Chapter 6

Branch-and-Bound

Preview

- Overview
- Lecture Notes
  - Illustrating Branch-and-Bound with the 0-1 Knapsack problem
    - Breadth-First Search with Branch-and-Bound Pruning
    - Best-First Search with Branch-and-Bound Pruning
  - The Traveling Salesperson Problem
  - Abductive Interference (Diagnosis)
- Quick Check
- Classroom Discussion
- Homework
- Keywords
- Important Links

Overview

We have given the thief in our 0-1 Knapsack problem two algorithms to work with so far. The dynamic programming and backtracking algorithms are both exponential-time algorithms. In the worst case, both could take years to run. The **branch-and-bound** algorithm is an improvement on the backtracking algorithm.

The branch-and-bound design strategy is very similar to backtracking in that a state space tree is used to solve a problem. The differences are that the branch-and-bound method are as follows:

- Does not limit us to any particular way of traversing the tree
- Used only for optimization problems.

A branch-and-bound algorithm computes a number (**bound**) at a node to determine whether the node is promising. The number is a bound on the value of the solution that could be obtained by expanding beyond the node. If that bound is no better than the value of the best solution found so far, the node is nonpromising. Otherwise it is promising. Because the optimal value is a minimum in some problems and a maximum in others, by "better" we mean smaller or larger, depending on the problem.

The backtracking algorithm for the 0-1 Knapsack problem in Section 5.7 is actually a branch-and-bound algorithm. In that algorithm, the promising function returns false if the value bound is not grater than the current value of the *maxprofit*. Besides using the bound to determine whether a node is promising, we can compare the bounds of promising nodes and visit the children of the one with the best bound. In this way, we can we can often arrive at an optimal solution faster than we would by methodically visiting the nodes in some predetermined order. This approach is called **best-search first with branch and bound pruning.** The implementation of this approach is a simple modification of another methodical approach called **breadth-first search with branch-and-bound pruning**.

Recall that the breadth-first search consists of visiting the root first, followed by all the nodes at level 1, then the nodes at level 2, etc.

Unlike depth-first search, there is no simple recursive algorithm for breadth-first search. We can implement one using a **queue**. We insert an item at the end of the queue with a procedure called *enqueue*, and we remove an item from the front with a procedure called *dequeue*.

Lecture Notes

Illustrating Branch-and-Bound with the 0-1 Knapsack problem

Now we will illustrate the branch-and-bound design strategy by applying it to the 0-1 Knapsack problem. First we discuss a simple version called breadth-first search with branch-and-bound pruning. After that, we show an improvement on the simple version called best-first search with branch-and-bound pruning.

Breadth-First Search with Branch-and-Bound Pruning

The 0-1 Knapsack Problem, is an optimization problem, so we can use branch and bound for it. We draw the tree in the same way as for the backtracking algorithm. When doing the *breadth-first* search, we place child items into a queue for later processing if they are promising. If they are not promising we leave them out of the queue, so that branch will not be processed.

A branch-and-bound algorithm decides whether to expand beyond a node by checking whether its bound is better than the best solution found so far. Therefore, when a node is nonpromising because its weight is not less than *W*, we set its bound to $0. In this way, we ensure that its bound cannot be better than the value of the best solution found so far.

In the case of a breadth-first search, this node is the third node visited. At the time it is visited, the value of the *maxprofit* is only $40. Because its bound $82 exceeds *maxprofit,* we expand beyond the node. In a simple breadth-first approach the value of the *maxprofit* can change by the time we visit the children. We waste our time checking these children. We can avoid this in a best-first search.

In a breadth-first search with branch-and-bound pruning, we visit a node's children only of its bound is better than the value of the current best solution. In some applications, there is no solution at the root because we must be at a leaf in the state space tree to have a solution. In these cases, we initialize *best* to a value that is worse than that of any solution.

Refer to the text for details about Algorithm 6.1, the breadth-first search with branch-and-bound pruning algorithm for the 0-1 Knapsack problem.

Function *bound* is essentially the same as function *promising* in algorithm 5.7. The difference is that we have written *bound* according to the guidelines, and *bound* returns an integer. Function *promising* returns a Boolean value because it was written according to backtracking guidelines. In the branch-and bound algorithm, the comparison with *maxprofit* is done in the calling procedure.

Algorithm 6.1 does not produce an optimal set of items; it only determines the sum of the profits in an optimal set. The algorithm can be modified to produce an optimal set as follows. At each node we also store a variable *items*, which is the set of items that have been included up to the node, and we maintain a variable *bestitems*, which is the current best set of items.

Best-First Search with Branch-and-Bound Pruning

In general, the breadth-first search strategy has no advantage over a depth-first search (backtracking). However, we can improve our search by using our bound to do more than just determine whether a node is promising. After visiting all the children of a given node, we can look at all the promising, unexpanded nodes and expand beyond the one with the best bound. In this way we often arrive at an optimal solution more quickly than if we simply proceeded blindly in a predetermined order.

Let's look at the instance of the 0-1 Knapsack problem in Example 6.1. A best-first search produces the pruned state space tree in Figure 6.3. The values *profit*, *weight*, and *bound* are specified from top to bottom at each node of the tree. We refer to a node by its level and position from the left in the tree. We only mention when a node is found to be nonpromising; we do not mention it when it is found to be promising.

Using the best-first search, we have checked only 11 nodes, which is 6 less than the number checked using breadth-first search. This savings does not appear to be significant, but in a large state space tree, the savings can be very significant when the best-first quickly hones in on an optimal solution. There is no guarantee that the node that appears to be best will actually lead to an optimal solution.

The implementation of best-first search consists of a simple modification to breadth-first search. Instead of using a queue, we use a **priority queue**.

Besides using a priority queue instead of a queue, we have added a check following the removal of a node from the priority queue. The check determines if the bound for the node is still better than *best*. This is how we can determine that a node has become nonpromising after visiting the node. Refer to the text for the specific algorithm.

The Traveling Salesperson Problem

Let's recall Nancy's struggles in the Traveling Salesperson problem. When her sales territory was expanded to 40 cities, she found that her dynamic programming algorithm would take six years to find an optimal route for covering the territory. She became content with just finding any tour, and she used the Hamiltonian Circuits problem to do this. Even if this algorithm did find a tour efficiently, the tour might not be optimal. Nancy doesn't want to waste a lot of time, so she has decided to try the branch-and-bound algorithm.

The goal of this problem is to find the shortest path in a directed graph that starts at a given vertex, visits each vertex in the graph exactly once, and ends up back at the starting vertex. Such a path is an *optimal tour*.

An obvious state space tree for this problem is one in which each vertex other than the starting one is tried as the first vertex at level 1, each vertex other than the starting one and the one chosen at level 1 is tried as the second vertex at level 2, etc. A portion of this state space tree, in which there are five vertices and in which there is an edge fro every vertex to every other vertex is shown in Figure 6.5.

Each leaf represents a tour. We need to find a leaf that contains an optimal tour. We stop expanding the tree when there are four vertices in a path stored at a node because the fifth one is uniquely determined. To use the best-first search, we need to be able to determine a bound for each node. IN this problem, we need to determine a lower bound on the length of any tour that can be obtained by expanding beyond a given node, and we call the node promising only if its bound is less than the current minimum tour length. Because a tour must leave every vertex exactly once, a lower bound on the length of a tour is the sum of the minimums. Refer to the text for a more detailed discussion of the algorithm.

To use best-first search, we need to be able to determine a bound for each node. In this problem, we need to determine a lower bound on the length of any tour that can be obtained by expanding beyond a given node, and we call that bound promising only if its bound is less than the current tour length. IN any tour, the length of the edge taken when leaving a vertex must be at least as great as the length of the shortest edge emanating from that vertex. Because a tour must leave every vertex exactly once, a lower bound on the length of a tour is a sum of the minimums. The result does not say that there is a tour of that particular length. It says there can be no tour with a shorter length. Refer to the text for a more detailed discussion of determining the bounds.

A problem does not necessarily have a unique bounding function. In the Traveling Salesperson problem, we can observe that every vertex must be entered and visited exactly once. For a given edge, we could associate half its weight with the vertex it leaves and the other half with the vertex it enters. The cost of visiting the vertex is then less than the sum of the weights associated with entering and exiting it.

When two or more bounding functions are available, one bounding function may produce a better bound at one node, and another might produce a better bound at another node. When this is the case, the algorithm can compute bounds using all available bounding functions and then use the best bound. The goal is not to visit as few nodes as possible but to maximize the efficiency of the algorithm. The extra computations may not offset the savings realized by visiting fewer nodes.

Another approach to handling problems like the Traveling Salesperson problem is to develop **approximation algorithms**. Approximation algorithms are not guaranteed to yield optimum solutions but to yield solutions that are reasonably close to optimal.

Abductive Interference

This section requires knowledge of **probability theory** and **Bayes' theorem**. An important problem in artificial intelligence and expert systems is determining the most probable explanation for some findings.

The process of determining the most probable explanation for a set of findings is called **abductive inference**.

For example, in medicine we want to determine the most probable set of diseases, given a set of symptoms. Assume that there are *n* diseases, *d*1, *d2*, . . . , *dn,* each of which may be present in a patient. We know that the patient has a certain set of symptoms *S*. Our goal is to find the set of diseases that are most probably present. Technically, there could be two or more sets that are probably present. However, we often discuss the problem as if a unique set is most probably present.

The **Bayesian network** has become a standard for representing probabilistic relationships like those between diseases and symptoms. For many Bayesian network applications, there exist efficient algorithms for determining the prior probability that a particular set of diseases contains the only diseases present in the patient.

Given that we can compute these probabilities, we can solve the problem of determining the most probable set of diseases using a state space tree like the one in the 0-1 Knapsack problem. We go to the left of the root to include *d*1, and we go to the right to exclude it. Similarly, we go to the left of a node at level 1 to include *d*2, and we go to the right to exclude it, and so on. Each leaf in the state space tree represents a possible solution. To solve the problem, we compute the conditional probability of the set of diseases at each leaf, and determine which one has the largest conditional probability. For a given node, let *D* be the set of diseases that have been included up to that node, and for some descendant of that node, let *D'* be the set of diseases that have been included up to that descendant. Refer to the text for a more detailed discussion of the problem.

Quick Check

1. In branch-and-bound problem, we are limited in a fixed way to traverse the tree. (True or False)
   Answer: False
2. Since the 0-1 Knapsack Problem, is a(n) ____ problem, we can use branch and bound for it.
   Answer: optimization
3. In best-first search, unlike breadth-first search, we put items into a(n) ____.
   Answer: priority queue
4. From one set of data, we can know how quickly the Branch-and-bound solves the traveling salesman problem. (True or False)
   Answer: False
5. Abductive inference is the process of determining the most probable explanation for a set of findings (True or False)
   Answer: True

Classroom Discussion

- Discuss three applications of the branch-and-bound design strategy.

Homework

Assign Exercises 7, and 10.

Keywords

- **Abductive inference –** process of determining the most probable explanation for a set of findings.
- **Approximation algorithms –** algorithms that are not guaranteed to yield optimal solutions, but rather yield solutions that are reasonably close to optimal.
- **Best-first search** – a state-space search that considers the estimated best partial solution next.
- **Best-first search with branch-and-bound pruning –** a technique for comparing the bounds with promising nodes and visiting the children of the one with the best bound.
- **Bound –** upper or lower limits of the quantity being optimized.
- **Branch–and–bound algorithm –** an algorithmic technique to find the best possible solution by keeping the best solution found so far.
- **Breadth-first search –** the search consists of visiting the root first, followed by all the nodes at level 1, followed by all the nodes at level 2, etc.
- **Breadth-first search with branch-and-bound pruning –** a technique for comparing the bounds of promising roots first.
- **Bayes' theorem –** shows the relationship between one conditional probability and its inverse.
- **Bayesian network –** a standard for representing probabilistic relationships.
- **Depth-first search –** the process of visiting the nodes of an algorithm in s predetermined order.
- **Nonpromising node –** if when visiting the node we determine that it cannot possibly lead to a solution.
- **Optimal tour –** shortest path in a directed graph that starts at a given vertex, visits each vertex in the graph exactly once, and ends up back at the starting vertex.
- **Probability theory –** the branch of mathematics concerned with the analysis of random phenomena.
- **Priority queue –** an abstract data type that supports finding the item with the highest priority across a series of operations.
- **Profit –** sum of the profit that have been included up to a node.
- **Promising node –** if when visiting the node we determine that it will lead to a solution.
- **Queue –** a collection of items in which only the earliest added item can be accessed.
- **Weight –** sum of the weights that have been included up to a node.

Important Links

- http://www.cs.pitt.edu/~kirk/algorithmcourses/ (Algorithms Courses on the Internet)
- www.sciencedirect.com/science/journal/01966774 (Journal of Algorithms)
- http://www.devarticles.com/c/a/Development-Cycles/Branch-and-Bound-Algorithm-Technique/ (article on branch and bound algorithms)