Backtracking

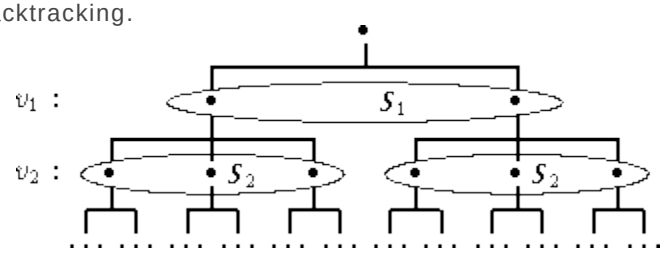
Backtracking is a refinement of the brute force approach, which systematically searches for a solution to a problem among all available options. It does so by assuming that the solutions are represented by vectors (v_1,v_2, ..., v_m) of values and by traversing, in a depth first manner, the domains of the vectors until the solutions are found.

When invoked, the algorithm starts with an empty vector. At each stage it extends the partial vector with a new value. Upon reaching a partial vector (v_1,v_2, ..., v_i) which can't represent a partial solution, the algorithm backtracks by removing the trailing value from the vector, and then proceeds by trying to extend the vector with alternative values.

```
ALGORITHM :
try(v1,v2, ..., vi){
   IF (v1,v2, ..., vi) is a solution
      THEN RETURN (v1,v2, ..., vi)
   FOR each v DO
      IF (v1,v2, ..., vi,v) is acceptable vector
         THEN sol = try(v1,v2, ..., vi,v)
         IF sol ≠ () THEN RETURN sol
      END
   END
   RETURN ()
}
```

If S_i is the domain of v_i , then $S_1 \times ... \times S_m$ is the solution space of the problem. The validity criteria used in checking for acceptable vectors determines what portion of that space needs to be searched, and so it also determines the resources required by the algorithm.

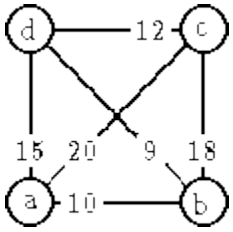
The traversal of the solution space can be represented by a depth-first traversal of a tree. The tree itself is rarely entirely stored by the algorithm in discourse; instead just a path toward a root is stored, to enable the backtracking.



1. Traveling Salesperson

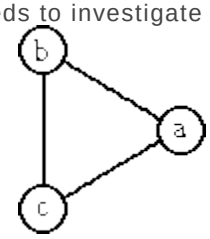
The problem assumes a set of n cities, and a salesperson which needs to visit each city exactly once and return to the base city at the end. The solution should provide a route of minimal length. The route (a, b, d, c) is the shortest

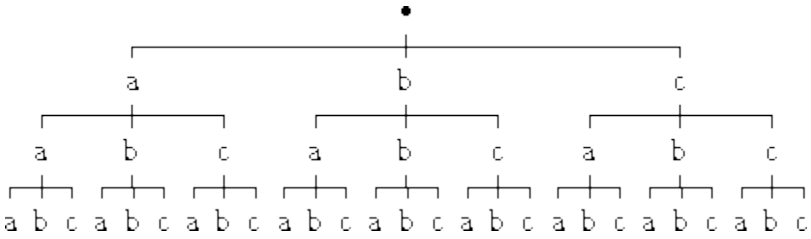
one for the following one, and its length is 51.



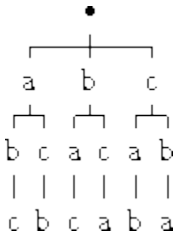
The traveling salesperson problem is an NP-hard problem, and so no polynomial time algorithm is available for it. Given an instance G = (V, E) the backtracking algorithm may search for a vector of cities (v_1, ..., $v_{|V|}$) which represents the best route.

The validity criteria may just check for number of cities in of the routes, pruning out routes longer than |V|. In such a case, the algorithm needs to investigate $|V|^{|V|}$ vectors from the solution space.





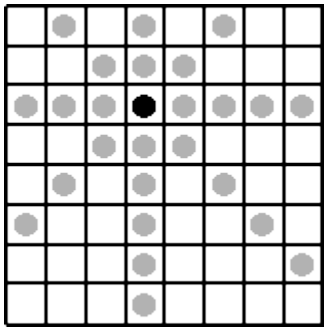
On the other hand, the validity criteria may check for repetition of cities, in which case the number of vectors reduces to |V |!.



2. The N Queens Problem

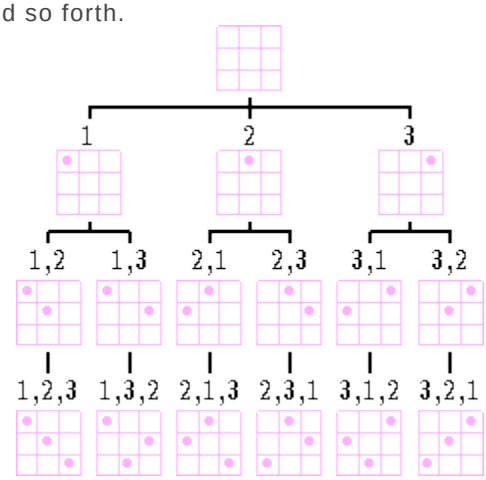
Consider a n by n chess board, and the problem of placing n queens on the board without the queens threatening

one another.

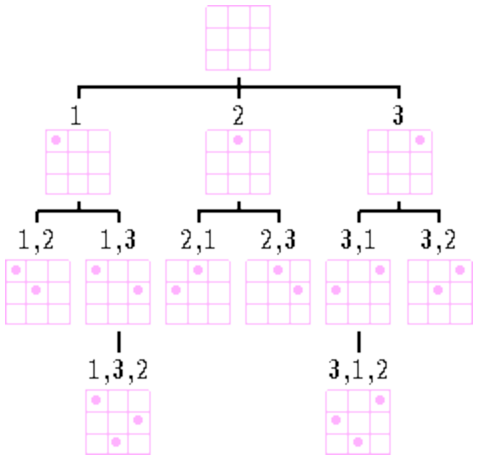


The solution space is $\{1, 2, 3, ..., n\}^n$. The backtracking algorithm may record the columns where the different queens are positioned. Trying all vectors $(p_1, ..., p_n)$ implies n^n cases. Noticing that all the queens must reside in different columns reduces the number of cases to n!.

For the latter case, the root of the traversal tree has degree n, the children have degree n - 1, the grand children degree n - 2, and so forth.

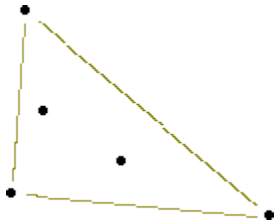


Checking for threatening positions along the way my further reduce the number of visited configurations.

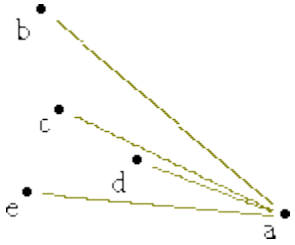


3. Convex Hull (Graham's Scan)

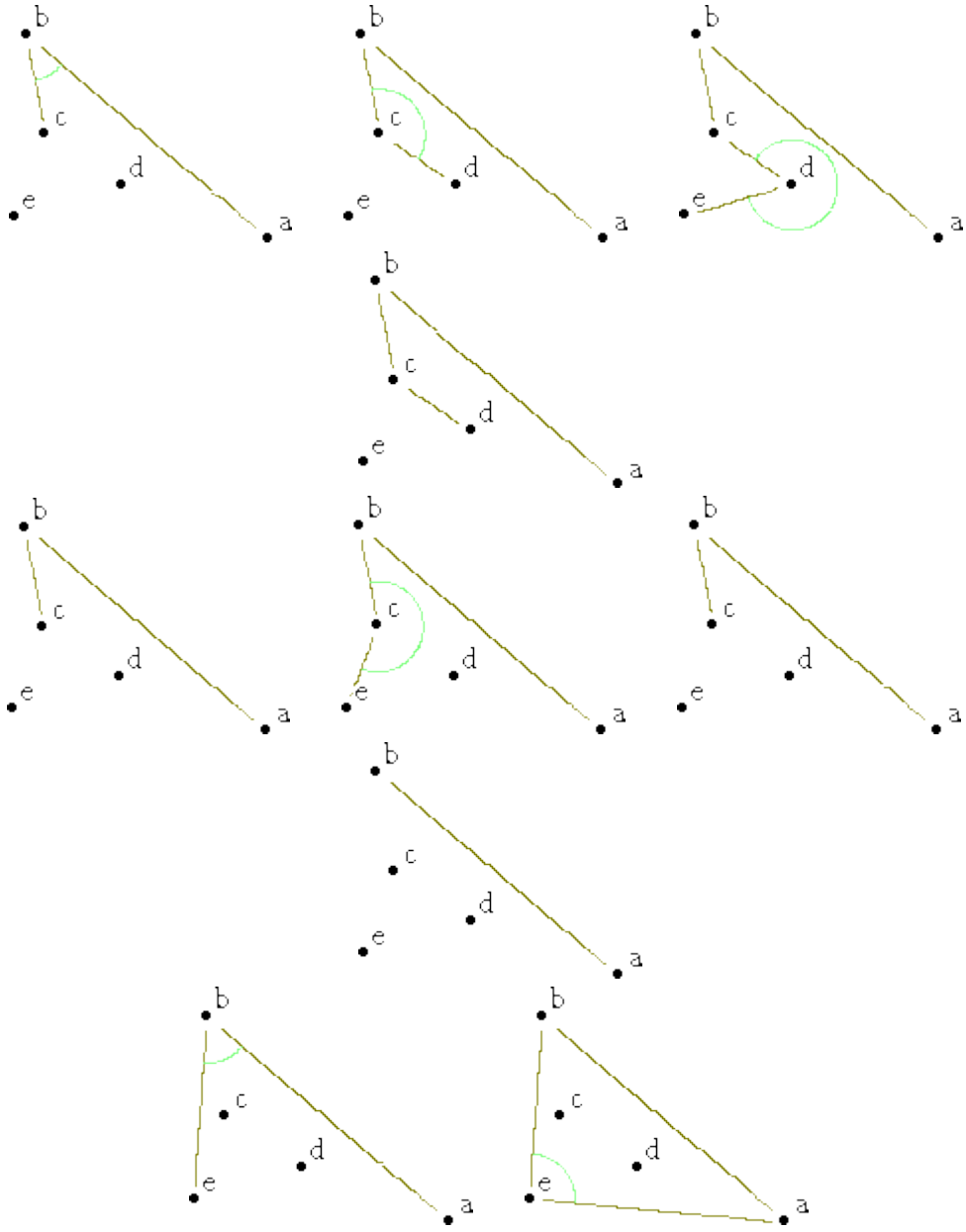
The problem asks to construct the shortest polygon which encloses a given set of points on a plan. Intuitively, the polygon can be determined by placing a rubber around the set.



Determine an extreme point with the largest x coordinate. Sort the points in order of increasing angles, relatively to the extreme point.



Traverse the points in the sorted order, adding them to the partial solution upon making a turn of less than 180 degrees, and backtracking when making a larger turn.

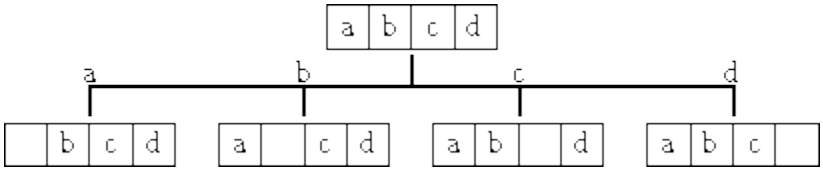


The algorithm requires O(n log n) time for sorting the points, and O(n) time to select an appropriate subset.

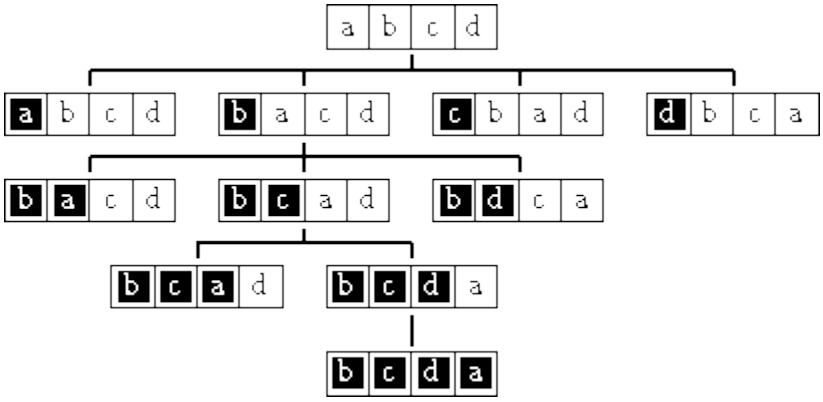
4. Generating Permutations

A permutation can be obtained by selecting an element in the given set and recursively permuting the remaining elements.

$$P(a_1, ..., a_N) = \begin{cases} a_i, P(a_1, ..., a_{i-1}, a_{i+1}, ..., a_N) & \text{if } N > 1 \\ a_N & \text{if } N = 1 \end{cases}$$



At each stage of the permutation process, the given set of elements consists of two parts: a subset of values that already have been processed, and a subset that still needs to be processed. This logical separation can be physically realized by exchanging, in the i^{th} step, the i^{th} value with the value being chosen at that stage. That approaches leaves the first subset in the first i locations of the outcome.




```
permute(i)
{
  if i == N output A[N]
  else
     for j = i to N do
         swap(A[i], A[j])
         permute(i+1)
         swap(A[i], A[j])
}
```

Hence, you can see backtracking works almost similar to Brute force algorithm. It is better than Brute force approach in a way, that it discards a partial solution state, if it discovers at any stage of the search that partial solution can't lead us to a complete solution. This way it reduces the number of unsuccessful searches.

Source:

http://www.learnalgorithms.in/#