Appendix A
Search Procedures and Computational Complexity

A.1 Nondeterministic Problem Solving

A problem-solving procedure is a computational procedure whose input is some problem P that has some (possibly empty) set of solutions. The procedure may *succeed*, in which case it returns some value v that purports to be a solution for P; or it may *fail*, in which case it either does not terminate or else returns some special value such as failure to indicate that it cannot solve the problem.

Most of the problem-solving procedures in this book are *nondeterministic*, i.e., they involve nondeterministic choices, which we will write in the pseudocode for our procedures as

```
nondeterministically choose $v \in V$
```

where V is some finite set.[1]

If a procedure does not involve such a nondeterministic choice, then it is *deterministic*. There are several theoretical models of nondeterminism [133, 206, 428]. Probably the easiest one to understand intuitively is based on an analogy to parallel computing, with an unlimited supply of CPUs. If b is the number of elements in V, then we start b CPUs running in parallel, one for each value of V. Each CPU will have a different execution trace. If one or more solution traces succeed, then the nondeterministic procedure may immediately terminate and return a solution found by any successful execution trace. Otherwise, the nondeterministic procedure fails.

In nondeterministic search procedures, the nondeterministic choice will be embedded in a loop or a recursive call, so that the procedure makes a sequence

1. Note that these nondeterministic choices are different from the nondeterministic actions that are introduced briefly in Chapter 1 and are the focus of Chapter 17. The multiple meanings for "nondeterministic" are unavoidable because the terminology was developed independently in two different fields.

of nondeterministic choices. Because each nondeterministic choice corresponds to spawning a set of parallel CPUs, the set of execution traces can be represented as an *execution tree* in which each node represents one of the iterations or recursive invocations of the procedure, the children of each node represent the subprocesses spawned by the procedure at this point of its iteration or recursion, and each path in the tree represents an execution trace. This tree is called the procedure's *search tree* or *search space.*

Here are some of the properties that a problem-solving procedure may have.

- *Soundness.* A deterministic procedure is sound if, whenever it is invoked on some problem P and returns a value $v \neq$ failure, v is guaranteed to be a solution for P. A nondeterministic procedure is sound if every successful execution trace returns a value that is guaranteed to be a solution to P.
- *Completeness.* A deterministic procedure is complete if, whenever it is invoked on a solvable problem P, it is guaranteed to return a value $v \neq$ failure. A nondeterministic procedure is complete if, whenever it is invoked on a solvable problem P, at least one of its execution traces will return a value $v \neq$ failure whenever P is solvable.
- *Admissibility.* If there is some measure of optimality for solutions to a problem P, then a deterministic procedure is admissible if it is guaranteed to return an optimal solution whenever P is solvable. Note that if a procedure is admissible, then it is also sound and complete.

A.2 State-Space Search

A well-known class of search problems are *state-space search problems.* The *state space* is a set of nodes called *states,* and the objective is to find a state s that satisfies some *goal condition* g. The set of all states is not given at the outset. Instead, an initial state s_0 is given, and the other states are generated as needed, by applying *state-space operators* to existing states. The problem specification is a triple (s_0, g, O), where s_0 is the *initial state,* O is the set of *operators,* and g is the *goal condition.* The set S of all possible states in the state space is defined recursively as follows: $s_0 \in S$; and if s is a state and o is an operator such that $o(s)$ is defined, then $o(s)$ is also a state. Each state $o(s)$ is called a *child* of s. Figure A.1 shows a nondeterministic state-space search procedure.

Any practical implementation of a nondeterministic search procedure must be deterministic rather than nondeterministic; thus it must have a control strategy for visiting the nodes in the execution tree. In the following subsections we discuss several well-known control strategies.

Breadth-First Search. The primary advantage of breadth-first search is that if a nondeterministic procedure p is complete, then the breadth-first version of p will

```
State-space-search($s, g, O$)
    if $g(s)$ then return $s$
    applicable $\leftarrow$ {all operators applicable to $t$}
    if applicable $= \emptyset$ then return failure
    nondeterministically choose $o \in$ applicable
    $s' \leftarrow o(s)$
    return State-space-search($s', g, O$)
```

Figure A.1 Nondeterministic state-space search procedure.

also be complete. However, in most cases the breadth-first search procedure will have a huge space requirement. For example, suppose that every node of the search space has b children, the terminal nodes are at depth d, the time needed to visit each node is $\Theta(1)$, and the space needed to store each node is also $\Theta(1)$. Then the running time for a breadth-first search will be $\Theta(b^d)$, and because a breadth-first search must keep a record of each node visited, then the memory requirement will also be $\Theta(b^d)$.

Depth-First Search. If a nondeterministic search procedure is complete, then the corresponding depth-first search will be complete if the search space contains no infinite paths or if the procedure can reliably detect and prune every infinite path. If there are infinitely many possible different nodes, then there is usually no good way to accomplish this. However, if there are only finitely many possible different nodes, then it can be accomplished by keeping track of the nodes on the current path and backtracking whenever the search procedure generates a node that is already in that path.

In the worst case, a depth-first search may need to examine the entire search space before finding a solution; then its running time may be worse than that of a breadth-first search. However, if the search procedure has a good heuristic for selecting which of a node's children to visit first, it may be able to find a good solution very quickly. In this case, its running time will usually be much better than that of a breadth-first search.

Because a depth-first search keeps track of only the nodes on the current path (plus possibly the siblings of those nodes), its space requirement is $\Theta(d)$, where d is the depth of the deepest node it visits.

Best-First Search. In some cases, the objective is to find a goal state s that minimizes some objective function $f(s)$. In this case, a nondeterministic search will not work properly: instead of returning the solution found in a single execution trace, we must look at *all* of the execution traces to see which one leads to the best solution.

One way to do this deterministically is to use a best-first search. This is like a breadth-first search in the sense that it maintains an *active list* of nodes that have been generated but not yet visited. However, instead of using this list as a queue the way a breadth-first search does a best-first search instead uses the list as a priority queue: the next node chosen from the active list will be the one whose f value is smallest. Best-first search is sound. If f is monotonically nondecreasing, i.e., if $f(s) \leq f(s')$ whenever s' is a child of s, then a best-first search will never return a nonoptimal solution and is admissible in finite search spaces. If there is a number $\delta > 0$ such that if $f(s) + \delta \leq f(s')$ whenever s' is a child of s, then a best-first search is admissible even in infinite search spaces.

The well-known A* search procedure is a special case of best-first state-space search, with some modifications to handle situations where there are multiple paths to the same state.

Depth-First Branch-and-Bound Search. Another search procedure for minimizing an objective function is branch-and-bound. The most general form of branch-and-bound is general enough to include nearly all top-down search procedures as special cases [415]. The best-known version of branch-and-bound is a simple depth-first version similar to the procedure shown in Figure A.2. In this procedure, s^* is a global variable that holds the best solution seen so far, with s^* equal to some dummy value and $f(s^*) = \infty$ when the procedure is initially invoked. If the state space is finite and acyclic and if f is monotonically nondecreasing, then Depth-first-BB is admissible. If f is nonmonotonic, then Depth-first-BB may not be admissible, and if the state space is infinite, then Depth-first-BB may fail to terminate.

Greedy Search. A greedy search is a depth-first search procedure with no backtracking. It works as follows. If s is a solution, then return it; otherwise, repeat the search at the child $o(s)$ whose f value is smallest. There are no guarantees of whether this procedure will find an optimal solution, but it can sometimes save

```
Depth-first-BB($s, g, f, O$)
    if $g(s)$ = true and $f(s) < f(s^*)$ then
        $s^* \leftarrow s$
        return $s$
    applicable ← {all operators applicable to $s$}
    if applicable = ∅ then return failure
    for every $o \in$ applicable
        Depth-first-BB($o(s), g, f, O$)
    return $s^*$
```

Figure A.2 A branch-and-bound version of state-space search.

a huge amount of time over what would be needed to find a guaranteed optimal solution.

Hill-Climbing Search. This is similar to greedy search, except that in a hill-climbing problem, every node is a solution, and a hill-climbing procedure will only go from s to $o(s)$ if $f(o(s)) < f(s)$.

Most of the above search procedures can be modified using a variety of heuristic techniques. These can be divided roughly into two classes.

1. *Pruning techniques.* These are ways to determine that some nodes will not lead to a solution (or to a desirable solution), so that the procedure can prune these nodes (i.e., remove them from the search space).
2. *Node-selection techniques.* These are ways to guess which nodes will lead to desirable solutions (e.g., solutions that are optimal or near-optimal, or solutions that can be found quickly), so that the procedure can visit these nodes first.

In general, pruning and node-selection heuristics will not necessarily preserve completeness or admissibility. However, there are a number of special cases in which these properties can be preserved.

Iterative deepening is a technique that can be used in conjunction with a depth-first search procedure to make it complete. It can be done in two ways: breadth-first or best-first.

Breadth-first iterative deepening does a depth-first search that backtracks whenever it reaches depth i and repeats this search for $i = 1, 2, \ldots$ until a solution is found. Like ordinary breadth-first search, breadth-first iterative deepening is complete but not admissible.

Best-first iterative deepening does a depth-first search that backtracks during its ith iteration whenever it reaches a node s such that $f(s) \geq f_i$, where $f_0 = f(s_0)$, and for $i > 0$,

$$f_i = \min\{f(s) \mid \text{the search backtracked at } s \text{ during iteration } i-1\}$$

As before, it repeats this search for $i = 1, 2, \ldots$ until a solution is found. The well-known IDA* procedure uses best-first iterative deepening.

The admissibility properties of best-first iterative deepening are the same as for breadth-first search. If f is monotonically nondecreasing, then best-first iterative deepening will never return a nonoptimal solution and is admissible in finite search spaces. If there is a number $\delta > 0$ such that if $f(s) + \delta \leq f(s')$ whenever s' is a child of s, then best-first iterative deepening is admissible even in infinite search spaces.

For either kind of iterative deepening, if the time taken by the depth-first search grows exponentially at each iteration, then the total time taken by the iterative-deepening procedure is proportional to the time needed by the last depth-first search. Mathematically, if the ith depth-first search takes time $T(i) = \Omega(2^i)$ for each i, then the iterative-deepening procedure takes time $\Theta(T(n))$, where n is the total number of iterations.

A.3 Problem-Reduction Search

Another kind of search space is a *problem-reduction space*, in which each state s represents a problem to be solved, and each operator $o(s)$ produces not just a single child state s' but an entire set of children $\{s_1, \ldots, s_k\}$ (the number of children may vary from one state to another). The children are called *subproblems* of s, and a solution for one of them represents a portion of the solution to s. Thus, to solve s it is not sufficient just to find a solution below some descendant of s. Instead, the search space is an AND/OR graph, and a solution for s consists of a set of solutions $\{v_1, \ldots, v_k\}$ that are the leaf nodes of a *solution graph* rooted at s.

Figure A.3 gives an example of a solution graph in an AND/OR graph. Each AND-branch is indicated by drawing an arc to connect the edges in the branch. In some cases, the order in which the subproblems are solved may matter, so the

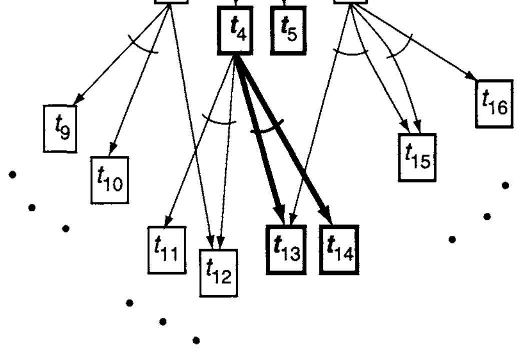

Figure A.3 A portion of an AND/OR graph. The AND-branches include $(t_0, \{t_1, t_2, t_3\})$, $(t_0, \{t_4, t_5\})$, $(t_0, \{t_6, t_7\})$, $(t_0, \{t_8\})$, $(t_3, \{t_9, t_{10}\})$, and so forth. If t_5, t_{13}, and t_{14} satisfy the goal condition, then the portion of the graph shown in boldface is a solution graph.

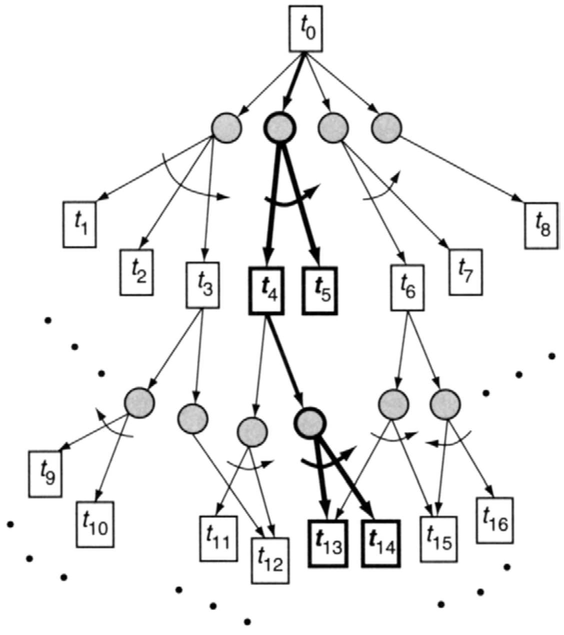

Figure A.4 An AND/OR graph similar to Figure A.3, but this time the OR-branches are shown explicitly, and orderings are imposed as indicated with arrows for each of the AND-branches.

ordering may be indicated by putting arrows on the arcs, as shown in Figure A.3. OR-branches may be indicated either implicitly by having more than one AND-branch coming out of a node, as shown in Figure A.3, or explicitly by introducing additional nodes and edges into the graph, as shown in Figure A.4.

```
Problem-reduction-search($s, g, O$)
    if $g(s)$ then return $s$
    *applicable* $\leftarrow$ {all operators applicable to $t$}
    if *applicable* $= \emptyset$ then return failure
    nondeterministically choose $o \in$ *applicable*
    $\{s_1, \ldots, s_k\} \leftarrow o(s)$
    for every $s_i \in \{s_1, \ldots, s_k\}$
        $v_i \leftarrow$ Problem-reduction-search($s_i, g, O$)
        if $v_i =$ failure then return failure
    return $\{v_1, \ldots, v_k\}$
```

Figure A.5 Nondeterministic problem-reduction search procedure.

Figure A.5 shows a nondeterministic procedure for searching a problem-reduction space. Instead of corresponding to a path through a state space, each successful execution trace of this algorithm corresponds to a solution graph of the AND/OR graph.

Several of the search strategies discussed earlier can be adapted to work on AND/OR graphs. However, the details are complicated and are not discussed here.

A.4 Computational Complexity of Procedures

Suppose $p(s)$ is a computational procedure that takes a single argument s. Then we will let $T(p, s)$ and $S(p, s)$, respectively, be the running time and the space requirement for $p(s)$. If $p(s)$ is nondeterministic, this is the minimum running time and space requirement of any of $p(s)$'s execution traces. If $p(s)$ does not terminate (or if $p(s)$ is nondeterministic and none of its execution traces terminates), then $T(p, s)$ and $S(p, s)$ are undefined.

In any implementation of p, the input s will be a string of symbols. For example, if s is a number, then it might be expressed as a sequence of binary digits or as an ASCII character string, or if s is a classical planning problem, it might be expressed as a string of symbols that define the initial state, the goal, and the planning operators. Let $|s|$ be the length of such a string. Then the procedure p's worst-case running time and space requirement are defined, respectively, as follows:

$$T_{\max}(p, n) = \max\{T(p, s) \mid |s| = n\}$$

$$S_{\max}(p, n) = \max\{S(p, s) \mid |s| = n\}$$

Complexity analyses of $T_{\max}(p, n)$ and $S_{\max}(p, n)$ normally focus not on the exact values of $T_{\max}(p, n)$ and $S_{\max}(p, n)$ but instead on their "big O" values. Intuitively, $O(f(n))$ is the set of all functions that grow no faster than proportional to $f(n)$. Mathematically, a function $f(n)$ is in the set $O(g(n))$ if there are numbers c and n_0 such that:

$$\forall n > n_0, 0 \leq f(n) \leq cg(n)$$

It is standard usage to write $f(n) = O(g(n))$ to mean $f(n) \in O(g(n))$.

A function $f(n)$ is said to be *logarithmically bounded* if $f(n) = O(\log n)$; *polynomially bounded* if there is a constant c such that $f(n) = O(n^c)$; and *exponentially bounded* if there is a constant c such that $f(n) = O(c^n)$.

Example A.1 Suppose that by analyzing p's behavior, we can show that $T_{\max}(p, n)$ is a polynomial of the form $a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0$. Then from the previous definition, it follows that $T_{\max}(p, n) = O(n^k)$.

Thus in this case $T_{\max}(p, n)$ is polynomially bounded. ∎

A.5 Computational Complexity of Problems

Traditional analyses of the complexity of computational problems (e.g., [206, 428]) have focused on *language-recognition problems.* Mathematically, a *language* is a set $\mathcal{L}$ of character strings over some finite alphabet; i.e., if L is a finite set of characters and L^* is the set of all strings of characters of L, then $\mathcal{L}$ may be any subset of L^*. The language-recognition problem for $\mathcal{L}$ is the following problem:

Given a character string s, is $s \in \mathcal{L}$?

A computational procedure p is a *recognition procedure* for $\mathcal{L}$ if for every string $s \in L^*$, $p(s)$ returns yes if $s \in \mathcal{L}$, and $p(s)$ does not return yes if $s \notin \mathcal{L}$. In the latter case, $p(s)$ either may return another value such as no or may fail to terminate.

For a language-recognition procedure, the worst-case running time and space requirement are defined slightly differently than before:

$$T_{\max}(p, n, \mathcal{L}) = \max\{T(p, s) \mid s \in \mathcal{L} \text{ and } |s| = n\}$$
$$S_{\max}(p, n, \mathcal{L}) = \max\{S(p, s) \mid s \in \mathcal{L} \text{ and } |s| = n\}$$

In other words, if p is a procedure for recognizing a language $\mathcal{L}$, then we do not care how much time or space $p(s)$ takes if $s \notin \mathcal{L}$. To readers who have never seen it before, this may seem counterintuitive, but it has some theoretical advantages. For example, if p is a procedure that is not guaranteed to terminate when $s \notin \mathcal{L}$, then $T_{\max}(p, n)$ and $S_{\max}(p, n)$ will not always be defined.

If $\mathcal{L}_1$ is a language over some alphabet L_1 and $\mathcal{L}_2$ is a language over some alphabet L_2, then a *reduction* of $\mathcal{L}_1$ to $\mathcal{L}_2$ is a deterministic procedure $r : L_1^* \to L_2^*$ such that $s \in \mathcal{L}_1$ iff $r(s) \in \mathcal{L}_2$. This definition is useful because if r is a reduction of $\mathcal{L}_1$ to $\mathcal{L}_2$ and p is a procedure for recognizing $\mathcal{L}_2$, then the composite procedure $p(r(s))$ recognizes $\mathcal{L}_1$ because it returns yes iff $r(s)$ is in $\mathcal{L}_2$, which happens iff $s \in \mathcal{L}_1$.

If r's worst-case running time is polynomially bounded, then we say that $\mathcal{L}_1$ is *polynomially reducible* to $\mathcal{L}_2$. In this case, the worst-case running time and space requirement of $p(r(s))$ are at most polynomially worse than those of $p(s)$. This means that we can recognize strings of $\mathcal{L}_1$ with only a polynomial amount of overhead beyond what is needed to recognize strings of $\mathcal{L}_2$.

Here are some complexity classes dealing with time.

- P is the set of all languages $\mathcal{L}$ such that $\mathcal{L}$ has a deterministic recognition procedure whose worst-case running time is polynomially bounded.

- NP is the set of all languages $\mathcal{L}$ such that $\mathcal{L}$ has a nondeterministic recognition procedure whose worst-case running time is polynomially bounded.
- EXPTIME is the set of all languages $\mathcal{L}$ such that $\mathcal{L}$ has a deterministic recognition procedure whose worst-case running time is exponentially bounded.
- NEXPTIME is the set of all languages $\mathcal{L}$ such that $\mathcal{L}$ has a nondeterministic recognition procedure whose worst-case running time is exponentially bounded.

Here are some complexity classes dealing with space.

- NLOGSPACE is the set of all languages $\mathcal{L}$ such that $\mathcal{L}$ has a nondeterministic recognition procedure whose worst-case space requirement is logarithmically bounded.
- PSPACE is the set of all languages $\mathcal{L}$ such that $\mathcal{L}$ has a recognition procedure whose worst-case space requirement is polynomially bounded. It makes no difference whether the procedure is deterministic or nondeterministic; in either case we will get the same set of languages.
- EXPSPACE is the set of all languages $\mathcal{L}$ such that $\mathcal{L}$ has a recognition procedure whose worst-case space requirement is exponentially bounded. It makes no difference whether the procedure is deterministic or nondeterministic; in either case we will get the same set of languages.

It can be shown [428] that

$$\text{NLOGSPACE} \subseteq \text{P} \subseteq \text{NP} \subseteq \text{PSPACE} \subseteq \text{EXPTIME} \subseteq \text{NEXPTIME} \subseteq \text{EXPSPACE}$$

Some of these sets are known to be unequal (e.g., P $\neq$ EXPTIME), but it is unknown whether all of them are unequal (e.g., the question of whether or not P = NP is the most famous unsolved problem in the entire theory of computing).

If C is one of the complexity classes and $\mathcal{L}$ is a language, then $\mathcal{L}$ is *C-hard* if every language in C is reducible to $\mathcal{L}$ in polynomial time. $\mathcal{L}$ is *C-complete* if C is $\mathcal{L}$-hard and $\mathcal{L} \in C$. Intuitively, if $\mathcal{L}$ is C-complete, then $\mathcal{L}$ is one of the hardest languages in C: if we can recognize $\mathcal{L}$, then we can also recognize any other language in C with at most a polynomial amount of additional overhead.

A.6 Planning Domains as Language-Recognition Problems

In order to discuss the computational complexity of planning problems, the problems must be reformulated as language-recognition problems. There are several standard ways to do this. Given an alphabet L in which to write statements of planning problems, we can define the following languages.

- PLAN-EXISTENCE is the set of all strings $s \in L^*$ such that s is the statement of a solvable planning problem.
- PLAN-LENGTH is the set of all strings of the form (s, k) such that s is the statement of a solvable planning problem, k is a nonnegative integer, and s has a solution plan that contains no more than k actions.

The definition of PLAN-LENGTH follows the standard procedure for converting optimization problems into language-recognition problems (cf. [206, pp. 115–117]). What really interests us, of course, is not the problem of determining whether there is a plan of length k or less but the problem of finding the shortest plan. If the length of the shortest plan is polynomially bounded, then it can be shown that the two problems are polynomially reducible to each other. However, if the length of the shortest plan is not polynomially bounded, then finding the shortest plan can be much harder than determining whether there is a plan of length k or less. For example, in the well-known Towers of Hanoi problem [5] and certain generalizations of it [248], the length of the shortest plan can be found in low-order polynomial time—but actually producing a plan of that length requires exponential time and space because the plan has exponential length.

A.7 Discussion and Historical Remarks

Some of the best-known heuristic-search algorithms include the A* algorithm [426] for searching state-space graphs, the HS [380] and AO* [426] algorithms for searching AND/OR graphs, the alpha-beta algorithm [328, 426] for searching game trees, and a variety of branch-and-bound algorithms [44, 415]. All of these algorithms can be viewed as special cases of a general branch-and-bound formulation [415]. Breadth-first iterative deepening was first used in game-tree search [468]. Best-first iterative deepening was first used in Korf's IDA* procedure [335]. For further reading on heuristic search, see Pearl [430] and Kanal and Kumar [307].

For further reading on algorithms and complexity, see Cormen *et al.* [133] and Papadimitriou [428]. Also, Garey and Johnson [206] is a compendium of a large number of NP-complete problems.