Chapter 5

Backtracking

Preview

- Overview
- Lecture Notes
  - The Backtracking Technique
  - The n-Queens Problem
  - Using a Monte Carlo Algorithm to Estimate the Efficiency of a Backtracking Algorithm
  - The Sum-of-Subsets Problem
- Quick Check
  - Graph Coloring
  - The Hamiltonian Circuits Problem
  - The 0-1 Knapsack Problem
    - A Backtracking Algorithm for the 0-1 Knapsack Problem
    - Comparing the Dynamic Programming Algorithm and the Backtracking Algorithm for the 0-1 Knapsack Problem
- Quick Check
- Classroom Discussion
- Homework
- Keywords
- Important Links

Overview

This chapter introduces the concept of Backtracking, which is a very useful algorithm that can be used in many problems. Backtracking is kind of like trying to find your way out of a maze. You try heading in one direction, and if you hit a dead end, you go back to the last intersection and try a different direction. In the worst case, you end up trying every possible passage in the maze. An example of this would be the 0-1 Knapsack Problem (we did a dynamic programming solution). We can do a solution to this problem using backtracking. Recall from Section 4.4.1 that there are 2 *n* subsets, which means that this brute-force method is feasible only for small values of *n*. However, if while generating the subsets we can find signs that tell us that many of them do not need to be generated, we can often avoid unnecessary labor. Backtracking algorithms for problems such as the 0-1 Knapsack problem are still exponential-time in the worst case. They are useful because they are efficient for many large instances.

Lecture Notes

The Backtracking Technique

Backtracking is used to solve problems in which a **sequence** of objects is selected from a specified **set** so that the sequence satisfies some **criterion**. The goal in this problem is to position *n* queens on an *n* x *n* chessboard so that no two queens threaten each other. No two queens can be in the same row, column, or diagonal. The sequence for the problem is the *n* positions in which the queens are placed. The set for each choice is *n* (squared) possible positions on the chessboard, and the criterion is that no two queens can threaten each other. The *n*-Queens problem is a generalization of its instance when n=8, which is the instance using a standard chessboard.

Backtracking is a modified **depth-first search** of a tree. A **preorder** tree traversal is a depth-first search of a tree. This means that the root is visited first, and a visit to a node is followed immediately by visits to all descendants of the node. A depth-first search does not require that the children be visited in any particular order, but we will visit the **children** of a node from left to right.

To return to the *n*-Queens problem when *n* = 4, our task is to position four queens on a 4 x 4 chessboard so that no two queens threaten each other. We already know that no two queens can be in the same row. The instance can be solved by assigning each queen a different row and checking which column combinations yield solutions. In this case there are 4 x 4 x 4 x 4=256 candidate solutions.

We can create the candidate solutions by constructing a tree in which the column choices for the first queen (in row 1) are stored in level-1 nodes in the tree, the column choices for the second queen are in level-2 nodes, and so on. A path from the root to the **leaf** is a candidate solution. This tree is called a **state space tree**. A simple depth-first search of a state space tree is like following every path in a maze until you reach a dead end. It does not take advantage of any signs along the way. We can make the search more efficient by looking for these matches. For example, no two queens can be in the same column. So, there is no point in constructing or checking any paths emanating from the node containing <2.1>. Other nodes that can be eliminated are discussed in the text. We call a node **nonpromising** if we can determine that it cannot possibly lead to a solution. It there is a potential that a solution might be found, we call the node **promising**.

Backtracking consists of doing a depth-first search of a state space tree, checking whether each node is promising, and if it is nonpromising, backtracking to the node's parent. This process is called **pruning**. A subtree consisting of the visited nodes is called the **pruned state space tree.**

The n-Queens Problem

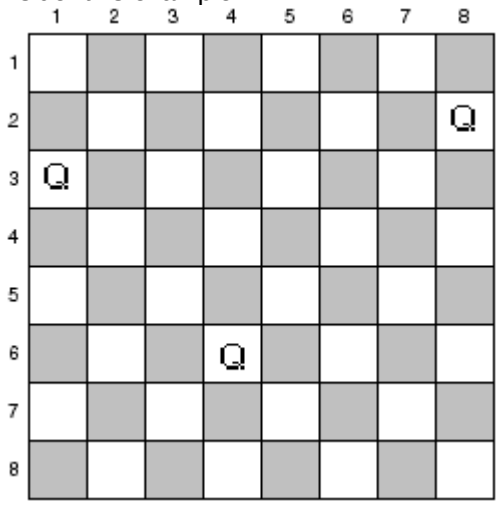
The goal of the *n*-Queens problem is to place *n* queens on an *n* x *n* board so that none of the queens can attack another queen. Remember that queens can move horizontally, vertically, or diagonally any distance. The sequence to be found is the N positions for the queens and the set from which those positions are chosen is the n^2 possible positions on the board, the criterion that the sequence must satisfy is that none of the queens can attack another.

The promising function must check whether two queens are in the same column or diagonal. If w

To solve this problem with backtracking, we have to do several things. *First*, we must visualize a tree model for our solution space. We can simplify our tree by recognizing that no two queens can be on the same row. So the children of the root will show the position of the queen on row 1. For an *n* x *n* size board, there will be *n* children of the root. *Second*, we must come up with ways of eliminating branches before we actually traverse them. In the *n*-Queens problem, one way of eliminating a branch is if it would put two queens in the same column. Another way of eliminating branches is to recognize that no two queens can be on the same diagonal. So if we choose [1, 1] for the queen in row 1, then we do not need to evaluate [2, 2] for the queen in row 2. Using these two rules, we can eliminate quite a bit of the solution space. When we identify a node as nonpromising, we go back to the parent and try another branch. Eventually we will get to one or more leaf nodes and have several candidate sequences to evaluate. In the *n*-Queens problem, if we get to any leaf node, we have our solution.

Let *col (i)* be the column where the queen in the ith row is located, then to check whether the queen in the kth row is in the same column, we need to check whether *col (i) = col (k)*

Now to check the diagonals, consider this example:



In this example, the queen in row 6 is being threatened in its left diagonal by the queen in row 3, and in its right diagonal by the queen in row2. Notice that

col (6) - col (3) = 4 -1 = 3 = 6- 3.

That is, for the queen threatening from the left, the difference in the columns is the same as the difference in the rows. Furthermore,

col (6) - col (2) = 4 -8 = -4 = 2 - 6.

That is, for the queen threatening from the right, the difference in the columns is the negative of the difference in the rows.

Therefore, if the queen in the kth row threatens the queen in the ith row along one of its diagonals, then

col (*i*) - col (*k*) = *i* - *k* or col (*i*) - col (*k*) = *k* - *i*.

When an algorithm consists of more than one routine, we do not order the routines according to the rules of any particular programming language. We just present the main routine first. Algorithm 5.1 produces all solutions to the *n*-Queens problem because that is how we stated the problem. This makes the algorithm less cluttered. Actually running an algorithm to determine its efficiency is not really an analysis. The purpose of an analysis is to determine ahead of time whether an algorithm is efficient.

The time spent in the promising function is a consideration in any backtracking algorithm. A very time-consuming promising function could offset the advantage of checking fewer nodes. In the case of Algorithm 5.1, the promising function can be improved by keeping track of the sets of columns, left diagonals, and right diagonals of the queens already place. In this way, it is not necessary to check whether the queens already positioned threaten the current queen. We need only check if the current queen is being placed in a controlled column or diagonal.

Using a Monte Carlo Algorithm to Estimate the Efficiency of a Backtracking Algorithm

Monte Carlo algorithms are **probabilistic** algorithms, which are algorithms in which the next instruction executed is sometimes determined at random according to some probability distribution. A **deterministic algorithm** is one in which this cannot happen. **A Monte Carlo algorithm** estimates the expected value of a random variable, defined on a sample space, from its average value on a random sample of the sample space. There is no guarantee that the estimate is close to the true expected value, but the probability that it is close increases as the time available to the algorithm increases.

We can use a Monte Carlo algorithm to estimate the efficiency of a backtracking algorithm. First, we generate a **typical path**, which consists of nodes that would be checked given that instance, and then estimate the number of nodes in this tree from the path. The estimate is of the total number of nodes that would be checked to find all solutions.

These conditions should be satisfied so that we can apply the above technique:

- The same promising function must be used on all nodes at the same level in the state space tree.
- Nodes at the same level in the state space tree must have the same number of children.

The Monte Carlo technique requires that we randomly generate a promising child of a node according to the uniform distribution. By this we mean that a random process is used to generate the promising child. When implementing the technique on the computer, we can generate only a pseudorandom promising child. See the text for details about the technique. The process continues until no promising children are found.

When a Monte Carlo algorithm is used, the estimate should be run more than once, and the average of the results should be used as the actual estimate. Twenty trials are normally sufficient. Although the probability of obtaining a good estimate is high when the Monte Carlo algorithm is run many times, but there is never a guarantee that it is a good estimate. The estimate produced for any one application of the Monte Carlo technique is for one particular instance. The estimate obtained using a Monte Carlo estimate is not necessarily a good indication of how many nodes will be checked before the first solution is found. To obtain only one solution, the algorithm may check a small fraction of the nodes it would check to find all solutions.

The Sum-of-Subsets Problem

Recall the thief and the 0-1 Knapsack problem from section 4.4.1. In this problem, there is a set of items the thief can steal, and each item has its own weight and profit. The knapsack will break of the total weight of the items in it exceeds *W*. The goal is to maximize the total value of the stolen items while not making the total weight exceed *W*. An optimal solution for the thief is a set of items that maximizes the total weight subject to the constraint that the total weight must not exceed *W*. The thief might first try to

determine whether there was a ser whose total weight equaled *W*. This is called the Sum-of-Subsets problem.

For large values of *n*, a systematic approach is necessary. One approach is to create a state space tree.

If we sort the weights in nondecreasing order before doing the search, there is an obvious sign that a node is nonpromising. For more discussion of the solution, refer to the text.

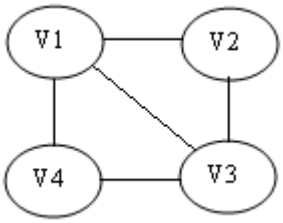
Quick Check

1. Preorder traversal is one way to perform a depth-first search on a tree. (True or False)
   Answer: True
2. In the n-queen problem the idea that no two queens can attack each other is called, the ____.
   Answer: criterion
3. After visiting a node, if we determined that it can possibly lead to a solution. The node is called ____.
   Answer: promising
4. A Monte Carlo algorithm is a(n) ____ algorithm.
   Answer: probabilistic
5. In the sum-of-subsets problem when W = weight we get a solution, and we can get another by including more items. (True or False)
   Answer: False

Graph Coloring

In the m-Coloring problem we are concerned in finding all ways to color an undirected graph using at most m different colors, so that no two adjacent vertices are the same color. For example consider the following graph:

In this graph we have no solution to the 2-coloring problem, because there is no way to color the vertices so that no adjacent vertices are the same color:



In case of the 3-coloring problem, we may have 6 different solutions that differ in the way the colors are permuted.

An important application of graph coloring is the coloring of maps. A graph is called **planar** if it can be drawn in a plane in such a way that no two edges cross each other. To every map there corresponds a planar graph. Each region of the map is represented by a vertex. If one region is adjacent to another region, we join their corresponding vertices by an edge. The *m*-Coloring problem for planar graphs is to determine how many ways the map can be colored, so that no two adjacent regions are the same color.

A straightforward state space tree for the *m*-Coloring problem is one in which every possible color is tried for vertex *v* 1 at level 1, every possible color is tried for *v* 2 at level 2, and so on. Each path from the root to the leaf is a candidate solution. We check whether a candidate solution is a solution by determining whether any two adjacent vertices are the same color.

We can backtrack in this problem because a node is nonpromising if a vertex that is adjacent to the vertex being colored at the node has already been colored the color that is being used at that node.

For small values of m (colors) and a small number of nodes, we can directly solve the problem. However, as m and the number of nodes increases, the number of possible solutions will become so large!

The Hamiltonian Circuits Problem

Remember Example 3.12, in which Ralph and Nancy are competing for the same sales position. The person who covered all 20 cities in the sales territory first wins the job. Nancy found the shortest **tour** in 45 seconds. Ralph's algorithm would take him 3800 years. So, we know Nancy won the job. Now suppose her boss doubled her territory, giving her 40 cities to cover. In her new territory, not every city is connected to another city by a road. Nancy's dynamic programming algorithm took 1 microsecond to process the basic operation. For 40 cities, this algorithm would take 6.46 years to process. Nancy needs to find a better way.

A **Hamiltonian Circuit** (tour) is a path that starts at a given vertex, visits each vertex in the graph exactly once, and ends at the starting vertex. Some connected graphs do not contain Hamiltonian circuits and others do. The Hamiltonian circuit problem determines if a particular connected graph has Hamiltonian circuits.

A state space tree for this problem is as follows. Put the starting vertex at level 0 in the tree; call this the zeroth vertex on the path. At level 1, create a child node for the root node for each remaining vertex that is adjacent to the first vertex. At each node in level 2, create a child node for each of the adjacent vertices that are not in the path from the root to this vertex, and so on.
In order to backtrack in this state space tree:

- The *i*th vertex on the path must be adjacent to the (*i* - 1)st vertex on the path.
- The (*n* - 1)st vertex must be adjacent to the 0th vertex (the starting point).
- The *i*th vertex cannot be one of the first *i* - 1 vertices.

A more detailed discussion of the algorithm can be found in the text. In Nancy's dilemma, it is possible that the backtracking algorithm (for the Hamiltonian Circuits problem) will take even longer than the dynamic programming algorithm. Nancy can use the Monte Carlo technique to estimate the efficiency of her instance. The Monte Carlo algorithm estimates the time to find all the circuits. Because Nancy needs only one circuit, she can have the algorithm stop when the first circuit is found.

The 0-1 Knapsack Problem

In the previous chapter, we solved the 0-1 Knapsack Problem using dynamic programming, now we will solve it using backtracking. Then we will compare both solutions.

A Backtracking Algorithm for the 0-1 Knapsack Problem

In this problem, the thief wants to choose a sequence of items such that the profit is maximized and the weight of the items is not above *W*. We're choosing a sequence according to a constraint, so backtracking can be used.

As we did in Sum of Subsets problem, we will build a tree for this problem. It is a tree where choosing the left child of a node means that you include an item, and choosing the right child means that you do exclude an item. We then go to node at level 1 continue same way. This problem is different in that it is an optimization problem. This means that we do not know if a node contains a solution until the search is over. We have to find all solutions, but we also have to keep track of which one is best. For example, the maximum profit might come from items whose total weight is less than *W*.

We won't know that was the best solution until we eliminate the other possibilities. Therefore, we backtrack a little differently from the previous ways. If the items included in the node have a greater total profit than the best solution so far, we can change the value of the best solution so far. We may still find a better solution at one of the node's descendants (by stealing more items). For optimization problems, we always visit a promising node's children. An example of a general algorithm for backtracking in optimization problems is in the text.

Now we need to decide how to eliminate branches. One obvious way is to use the weight constraint which says that the weight of items must not exceed *W*. So we know that if the weight of items chosen is greater than *W*, we can eliminate that branch. Further, if the weight of items chosen so far is equal to *W*, we know that we do not have to check that node's children because adding any weight would cause it to exceed *W*. So a node is nonpromising if the total weight thus far is greater than or equal to W.

We can use considerations from the greedy approach to find a less obvious sign. This approach failed to give an optimal solution in Section 4.5. Here we use greedy algorithms to limit our search. The profit realized in the Fractional Knapsack Problem will always be equal to or greater than the profit in the 0-1 Knapsack Problem. So, if we are at a node with a total profit so far, and we calculate the amount of profit we could get by greedily choosing from the remaining items, we end up with some value that represents an upper bound on profit if we count this node as part of the solution. If that upper bound is less than the best solution we have picked so far, we do not need to go down that branch. None of the solutions in that branch will be better than the solution we have already found.

Comparing the Dynamic Programming Algorithm and the Backtracking Algorithm for the 0-1 Knapsack Problem

Backtracking algorithms depend on the data they are given. For example, with one set of data, the 0-1 Knapsack Problem might have to complete a depth-first search of the tree without eliminating any branches. With another set of data, the 0-1 Knapsack Problem might eliminate most of the branches. This is because the decision to eliminate branches depends on the data you are processing.

In backtracking algorithms, the worst case gives little insight into how much checking is usually saved by backtracking. With so many considerations, it is difficult to analyze theoretically the relative efficiencies of the two algorithms. In these cases, the algorithms can be compared by running them on many sample instances and seeing which algorithm usually performs better.

Quick Check

1. A graph which can be drawn in a plane in such a way that no two edges cross each other is called, ____.
   Answer: Planner
2. A Hamiltonian Circuit can be also called a tour. (True or False)
   Answer: True
3. A problem where we can not determine if a node contains a solution until the search is over is called a(n) ____ problem.
   Answer: optimization
4. In the 0-1 Knapsack Problem, a node is nonpromising if the total weight thus far is greater than or equal to W. (True or False)
   Answer: True
5. Backtracking algorithm don't depend on the set of data they are given. (True or False)
   Answer: True

Classroom Discussion

- Can you think of some applications of backtracking? If yes discuss one.
- Discuss some of the practical applications that could be represented in terms of the m-coloring problem

Homework

Assign Exercises 11, 16, 23, and 30.

Keywords

- **Backtracking –** a procedure whereby after determining that a node can lead to nothing but dead ends, we go back.
- **Checknode –** to check whether the node is promising after passing it to the procedure.
- **Children –** nodes of trees referred to by their parent node (s).
- **Criterion –** an established rule that a certain sequence follows.
- **Depth first search –** a preorder traversal, where the root of the tree is visited first.
- **Deterministic algorithm –** an algorithm where there is no random execution of instructions
- **Leaf –** a node with no children in a tree.
- **Hamiltonian Circuit –** a path that starts at a given vertex, visits each vertex in the graph exactly once, and ends at the starting vertex.
- **Implicitly –** a state space tree exists *implicitly* in the algorithm if it is not actually constructed.
- **Leaf –** a node with no children.
- **Monte Carlo algorithm –** an algorithm that estimates the expected value of a random variable, defined on a sample space, from its average value on a random sample of the sample space.
- **Nonpromising node –** if when visiting the node we determine that it cannot possibly lead to a solution.
- **Planar graph –** A graph that can be drawn in a plane in such a way that no two edges cross each other.
- **Preorder –** a kind of tree traversal.
- **Probabilistic algorithm –** an algorithm in which the next instruction executed is sometimes determined at random according to some probability distribution.
- **Promising function –** the function *promising* in each application of backtracking.
- **Promising node –** if when visiting the node we determine that it will lead to a solution.
- **Pruned state space tree –** sub-tree consisting of the visited nodes.
- **Pruning state space tree –** doing a depth first search of a state space tree, checking whether. each node is promising, and, if it is nonpromising, backtracking to the node's parent.
- **Sequence –** a group of things arranged in an order.
- **Set –** a group of naturally connected things.
- **State space tree –** a tree in which the column choices for the first queen are stored in level-1 nodes in the tree, the column choices for the second queen are stored in level-2 nodes, and so on(in the *n*-Queens problem).
- **Tour –** another term for Hamiltonian Circuit.
- **Typical path –** nodes that would be checked in a particular instance.
- **Weight –** sum of the weights that have been included up to a node at level i (in sum of subsets problem).

Important Links

- http://cgm.cs.mcgill.ca/~godfried/teaching/algorithms-web.html (Web data structures and algorithms)
- http://algo.inria.fr/AofA/ (Analysis of Algorithms)