Problem Solving:
Backtracking & Heuristic

Paul S. Wang, Sofpower.com

May 13, 2023

In this article we further discuss computational thinking for problem solving. First let's look at the *backtracking paradigm*.

This post is part of our *Computational Thinking* (CT) blog where you can find many other interesting and useful articles.

Eight Queens

The Bavarian chess player Max Bezzel formulated the Eight Queens problem in 1848. The task is to place eight queens on a chess board so that no queen can attack another on the board. As you may know, a queen can attack another piece on the same row, column, or diagonal. And the question is how many solutions are there.

It turns out that there are 12 basic solutions (Figure 1 shows one). Other solutions can be derived from these by board rotations or mirror reflections for a total of 92 distinct solutions.

We know a necessary condition for a solution is that each column and each row must contain one and only one queen. To satisfy this necessary condition, there are 8 possible column positions for row 1, 7 for row 2, and so on, for a total of $8! = 40320$ queen placements. A brute-force way to find solutions is to check all 8! cases for diagonal attacks.

But we can do better than that by using a solution technique called *backtracking*. The idea is to place the queens, one at a time, making sure the next queen is placed in a nonattacking position in relation to previously placed queens. If the procedure finished placing all 8 queens, we have a solution. If it got stuck along the way, we backtrack to the previous queen

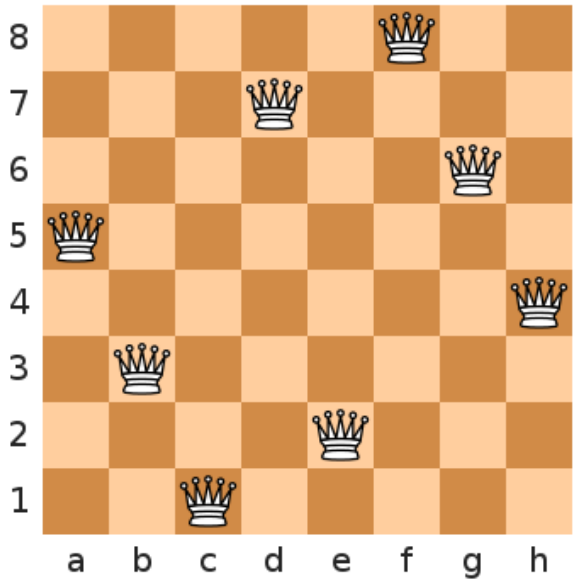

Figure 1: Eight Queens

and move it to its next possible position. If it has no next position, then we backtrack further. Compared to the brute-force method, backtracking examines far fewer cases.

To illustrate backtracking, let's look at a Four Queens problem. We begin by placing the first queen in the first column at board position (1,a) (Figure 2 left). Next we place our second queen in the second column at board position

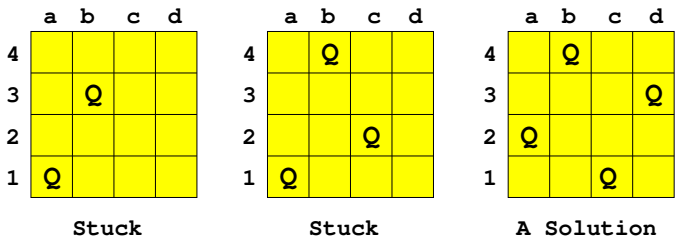

Figure 2: Four Queens Backtracking

(3,b). We then found that there is no place for the third queen in the third column. We are stuck.

So, we go back and move the second queen to the next possible position in the column at board position (4,b). This allows us to place the third queen

in the third column at position (2,c), only to find there is no position for the fourth queen (Figure 2 middle).

Again, we need to go back and change the position of a previous queen. It turns out that we have no next positions for the third or second queen. We are forced to move the first queen to its next possible position (2,a). From this position, proceeding in the same manner as before, we finally arrive at a solution (Figure 2 right). This is clearly faster than the brute-force approach. The efficiency of backtracking comes from abandoning further queen placements after getting stuck.

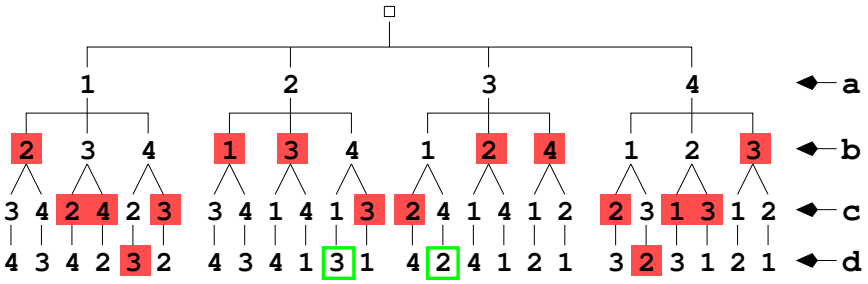

Figure 3: A Solution Tree

To illustrate the savings, let's look at the solution tree for the Four Queens problem (Figure 3), where the first level nodes give the row positions for the first queen (in column `a` on the board), the second level positions for the second queen (column `b`), and so on.

The solid shaded nodes are deadends. The paths leading to the two boxed nodes represent the only two solutions. You can see in Figure 3 how many tree branches are pruned by backtracking, resulting in significant computational savings.

Backtracking Implementation

Let's see how backtracking is applied to solve the Eight Queens problem by showing an algorithm for it. Our algorithm uses the following quantities:

- The size of the board is `N` by `N`; both rows and columns are numbered from 1 to `N`
- The integer array `qn`, with elements `qn[0]` through `qn[N]`

- The quantity `qn[c]` is the row position of the queen in column `c`, where $1 \leq c \leq N$

The backtracking algorithm attempts to position a queen in each successive column and is specified as a recursive function `queens`:

Algorithm `queens(r0, c)`:
Input: Integer `r0` (starting row), `c` (current column)
Effect: queens(1,1) displays all solutions for the `N` queens problem

1. If `(c < 1)`, then return (finished)
2. If `(c > N)`, then (found one solution)
   (a) Display `qn[1]` through `qn[N]` as the solution found
   (b) Call `queens(qn[c-1]+1, c-1)` (for more solutions)
   (c) Return
3.
```
for (r=r0; r <= N; r=r+1)
{  if ( safep(r, c) )
   {  set qn[c] = r;
      queens(1, c+1); (proceeds to next column)
      return;
   }
}
```
4. (Backtrack) Call `queens(qn[c-1]+1, c-1)`

After setting `N=8`, the call `queens(1, 1)` produces all 92 solutions for the Eight Queens problem.

The function terminates (Step 1) if `c` is zero (backtracked to the left of column 1). Otherwise (Step 2), if `c` exceeds `N` (a queen in each column), it displays the solution and continues (Step 2b) to check for more solutions before returning.

When Step 3 is reached, the function tries to place a queen in column `c` at a valid row between `r0` and `N` inclusive. After each successful row placement, it continues to the next column by calling `queens(1, c+1)`. Finally (Step 4), having processed all rows between `r0` and `N`, it backtracks to the previous column for more solutions.

The predicate `safep` checks the validity of position `(r, c)` for placing a queen and returns true or false.

```
safep(r, c)
{   for (y=1; y < c; y=y+1)
    {  if (qn[y] == r || abs(qn[y]-r)==abs(c - y))
       return 0;
    }
    return 1;
}
```

The function makes sure that the position `(r, c)` is not on an occupied row or diagonal by a queen in an earlier column.

General Backtracking

We used the Eight Queens problem to introduce the backtracking technique, which is generally applicable to solve problems by building a solution one element at a time. For a queens problem, we can place one queen at a time until all queens are placed. Other such problems include maze escaping (Figure 4), crossword and Sudoku puzzles.

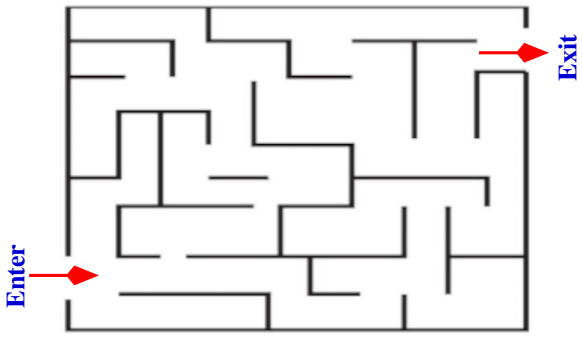

Figure 4: A Maze Puzzle

But backtracking is not just for games. It has wide applicability in solving practical problems, such as the *knapsack problem*, packing items of different weight or size into a container. The goal is to maximize the total dollar value, for example, of the packed items.

Similar to the queens problem, the knapsack problem is a type of *combinatorial optimization* problem where different combinations of items, satisfying given conditions known as *constraints*, are examined to optimize certain desired values. For the queens problem, we have the nonattacking constraint,

and we want to find different ways to place the maximum number of queens on the board. For the knapsack problem, we want to find different ways to pack the given items in order to maximize the value of the packed items under certain measures.

Tree Traversals

Perhaps you already know that files in a computer are organized in a tree structure Figure 5.

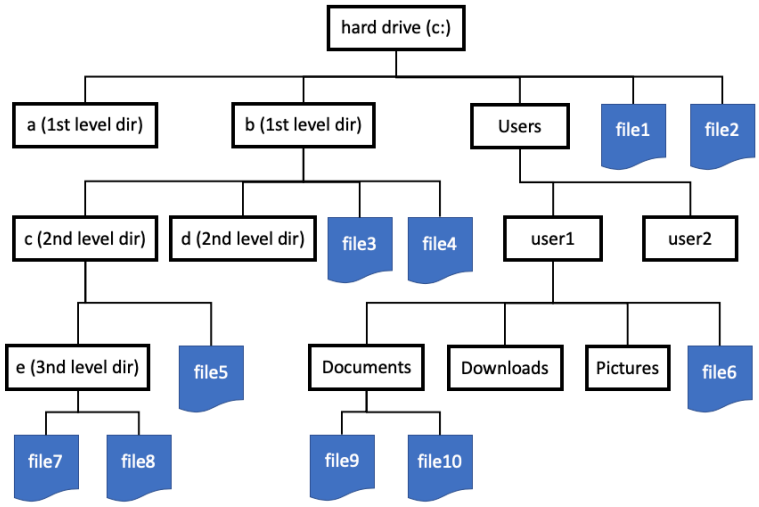

Figure 5: Windows File Tree

In general, a tree structure is a very useful way to organize hierarchical data. Family trees are well-known. Internet domain names are also organized into a tree structure. On a computer, sometimes we need to search for a file because we have forgotten its folder location, or we are not sure of the file's precise name. Or, we want to find all files whose name contains a certain character string. Most operating systems provide a way for users to do such searches. It can be as easy as typing in a substring of the file name and a computer program will look for files with matching names in the entire file tree. This is very convenient, indeed. In fact, you can also find files containing certain words. That can come in handy when you remember parts of the file contents but not the file name.

But, how can such search operations be performed? Well, we need a systematic way to visit each node on the file tree, known as a *tree traversal*. A file tree traversal enables a program to visit all files, following folders and subfolders, and to match file names or contents with user input.

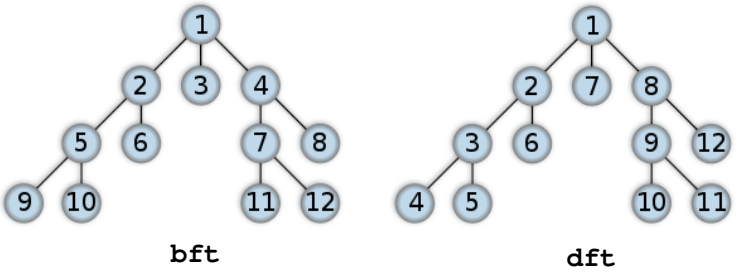

Figure 6: Tree Traversals

The two most common tree traversal algorithms are *depth-first traversal* (dft) and *breadth-first traversal* (bft). With bft, we visit the root, then its child nodes, then grandchild nodes, and so on. With dft, we visit the root, then we **dft** the first child branch, **dft** the 2nd child branch, and so on. Figure 6 shows the node-visit order for a tree using bft and dft.

Implementation of the bft is straightforward. Because dft is defined recursively, we can use a recursive algorithm to implement it.

Algorithm `dft(nd)`:
Input: `nd` the starting node for dft traversal
Effect: visiting every node in the tree rooted at `nd` in dft order

1. Visit `nd`
2. If `nd` is a leaf node (no children), then return
3. Otherwise, for each child node `c` of `nd`, from first child to last child, call `dft(c)`

Trace this algorithm when called on the root of the tree in Figure 6 and verify the dft order given in the figure.

Make use of the tree structures: *Keep the tree structure in mind. It can be found everywhere and can be used to advantage in many situations.*

Let's take a fresh look at the solution tree (Figure 3) for the Four Queens problem we have seen. Now, we see that the backtracking algorithm employed there is simply a dft of the solution tree while applying the nonattacking condition at each node. A solution is found whenever a valid last level leaf is reached.

Tree traversal is important in programming because tree structures are found in many varied situations. For example, markup languages, including HTML (Hypertext Markup Language), organize a documents into a tree structure of the top element containing data and child elements that may in turn contain data and other elements.

Complexity

The speed of modern computers adds a new dimension to problem solving, namely by brute force. We are no longer limited by our own speed or processing capabilities. Instead, we can ask the computer to examine all cases, or explore all possibilities, even when their numbers become quite large.

For example, we can solve the Eight Queens problem by checking each of 8! ways of placing the queens. The backtracking algorithm is a faster way to explore all of the solution tree, which has 8! branches. However, if the number of queens, n, increases much beyond 8, then the brute-force method, even with backtracking, will soon prove to be too slow. This is because the number of possible solutions $n!$ grows big very quickly as n increases.

Weigh Speed vs. Complexity: *Fast computers enable solutions by brute force. But, they are no match for rapidly growing problem complexities.*

To get a feel of how fast $n!$ grows as n increases, let's consider a task of simply running a loop that does nothing for 20! iterations. Assuming a fast computer with a CPU (Central Processing Unit) clock rate of 10 GHz (10^{10} Hz), and each iteration takes just 1 clock cycle, we can compute how long the task will take:

$20! = 2432902008176640000$

$\frac{20!}{10^{10}} = 243290200.817664$ seconds

$\frac{243290200.817664}{24 \times 60 \times 60} = 2815.85880576$ days

That is more than 7.5 years! It will take much, much longer if each loop iteration actually does something.

The term *complexity* is used in computer science in two ways: (i) the inherent difficulty of computational problems, and (ii) the growth of time/space required by an algorithm as its input problem size increases.

Indeed, not all problems or algorithms are created equal. For example, binary search grows proportional to $log_2(n)$ in complexity, where n is the length of the sorted list. Finding the maximum/minimum value in an arbitrary list grows linearly with the list size. The bubble sort algorithm has complexity n^2, while the quicksort algorithm has an average complexity of $n \times log_2(n)$.

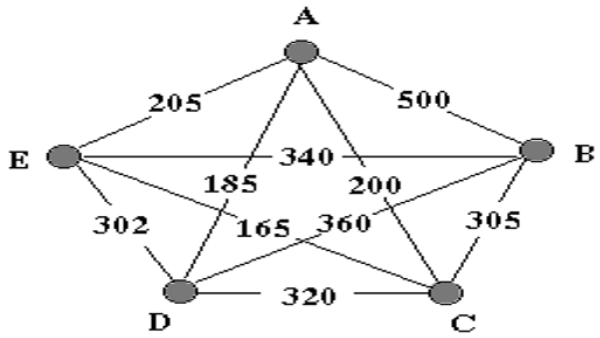

Figure 7: Traveling Salesman Problem

In computer science, the well-known *traveling salesman* problem asks this simple question: Given a set of cities and the distances between each pair of cities, what is the shortest route to visit each city and return to the starting city? Figure 7 shows an instance of this problem involving five cities. Such a problem has proven to be difficult when the number of cities grows. For n cities, there are $(n-1)!$ possible routes to check. This *combinatorial growth* is also seen in the the queens problem.

Heuristics

When faced with a high-complexity problem that quickly outstrips the computational powers of computers, what is a problem solver to do? Well, giving up is the last option. We must be resourceful and try our best to come up with something: a shortcut, an approximation, an oversimplification, or an experience-based rule of thumb. In other words, we will try *heuristics*.

In computer science, a *heuristic* is a technique to solve a problem more quickly or efficiently when brute-force or rigorous algorithmic methods are too expensive (practically impossible), or to get some solution instead of insisting on an exact or optimal one. This is often achieved by trading accuracy, precision, optimality, or completeness for computational feasibility.

In daily life, well-known examples of heuristics include stereotyping and profiling. Let's look at some examples in computing. We already know that the Eight Queens puzzle becomes too big for a slightly larger number of queens. But, we can apply the following heuristic to at least get a solution.

Don't forget heuristics: *Apply heuristics when problem complexity outstrips computational power.*

Here is a *queens heuristic*, where we form a list, *positions*, of row positions for queen placement based on the number of queens:

1. Let $N \geq 4$ be the number of queens. Let $even = (2, 4, 6, ...)$ be the list of even numbers, and $odd = (1, 3, 5, ...)$ be the list of odd numbers, less than or equal to N.

2. If $N \ mod \ 6 = 2$, then swap 1 and 3 in *odd* and move 5 to the end

3. If $N \ mod \ 6 = 3$, then move 2 to the end of *even*, and move 1 and 3 to the end of *odd*

4. *positions* = *even* followed by *odd*

Applying this heuristic to $N = 7$, we get $positions = (2, 4, 6, 1, 3, 5, 7)$. For $N = 8$, we get $positions = (2, 4, 6, 8, 3, 1, 7, 5)$. For $N = 20$, we get

$$positions = (2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 3, 1, 7, 9, 11, 13, 15, 17, 19, 5),$$

and we did not wait for years!

For the traveling salesman problem (TSP), there are quite a few heuristics. A TSP *tour* is a round trip visiting each city once and returning to the origin city. The simplest and most intuitive is the *nearest city* heuristic, which calls for always traveling to the nearest new city from the current city. To find the nearest city at every step, we need to perform a total of

$$(n-1) + (n-2) + ... + 2$$

comparisons. The number is proportional to n^2.

The *greedy* heuristic sorts all, at most $\frac{n(n-1)}{2}$, segments between city pairs and form a sorted list, `L`. It then constructs a tour, usually shorter than the nearest city heuristic, by adding segments, one at a time, to the tour and removing them from `L`:

1. Add the shortest segment from `L` to the tour and remove it from `L`.
2. From `L`, add the shortest **valid** segment to the tour. A segment is invalid if it causes a city to be visited twice unless it's the nth segment.
3. Repeat step 2 until the tour is complete.

Note that segments of the tour may not be connected until construction is complete. Another TSP heuristic constructs a complete tour by forming dis-

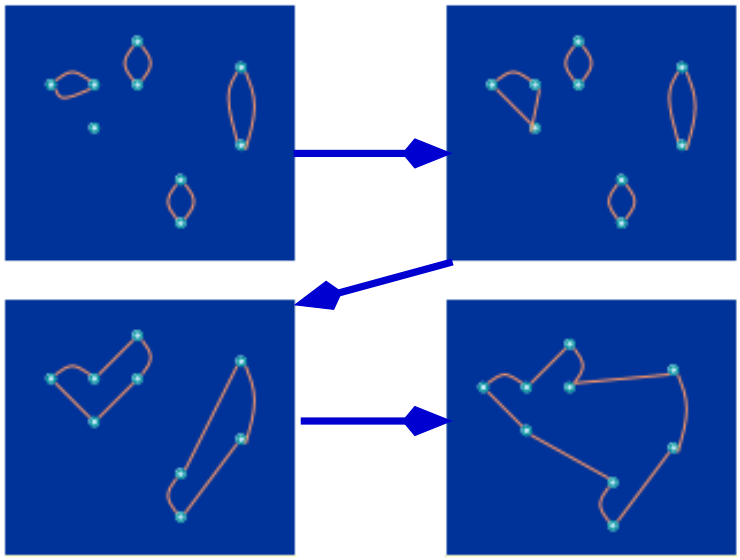

Figure 8: Neighborhood Tour Heuristic

joint neighborhood subtours and merging them together. Figure 8 illustrates the neighborhood tour heuristic for TSP.

A subtour is a tour involving a subset of the cities. The heuristic starts with n subtours, each consisting of an individual city. Then it merges subtours following these rules:

- From all current subtours, pick the two closest subtours and merge them into a larger subtour.
- When merging two subtours, find the best way that minimizes the merged subtour.

Finding good algorithms to automate problem solution is at the center of modern computing. There are many good techniques including, brute-force iteration, chipping away, recursion, top-down divide and conquer, bottom-up building blocks, tree traversal, backtracking, and others. For problems that are too complex, the challenge is to come up with clever heuristics to at least get some results.

Next: Problem Solving in General

In this article we have seen more applications of computational thinking in interesting specific cases. All these details bring us to the question "What are the general principles of computational thinking for problem solving that have wider applicability and usefulness?"

That is exactly the topic of "*Problem Solving: Paradigms & Applications*."