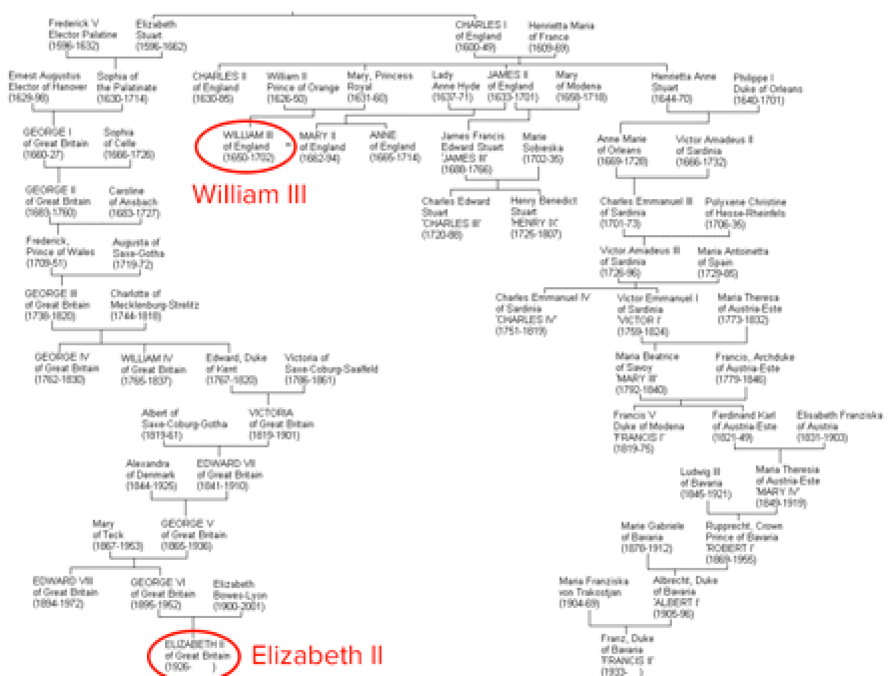
Route-finding

Sometimes, very different-sounding problems turn out to be similar when you think about how to solve them. What do Pac-Man, the royal family of Britain, and driving to Orlando have in common? They all involve route-finding or path-search problems:

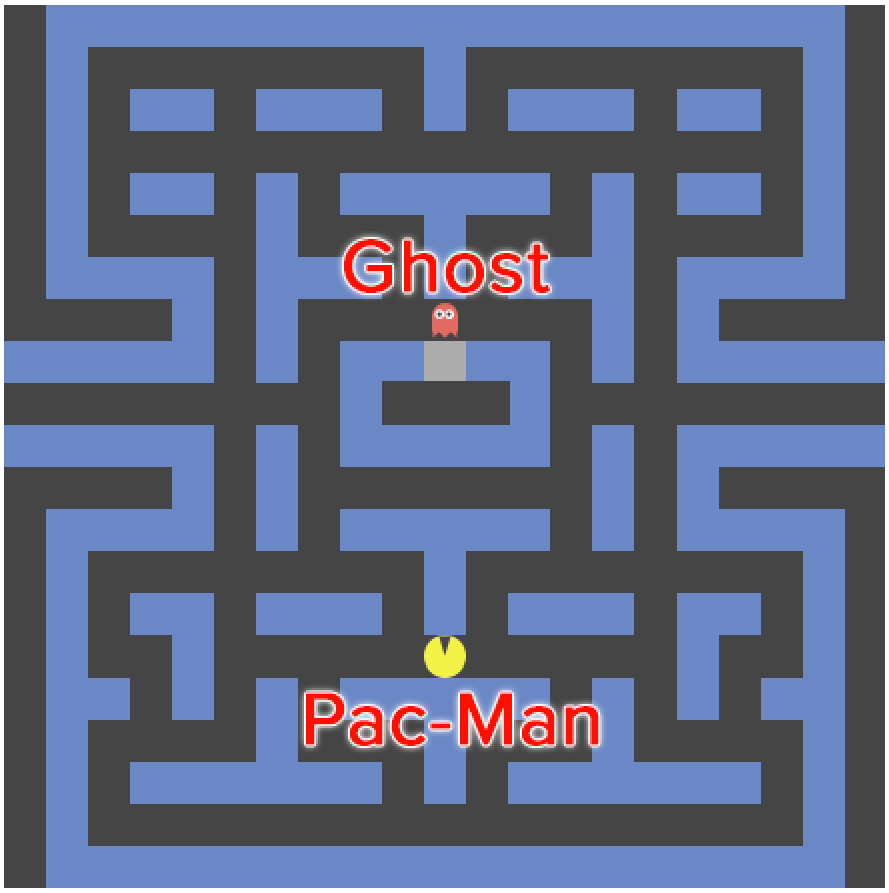
- How is the current Prince William related to King William III, who endowed the College of William and Mary in 1693?
- What path should a ghost follow to get to Pac-Man as quickly as possible?
- What's the best way to drive from Dallas, Texas to Orlando, Florida?

We have to be given some information to answer any of these questions.

For example, a family tree of the royal family of Britain would show connections between people who were directly related. Prince William is the son of Charles Philip Arthur Windsor. Charles is the son of Queen Elizabeth II. The problem is to find a short chain on the family tree connecting Prince William and William III, using these direct connections. As you can see from the tree below, it might take quite a few connections.



For Pac-Man, we need a map of the maze. This map shows connections between adjacent open squares in the maze—or lack of connections, if there is a wall in between—and the problem is to find a path along black squares that leads the ghost to Pac-Man.

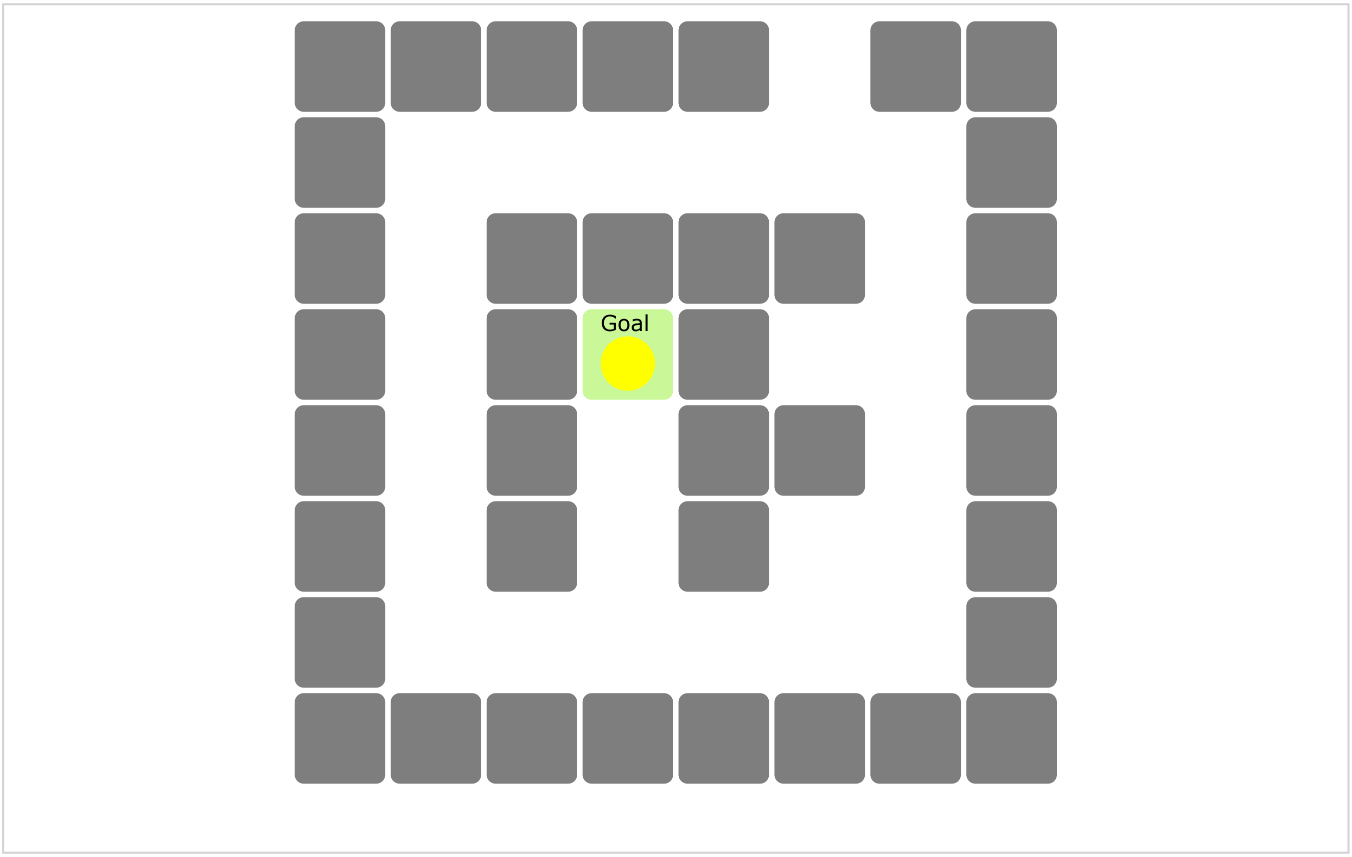


In order to find a driving route from Dallas to Orlando, we might use a map of the United States, showing connections, roads, between nearby cities. No single road directly connects Dallas to Orlando, but several sequences of roads do.

Dallas
Orlando

Exploring a maze

Let's look more deeply at something like Pac-Man, a computer game in which the main character is controlled by clicking on destinations in a maze. The game is below. Try clicking on a few locations to move the character, represented by the yellow circle, to the goal, represented by the green square.



Notice how the character moved to the goal? To make that happen, the program needs to determine the precise set of movements that the character should follow to get to where the user clicked and then animate those movements. There may be multiple paths for the character to follow, and the program needs to choose the best of those paths.

Before deciding on an algorithm, the movement rules first need to be established: walls are made of gray squares and legal locations to travel are empty. In each step, the character can move from one square to an adjacent square. This character, like a chess rook, cannot move diagonally.

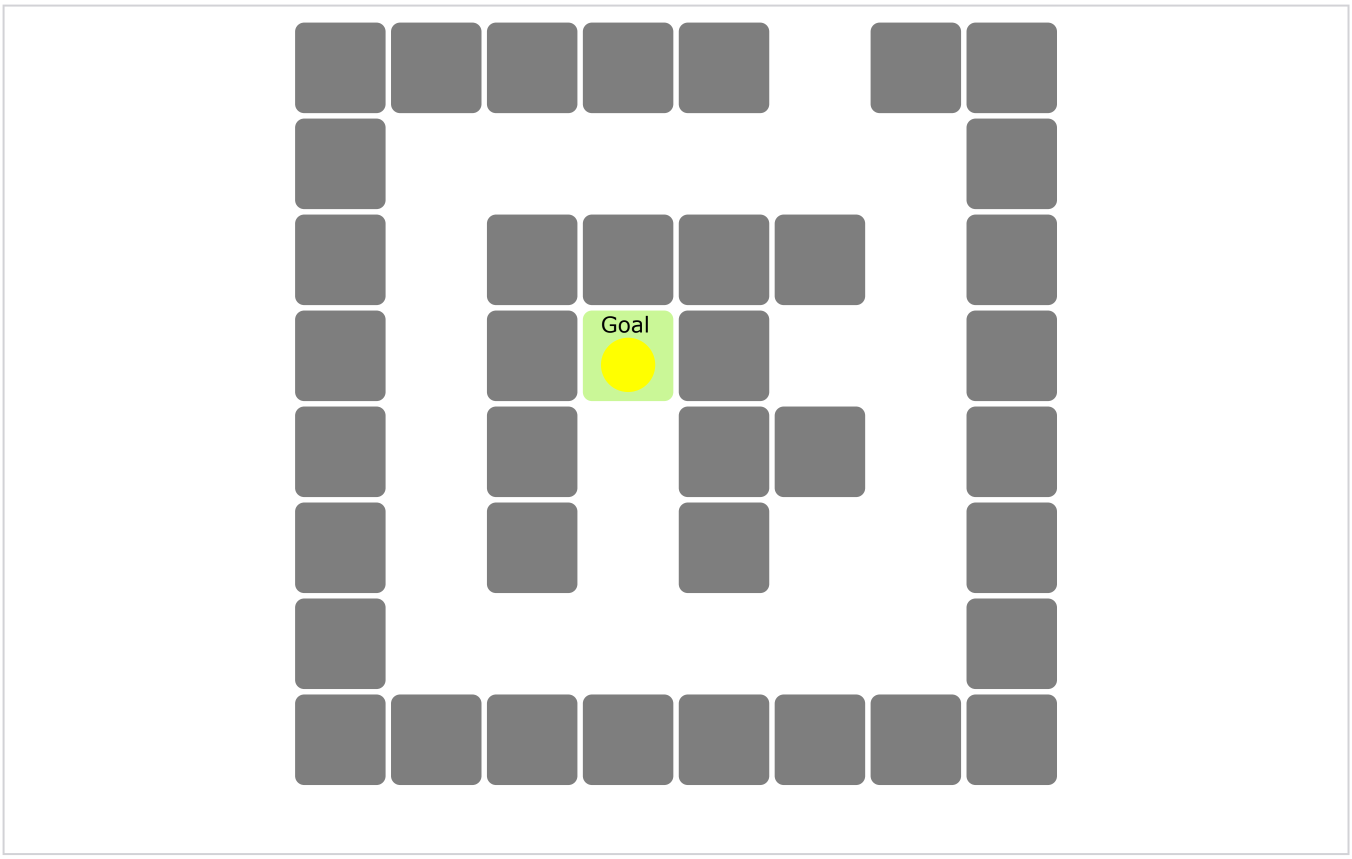
Here's the idea behind the algorithm that this program uses: move 1 square closer to the goal—the place the user clicked on—in each step. But what does "closer to the goal" mean? Traveling in a straight line toward the goal will often cause the character to smack into a wall. The algorithm needs to determine which of the surrounding squares are indeed "closer to the goal", and we can do that by assigning a "cost" to each square that represents the

minimum number of steps the character would have to take to get from that vertex to the goal. Here's an algorithm for assigning a cost to each square:

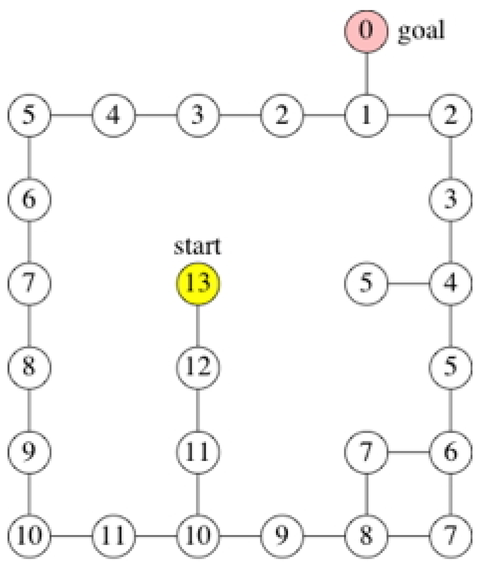
1. Start on the goal square. How far is the goal from the goal? Zero steps, mark the goal with the number 0.
2. Find all squares in the maze that are exactly one step away from the goal. Mark them with the number 1. In this maze, if the goal is the exit square, then there is only one square that is exactly one step away.
3. Now find all squares in the maze that are exactly two steps away from the goal. These squares are one step away from those marked 1 and have not yet been marked. Mark these squares with the number 2.
4. Mark all squares in the maze that are exactly three steps away from the goal. These squares are one step away from those marked 2 and have not yet been marked. Mark these squares with the number 3.
5. Keep marking squares in the maze in order of increasing distance from the goal. After marking squares with the number **k**, mark with the number **k+1** all squares that are one step away from those marked **k** and have not yet been marked.

Eventually, the algorithm marks the square where the character starts. The program can then find a path to the goal by choosing a sequence of squares from the start such that the numbers on the squares always decrease along the path. If you view the number as the height of the square, it would be like going downhill.

You can play through the cost-marking algorithm below.



What if the user were trying to get from the start square to the goal? Using the square-marking algorithm, the start square is 13 steps away from the goal. Here's a picture showing the connections between possible locations for the character, the start, the goal, and the shortest distance of each location from the goal:



There's a square immediately to the south of the start (row four, column three) that is only 12 steps from the goal. So the first move is "south". South of that square is an 11. South again. South again to a 10. Then east to a 9. East twice more to a 7, then north five times to a 2. Finish up by going west once, to a 1, and finally north once, to the goal.

We won't discuss exactly how to implement this maze search algorithm right now, but you might find it fun to think about how you might represent the maze and the character and how you might implement the algorithm (of course in a programming language of your choice).