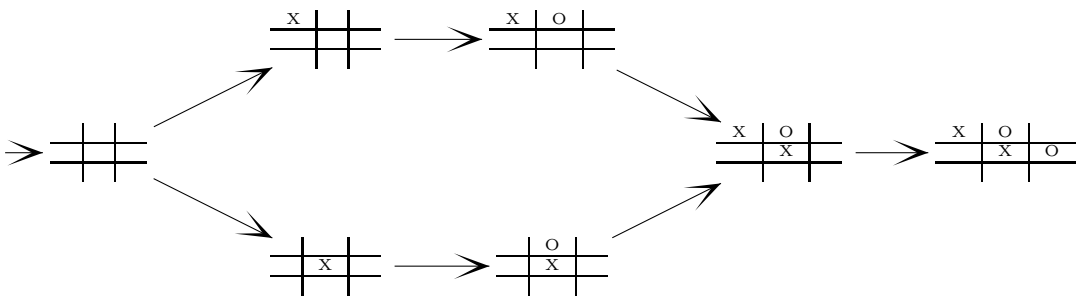


State merger is even more important when states are generated dynamically. For example, suppose that we are trying to find an optimal strategy for playing a game like tic-tac-toe. Blindly enumerating sequences of moves might create state diagrams such as:



Searching through the same sequence of moves twice wastes time as well as space. If we build an index of states we've already generated, we can detect when we get to the same state a second time. We can then use the results of our previous work, rather than repeating it. This trick, called *dynamic*

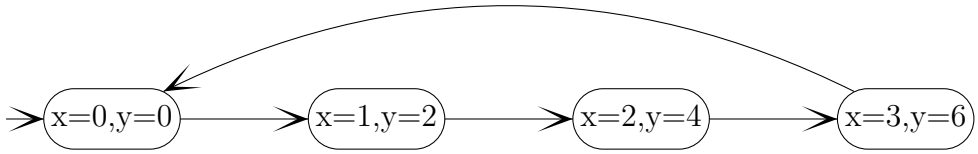
programming, can significantly improve the running time of algorithms.



We can also use state diagrams to model what happens when computer programs run. Consider the following piece of code

```
cyclic()
    y = 0
    x = 0
    while (y < 100)
         x = remainder(x+1,4)
         y = 2x
```

We can represent the state of this machine by giving the values of x and y. We can then draw its state diagram as shown below. By recognizing that we return to the same state after four iterations, we not only keep the diagram compact but also capture the fact that the code goes into an infinite loop.



19.7 Counting states

The size of state diagrams varies dramatically with the application. For example, the wolf-goat-cabbage problem has only $2^4 = 16$ possible states, because we have two choices for the position of each of the four key objects (wolf, goat, cabbage, boat). Of these, six aren't legal because they leave a hungry animal next to its food with the boat on the other side of the river. For this application, we have no trouble constructing the full state diagram.

Many applications, however, generate quite big state diagrams. For example, a current speech understanding system might need to store tens of thousands of words and millions of short phrases. This amount of data can be packed into computer memory only by using high-powered storage and compression tricks.

Eventually, the number of states becomes large enough that it simply isn't possible to store them explicitly. For example, the board game Go is played on a 19 by 19 grid of positions. Each position can be empty, filled with a black stone, or filled with a white stone. So, to a first approximation, we have 3^{361} possible game positions. Some of these are forbidden by the rules of the game, but not enough to make the size of the state space tractable.

Some games and theoretical models of computation use an unbounded amount of memory, so that their state space becomes infinite. For example, Conway's "Game of Life" runs on an 2D grid of cells, in which each cell has 8 neighbors. The game starts with an initial configuration, in which some set of cells is marked as live, and the rest are marked as dead. At each time step, the set of live cells is updated using the rules:

- A live cell stays live if it 2 or 3 neighbors. Otherwise it becomes dead.
- A dead cell with exactly 3 neighbors becomes live.

For some initial configurations, the system goes into a stable state or oscillates between several configurations. However, even if the initial set of live cells is finite, the set of live cells can grow without bound as time moves forwards. So, for some initial configurations, the system has infinitely many states.

When a system has an intractably large number of states, whether finite or infinite, we obviously aren't going to build its state space explicitly. Analysis of such systems requires techniques like computing high-level properties of states, generating states in a smart way, and using heuristics to decide whether a state is likely to lead to a final solution.

19.8 Variation in notation

State diagrams of various sorts, and constructs very similar to state diagrams, are used in a wide range of applications. Therefore, there are many different sets of terminology for equivalent and/or subtly different variations on this idea. In particular, when a state diagram is viewed as an active device, i.e. as a type of machine or computer, it is often called a *state transition* or an *automaton*.

Start states are also known as *initial* states. End states can be called *end* states or *goal* states.

There are two choices for setting up the transition function δ. State diagrams where the transition function returns a set of states are called *non-deterministic*. Those where δ returns a single state are called *deterministic*. Deterministic state diagrams are less flexible but easier to implement efficiently.