Chapter 5

Backtracking Algorithms

A backtracking algorithm is an organized exhaustive search which often avoids searching all possibilities. Thus, all **feasible** solutions are checked, though it is possible, via **pruning**, to avoid generating all the feasible solutions. We represent the whole process by a **state space tree** in which we try to recognize *good* and *bad* branches and prune the tree by abandoning the bad branches (thus cutting down the number of feasible solutions to be checked). Pruning is often done by using a **bounding function**. Backtracking algorithms generally make use of a bounding function to create a **depth first** state space tree. **Branch and bound** is a backtracking technique.
We revisit the traveling salesperson problem and show how branch and bound can be used to solve this combinatorial optimization problem.

5.1 The Traveling Salesperson Problem

Consider the following EXAMPLE:

	1	2	3	4	5
1	∞	25	40	31	27
2	5	∞	17	30	25
3	19	15	∞	6	1
4	9	50	24	∞	6
5	22	8	7	10	∞

The above table represents the cost matrix of visiting a certain city from any other of the remaining 4 cities. For example, the entry in row 3 and column 5 is the cost of going from city 3 to city 5. Recall that with the traveling salesman problem, we want to find the cheapest Hamiltonian circuit. Let us represent the above information by a cost matrix M.

$$M = \begin{bmatrix} \infty & 25 & 40 & 31 & 27 \\ 5 & \infty & 17 & 30 & 25 \\ 19 & 15 & \infty & 6 & 1 \\ 9 & 50 & 24 & \infty & 6 \\ 22 & 8 & 7 & 10 & \infty \end{bmatrix}$$

Next, we search for a lower bound on the cheapest Hamiltonian tour. Reduce M to a matrix that has a 0 in every row and in every column.

$$
\begin{array}{cc}
 & \text{row sub-} \\
 & \text{tractions} \\
\left[\begin{array}{ccccc}
\infty & \mathbf{0} & 15 & 6 & 2 \\
\mathbf{0} & \infty & 12 & 25 & 20 \\
18 & 14 & \infty & 5 & \mathbf{0} \\
3 & 44 & 18 & \infty & \mathbf{0} \\
15 & 1 & \mathbf{0} & 3 & \infty
\end{array}\right] &
\begin{array}{c}
25 \\ 5 \\ 1 \\ 6 \\ 7 \\ \hline 44
\end{array}
\end{array}
$$

$$
\begin{array}{cc}
 & \left[\begin{array}{ccccc}
\infty & 0 & 15 & 3 & 2 \\
0 & \infty & 12 & 22 & 20 \\
18 & 14 & \infty & 2 & 0 \\
3 & 44 & 18 & \infty & 0 \\
15 & 1 & 7 & \mathbf{0} & \infty
\end{array}\right] \\
\text{column subtrac-tions} & \begin{array}{ccccc} & & & 3 & \end{array}
\end{array}
$$

Hence, the total amount subtracted is: $44 + 3 = 47$.

Fact: Any Hamiltonian circuit will have a cost greater or equal to 47. Therefore 47 is a lower bound and we can use it as our bounding function.

Next, we are interested in finding one Hamiltonian path. One of the ways of finding such a path is demonstrated in the following steps. The Hamiltonian path will constitute the leftmost path in the state space tree. Recall that the construction of the state space tree is a depth-first construction.

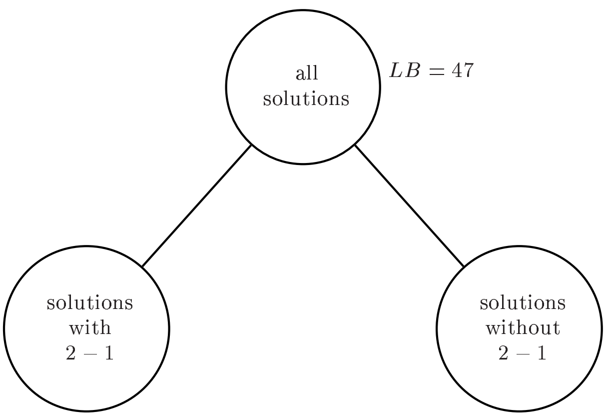
Find the minima (diferrent from the first zero) of each row and column.

$$
\begin{array}{ccl}
 & & \text{minima} \\
 & \left[\begin{array}{ccccc}
\infty & \mathbf{0} & 15 & 3 & 2 \\
\mathbf{0} & \infty & 12 & 22 & 20 \\
18 & 14 & \infty & 2 & \mathbf{0} \\
3 & 44 & 18 & \infty & \mathbf{0} \\
15 & 1 & \mathbf{0} & 0 & \infty
\end{array}\right] &
\begin{array}{l}
2 \\ 12 \text{ highest!} \\ 2 \\ 3 \\ 0
\end{array} \\
\text{minima} & \begin{array}{ccccc} 3 & 1 & 12 & 0 & 0 \end{array} &
\end{array}
$$

Suppose that we choose edge 2-1 as 1^{st} edge (since 12 is highest).

We start building the state space tree. The root node represents **all** possible solutions.

The left child of the root node represents all solutions that include the edge 2-1 while the right child represents all solutions that do not include the edge 2-1. So far we have constructed:



We will continue by considering the left child since this is a depth-first procedure.

Left child of the root node:

Delete row 2 and column 1 since by choosing 2-1 we do not want to leave from 2 anymore and we do not want to get to 1 from anywhere. Moreover, entry $(1, 2)$, i.e. cost from going from 1 to 2 is now ∞ since it is an impossible path (no cycles). Therefore, the correspinding cost matrix is:

	2	3	4	5
1	∞	15	3	2
3	14	∞	2	0
4	44	18	∞	0
5	1	0	0	∞

Again, we will reduce the new cost matrix to a matrix, where every row and every column has a 0 (by subtracting 1 in the first column and 2 in the first row). So we obtain:

	2	3	4	5
1	∞	13	1	0
3	13	∞	2	0
4	43	18	∞	0
5	0	0	0	∞

Since we subtracted $1 + 2 = 3$, the lower bound at node solution with $2 - 1$ is $LB = 47 + 3 = 50$. We choose edge $4 - 5$. Again we get a new cost matrix:

	2	3	4
1	∞	13	1
3	13	∞	2
5	0	0	∞

The reduction yields

	2	3	4
1	∞	12	0
3	11	∞	0
5	0	0	∞

So lower bound at solution with $4 - 5$ is $LB = 50 + 3 = 53$. Now we choose edge $1 - 4$, with a cost matrix

	2	3
3	11	∞
5	0	0

and a lower bound $LB = 53 + 11 = 64$, since we reduce this matrix to the matrix below by subtracing 11 from the first row. Hence the matrix at solution with $1-4$ is

	2	3
3	0	∞
5	0	0

Since the remaining table has 0's and ∞'s only, we can deduce the circuit corresponding to this path (left most) in the state space tree.
In the process, we chose the following edges: $2-1$, $4-5$, $1-4$, $3-2$, $5-3$, so we have $1-4-5-3-2-1$ as circuit with cost 64.
So 64 is the cost of the current best circuit.
At this point, we are done with the leftmost path in the state space tree and have found a Hamiltonian path of cost 64.

So now we backtrack to node solution without $1-4$ and we get as new cost matrix

	2	3	4
1	∞	12	∞
3	11	∞	0
5	0	0	∞

So the lower bound at node solution without $1-4$ is $LB = 53 + 12 = 65$, but $65 \geq 64$, therefore *prune* tree at this node, i.e. **stop** this path in the state space tree. So backtrack to solution without $4-5$. The cost matrix is

	2	3	4	5
1	∞	13	1	0
3	13	∞	2	0
4	43	18	∞	∞
5	0	0	0	∞

and after the reduction (subtract 18 from the elements in the third row) we have:

	2	3	4	5
1	∞	13	1	0
3	13	∞	2	0
4	25	0	∞	∞
5	0	0	0	∞

Therefore lower bound at solution without $4-5$ is $LB = 50 + 18 = 68$ and again $68 \geq 64$,

and backtrack to the right child of the root node.

Right child of the root node:

Backtracking to the right child of the root node take us to: solution without $2-1$.

The cost matrix is

	1	2	3	4	5
1	∞	0	15	3	2
2	∞	∞	12	22	20
3	18	14	∞	2	0
4	3	44	18	∞	0
5	15	1	0	0	∞

and after subtracting 12 in the second row and 3 in the first column, we have

	1	2	3	4	5
1	∞	0	15	3	2
2	∞	∞	0	10	8
3	15	14	∞	2	0
4	0	44	18	∞	0
5	12	1	0	0	∞

with a lower bound of $LB = 47 + 15 = 62 \leq 64$. So continue working along this path in the tree. We choose edge 4-1, and the new cost matrix is already reduced:

	2	3	4	5
1	0	15	∞	2
2	∞	0	10	8
3	14	∞	2	0
5	1	0	0	∞

so at node solution with $4-1$ $LB = 62 + 0 = 62$. We continue and now we choose edge $2-3$, obtaining the cost matrix

	2	4	5
1	0	∞	2
3	∞	2	0
5	1	0	∞

that is again already reduced.
After choosing edge $3-5$ the cost matrix still remains reduced:

	2	4
1	0	∞
5	1	0

So choose edges $1-2$ and $5-4$ with lower bound $LB = 62+0$. Stop this branch with current optimal solution $= 62$ and circuit $4-1-2-3-5-4$. Forget about the previous $LB = 64$ and now use the current optimal $LB = 62$.

Backtrack to solution without $5-4$ with cost matrix

	2	4
1	0	∞
5	1	∞

But here we have to subtract 1 (second row), so $LB = 62+1 = 63 > 62$. Therefore stop here and backtrack to solution without $3-5$. But the cost matrix

	2	4	5
1	0	∞	2
3	∞	2	∞
5	1	0	∞

has to be recuded (subtract 2 in second row and in third column), so $LB = 62+4 = 66 > 62$, again stop this path and backtrack to solution without $2-3$.

	2	3	4	5
1	0	15	∞	2
2	∞	∞	10	8
3	14	∞	2	0
5	1	0	0	∞

Reduce by subtracting 8 in 2^{nd} row. Therefore $LB = 62+8 = 70$, so stop this path and backtrack to solution without $4-1$. The cost matrix is

	1	2	3	4	5
1	∞	0	15	3	2
2	∞	∞	0	10	8
3	15	14	∞	2	0
4	∞	44	18	∞	0
5	12	1	0	0	∞

that has to be reduced by subtracting 12 in the first column, therefore $LB = 62 + 12 = 74$. So stop (prune).

The best solution is $4 - 1 - 2 - 3 - 5 - 4$ with cost $= 62$.

Note that we could have stopped at the node corresponding to *solutions with edge* $3-5$ of cost$= 62$, since its lower bound is 52 which is equal to the lower bound of the node corresponding to *solutions without edge* $2 - 1$. There is no way one can find a cheaper cost (than 62) by considering the right subtree of the node corrresponding to *solutions without arc* $2 - 1$ since

$$\begin{array}{c}\text{lower bound}\\ \text{of node}\end{array} \leq \begin{array}{c}\text{lower bound}\\ \text{of any descendant}\\ \text{of that node}\end{array}$$

The state space tree is given by Figure 5.1.

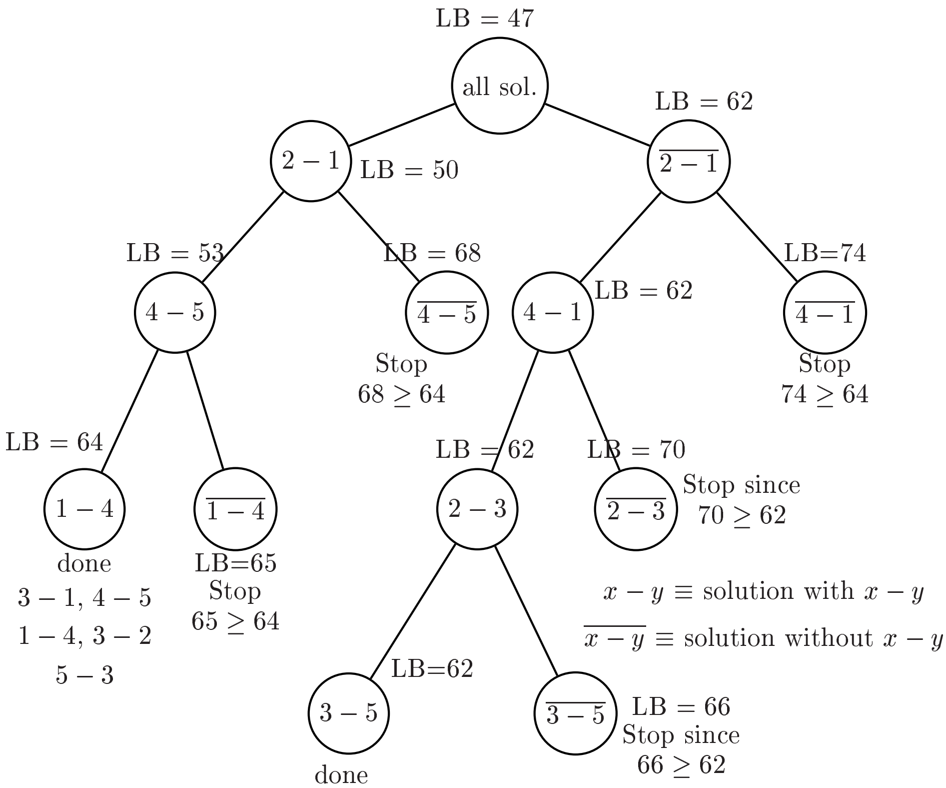


Figure 5.1: The State Space Tree.

Next, we revisit the 0-1 knapsack problem and give two backtracking algorithms to sole the problem.

5.2 Backtracking and the 0-1 Knapsack Problem

In this section, we discuss two methods for solving the 0-1 knapsack problem. While the first is very intuitive and easy to understand, the second is more efficient [Sti87].

First method: Let $\mathbf{x} = (x_1, x_2, \ldots, x_{lev}, -, -, \ldots, -)$ denote a partial solution during the construction of a feasible solutions. Recall that

$$x \text{ is feasible } \Leftrightarrow \sum_{i=1}^{lev} w_i x_i \leq M \qquad 1 \leq lev \leq n \quad \begin{array}{c}\text{where } M \text{ is the}\\ \text{knapsack capacity}\end{array}$$

$\sum_{i=1}^{lev} w_i x_i$ can thus be used to prune the state space tree.

EXAMPLE:

objects	1	2	3	4	5
weights	11	12	8	7	9
profits	23	24	15	13	16

$M = 26$

Note that the ordering of the objects $\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \cdots \geq \frac{p_n}{w_n}$ is not important here.

Second method: Use a bounding function $B(x)$ where $\mathbf{x} = \{x_1, x_2 \ldots, x_{lev}, \ldots\}$. Let $currw := \sum_{i=1}^{lev} w_i x_i$ and define $RK(lev, M')$ to be the optimal profit of the rational knapsack problem using objects $lev+1, lev+2, \ldots, n$ and capacity $M' = M - currw$. Define the bounding function $B(\mathbf{x})$ where $\mathbf{x}$ is a feasible partial solution:

$$B(\mathbf{x}) = \sum_{i=1}^{lev} p_i w_i + RK(lev, M - currw).$$

In other words, $B(\mathbf{x})$ is the sum of

- the profit obtained from objects $1, 2, \ldots, lev$, plus
- the profit from the remaining objects, using the remaining capacity $M - currw$, but allowing rational x_i's.

EXAMPLE: Consider the previous one.

1. $\mathbf{x} = (-, -, -, -, -)$; the optimal solution using objects $1, 2, 3, 4, 5$ is given by $RK(0, M)$ ($M' = M$, since $currw = 0$), where

   $$RK(0, M) = 23(1) + 24(1) + 15(\frac{3}{8}) = 52.625,$$

   since $(1, 1, \frac{3}{8}, 0, 0)$ is the solution vector $\Rightarrow \; B(\mathbf{x}) = 0 + 52.625$

2. $\mathbf{x} = (1, -, -, -, -)$ with $currw = 11$; optimal solution using $2, 3, 4, 5$ and $M' = 26 - 11 = 15$ is

   $$RK(1, M') = 24(1) + 15(\frac{3}{8}) = 29.625$$

   Therefore

   $$B(\mathbf{x}) = 23 \cdot (1) + 29.625$$

3. $\mathbf{x} = (1, 1, -, -, -)$, $currw = 23$ and $M' = 26 - 23 = 3$

   $$RK(2, M') = \frac{3}{8} \cdot 15 = 5.625$$

   Therefore

   $$B(\mathbf{x}) = [23(1) + 24(1)] + [5.625] = 52.625$$

4. $\mathbf{x} = (1, 1, 1, -, -)$, is not a feasible solution since $currw = 31 > 26$; so backtrack and consider:

5. $\mathbf{x} = (1, 1, 0, -, -)$, $currw = 23$ and $M' = 26 - 23 = 3$; now look for optimal solution using objects 4 and 5 to fill knapsack of capacity 3

$$RK(3, M') = \frac{3}{7} \cdot 13 = 5.5705$$

$$\Rightarrow B(\mathbf{x}) = [23(1) + 24(1) + 15(0)] + 5.5705 = 52.5705$$

6. $\mathbf{x} = (1, 1, 0, 1, -)$ is not a feasible solution, therefore backtrack and consider:

7. $\mathbf{x} = (1, 1, 0, 0, -)$ with $currw = 23$ and $M' = 26 - 23 = 3$; so

$$RK(4, M') = \frac{3}{9} \cdot 16 = 5.3333$$

Therefore

$$B(\mathbf{x}) = 47 + 5.3333 = 52.3333$$

8. $\mathbf{x} = (1, 1, 0, 0, 1)$ is not a feasible solution; backtrack and consider:

9. $\mathbf{x} = (1, 1, 0, 0, 0)$; profit $= 23(1) + 24(1) = 47$; therefore let $opt = 47$ (current optimum profit); now backtrack almost to the beginning:

10. $\mathbf{x} = (1, 0, -, -, -)$, $currw = 11$ and $M' = 26 - 11 = 15$

$$RK(2, M') = 1(15) + 1(13) = 28$$

Therefore

$$B(\mathbf{x}) = 23 + 28 = 51$$

11. $\mathbf{x} = (1, 0, 1, -, -)$, $currw = 19$ and $M' = 26 - 19 = 7$

$$RK(3, M') = 1(13) = 13$$

Therefore

$$B(\mathbf{x}) = (23 + 15) + 13 = 51$$

12. $\mathbf{x} = (1, 0, 1, 1, -)$ with $currw = 26$ and $M' = 26 - 26 = 0$

$$RK(4, M') = 0$$

So

$$B(\mathbf{x}) = (23 + 15 + 13) + 0 = 51$$

13. $\mathbf{x} = (1, 0, 1, 1, 1)$ is not a feasible solution, therefore backtrack and consider

14. $\mathbf{x} = (1, 0, 1, 1, 0)$ with $currw = 26$, $B(\mathbf{x}) = P = 51$; since $51 > 47$, let $opt = 51$ (that is the new current optimal profit); backtrack and consider

15. $\mathbf{x} = (1, 0, 1, 0, -)$, $currw = 19$ and $M' = 26 - 19 = 9$

$$RK(4, M') = \frac{7}{9} \cdot 16$$

$$B(\mathbf{x}) = (23 + 15) + \frac{7}{9} \cdot 16 = 50.52 < 51 \text{ so prune}$$

Again backtrack and consider

16. $\mathbf{x} = (1, 0, 0, -, -)$ with $currw = 11$ and $M' = 26 - 11 = 15$

$$RK(3, M') = 1(13) + \frac{8}{9} \cdot 16 = 13 + 14.08$$

$$B(\mathbf{x}) = 23 + 27.08 < 51 \text{ so again prune}$$

backtrack and consider

17. $\mathbf{x} = (0, -, -, -, -)$, $currw = 0$ and $M' = 26 - 0 = 26$; so

$$RK(1, M') = 1(24) + 1(15) + \frac{6}{7} \cdot 13 = 50.141$$

and

$$B(\mathbf{x}) = 0 + 50.141 < 51 \text{ so prune}$$

Finally, we are done with the solution $(1, 0, 1, 1, 0)$ and an optimal profit $opt = 51$.

Remark: We know that

$$\text{optimal solution of 0-1 knapsack} \quad \leq B(\mathbf{x})$$

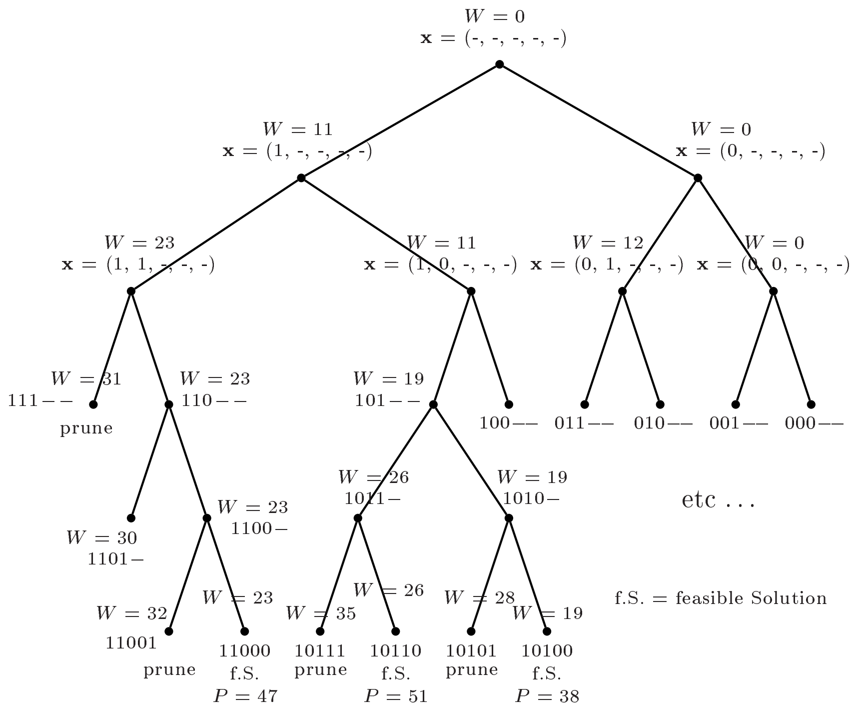
So if at some stage during the construction of the state space tree we have

$$B(\mathbf{x}) \leq opt$$

(where *opt* is the optimal profit found so far) then

$$\text{optimal solution of 0-1 knapsack} \ \leq B(\mathbf{x}) \leq opt$$

we can therefore ignore **all** descendants of $\mathbf{x}$ (i.e. prune tree) since none of them will produce a profit higher than *opt*.



All feasible solutions appear at the leaves of the state space tree. Choose the one with highest profit.

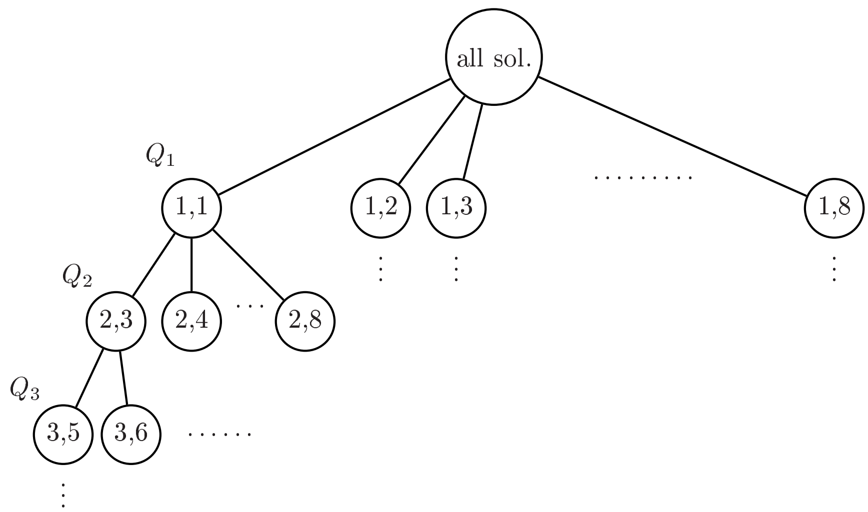
We now consider three more problems that can be solved by using backtracking.

5.3 More Backtracking Examples

There are problems that can be solved only by performing an exhaustive search of all possible solutions. One way to search systematically for a solution is to use a decision tree, where each internal vertex represents a decision and each leaf a possible solution. Recall that to find a solution via backtracking, first make a sequence of decisions in an attempt to reach a solution as long as this is possible. The sequence of decisions can one represent by a path in the decision tree (state space tree). Once it is known that no solution can result from any further sequence of decisions, backtrack to the parent of the current vertex and work toward a solution with another series of decisions, if this is possible. The procedure continues until a solution is found, or it is established that no solution exists. The problems discussed in this section are taken from [Ros95, pp. 585-588].

5.3.1 The Eight Queens Problem

The n-queens problem asks how n queens can be placed on an $n \times n$ chessboard so that no two queens can attack one another. We have to find n positions on an $n \times n$ chessboard so that no two of these positions are in the same row, same column, or in the same diagonal. We will use backtracking to solve the n-queens problem.



Positions on the board are represented by (a, b); a: row, b: column. So the levels in the tree are the rows. The 1^{st} node has 8 children but subsequent nodes will have less than 8 children since we know that if we place a queen on $(1, 1)$ then we don't need to consider $(2, x)$ for $x = 1, 2$. So now place a queen on $(1, 1)$. Consider the children of $(1, 1)$. Place the second queen Q_2 on $(2, 3)$. Consider the children of $(2, 3)$. Place the 3^{rd} queen Q_3 on $(3, 5)$ etc. until we are stuck. Backtrack to the most recent parent, consider the next child and go on until stuck again or all 8 queens are placed on the board. Note that this method will yield a solution with Q_1 on $(1, 1)$. If we are interested in more than one solution, we can finally backtrack to $(1, 2)$, place Q_1 on $(1, 2)$ and use the same method to generate solutions with Q_1 on $(1, 2)$.

Possible solutions:

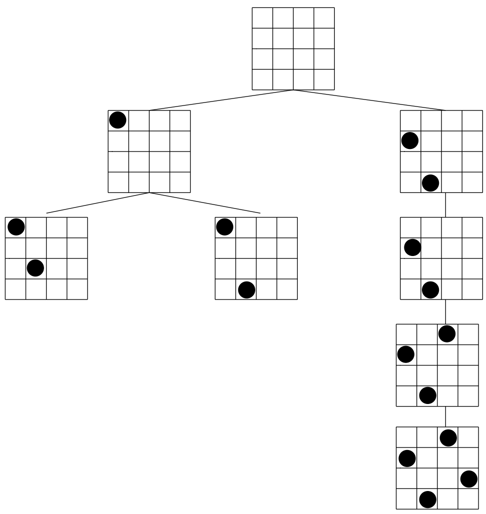
1. (1, 3), (2, 5), (3, 8), (4, 4), (5, 1), (6, 7), (7, 2), (8, 6)
2. (1, 1), (2, 7), (3, 4), (4, 6), (5, 8), (6, 2), (7, 5), (8, 3)

There are $\binom{64}{8} = 4,426,165,368$ ways of choosing 8 out of (8×8). By allowing the placement of queens only on distinct rows and columns, we require the examination of at most 8! 8-tuples.

Another way of seeing the 8! is by observing that in the (a, b) notation we can omit **a** (since we have $a = 1$, $a = 2$, ..., $a = 8$ in that order) and represent the solution with $(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$ where the x_i are the b's i.e. the columns. So we have a permutation of 8 queens. As a matter of fact, Sahni and Horowitz used this fact and have an elegant solution in "Fundamentals of Computer Algorithms" [HS78].

As a second example, Rosen considers the Four-Queens problem. He uses backtracking to solve the problem. At stage $(k + 1)$ we attempt putting an additional queen on the board in the $(k + 1)$st column, where there are already queens in the first k columns. We examine squares in the $(k + 1)$st column starting with the square in the first row, looking for a position to place this queen so that it is not in the same row or in the same diagonal as a queen already on the board. If it is impossible to place the queen in the $(k + 1)$st column, backtrack to the placement of the queen in the kth column, and place this queen in the next allowable row in this column, if such a row exists. If no such row exists, backtrack further.

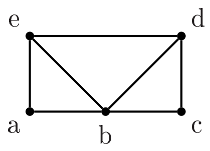
The following is a solution to the 4-queens problem, that works in the way described above.

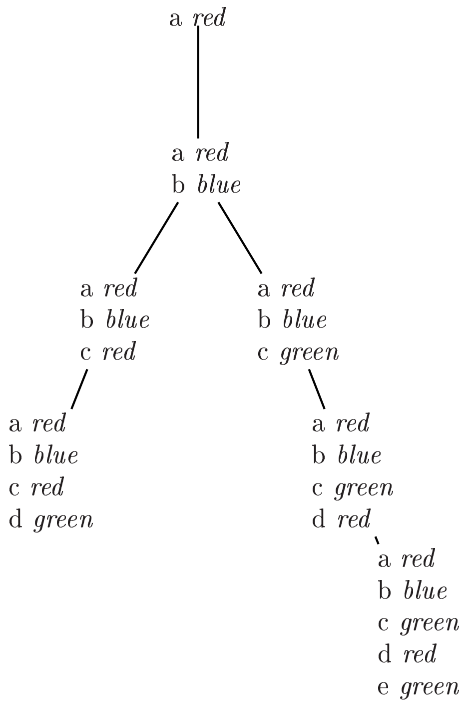


5.3.2 Vertex coloring problem

Find the minimum number of colors needed to color the vertices of a graph such that no two adjacent nodes have the same color. In this example, we would like to know if we can color the graph with 3 colors only. The tree shown in the figure illustrates how backtracking can be used to construct a 3-coloring. In this procedure, red is used first, then blue, and finally green.

In this tree, the initial path from the root, which represents the assignment of red to a, leads to a coloring with a red, b blue, c red, and d green. It is impossible to color e using any of the three colors when a, b, c, and d are colored in this way. So, backtrack to the parent of the vertex representing this coloring. Since no other color can be used for d, backtrack one more level. Then change the color of c to green. We obtain a coloring of the graph by then assigning red to d and green to e. □

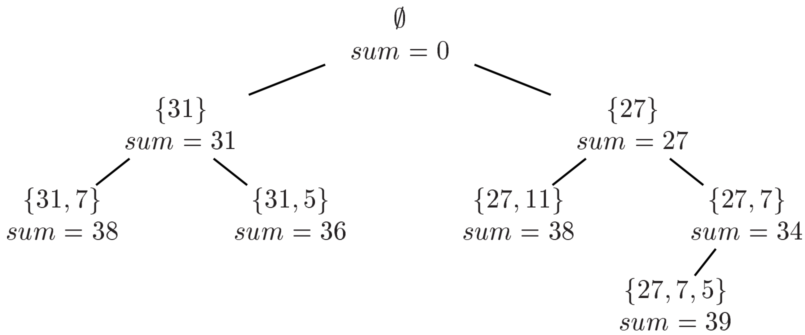




5.3.3 Sum of subsets

Given a set of positive integers $x_1, x_2, \ldots, x_n$ find a subset that has M as its sum.

EXAMPLE: $S = \{31, 27, 15, 11, 7, 5\}, M = 39$



To find a solution, we start with a sum with no terms. We build up the sum by successively adding terms. An integer in the sequence is included if the sum remains less than M when this integer is added to the sum. If a sum is reached so that the addition of any term is greater than M, backtrack by dropping the last term of the sum. The above figure displays a backtracking solution to the problem of finding a subset of $\{31, 27, 15, 11, 7, 5\}$ with sum equal to 39. □

This concludes the algorithmic techniques we wanted to study; namely: greedy, divide and conquer, dynamic programming and backtracking. Before continuing and studying topics, we end the introduction to algrithmic techniques by giving a very brief classification of problems.

5.4 Classification of Problems

Roughly speaking, problems can be classified in three categories:

tractable: solvable in polynomial-time
intractable: solvable but not in polynomial-time
unsolvable: problems that **cannot** be solved (example: halting problem)

NP-Complete Problems: Problems for which no polynomial-time algorithm is known, but no one has been able to show that such algorithms do not exit. NP-complete problems are equivalent to the sense that if any one of them has a polynomial-time solution, then all of them do. One of the major open questions in computer science is whether there is a polynomial-time solution for any of the NP-complete problems. Almost all experts conjecture strongly that the answer is **no**. The evidence supporting this conjecture, that $P \subset NP$ and not $P = NP$, is only the failure to find such a polynomial-time solution to any of the NP-complete problems, in over thirty years of research. Steven Cook defined the notion of NP-completeness and showed the NP-completeness of a collection of problems, mostly from the field of mathematical logic. The first problem he tracked is the Satisfiability Problem. In 1970's, he proved that this problem is NP-complete.

Roughly speaking, he defined NP-complete problems to be problems that have the property that we can verify in polynomial-time whether a supplied solution is correct. For example, if we did not have to find an optimal filling of the 0-1 Knapsack Problem but simply to verify that a given filling has a profit value of at least k, where k is a given integer, it would be easy to write a polynomial-time algorithm to do so.

Since then, many hundreds of natural problems from many areas of computer science and mathematics such as graph theory, combinatorial optimization, scheduling and symbolic computation have been proven to be NP-complete.

There are several reasons for which polynomial-time problems are considered tractable:

a) Most practical problems can be solved in $O(n^4)$ and do not require higher-degree polynomial-time algorithms.

b) Portability of polynomial-time problems (from one model to another).

c) The class of polynomial-time problems has nice closure properties because polynomials are closed under $+$, $\times$ and $\circ$ (polynomial composition).

In practice, we still need to solve many problems that are known to be NP-complete. There are several ways of dealing with NP-complete problems:

1. We can sometimes use dynamic programming or branch and bound techniques to cut down the running time from the brute force exhaustive search. This might work only if the problem size is not too big.

2. In some cases, algorithms have non-polynomial behavior in the worst-case, but are efficient in some cases (with certain input data).

3. Give up the exact algorithm and settle for an approximate algorithm that will generally produce a suboptimal solution. Approximation algorithms (unlike heuristics) produce solutions with some guarantee of how close they are to optimal solutions.

4. Heuristics such as local search, simulated annealing, genetic algorithms, neural networks, and many others can be used to improve the solution quality or the running time. However, the solutions produced cannot be guaranteed to be within a certain distance from the optimal solution. Rigorous analysis of heuristic algorithms is generally very difficult.

5. The problem at hand may be more specialized than the general one that was proven to be NP-complete.

Bibliography

[HS78] E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms.* Computer Science Press, 1978.

[Ros95] Kenneth H. Rosen. *Discrete Mathematics and Its Applications.* McGraw-Hill, 1995.

[Sti87] D.R. Stinson. *An Introduction to the Design and Analysis of Algorithms.* Charles Babbage Research Centre, Manitoba, Canada, 1987.