Heuristic Search

12

This text has focused on the interaction of algorithms with data structures. Many of the algorithms presented in this text deal with search and how to organize data so searching can be done efficiently. Many problems involve searching for an answer among many possible solutions, not all of which are correct. Sometimes, there are so many possibilities, no algorithm can be written that will efficiently find a correct solution amongst all the possible solutions. In these cases, we may be able to use a *rule of thumb*, most often called a *heuristic* in computer science, to eliminate some of these possibilities from our search space. If the *heuristic* does not eliminate possible solutions, it may at least help us order the possible solutions so we look at *better* possible solutions first, whatever *better* might mean.

In Chap. 7 depth first search of a graph was presented. Sometimes search spaces for graphs or other problems grow to such an enormous size, it is impossible to blindly search for a goal node. This is where a heuristic can come in handy. This chapter uses searching a maze, which is really just a type of graph, as an example to illustrate several search algorithms that are related to depth first or breadth first search. Several applications of these search algorithms are also presented or discussed.

Heuristic search is often covered in texts on Artificial Intelligence [3]. As problems in AI are better understood, algorithms arise that become more commonplace over time. The heuristic algorithms presented in this chapter are covered in more detail in an AI text, but as data sizes grow, heuristic search will become more and more necessary in all sorts of applications. AI techniques may be useful in many search problems and so are covered in this chapter to provide an introduction to search algorithms designed to deal with large or infinite search spaces.

12.1 Chapter Goals

By the end of this chapter you will have been presented with examples of depth first and breadth first search. Hill climbing, best first search, and the A* (pronounced A star) algorithm will also be presented. In addition, heuristics will be applied to the search in two person game playing as well.

K.D. Lee and S. Hubbard, *Data Structures and Algorithms with Python*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-13072-9_12

While heuristic search is not the solution to every problem, as data sizes grow, the use of heuristics will become more important. This chapter provides the necessary information to choose between at least some of these techniques to improve performance and solve some interesting large problems that would otherwise be unsolvable in a reasonable amount of time.

12.2 Depth First Search

We first encountered depth first search in Chap. 6 where we discuss search spaces and using depth first search to find a solution to some sudoku puzzles. Then, in Chap. 7 the depth first search algorithm was generalized a bit to handle search spaces that include cycles. To prevent getting *stuck* in a cycle, a visited set was used to avoid looking at vertices that had already been considered. A slightly modified version of the depth first search for graphs is presented in Sect. 12.2.1. In this version the path from the start to the goal is returned if the goal is found. Otherwise, the empty list is returned to indicate the goal was not found.

12.2.1 Iterative Depth First Search of a Graph

```
def graphDFS(G, start, goal):
  # G = (V,E) is the graph with vertices, V, and edges, E.
  V,E = G
  stack = Stack()
  visited = Set()
  stack.push([start]) # The stack is a stack of paths

  while not stack.isEmpty():
    # A path is popped from the stack.
    path = stack.pop()
    current = path[0] # the last vertex in the path.
    if not current in visited:
      # The current vertex is added to the visited set.
      visited.add(current)

      # If the current vertex is the goal vertex, then we discontinue the
      # search reporting that we found the goal.
      if current == goal:
        return path # return path to goal

      # Otherwise, for every adjacent vertex, v, to the current vertex
      # in the graph, v is pushed on the stack of paths yet to search
      # unless v is already in the path in which case the edge
      # leading to v is ignored.
      for v in adjacent(current,E):
        if not v in path:
          stack.push([v]+path)

  # If we get this far, then we did not find the goal.
  return [] # return an empty path
```

The algorithm in Sect. 12.2.1 consists of a while loop that finds a path from a *start* node to a *goal* node. When there is a choice of direction on this path, all choices are pushed onto the stack. By pushing all choices, if a path leads to a dead end, the algorithm just doesn't push anything new onto the stack. The next time through the loop, the next path is popped from the stack, resulting in the algorithm *backtracking* to a point where it last made a decision on the direction it was going.

12.2.2 Maze Representation

How should the maze be represented? Data representation is such an important part of any algorithm. The maze consists of rows and columns. We can think of each location in the maze as a tuple of (row, column). These tuples can be added to a hash set for lookup in O(1) time. By using a hash set we can determine the adjacent (row,column) locations in O(1) time as well for any location within the maze. When a maze is read from a file, the (row, column) pairs can be added to a hash set. The adjacent function then must be given a location and the maze hash set to determine the adjacent locations.

12.2.3 DFS Example

Consider searching the maze in Fig. 12.1. Let's assume that our depth first search algorithm prefers to go up if possible when searching a maze. If it can't go up, then it prefers to go down. Next preference is given to going left in the maze, followed lastly by going right. Assume we start at the top of the maze and want to exit at the bottom. Note that going on the diagonal is not considered in the examples presented in this chapter since otherwise moves where two corners in the maze meet would be

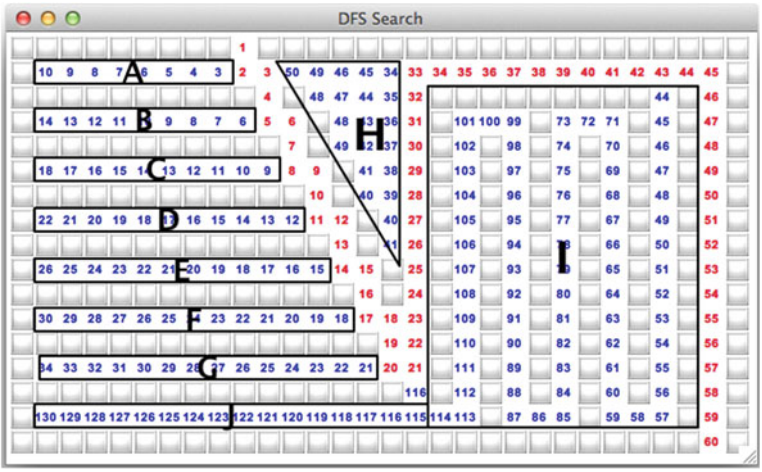

Fig. 12.1 Depth First Search of a Maze

possible. Diagonal moves would have the affect of moving through what looks like walls in the maze in some circumstances.

According to our direction preference, the algorithm proceeds by making steps 1 and 2 in red. Then it proceeds to travel to the left into region A. When it gets to 10 in region A, there are no possible moves adjacent to step 10 that have not already been visited. The code in lines 23–25 cannot find anything to push onto the stack.

However, when step 2 was originally considered, all the other choices were pushed onto the stack including the red three that appears to the right of step 2. When nothing is pushed onto the stack while looking at step 10 in region A, the next top value on the stack is the red step 3. The unvisited nodes adjacent to the red step 3 are then pushed onto the stack. The last location pushed is the red step 4, which leads to the red step 5 being pushed and considered next. Then the depth first search proceeds to the left again, examining all the locations in region B.

When region B is exhausted, backtracking occurs again, resulting in taking the red step 6. This leads to the search entering region D next, exhausting the possibilities on this path and backtracking occurring to take the search to step 12 in red. Likewise, regions E, F, and G are explored. When the search gets to red step 21 the depth first search prefers to go up and proceeds to the top of the maze and enters region H.

We can tell by looking at the maze that entering region H will lead nowhere. But depth first search does not know or care about this. It just blindly considers the next possible path to the goal until that path leads to the goal or we have exhausted all possible next steps and backtrack. Backtracking out of region H leads to step 34 in red. When we reach step 44 the algorithm prefers to go down first and proceeds on a wild goose chase leading from region I to region J where it runs out of possible next steps and backtracks to the red step 44. Finally, that path leads to the goal.

There are some things to notice about this search. First, as mentioned before, it was a blind search that uses backtracking to eventually find the goal. In this example the depth first search examined every location in the maze, but that is not always the case. Depth first search did find a solution, but it wasn't the optimal solution. If the depth first search were programmed to go right first it would have found a solution much faster and found the optimal solution for this maze. Unfortunately of course, that won't work for all mazes.

While the maze search space is finite, what if the *maze* was infinite in size and we went to the left while we should have started going right? The algorithm would blindly proceed going left forever, never finding a solution. The drawbacks of depth first search are as follows.

- Depth first search cannot handle infinite search spaces unless it gets lucky and proceeds down a path that leads to the goal.
- It does not necessarily find an optimal solution.
- Unless we are lucky, the search order in a finite space may lead to exhaustively trying every possible path.

We may be able to do better using either breadth first search or a heuristic search. Read on to see how these algorithms work.

12.3 Breadth First Search

Breadth First Search was first mentioned in Chap. 7. The code for breadth first search differs in a small way from depth first search. Instead of a stack, a queue is used to store the alternative choices. The change to the code is small, but the impact on the performance of the algorithm is quite big.

Depth first search goes down one path until the path leads to the goal or no more steps can be taken. When a path is exhausted and does not end at the goal, backtracking occurs. In contrast, breadth first search explores all paths from the starting location at the same time. This is because each alternative is enqueued onto the queue and then each alternative is dequeued too. This has an effect on how the search proceeds.

12.3.1 BFS Example

Breadth first search takes a step on each path each time through the while loop in Sect. 12.2.1. So, after step 2 in Fig. 12.1 the two step 3's occur next. Then the three step 4's occur. The three step 5's are next. The five step 6's are all done on the next five iterations of the while loop.

You can see that the number of alternatives is growing in this maze. There were 2 step 2's on up to five step 6's. The number of choices at each step is called the *branching factor* of a problem. A *branching* factor of one would mean that there is no choice from one step to the next. A branching factor of two means the problem doubles in size at each step.

Since breadth first search takes a step in each direction at each step, a branching factor of two would be bad. A branching factor of two means the size of the search space grows exponentially (assuming no repeated states). Breadth first search is not a good search in this case unless the goal node is very near the start node.

The breadth first search shown in Fig. 12.2 covers nearly as much of the maze as the blind depth first search did. Only a few locations are left unvisited. The breadth first search found the optimal solution to this maze. In fact, breadth first search will always find the optimal solution if it is given enough time.

Breadth first search also deals well with infinite search spaces. Because breadth first search branches out from the source exploring all possible paths simultaneously, it will never get stuck going down some infinite path forever. It may help to visualize pouring water into the maze. The water will fill the maze from the source and find the shortest way to the goal.

The advantages and disadvantages of breadth first search are as follows.

- Breadth first search can deal with infinite search spaces.
- Breadth first search will always find the optimal goal.
- It may not perform well at all when the problem has too high a branching factor. In fact, it may take millions of years or more to use breadth first search on some problems.

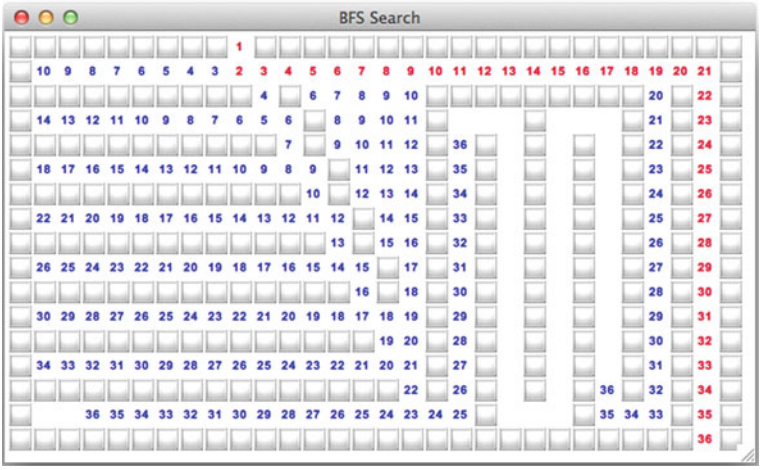

Fig. 12.2 Breadth First Search of a Maze

While it would be nice to be able to find optimal solutions to problems, breadth first search is not really all that practical to use. Most interesting problems have high enough branching factors that breadth first search is impractical.

12.4 Hill Climbing

Depth first search was impractical because it blindly searched for a solution. If the search is truly blind then sometimes we'll get lucky and find a solution quickly while other times we might not find a solution at all depending on the size of the search space, especially when there are infinite branches.

If we had some more information about where the goal is, then we might be able to improve the depth first search algorithm. Think of trying to summit a mountain. We can see the peak of the mountain so we know the general direction we want to take to get there. We want to climb the hill. That's where the name of this algorithm comes from.

Anyone who has climbed mountains knows that sometimes what appears to be a route up the mountain leads to a dead end. Sometimes what appears to be a route to the top only leads to a smaller peak close by. These false peaks are called localized maxima and hill climbing can suffer from finding a localized maximum and thinking that it is the overall goal that was sought.

12.4.1 Hill Climbing Example

Figure 12.3 features the same maze with hill climbing applied to the search. To climb the hill we apply a heuristic to help. In searching a maze, if we know the exit point of

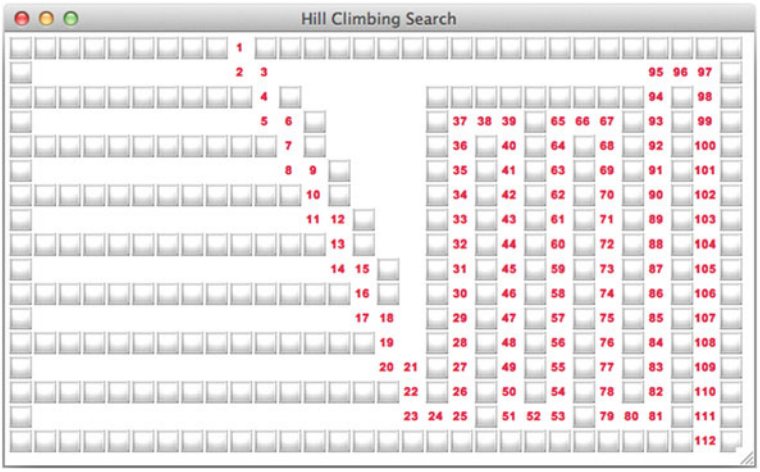

Fig. 12.3 Hill Climbing Search of a Maze

the maze we can employ the *Manhattan distance* as a heuristic to guide us towards the goal. We don't know the length of the path that will lead to the solution since we don't know all the details of the maze, but we can estimate the distance from where we are to the goal if we know the location of the goal and our current location.

The *Manhattan distance* is a measure of the number of rows and columns that separate any two locations on a maze or map. In Fig. 12.3 the Manhattan distance from the start to the goal is 36. We have to go down one row, then right 20 columns, and down 15 rows. This distance is called the Manhattan distance because it would be like walking between buildings in Manhattan or city blocks in any city.

The Manhattan distance would be either exact or an under-estimate of the total distance to the goal. In Fig. 12.3 it is an exact estimate, but in general a direct route to the goal may not be possible in which case the Manhattan distance would be an under-estimate. This is important because over-estimating the distance will mean that hill climbing will end up working like depth first search again. The heuristic would not affect the performance of the algorithm. For instance, if we took the easy approach and said that our distance was always 100 from the goal, hill climbing would not really occur.

The example in Fig. 12.3 shows that the algorithm chooses to go down first if possible. Then it goes right. The goal location is known and the minimum Manhattan distance orders the choices to be explored. Going left or up is not an option unless nothing else is available. So the algorithm proceeds down and to the right until it reaches step 25 where it has no choice on this path but to go up.

Hill climbing performs like depth first search in that it won't give up on a path until it reaches a dead end. While hill climbing does not find the optimal solution in Fig. 12.3, it does find a solution and examines far fewer locations in this case than breadth first or depth first search. The advantages and disadvantages of hill climbing are as follows.

- The location of the goal must be known prior to starting the search.
- You must have a heuristic that can be applied that will either under-estimate or provide an exact length of the path to the goal. The better the heuristic, the better the hill climbing search algorithm.
- Hill climbing can perform well even in large search spaces.
- Hill climbing can handle infinite search branches if the heuristic can avoid them.
- Hill climbing may suffer from local maxima or peaks.
- Hill climbing may not find an optimal solution like breadth first search.

To implement hill climbing the alternative choices at each step are sorted according to the heuristic before they are placed on the stack. Otherwise, the code is exactly the same as that of depth first search.

12.4.2 Closed Knight's Tour

Hill climbing can be used in solving the closed Knight's Tour problem. Solving this problem involves moving a knight from the game of chess around a chess board (or any size board). The knight must be moved two squares followed by one square in the perpendicular direction, forming an L on the chessboard. The closed knight tour problem is to find a path that visits every location on the board through a sequence of legal knight moves that starts and ends in the same location with no square being visited twice except the starting and ending location.

Since we want to find a path through the board, the solution can be represented as a path from start to finish. Each node in the path is a move on the board. A move is valid if it is on the board and is not already in the path. In this way, the board itself never has to be explicitly built.

Generating possible moves for a knight could be rather complex if you try to write code to deal with the edges of the board. In general, when adjacent nodes have to be generated and special cases occur on boundaries, it is far easier to generate a set of possibly invalid moves along with the valid moves. In the case of moving a knight around, there are eight possible moves in the general case. After generating all possible moves, the invalid moves are obvious and can be filtered out. Using this technique, boundary conditions are handled in a uniform manner once instead of with each separate possible move. The code is much cleaner and the logic is much easier to understand.

Figure 12.4 provides a solution to the closed knight's tour problem for a 12×12 board. The tour starts in the lower left corner where two edges were not drawn so you can see where the tour began and ended while it was being computed. The tour took a few minutes to find using a heuristic to sort the choices of next location. A least constrained heuristic was applied to sort the new choices before adding them to the stack. The least constrained next choice was the choice that would have the most choices next. Sorting the next moves in this fashion avoids looking at paths that lead to dead ends by generally staying closer to the edges of the board where the next move has the most choices. In other words, it avoids moving to the middle

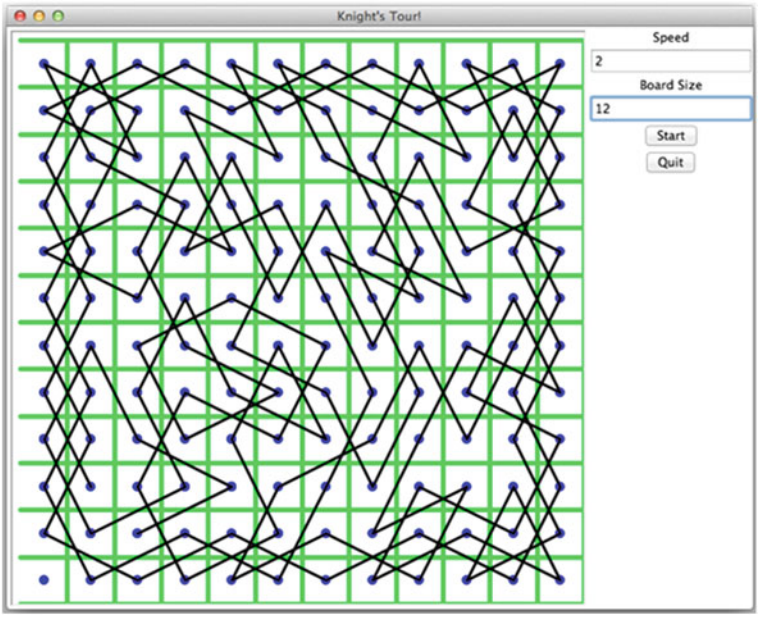

Fig. 12.4 A Closed 12 × 12 Knight's Tour

and getting stuck in the middle of the board. This heuristic is not perfect and some backtracking is still required to find the solution. Nevertheless, without this heuristic there would be no hope in solving the problem in a reasonable amount of time for a 12 × 12 board. In fact, the 8 × 8 solution can't be found in a reasonable amount of time with simple depth first search, unless you get lucky and search in the correct direction at each step. With the heuristic and hill climbing applied, the 8 × 8 solution can be found in just a few seconds.

12.4.3 The N-Queens Problem

To solve the N-Queens problem, N queens must be placed on an $N \times N$ chess board so that no two queens are in the same column, row, or diagonal. Solving this using depth first search would not work. The search space is too large and you would simply have to get very lucky to find a solution using brute force.

The N-Queens problem does have the unique feature that when a queen is placed on the board, all other locations in the row, column, or the diagonals it was placed in are no longer possible candidates for future moves. Removing these possible moves from the list of available locations is called *forward checking*. This forward checking decreases the size of the search space at each step.

The choice of the next row to place a queen is another unique feature of the N-Queens problem. The solution won't be easier to find if a random row is picked or

if we simply pick the next row in the sequence of rows. So the search for the solution is only what column to place the next queen in.

To aide in forward checking, the board can be represented as a tuple: (queen locations, available locations). The first item in the tuple is the list of placed queens. The second item of the tuple is the list of available locations on the board. The forward checking can pick one of the available locations for the next row. At this point all locations in the second part of the tuple that conflict with the choice of the next queen placement can be eliminated. Thus forward checking removes all the possible locations that are no longer viable given a choice of placement for a queen.

The hill climbing part of solving the N-Queens problem comes into play when the choice of which column to place a queen is made. The column chosen is the one that least constrains future choices. Like the Knight's Tour, the N-Queens problem benefits when the next choice made leaves the maximum number of choices later. Using this heuristic, forward checking, and the simple selection of the next row in which to place a queen, it is possible to solve the 25-Queens problem in a reasonable amount of time. One solution is shown in Fig. 12.5.

To review, implementing hill climbing requires the alternative choices at each step be sorted according to the heuristic before they are placed on the stack. Otherwise, the code is exactly the same as that of depth first search. In some cases, like the Knight's Tour and the N-Queens problem, any solution is an optimal solution. But, as noted above when searching a maze, hill climbing does not necessarily find an optimal solution. Wouldn't it be nice if we could combine breadth first search and hill climbing.

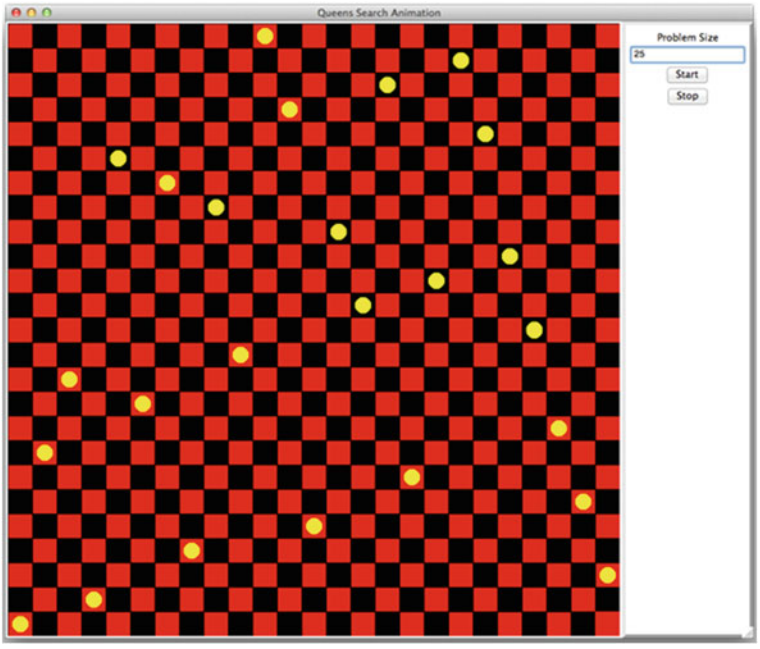

Fig. 12.5 A 25-Queens Solution

12.5 Best First Search

So, breadth first search can find an optimal solution and deal with infinite search spaces, but it is not very efficient and can only be used in some smaller problems. Hill climbing is more efficient, but may not find an optimal solution. Combining the two we get best first search. In best first search we order the entire queue according to the distance of each current node to the goal using the same heuristic as hill climbing.

12.5.1 Best First Example

Consider the example in Fig. 12.6. Step 3 moves closer by moving down one row to step 4. Now, to the right of step 3 is an equally good move (actually better knowing the optimal solution), but the next step is *better* at step 5 than to the right of 3 because it is closer to the eventual goal. So best first proceeds down and to the right like hill climbing until it gets to step 26 in red. At this step it is forced to move up and away from the goal. In this case red step 28 looks just as good as blue step 24 along the bottom. The Manhattan distance of both is 14. When we reach red step 29 then blue step 22 in the middle of the maze looks just as good. The effect of heading away from the goal is to start search all paths simultaneously. That's how best first works. It explores one path while it is moving towards the goal and multiple paths when moving away from the goal.

The code for best first search is a lot like breadth first except that a priority queue is used to sort the possible next steps on the queue according to their estimated distance from the goal. Best first has the advantage of considering multiple paths, like breadth first search, when heading away from the goal while performing like hill climbing when heading toward the goal.

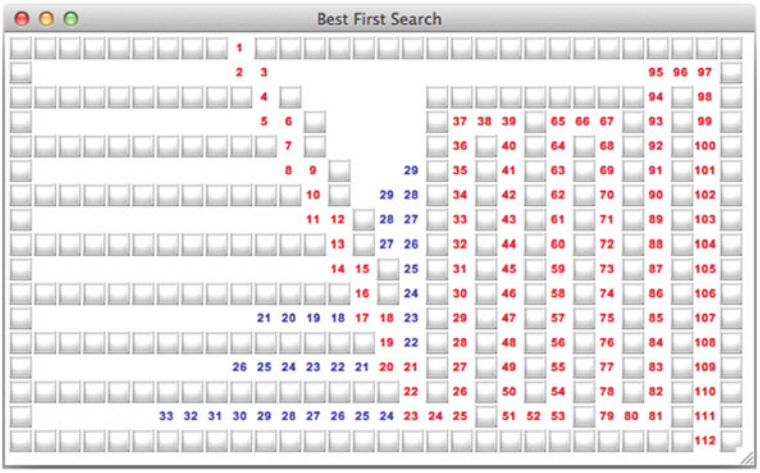

Fig. 12.6 Best First Search of a Maze

In the example shown here we did not do better than hill climbing. Of course, that is only this example. In general hill climbing may do worse than best first. It all depends on the order that locations are searched in the search space. However, neither hill climbing or best first found the optimal solution like breadth first search. They both got stuck heading into the long path in the middle of the maze.

12.6 A* Search

Wouldn't it be nice to be able to give up on some paths if they seem too long? That's the idea behind the A* algorithm. In this search, the next choices are sorted by their estimate of the distance to the goal (the Manhattan distance in our maze examples) and the distance of the path so far. The effect of this is that paths are abandoned (for a while anyway) if they appear to be taking too long to reach the goal.

12.6.1 A* Example

In Fig. 12.7 the same path is first attempted by going down and to the right until step 25 at the bottom of the maze is reached. Then that path is abandoned because the length of the path plus the Manhattan distance at step 4 in red is *better* than taking another step (step 26) at the bottom of the maze. Again the search goes down and to the right eventually filling the same region H from Fig. 12.1. At this point the search continues across the top to step 19 where it again goes down to step 33 at which point step 20 in red looks better than taking step 34 to the left. The search gives up on the blue path at step 33 and then proceeds to the goal from red step 20.

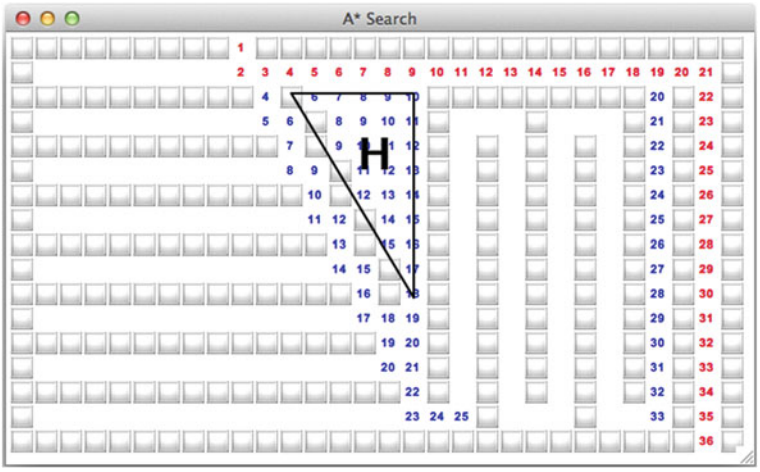

Fig. 12.7 A-Star Search of a Maze

The A* algorithm finds an optimal solution in this example because it gives up on two paths that are getting *too long* according to the heuristic plus total cost so far. Of course the optimality of the solution depends on the heuristic and the total cost. For instance, the heuristic should be good enough to return 0 as the cost of getting from the goal to the goal. The heuristic cannot over-estimate the cost of getting to the goal from the current node.

The A* algorithm was used to solve a problem in the Infinite Mario AI competition. In this competition programmers from around the world were given the task of writing code that would guide Mario through a programmable version of the Nintendo game *Mario Brothers*. The idea was that machine learning techniques would be employed to *teach* Mario to make good decisions while navigating the game's *world*. Instead of using machine learning, Robin Baumgarten solved the problem of getting Mario through the game by implementing the A* algorithm. In Robin's solution Mario makes choices based on the path length so far plus a heuristically computed cost to get to the goal. The A* implementation solved the problem and was a hit on the internet. You can read more about Robin's solution and how he developed it at http://aigamedev.com/open/interviews/mario-ai/.

12.7 Minimax Revisited

In Chap. 4 tictactoe was presented as a demonstration of two dimensional arrays. The outline of the minimax algorithm was presented. The minimax algorithm is used by computer games to provide a computer opponent. Minimax only applies in two person games of what is called *perfect information*. These are games where everyone can see everything. Poker is not a game of perfect information. Tic tac toe is a game of perfect information.

The game of tic tac toe is small enough that adults can generally look far enough ahead so they never lose, unless they make a careless mistake. Children on the other hand sometimes can't get enough! If you don't have children or younger brothers and sisters, someday you will understand. Tic tac toe is also small enough to be solvable by a computer. The minimax algorithm can play the game to its conclusion to insure that it never loses, just like an adult.

The game of connect four is a bit different. In this game, black and red checkers are dropped down slots in a vertically positioned board. The board is seven checkers wide by six checkers tall. Checkers always drop as far down as possible so there are at most seven choices at each turn in the game. The goal is to get four of your checkers in a row, a column, or on a diagonal. In Fig. 12.8 the computer has won with the four black checkers on the diagonal.

Playing connect four is not as easy as playing tic tac toe. With a branching factor of approximately seven at each turn, the number of possible boards quickly grows past what can be considered exhaustively in a reasonable amount of time. In these situations a heuristic must be added to the minimax algorithm to cut off the search. The algorithm is not repeated here. See Sect. 4.9 for a complete description of the

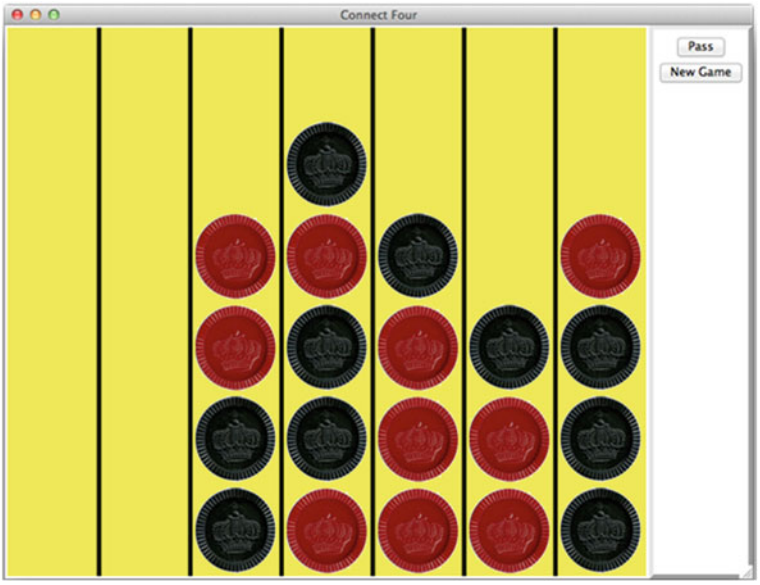

Fig. 12.8 The Connect Four Game

algorithm. The base cases for minimax are then modified as follows to incorporate the search cutoff heuristic.

1. The current board is a win for the computer. In that case minimax returns a 1 for a computer win.
2. The current board is a win for the human. In that case minimax returns a -1 for a human win.
3. The current board is full. In that case, since neither human or computer won, minimax returns a 0.
4. The maximum depth has been reached. Evaluate the board with no more search and report a number between -1.0 and 1.0. A negative value indicates the human is more likely to win given this board. A positive value indicates the computer is more likely to win.

To implement this last base case for the algorithm, a new depth parameter is passed to the minimax algorithm and possibly some other parameters as well. Early in a game the maximum depth may not be very deep. However, later in a game, when less choices are available, the maximum depth may be deeper. Increasing the depth of search is the best way to improve the computer's ability to win at games like this. A good heuristic can also help in earlier stages of the game when deeper search is not possible. Coming up with a good heuristic is a challenge. The trick is to keep it relatively simple to compute while encouraging moves in some fashion on the board.

We have developed a connect four implementation based on these ideas that runs on standard hardware without any special multiprocessing. Our version beats all commercially available apps and applications presently available when playing against them. Your challenge, should you care to take it on, is to build a better one. A front-end for this game is available in Sect. 20.6 or on the text's website so you can build your own connect four game.

12.8 Chapter Summary

This chapter covered heuristics and how they play a part in problems when the search space grows too large for an exhaustive search. Many problems that would otherwise be unsolvable are solvable when heuristics are applied. However, the nature of a heuristic is that sometimes they work well and other times they may not. If a heuristic worked every time, it would be called a technique and not a heuristic.

We must think carefully about whether heuristic search is really required or not when solving problems. Choosing the right problem representation, data structure, or algorithm is much more important than a brute force approach and applying a heuristic. It may be that a problem that seems too big to solve can be reduced to something that can be solved by the right algorithm. When there is no such reduction possible, heuristic search may be the answer.

The search algorithms hill climbing, best first, and A* are best remembered by comparing their algorithms to depth first search and breadth first search. Hill climbing is like depth first search except that a heuristic is applied to order the newly added nodes to the stack. Best first is like breadth first search except that all the nodes on the queue are ordered according to the heuristic. It is often implemented with a priority queue. The A* algorithm is like best first except that the queue is ordered by the sum of the heuristically estimated distance to the goal plus the distance travelled so far.

The minimax algorithm too uses a heuristic when the search space is too large. An effective game engine will always search as deep as possible, but when the search must be cut off, a good heuristic will help in estimating the worth of a move in the game.

12.9 Review Questions

Answer these short answer, multiple choice, and true/false questions to test your mastery of the chapter.

1. Which is faster, depth first search or breadth first search?
2. Which search, depth first or breadth first, may not complete in some situations? When could that happen?
3. When hill climbing, what could prevent the algorithm from finding a goal node?

4. Will the best first search algorithm find an optimal solution? Why or why not?
5. Will the A* algorithm find an optimal solution? Why or why not?
6. When would hill climbing be better to use than the A* algorithm?
7. What is forward checking and how does that help solve a problem?
8. Describe what happens in the depth first search algorithm when backtracking occurs. Be specific about how the algorithm behaves at the point when backtracking occurs in Sect. 12.2.1.
9. Name a game that cannot use the minimax algorithm other than poker.
10. What is the best way to insure that minimax behaves as desired: a really good heuristic or deeper search?

12.10 Programming Problems

1. Write a program that uses the five search algorithms in this chapter to search a maze as shown in the examples. Construct sample mazes by writing a text file where each space represents an open location in the maze and each non-space character represents a wall in the maze. Start the maze with the number of rows and columns of the maze on the first two lines of the file. Assume that you search the maze from top to bottom to find a way through it. There should be only one entry and one exit from your maze. Compare and contrast the different algorithms and their performance on your sample mazes. Be sure to download the maze searching front-end from the text's website so you can visualize your results. The architecture for communication between the front-end and your back-end code is provided in the front-end program file.
2. Write a program to solve the Knight's Tour problem. Be sure to use a heuristic in your search to narrow the search space. Make sure you can solve the tour quickly for an 8×8 board. Draw your solution using turtle graphics.
3. Write a program to solve the N-Queens problem. Use forward checking and a heuristic to solve the N-Queens problem for an 8×8 board. For an extra challenge try to solve it for a 25×25 board. The program will likely need to run for a while (a half hour?) to solve this one. Be sure to use the N-Queens front-end code provided on the text's website to visualize your result. The back-end code you write should follow the architecture presented at the top of the front-end program file.
4. Write the connect four program to challenge another student's connect four. You both must write programs that have a *pass* button. A flip of a coin can determine who goes first. The one who goes first should begin by pressing their pass button. Then you and the other student can flip back and forth while your computer programs compete. To keep things moving, your game must make a move within 30 s or it forfeits. You can use the front-end code presented in Sect. 20.6 as your front-end. You must write the back-end code. Follow the architecture to communicate with the front-end code presented at the top of the front-end program file.
5. For an extra challenge, write the connect four program and beat the program provided by the authors on text website. To run the author's code you must have

Python version 3 and Common Lisp installed. Both the front-end code and the author's back-end code must be in the same directory or folder to run the author's version of the program. You can get the author's front-end and back-end code from the text's website.