Chapter 16 Combinatorial Searching

"If he [Thomas Edison] had a needle to find in a haystack, he would not stop to reason where it was most likely to be, but would proceed at once with the feverish diligence of a bee, to examine straw after straw until he found the object of his search. ... Just a little theory and calculation would have saved him ninety percent of his labor. " - Nikola Tesla

16.1 Introduction

Combinatorics is the study of finite or countable[1] discrete structures. Graphs, trees, countable or finite sets, permutations, solution spaces to games such as chess, certain types of vector spaces and topological spaces, and various algebraic entities are examples of such discrete structures. Some common types of combinatorial problems include:

- counting how many structures satisfy certain conditions, such as the number of distinct binary trees of a given height, or the number of ways that n rooks can be placed on a chess board so that no two are attacking each other,
- finding a maximum or minimum of a set of structures, such as a shortest path through a graph, and
- deciding whether an object satisfies certain conditions or whether an object exists that satisfies a set of conditions, such as whether there is a proof of a theorem in a given theory, whether there is an assignment of truth values to a boolean expression that makes it true, whether a number is a prime number, or whether a graph can be colored with a given number of colors so that no two adjacent vertices have the same color.

Such problems are usually characterized by a ***search space***, which represents the set of possible solutions to the problem. Often the size of this search space is an exponential function of the size of the ***instance of the problem***. Problems and problem instances are different. A problem can be thought of as the set of all problem instances. For example

> Given any set of cities, find a shortest round trip that visits all of the cities.

is a problem, whereas

> Given the particular set of cities S, find a shortest round trip that visits all of the cities in S.

is an instance of the problem. Because the search space is so large, exhaustive search is usually not possible except for small problem instances. The objective of combinatorial searching is to find one or more solutions, possibly optimal solutions, in a large search space. Some problems for which it has been used include

- finding optimal layouts of circuits in VLSI design,
- protein structure prediction,
- internet packet routing, and
- playing games.

[1] A set is countable if it can be put into a one-to-one correspondence with the positive integers.

A ***decision problem*** is a problem that has a yes/no answer. In combinatorial searching, a decision problem is one that answers the question, does a solution exist, which can be rephrased as, is there a candidate solution that satisfies the stated conditions of a solution? Any decision problem has a corresponding search problem, and vice versa:

- Search variant: Find a solution for given problem instance (or determine that no solution exists)
- Decision variant: Determine whether solution for given problem instance exists

A ***combinatorial optimization problem*** is a problem in which a solution is sought that minimizes the value of some objective function. (See Chapter 10.) The purpose of this chapter is to explore a few different types of combinatorial search problems and algorithmic paradigms that can be used to solve them.

16.2 Search Trees

Many combinatorial search algorithms are characterized by their having a procedure which, given an instance Q of the problem, can either solve Q directly or derive from Q a set of subproblems $Q_1, Q_2, \ldots, Q_D$ such that the solution to Q can be constructed from the solutions to the subproblems. Such search problems can be represented by ***search trees***. A search tree is a tree in which the root node represents the original instance of the problem to be solved, and non-root nodes represent subproblems of the original problem. Leaf nodes represent subproblems that can be solved directly, and non-leaf nodes represent subproblems whose solution is obtained by deriving subproblems from it and combining their solutions. An edge from a parent to a child node represents the fact that the child node is a subproblem derived from the parent.

There are two types of non-leaf nodes in a search tree. An ***AND-node*** represents a problem that is solved only when all of its child nodes have been solved. An ***OR-node*** represents a problem that is solved when any of its children have been solved. A tree consisting entirely of AND-nodes is called an ***AND-tree***. A tree consisting entirely of OR-nodes is called an ***OR-tree***. A tree that contains both types of nodes is called an ***AND/OR-tree***. Figure 16.1 illustrates the three different types of trees.

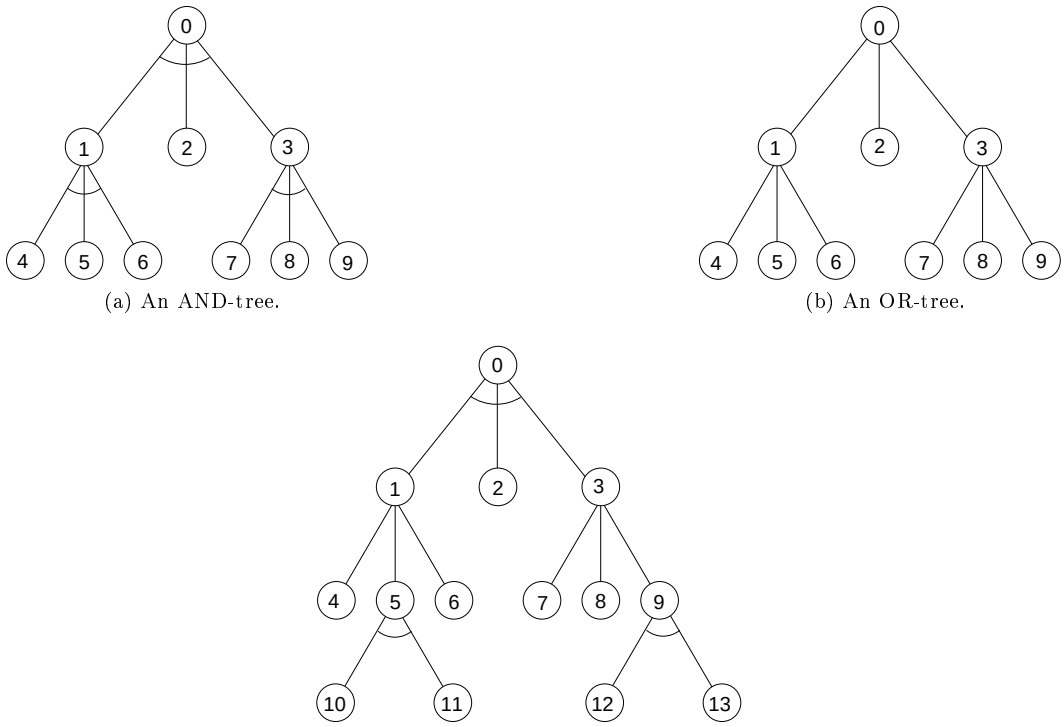
16.3 Divide and Conquer

Divide-and-conquer is a problem solving paradigm in which a problem is partitioned into smaller subproblems, each of which is solved using the divide-and-conquer paradigm if they are sufficiently large in size, and then solutions to the subproblems are combined into a solution to the original problem. If the subproblem instances are small enough, they are solved by some non-recursive method.

The simplest example of the divide-and-conquer paradigm is the binary search algorithm for searching for a key in a sorted, random-access list (e.g., an array.) Binary search can be represented by an OR-tree, because the entire set of subproblem solutions is not searched; the algorithm performs a ***dichotomous search***, meaning that it searches only one of two alternatives at each node. Quicksort, on the other hand, is represented by an AND-tree, because each subproblem's solution must be found. Both of these examples divide the problem into two subproblems, but the paradigm can be used to divide a problem into any number of subproblems.

The divide-and-conquer paradigm lends itself to parallelization because separate processes can search separate parts of the search space. This is easily accomplished on a shared memory multiprocessor, using a parallel language that supports the sharing of variables. The need to share data arises because the list of unsolved problems must be accessible to all processes, otherwise trying to load-balance the processes is difficult. With a single list of unsolved problems, treated like a stack, the processes can use a decentralized load-balancing algorithm, pushing and popping problems from the stack as needed. Of course the stack itself can become a bottleneck in this case.

On a multicomputer, the situation is different. There is no shared memory, ruling out the possibility of using a centralized stack. Instead the subproblems must be distributed to the processors, but this is not simple, because the subproblems are generated dynamically. Different solutions to this problem have been



(a) An AND-tree.

(b) An OR-tree.

(c) An AND-OR-tree.

Figure 16.1: Three different types of search trees. The tree in Figure (a) is an AND-tree. This implies that subproblem 1 is solved only when each of subproblems 4, 5, and 6 are solved, and more generally, the initial problem is solved if and only if solutions for problems 4 though 9 have been found. The tree in Figure (b) is an OR-tree. Therefore, the root problem is solved if and only if a solution is found for any of subproblems 4 through 9. The tree in Figure (c) is an AND/OR-tree. There are various combinations of solutions of subproblems that contribute to a solution to the original problem. For example, solutions to 4, 2, and 7 lead to a solution to the initial problem, as do solutions to 10, 11,, 12, 13, and 2.

proposed and studied, and two distinct ones emerge, one in which the initial problem is distributed among the memories of all processors, and the other in which a single processor stores the initial problem and ultimately its solution. In the latter design, processor utilization is low and gradually increases, as work is distributed among them. In the former, the limited memory prevents the solution from being highly scalable. We will not explore this class of problem solving paradigms here, preferring instead to focus on message-passing based algorithms.

16.4 Backtrack Search

Backtrack search is an ***enumerative method*** for solving combinatorial optimization problems. An enumerative method tries to exhaustively search through all possible solutions until it finds one. A problem that yields to a backtracking solution must have the property that it is possible to determine that some initial choices cannot lead to a solution. This makes it possible to stop a fruitless search without having to take it to a complete conclusion. Backtracking works by descending a tree (i.e., extending a partial solution) as far as it can go while searching for a complete solution, and if it fails to find one, then it backs up the tree (it backtracks) to the nearest ancestor that still has subtrees that have not yet been searched. It stops either when it finds a complete solution or no ancestor has unsearched paths down the tree.

The nodes of the tree are generated by ***node expansion.*** The node expansion operation, when applied to node v either determines that v is a leaf node or generates the children of v. A node can be expanded only

if it is the root of the tree or if it is a child of some node previously expanded. The search, starting with the root, successively applies node expansion to generate the nodes of the tree until one or more leaf nodes are identified as the solution.

Finding the exit in a maze is a good example of a backtrack search. A solution to the maze problem is a partial path from the entrance of the maze to the current cell. The idea is to keep extending the path from the entrance until it leads to an exit. If the current path reaches a dead end, then backup to the last place where the path could be extended in a direction that it did not yet explore. If no such place exists, then there is no way out of the maze.

The classic example of a backtracking algorithm is the one used to solve the eight queens problem. In that problem, the goal is to place eight queens on a chessboard so that none are attacking any of the others. Solutions to this problem can be found in virtually every textbook on algorithms. The eight queens problem is an example of a problem that is non-enumerative – it stops when a single solution is found. An enumerative problem is one that seeks all solutions, for example, all satisfiable assignments of truth values to variables in a boolean formula (in conjunctive normal form.)

To each search tree of a backtrack search, we can associate a second tree, the ***solution tree***, which Quinn calls the ***state space tree***. The root of the solution tree is the empty solution; all internal nodes represent ***partial solutions***. The children of a node represent extensions to the partial solution of their parent node. Leaf nodes represent either complete solutions or ***blocked solutions***, meaning those that cannot be extended to complete solutions, like the dead ends in the maze. Searching through the tree of subproblems corresponds in a natural way to searching through the tree of partial solutions.

16.4.1 Example: Crossword Puzzle Generation

Quinn uses a different example, the problem of generating a crossword puzzle [4]. The input is the rectangular array to be filled with a crossword puzzle, an indication of which cells are blacked-out (cannot contain letters), and a dictionary of words. Figure 16.2 is an example of the initially empty crossword puzzle input. The objective is to assign letters to each cell so that every horizontal row and every vertical column of two or more cells contains a word from the dictionary. It may not be possible; it might be the case that the shape prevents this. This can be viewed as a decision problem: for the given instance of the problem, is there a crossword puzzle that satisfies the constraints? It can also be viewed as a problem that seeks a solution and returns that solution if it exists. Here, we will try to find an actual solution or report that none exists.

In a crossword puzzle, there is a set of horizontal words and a set of vertical words. Numbers are associated to cells that contain the starts of words, either horizontal or vertical or both. The geometry of the puzzle can be used to determine exactly which cells are numbered and which are not. A word is complete if all letters of the word have been assigned to the cells of the puzzle. A word is incomplete, or partial, if one or more of its letters have not yet been assigned. A solution to the puzzle is complete if every horizontal and vertical word is complete.

There are various search strategies that could be used to find a solution. One is as follows: we look for the longest incomplete word in the puzzle and try to find a word in the dictionary that extends it to a complete word. If there are two or more longest incomplete words in the puzzle, we use some strategy for ordering them, such as horizontal before vertical, then ascending order of their numbers. If more than one dictionary word extends an incomplete word, we try the first word we have not yet tried before in that location. Implicit in this description is that we have to do some record-keeping to keep track of the last dictionary word we tried for the position. If there are no dictionary words that successfully extend the word, we have reached a dead end and must backtrack, which means that we must return to the parent node in the solution tree, unless we are at the root of the tree, in which case we have discovered that there is no solution at all to this particular problem instance.

In the context of this problem, the root of the solution tree has an empty crossword puzzle. The children of the root represent all possible words that can be used to fill in one of the puzzle's longest words. In general, an internal node of the solution tree represents a state of the puzzle in which some words have been assigned, and some particular dictionary word has been used to complete some specific incomplete word in the puzzle. Thus, two children of a node differ because they represent different choices of dictionary word to complete

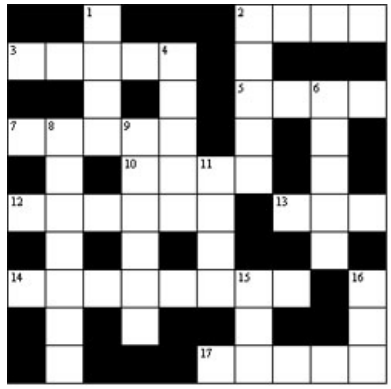


Figure 16.2: An initially empty crossword puzzle. The input to the crossword puzzle creator is the starting shape, which determines where the words start and how many words must be created.

an incomplete word in the parent. Leaf nodes represent either complete solutions in which every cell has been filled in, or incomplete solutions that cannot be extended because there are no dictionary words to complete one or more incomplete words. A solution tree has ***depth*** d if d assignments must be made to create a complete solution. If a solution tree has depth d, then any leaf node at depth d must be a solution. A leaf node at a depth less than d must represent a dead end.

16.4.2 Performance of Sequential Backtrack Search

Backtrack search in general has a worst case running time that is exponential in the depth of the solution tree, which is usually proportional to the size of the problem. To see this, let b be the ***average branching factor*** of the solution tree, which is the average number of branches from a node to its children. If the height of the tree, i.e., the depth of the deepest node in the tree, is d, then the number of nodes that must be searched in the worst case is

$$b^0 + b^1 + b^2 + \cdots + b^d = \sum_{k=0}^{d} b^k = \frac{b^{d+1} - 1}{b - 1} = \Theta(b^d)$$

which is an exponential function of d.

Although the running time is exponential in the worst case, the space requirement is not. At any given time, only a single path from the root to a leaf must be stored; the procedure only needs to expand the set of partial solutions of a node when it visits it. For some algorithms, the node expansion is only conceptual and for others it is explicit. In our crossword puzzle example, we do not have to generate the list of all possible extensions to a partial word when we generate the partial word. We only need to keep a list of those we have already tried. In the worst case, the memory associated with a node's expansion is proportional to the branching factor of the node. Therefore, the algorithm only requires an amount of memory proportional to the depth of the tree, which is proportional to the problem size, or to the maximum branching factor, whichever is greater. Therefore, the limiting factor in the size of problem that can be solved is the running time, not the memory.

16.5 Parallel Backtrack Search

We take the viewpoint here that we are developing a parallel backtrack search algorithm to improve the running time of a particular sequential backtrack search algorithm, and therefore we assume that we know the performance of the sequential backtrack search algorithm.

Before we look at possible parallelizations, we do a bit of complexity analysis. Suppose that Y is the solution tree for the particular problem. Suppose that Y has n nodes and that the deepest node is at depth d. Suppose

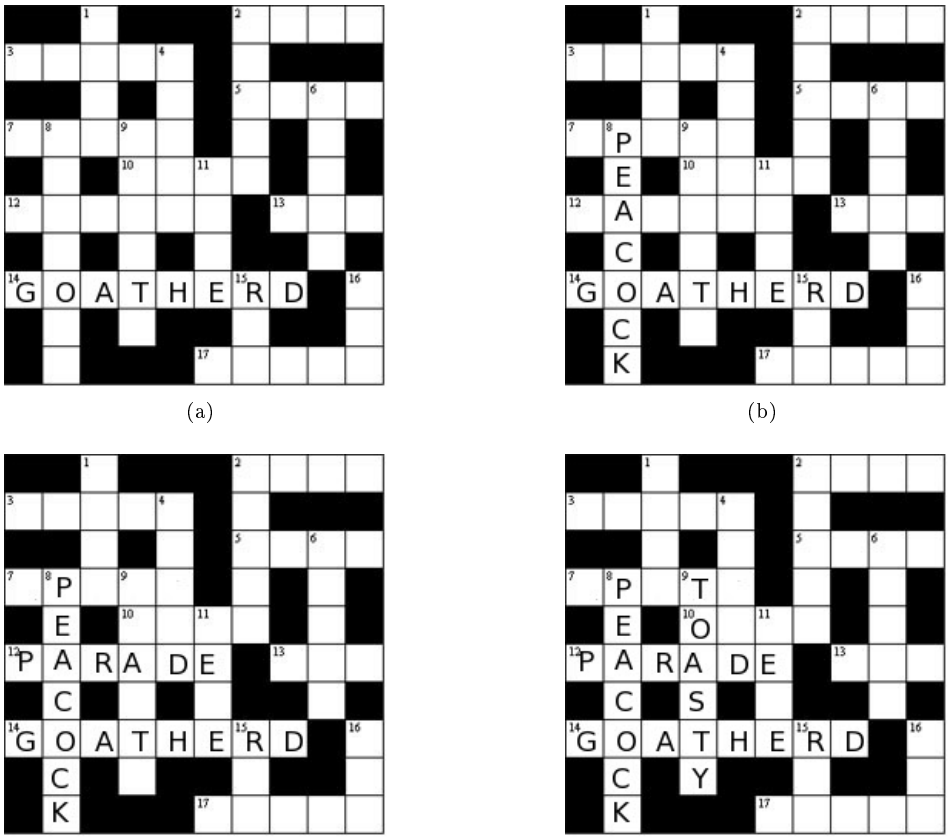


Figure 16.3: Nodes in the solution tree to the software puzzle problem. (a) is the first child node to be generated because the longest word to be filled in is word 14 across. (b) is a child of node (a) because 8 down is the second longest word. The next node to be generated is (c) and then (d). If our dictionary does not have a word that can complete 7 across, then the algorithm would backtrack up to the state of the solution represented by node (c) and would generate a different choice of word to complete 9 down.

the number of processes is p. In the worst case every node must be examined before a solution is found. This implies that there will be at least one process that must examine n/p nodes, so the worst case running time is at least n/p. However, it is also true that in the worst case, the deepest node in the tree must be examined, implying that the worst case running time is at least d. Therefore, any parallel backtrack algorithm that solves this problem will have a worst case running time of $\Omega(n/p + d)$. This is a lower bound on the running time of any algorithm. Any algorithm that can achieve this bound is therefore optimal.

The best way to understand how we can parallelize a backtrack search of a tree is to start out with a simple version of the problem. Let us assume for simplicity that every node has the same branching factor b. Let us further assume that the number of processes is a power of b, say $p = b^m$ for some $m > 0$. The levels of a tree of depth d are numbered 0, 1, 2, ..., d. At level k there are b^k nodes. The searches of any two subtrees are independent of each other in the sense that they do not need data from each other and have no need to synchronize, except when it is time to terminate, which we discuss later. Therefore, it makes sense to assign to each process a subtree whose root is at level m, because there are $p = b^m$ processes. Figure 16.4 depicts this assignment when $p = 9$ for a ternary solution tree.

In order for each process to search its own subtree starting at level m, all processes must descend the tree from the root until they reach depth $m - 1$, expanding the internal nodes on the path as they descend the tree. In the figure, for example, processes 0, 1, and 2 would each have to expand the root node, and then the leftmost child of the root node, and processes 3, 4, and 5 would each have to expand the root node and

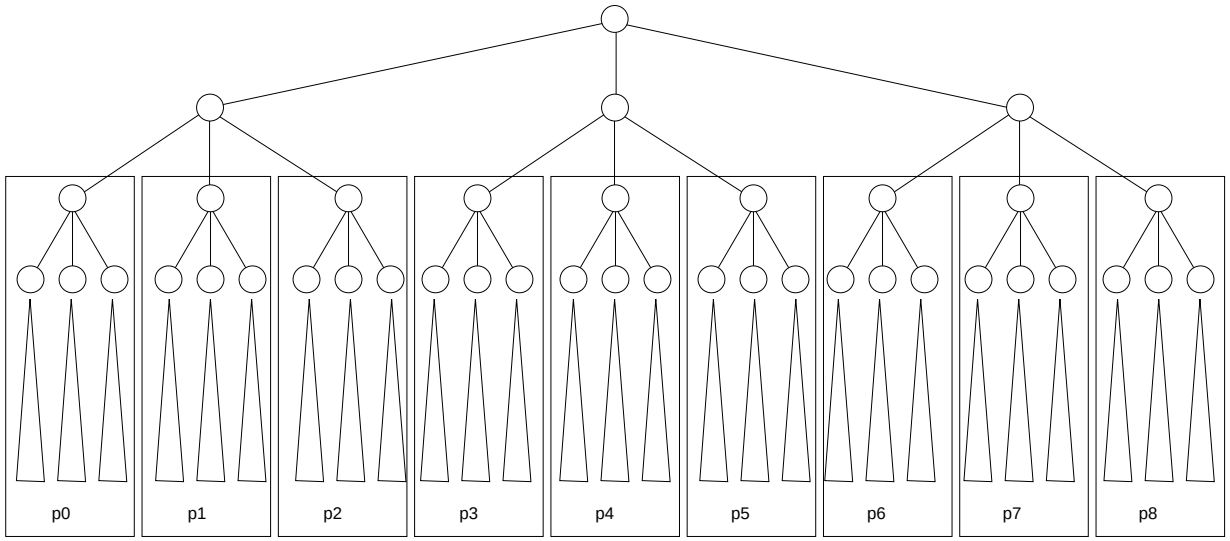


Figure 16.4: A search tree with a branching factor of 3, and a set of 9 processes that can search its subtrees in parallel. In order for the 9 processes to process their search trees in parallel, each must start the search from the root of the tree and expand every node until they reach level 2, which is the root of their search trees. Until they reach level 2, they are doing redundant work, as each is expanding the same set of nodes.

the middle child of the root. This implies that there is a certain amount of redundant work until they all reach a node at the level at which they can begin their parallel searches.

When the tree has a uniform branching factor b but the number of processes is not of the form b^m for any m, then the subtrees can be divided more evenly among the processes by descending far enough down into the tree so that the partitioning is relatively uniform. For example, if $p = 7$, then rather than distributing the 9 subtrees at level 2 to the 7 processes, which would require giving 2 subtrees to 2 processes, and 1 to the others, we could descend to level 3, where there are 27 subtrees, and give 4 subtrees to all processes except one, which would get 3. Descending the tree deeper before the parallel search starts increases the amount of redundant code that is executed, but as long as the search tree is deep enough, this is more than compensated for by the more even load balancing. Furthermore, if the tree is very deep, the amount of redundant code is very small in comparison to the number of nodes in each subtree being searched in parallel. For example, if the depth of the tree is 20, the branching factor is 5, and the tree is searched redundantly until level 10, i.e., halfway down, you might think at first that this is pretty far down the tree to go before the parallel search begins, but the number of nodes in the tree of depth 10 rooted at level 0 is $12,207,031$ whereas the total nodes in the tree is $119,209,289,550,781$; the fraction is 0.0000001. The point is that if the number of processes is large enough, there is much more work they can do in parallel while balancing their load by descending further in the tree.

It is more often the case than not that the branching factor is not uniform, and that the subtrees are of significantly different sizes. In this case, some other strategy must be used for partitioning the problem among the processes. The crossword generation problem is a good example of this, because some choices of words will lead to dead ends very quickly whereas others will not. The choices that lead to early dead ends are the roots of small subtrees.

One easy way to decrease load imbalance is to start the parallel search deeper in the tree. If the search starts deeper in the tree, say at level h where $h \gg m$, there will be many more subtrees than processes and each process will handle many subtrees. This increases the probability that the total number of nodes that each process will search will be about the same.

An alternative strategy, which may also be used in conjunction with starting the search deeper in the tree, is to use an interleaved strategy for selecting the subtrees at the level where they should search in parallel. If there are p processes and n nodes in the level in which the parallel search begins, then a cyclic decomposition of the nodes in this level will assign to process r every p^{th} node starting with node r, i.e.

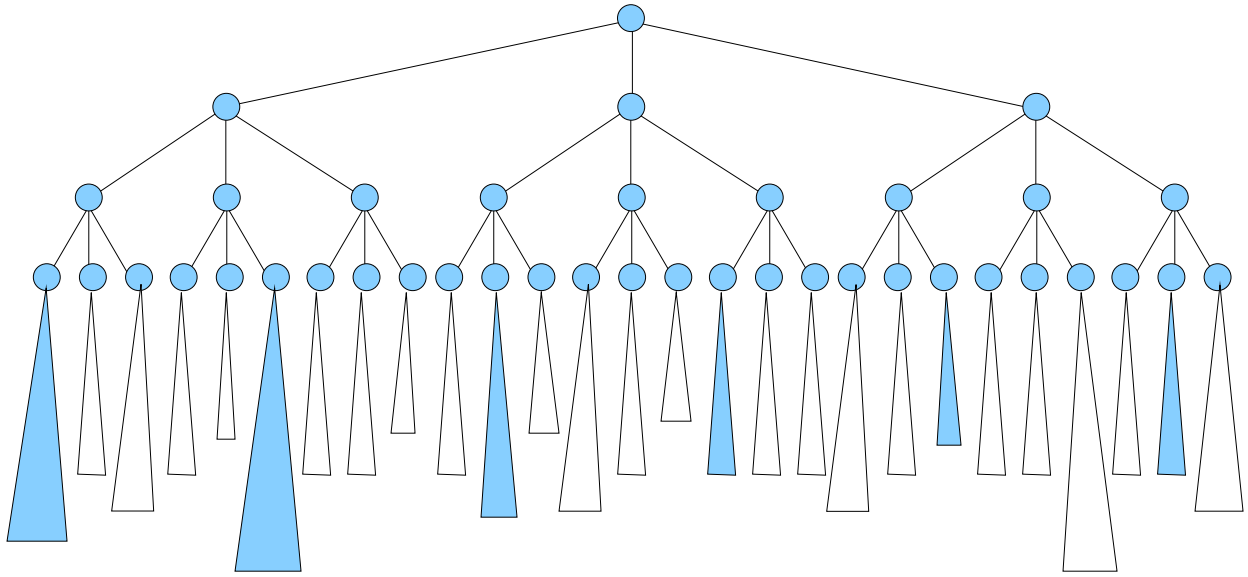


Figure 16.5: Parallel backtrack search using interleaving on a search tree with varying subtree sizes, with 5 processes. In this example, processes start the parallel search on level 3. The highlighted nodes and subtrees represent the nodes of the tree that a single process, say process 0, would search. Every process would search all of the nodes on levels 0, 1, and 2 because they all have to reach level 3 and expand all nodes on that level.

nodes r, $r+p$, $r+2p$, $\ldots, r+kp$, where kp is the largest multiple of p less than n. An interleaved strategy has the potential to improve the load balance for the same reason that it does in the various algorithms for processing arrays, because the distribution of the uneven work load within the tree might be concentrated in adjacent subtrees, and the interleaving helps to distribute the bigger trees to more processes. This is the strategy proposed by Quinn. This algorithm is a backtrack enumeration algorithm, because it searches for *all solutions*, not just one. We summarize it here.

1. All processes execute the same code; it is an SPMD algorithm, suitable for implementing in MPI.
2. The function `parallel_backtrack` is a recursive function that contains the backtrack logic and is passed three parameters:
   (a) a parameter named `board` that encapsulates all of the updates to the partial solution. The name is chosen to suggest that it could be the state of a board in a board game. Initially the board is an empty solution, and as the tree is descended, the board is updated to contain more and more of the solution. The board parameter can be thought of as a pointer to a structure containing this information.
   (b) an integer parameter named `level` that indicates the level of the tree in the current invocation of this recursive function.
   (c) the process rank, `id`.
3. The function accesses various variables that are global in the program but visible only to the individual process, i.e., they are private to the process. The use of global variables is just a convenience so that the function does not have to carry a large list of parameters. Some of these variables are unchanged by the function, and some are. In the following list, `[in]` means it is an input and remains unchanged, `[inout]` means its value is used and it is also changed. The global variables are:

`int cutoff_depth`	`[in]`	the level in the tree at which nodes represent the roots of subtrees that should be searched in parallel.
`int cutoff_count`	`[inout]`	a count of the number of nodes at the `cutoff_depth` that have been visited by the process.
`int depth`	`[in]`	the predetermined depth of the tree where the search will stop.
`int moves[depth]`	`[inout]`	an array that stores the move made at each level of the tree while searching for a solution.
`int p`	`[in]`	the total number of processes.

4. The call `parallel_backtrack(board, level, id)` does the following:

   (a) It first checks whether `level` is the depth where the search of the tree should stop, i.e., whether `level == depth`.

   (b) If it is, then the board is checked to see if it is complete solution. If it is, then the sequence of moves that resulted in this solution is printed, and the function returns.

   (c) Otherwise, if `level == cutoff_level`, then this implies that the search is at the level of the tree at which the process has to count how many nodes are at this level and only search those subtrees whose roots at this level are numbered id, $id + p$, $id + 2p$ and so on. This requires two steps:

      i. increment `cutoff_count`

      ii. check whether `cutoff_count mod p == id`. If so, then it is a node whose subtree should be searched by this process. If not, the process can ignore the node, which basically means that the function returns, in order to back up the tree.

   If the function did not return, then it needs to process the subtree whose node it has just visited. The algorithm uses a function `count_moves(board)`, which, given the current state of `board`, determines how many different possible partial solutions can extend `board` in this state. The algorithm also uses a function `make_move(board,i)` which, given `board` and the index of a valid move, `i`, updates the state of the board with the partial solution associated with the index `i`. It also uses a function `undo_move(board,i)` which restores the change made to the state by `make_move(board,i)`. After `make_move()` is called, the function is called recursively. When the recursive call returns, `undo_move()` allows the current invocation of the function to try the next possible move, whether or not the recursive call succeeded. This is how it explores the entire subtree. The logic is thus

```
num_possible_moves = count_moves(board);
for ( i = 0; i < num_possible_moves; i++ ) {
    make_move(board,i);      // update the board to reflect the move made.
    moves[level] = i;        // update moves to record the move that was made.
    parallel_backtrack(board, level+1);  // recursively process subtree.
    undo_move(board, i);     // restore the state of the board to what it was
                             // before the move.
}
```

5. It is important to understand that the backtrack may back up the tree to a level above the `cutoff_depth`, and then descend to that level again, like hail in a hailstorm. The different recursive sub-calls might descend and then backtrack, but no node at the cutoff depth is counted twice by any process, because each recursive call examines a different node.

The pseudocode is given in Listing 16.1 below.

Listing 16.1: Pseudocode for interleaved parallel backtrack of a search tree.

```

/* Global Per-process Variables */
```

```
int  cutoff_count;  /* This counts the number of nodes at the cutoff_depth that
    have been visited by the process. */
int  cutoff_depth;  /* This is the level of the tree at which the parallel search
    starts. */
int  depth;         /* This is the predetermined depth of the tree where the
    search will stop. */
int  moves[depth];  /* an array that indicates which moves were made at each level
int  p;             /* total number of processes */

/* Assume Board_State is a reference or pointer to a board state object */

void parallel_backtrack( Board_State board, int level, int id)
{
    if ( level == depth ) {
        if ( board represents a solution to the problem ) {
            print_solution(board, moves);
        }
        return;
    }
    else {
        if ( level == cutoff_level ) {
            cutoff_count++;
            if ( id == cutoff_count % p )
                return;  // not my subtree
        }
        num_possible_moves = count_moves(board);
        for ( int i = 0; i < num_possible_moves; i++ ) {
            make_move(board, i);
            moves[level] = i;
            parallel_backtrack(board, level+1, id);
            undo_move(board,i);
        }
        return;
    }
}
```

16.6 Distributed Termination Detection

The algorithm in Listing 16.1 finds every solution to the given problem, but some algorithms only need to find a single solution and then terminate. In this case, as soon as a single process finds a solution, all processes should be made to halt as well. This problem is known as the ***termination detection problem***, and the question is how to solve it correctly. It is easy to make a mistake and end up with an algorithm that deadlocks or fails to terminate properly because of the way in which the termination messages are passed.

Suppose that one process finds a solution and we want to terminate the program as soon as possible after this happens. The other processes might be anywhere in their code when the event occurs. In a UNIX environment, one could use the UNIX signal mechanism, write signal handlers, and create fairly complex code, but a simpler method exists that will usually not add very much additional delay to the termination of the entire set of processes. This alternative is a synchronous solution in which each process has a point in its code at which it periodically checks for messages from other processes, in essence asking the question, has any process sent a message that it has found a solution? If a process finds a message that a solution has been found, it exits, and otherwise it continues. The problem is how a process can check whether a message has been delivered to it without actually waiting for a message. The MPI library provides a function, `MPI_Iprobe`, that does just this: it checks whether a message has been sent to the process without causing the process to block while waiting for one. The *C* syntax is

```
#include <mpi.h>
```

```
int MPI_Iprobe(int source,        /* Source rank or MPI_ANY_SOURCE */
               int tag,           /* Tag value or MPI_ANY_TAG      */
               MPI_Comm comm,     /* Communicator handle           */
               int *flag,         /* Message-waiting flag (true if message available)*/
               MPI_Status *status /* Address of status object or MPI_STATUS_IGNORE */
)
```

The `MPI_Iprobe` function allows a program to check for incoming messages without actually receiving them and without blocking. If on return, the `flag` argument is `true`, a message is available and the program can then decide how to receive it, based on the information returned by the function in the `status` object. By passing `MPI_ANY_SOURCE` as the first argument and `MPI_ANY_TAG` as the second, the process indicates that it is interested in knowing whether any message has been sent by any process. This suggests that we can modify the algorithm described in Listing 16.1 by terminating a process whenever any of the following occurs:

- the process has found a solution and sent a message to the other processes;
- the process has received such a message from some process;
- the process has finished searching its portion of the solution tree.

Suppose we modify the pseudocode in Listing 16.1 by sending a message when a solution is found, and then having the process call `MPI_Finalize` to terminate, as follows:

```
if ( level == depth ) {
        if ( board represents a solution to the problem ) {
            print_solution(board, moves);
            send a message to all processes that a solution has been found;
            call MPI_Finalize;
        }
        return;
    }
```

We would also need to modify the code to check for messages. We could do this in the for-loop that repeatedly tries new moves and tests whether they lead to solutions:

```
for ( int i = 0; i < num_possible_moves; i++ ) {
    check for termination messages;
    if a termination message is received, then
        call MPI_Finalize;
    else {
        make_move(board, i);
        moves[level] = i;
        parallel_backtrack(board, level+1, id);
        undo_move(board,i);
    }
}
```

We would also make appropriate changes to terminate the process when the initial call

```
parallel_backtrack(board, 0, id);
```

returns. Is this correct? Suppose that a process p_m finds a solution and sends a message to all processes and immediately calls `MPI_Finalize` and then, shortly after, process p_k also finds a solution before it has checked for new messages, and tries to send a message saying that it has found a solution. The attempt to

send a message from process p_k will fail because process p_m has called `MPI_Finalize`, which results in a run-time error.

No process can be allowed to call `MPI_Finalize` if there are other processes that are active or might be sending or receiving messages. The problem is how a set of processes can coordinate their mutual termination. This problem is called the ***distributed termination detection problem***. Algorithms to solve this problem were published independently by Gouda in 1981 [3] and by Dijkstra, Feijen, and Gasteren in 1983 [2]. Both algorithms are based on arranging the processes into a ring and passing a token around the ring in a specific way, but they differ in what the token contains and how it is passed. The Quinn textbook [4] describes an algorithm that is a hybrid of the two methods but attributes the algorithm to Dijkstra, Feijen, and Gasteren[2]. Here we present the algorithm from [2] as it was originally published, because it is simpler.

16.6.1 Preliminaries.

1. A ***token*** is a message that has no content other than a ***color*** property, which is either black or white. Thus, it is just a message that stores a two-valued variable named `color`.
2. The processes are numbered $p_0, p_1, \ldots, p_{N-1}$ and are arranged in a ring for the purpose of passing the token. Specifically, p_0 can only pass the token to p_{N-1}, and for $j > 0$, p_j can only pass it to p_{j-1}.
3. Passing or receiving a token is not a message-passing action; the token is not considered to be a message. Sending or receiving messages refers to messages other than tokens.
4. A process is either ***active*** or ***passive***. Being active means that the process can send a message to another process, or it can become passive spontaneously. A passive process remains passive unless it receives a message, in which case it becomes active.
5. It follows that if all processes are passive, then the collection of processes is stable and remain passive because there is nothing that can make any passive process become active. This indicates that they can all terminate. If any process is active, it implies that it can activate any passive process by sending it a message, implying that no process can safely terminate while any process is active.
6. Every process will have a ***color*** property as well, and like tokens, it is either black or white.

We can now state the rules that govern how the Dijkstra-Feijen-Gasteren algorithm works. The fact that it is distributed means that there is no centralized control and that each process follows the rules described below. The only exception is that process p_0 is the distinguished process and is responsible for initiating the passing of the token around the ring and deciding whether the system is quiescent, implying that processes can terminate.

16.6.2 The Protocol

1. If a process p_{i+1} is active and is passed the token, it holds it; when it is passive it passes it to the next process in the ring, p_i. An active process eventually becomes passive and so does not hold the token indefinitely.
2. A process that sends a message (not a token) to any other process changes its color to black.
3. When a process passes the token along, if the process is black it makes the token black, and if it is white, it passes the token without changing its color. In short, the new color of the token is black if either it was black or the process was black, and otherwise it is white.
4. When process p_0 receives the token, if the token is black, or process p_0 is black, it makes itself white and the token white and initiates a new probe by passing the token to process p_{N-1}. If p_0 is white and the token is white, it broadcasts a messsage that all processes can terminate.

[2]His is not the only publication to do this. The algorithm he describes is called the Dijkstra & Safra algorithm and was published in 1999 [1].

5. When a process passes the token to the next process in the ring, it makes itself white.

This algorithm must be modified for the backtrack algorithm, because a process can be in one of several states:

- it has found a solution to the problem and has sent a message to that effect;
- it is still searching for a solution;
- it is no longer searching but has not found a solution.

The message traffic is fairly simple on the other hand because

- processes that are busy searching their trees do not send any messages;
- when a process finds a solution, it sends a "solution-found" message to the root process, p_0; (this is the only type of message, other than the passing of the token, that is sent to any process) and
- the only process that ever receives a message is p_0 (that is, until p_0 initiates a probe.)

We can still use colors to describe the algorithm. Each process is either still searching the tree or not. A process is colored black if it is still searching the tree, and is white if it has stopped searching. The token will have two properties, its color and a flag. When the token is white, it will imply that a solution to the problem has been found, and the flag will be ignored. When it is black, it will imply that it is being used to query the status of all processes, and the flag will be used. The protocol is therefore described by the following rules.

1. All processes are initialized to the color black (because they are searching their trees.)
2. When a process finds a solution or finishes searching its tree, it makes itself white.
3. A process other than p_0 that finds a solution sends a solution-found message to p_0.
4. If process p_0 receives a solution-found message, or if it finds a solution itself, if it has the token, it makes the token white and passes it to p_{N-1} and if it does not have the token, it waits for its return and then makes it white and sends it to p_{N-1}.
5. If process p_0 finishes searching its tree and has not found a solution, it makes the token black, sets the token flag to 0, and passes it to p_{N-1}.
6. When a white token returns to p_0, every process has been notified that it should stop searching; p_0 then sends a termination message to all processes.
7. When a black token returns to p_0, p_0 inspects its flag. If the token flag is 0, it sends a termination message to all processes, and otherwise, at least one process is still searching, so it resets the flag to 0 and passes it to p_{N-1} again.
8. When a process p_i, $i \neq 0$, receives a white token and is busy searching (and is therefore white), it stops searching, makes itself black, and passes the token to p_{i-1}. If the process was already black it just passes the token to p_{i-1}.
9. When a process p_i, $i \neq 0$, receives a black token, if the process is black (because it is still searching), it makes the token flag 1 and passes it to p_{i-1}. If the process is white, it just passes the token along without changing its flag.
10. No process can call a function to terminate, such as `MPI_Finalize`, until it receives a termination message from process p_0.

These rules will ensure that early termination of a process because it has found a solution will result in termination of all processes as soon as the token can be passed around the ring, and that if no process finds a solution, each will be allowed to search its entire tree until all have searched their trees, and no process will try to terminate by calling a library function until all processes are ready to terminate. This allows for the scenario in which one process finds a solution while others have finished searching their trees without success, and that process needs to notify the root process of its success. The rules also prevent more than one token from being passed around the ring at a time.

The only remaining problem is how we integrate these rules into the algorithm. The root process, p_0, will have to execute a different path through the code than the other processes, since it is acting as a coordinator of the group. We provide a pseudocode algorithm for process p_0 in Listing 16.2 below. The other processes have a much simpler implementation. A single function could be written for all processes, but the Listing makes it easier to understand what process p_0 must do.

Listing 16.2: Process 0 algorithm for recursive parallel backtrack with distributed termination detection.

```
/* Constants */
const int  SOLUTION_FOUND    = 1;
const int  NO_SOLUTION_FOUND = 0;
const int  CUTOFF_LEVEL; /* level of the tree at which parallel search starts.*/
const int  MAX_DEPTH;    /*  depth of the tree where the search will stop.    */


/* Types  */
typedef struct
{
    int color;
    int flag;
} TokenType;

/* Functions and types that depend on the particular application:
   Board_State is a reference or pointer to a board state object
   count_moves() counts the numebr of moves that can be made in a given board
   state.
   make_move() makes a move by altering the state of the board.
*/


/** black_token_probe()  sends a black token around the ring to detect whether any
     processes are still searching or all have stopped. It repeatedly does this
    until either all processes have stopped and a black token is returned with its
    flag set to 0, or it is interrupted by the receipt of a solution-found message,
     in which case it sends a white around the ring to stop remining processes from
     searching. In either case, it will issue a termination-message broadcast.  */
void black_token_probe()
{
    int        can_terminate = 0;
    TokenType  token;

    while (  !can_terminate ) {
        token.color = black;
        token.flag  = 0;
        send token to p-1;
        wait for the token to return by calling receive from process 1;
        if ( 1 == token.flag  ) {
            /* some processes are still searching. Check if a solution-found
                message is available */
            check whether there is a solution-found message;
            if ( there is a solution-found message )  {
                retrieve the message;
```

```
                record information about solution;
                token.color = white;
                send token to process p-1;
                wait for the token to return by calling receive from process 1;
                can_terminate = 1;
            }
            /* back to top of loop to resend token */
        }
        else { /* no process is searching and no solution found */
            can_terminate = 1;
        }
    }
}


/** parallel_backtrack()
  This is a recursive function that behaves differently depending on the
  level of the search tree at which it is called. This algorithm is strictly
  designed for process 0 to execute.

  Only invocations at the cut-off level will listen for incoming solution-found
  messages and then send white probes around the ring.
  Invocations below the cutoff level will return an indication of whether they
  found a solution, without listening for messages or checking for any probes.

  If the process fails to find any solutions, it sends a black token around the
  process ring.  It only knows it has failed to find a solution at the root
  level of the tree; therefore only the root level  invocation can send a black
  token around the ring.

  If the process  finds a solution, it implies that it descending to a leaf node
  at level = MAX_DEPTH, and then recursed back up the tree.  If the cutoff_level
  is above the leaf, then it will pass through that level on the way back up
  the tree, and when it does, it will send a white token around the ring to
  tell the remaining processes to stop searching and prepare to receive a
  termination-message broadcast.
*/

int parallel_backtrack( Board_State board, int level, int id)
{
    static    int  cutoff_count = 0;  /* number of nodes at the cutoff_level that
        have been visited by the process. */
    static    int  moves[MAX_DEPTH];  /* an array that indicates which  moves were
         made at each level */

    int         solution_found   = NO_SOLUTION_FOUND;
    int         can_terminate;
    int         token_is_out;
    int         num_possible_moves;
    int         i;
    TokenType   token;

    if ( level == MAX_DEPTH ) {
        if ( board represents a solution to the problem ) {
            print_solution(board, moves);
            return SOLUTION_FOUND;
        }
        else
            return NO_SOLUTION_FOUND;
```

```
    }


    if ( level != CUTOFF_LEVEL ) {
        /* level != CUTOFF_LEVEL; just expand node and check subtrees
           for solutions */
        num_possible_moves = count_moves(board);
        i = 0;
        solution_found = NO_SOLUTION_FOUND;
        while ( i < num_possible_moves && !solution_found ) {
            make_move(board, i);
            moves[level] = i;
            solution_found = parallel_backtrack(board, level+1, id);
            if ( !solution_found ) {
                undo_move(board,i);
                i++;
            }
        }
        if ( level == 0 ) {
            /* Process 0 has returned to the top of the tree. If it found a
                solution, that was intercepted before it reached level 0, so if we
                reach here, it means it did not find a solution and needs to
                initiate a probe with a black token. */
            black_token_probe();
            /* when this returns, either a solution message was found or the black
                 token eventually made it around the ring. In either case we can
                tell all processes to terminate. */
            broadcast message to terminate all processes;
            call MPI_Finalize;
        }
        else
            /* level is below 0, so just pass the solution_found result up to the
                caller without intercepting it. */
            return solution_found;
    }
    else { /* level == CUTOFF_LEVEL */
        cutoff_count++;
        if ( id != cutoff_count % p )
            return NO_SOLUTION_FOUND;

        /* Expanding nodes and listening for solution-found messages */
        num_possible_moves = count_moves(board);
        i = 0;
        solution_found = NO_SOLUTION_FOUND;
        token_is_out = 0;
        while ( !solution_found && !can_terminate && i < num_possible_moves ) {
            if ( token_is_out  )  {
                /* check whether token has returned */
                if ( the token has returned ) {
                    token_is_out  = 0;
                    issue a terminate broadcast to all processes;
                    can_terminate = 1;
                }
            }
            else { /* no token is out */
                check whether there is a solution-found message;
                if ( there is a solution-found message )  {
                    retrieve the message;
                    record information about solution;
```

```
                    token.color = white;
                    send token to process p-1;
                    token_is_out = 1;
                }
                else  { /* no messages, no tokens, so do some work */
                    make_move(board, i);
                    moves[level] = i;
                    solution_found = parallel_backtrack(board, level+1, id);
                    if ( solution_found  ) {
                        token.color = white;
                        send token to process p-1;
                        token_is_out = 1;
                    }
                    else {
                        undo_move(board,i);
                        i++;
                    }
                }
            }
        } /* end of while loop */
        if ( can_terminate )
            call MPI_Finalize;
        else
            /* no solution was found by this process and no solution-found
               messsage was received. The loop exited because all moves by
               this process were searched without success. Just return
               0. */
            return NO_SOLUTION_FOUND;
    }
}
```

16.7 To be continued...

References

[1] Edsger W. Dijkstra. Shmuel safra's version of termination detection. In M. Broy and R. Steinbruggen, editors, *Proceedings of the NATO Advanced Study Institute on Calculational System Design*, pages 297–301, 1999.

[2] Edsger W. Dijkstra, W.H.J. Feijen, and A.J.M. van Gasteren. Derivation of a termination detection algorithm for distributed computations. *Information Processing Letters*, 16(5):217–219, 1983.

[3] Mohamed G. Gouda. Distributed state exploration for protocol validation. Technical report, Austin, TX, USA, 1981.

[4] M.J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Higher Education. McGraw-Hill Higher Education, 2004.

Subject Index

AND-node, 2
AND-tree, 2
AND/OR-tree, 2
average branching factor, 5

backtrack search, 3
blocked solution, 4

combinatorial optimization problem, 2
Combinatorics, 1

decision problem, 2
dichotomous search, 2
distributed termination detection problem, 12
Divide-and-conquer, 2

enumerative method, 3

node expansion, 3

OR-node, 2
OR-tree, 2

partial solution, 4
problem instance, 1

search space, 1
search tree, 2
solution tree, 4
state space tree, 4

termination detection problem, 10