C^{++} Search Class Library

Peter Bouthoorn

3 December 1993

1 Introduction

One of our daily activities is solving problems of any kind. AI-research has shown that a lot of these problems can equally well or sometimes more easily be solved by computer programs. To be able to write such a problem-solving program it is necessary to give an exact description of the problem to be solved and to know how it can be defined in terms that facilitate the translation of the problem into a computer program. In this paper we first explain the theory of problem-solving in AI and next describe an implementation of some of the ideas presented (the description of the implementation, called the search class library, will be quite rough, because we will concentrate on how the search class library must be used. For details on the implementation see the comments accompanying the source code).

2 Problem representation and search techniques

2.1 State space representation and problem reduction

In this section, we will describe two methods commonly used in problem representation. As a sample problem the 8-puzzle will be used. The 8-puzzle consists of 8 numbered, movable tiles set in a 3 X 3 frame. One of the cells of the frame is always empty, which makes it possible to move the tiles.

A sample 8-puzzle is given in fig. 1. Consider the problem of transforming the first configuration into the second one, our goal.
To solve this puzzle we try out various moves producing new configurations until we produce the goal configuration. To give a more formal definition of this problem we say that we are trying to reach a certain *goal* configuration

2	1	6		1	2	3
4	0	8		8	0	4
7	5	3		7	6	5

Figure 1: The 8-puzzle.

starting with an *initial* configuration by using some set of *operators*. An operator transforms one configuration into another, in the case of the 8-puzzle it is most natural to think of 4 such operators, each corresponding to moving the empty tile: move empty tile left, move empty tile right, move empty tile up, move empty tile down.

What we just have done is defining the problem of solving the 8-puzzle in terms of a *state space search*. In a state space search the object is to reach a certain goal *state* starting with an initial state. In the case of the 8-puzzle the start and goal state are the configurations given in fig. 1. More generally we can say that every configuration we produce when trying to solve the 8-puzzle corresponds to one state in the state space. All these states together, i.e., *all possible* configurations make up the state space. It is possible of course to define the state space without explicitly enumerating all states. Indeed, most of the time this is impossible and the state space will be defined implicitly by providing rules specifying how each state can be derived from another. The state space may be small as in the case of the 8-puzzle, but for most every day problems or other board games it is quite large (e.g., in chess the total number of possible board configurations, this is the total number of possible states, equals roughly 10^{120}). Obviously it would be impossible to explore the entire state space and often this is not needed, because we are interested in finding only one solution to a problem, i.e., only one path leading from the start state to the goal state. This means that we do not have to search the state space *exhaustively*, but a small(er) portion instead. The problem of course is, which portion?

But before we are going to discuss this point it will be helpful to look somewhat further at the approach we have described so far. We said that to solve a problem it is necessary to represent the problem as a state space search. That is, we define a start state, a goal state and a set of operators that transform one state into another. The actual search consists in moving around in the state space, looking for a path from the initial state to the goal state. In this case the search process proceeds forward because we start with the initial state and move towards the goal state. Hence it is called a *forward reasoning system*. The opposite behaviour is also possible: a system starting the search with the goal state and moving backward to the initial state. In this method, often called *backchaining* we *reason backward* from the goal states. These two techniques can be combined, resulting in a *bidirectional search*. In the case of the 8-puzzle it does not make much difference if we move forward or backward, because about the same number of paths will be generated in either case, but some problems can be solved more efficiently when searching in one direction rather than the other (see Rich (1983), p.58).

A different technique, or rather a different way to represent problems, that has not been mentioned so far is that of *problem reduction*. In this type of representation each operator used may divide the problem into a set of sub-problems that each have to be solved seperately. Additionally, there may be restrictions on the order in which these sub-problems have to be solved[1]. The object of problem reduction is to eventually produce a set of *primitive problems* whose solutions are regarded as trivial: at this stage the process of dividing problems into sub-problems halts.

These two approaches, state space search and problem reduction, are two of the most common methods used in problem representation, although variations of these approaches, as used in, e.g., game-playing are also possible (see Barr(1981), p.84ff., Ritch(1983), p.113ff., Nilsson(1971), p.137ff).

2.2 Trees and graphs

So far we have seen that a state space representation consists of the following components:

- The state descriptions.
- A start state, describing the situation from which the problem-solving process may start.
- A goal state, describing an acceptable solution to the problem.
- A set of operators describing how to transform one state into another.

Also, we said that to solve a problem we do not need to search the entire state space but only that part which leads to a solution, i.e., we need to search for an appropriate operator sequence, transforming the initial state

[1]In the search class library it is assumed that sub-problems must be solved in the order in which they are generated.

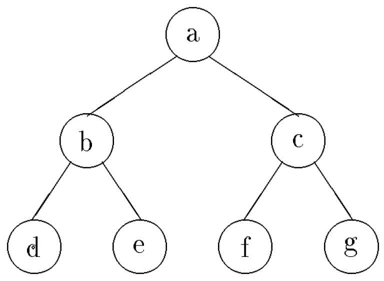

Figure 2: Start of a search tree

through a number of intermediate states into the goal state. To perform this search systematically we need a control strategy that decides which operator to apply next during the search. These strategies are commonly represented using *trees*[2]: construct a tree with the initial state as its root, next generate all the offspring of the root by applying all of the applicable operators to the initial state, next for every leaf node generate its successors by applying all of the applicable operators, etc. When these steps are performed a structure as displayed in fig. 2 structure will arise. In this representation every leaf node corresponds to a state, with the root node representing the initial state. Each operator application is represented by a connection between two nodes.

Trees are a special case of a more general structure called a *graph* [3]. A tree is a graph each of whose nodes has a unique parent (except for the root node, which has no parent). Searching a tree is easier than searching a graph, because when a new node is generated in the tree we can be sure it has not been generated before. This is true because every node has only one parent, so there cannot be two or more different paths leading to the same node. In a graph, however, nodes usually have more than one parent. Therefore, when searching a graph one should make provisions to deal with these situations. Saying that a node has more than one parent means that the node is generated by a different sequence of the same operators. That is, the same node may be part of several paths, and continuing the processing of both these nodes (which are really the same node) would be redundant and a waste of effort. This can be avoided at the price of additional bookkeeping.

[2]See Knuth(1979), p.305ff. for the concept of trees in computer science.

[3]See Nilsson(1971), p.22, Barr(1981), p.25/26, Ritch(1983), p.63

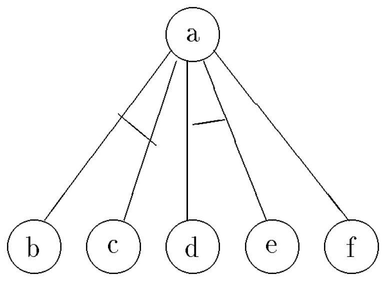

Figure 3: A structure showing alternative sets of subproblems for A.

Instead of traversing a tree we traverse a *directed graph*[4]: every time a node is generated we examine the set of nodes generated so far to see if this node already exists in the graph. If it does, we throw it away (note: see page 8 for exceptions to this rule), if not, we add it to the graph.

This way we will also avoid a related problem: if we did not check whether a node had been generated before, the search process would very likely end up in a *cycle*, in which the same set of nodes is generated over and over again. For instance, when we apply operator 'move empty tile left', next 'move empty right' and next move 'empty tile left' etc. to the 8-puzzle, the problem-solving process would go on producing the same nodes without end. When we modify the search procedure as described above, this situation will never arise, because we try to look up every node that is generated, before it is added to the graph.

A special kind of graph is the *AND/OR graph* that is used in problem-solving methods involving problem reduction. In the case of a normal graph, each node represents a different alternative state to be chosen next and the search process may continue along one of these nodes arbitratily. In a problem-reduction representation, however, we also need to deal with operators that divide the original problem into a set of sub-problems *each* of which need to be solved, instead of any of them. For example, suppose problem A can be solved either by solving problems B and C or by solving problems D and E or by solving problem F. This situation is depicted in fig. 3. In fig. 3 the nodes which form a set that has to be solved entirely are indicated by a special mark linking their incoming arcs. It is usual, however,

[4]Knuth(1979), p.371.

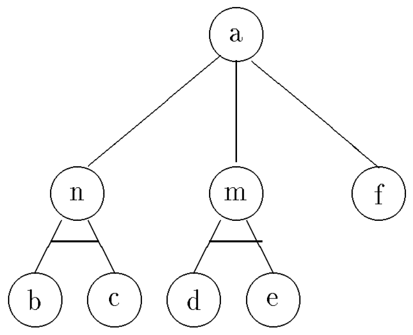

Figure 4: An AND/OR graph

to introduce some extra nodes into the structure so that each set containing more than one successor problem is grouped below its own parent node. With this convention the structure of fig. 3 becomes as shown in fig. 4. In this figure the added nodes labelled N and M serve as exclusive parents for sets {B,C} and {D,E}, respectively. This way one can think of N and M and F as *OR* nodes, because any of them may be solved to solve node A. Problem N, however, is reduced to a single set of sub-problems B and C, and *each* of these sub-problems must be solved to solve N. For this reason nodes B, C, D and E are called *AND* nodes. In fig. 4 AND nodes are indicated by a mark on their incoming arcs.

2.3 Basic search methods - depth-first, breadth-first

In section 2 we described the process of generating a search tree, in this section we will give a more precise description of this process.

- A start node is associated with the initial state description.
- The successors of a node are generated by applying all of the applicable operators to the state description associated with the node. We will call this procedure the *expansion* of a node.
- Pointers are setup from each successor back to its parent node. These pointers indicate the solution path in the game tree, leading from the goal node, once it is finally found, back to the start.

- Every successor node is checked to see if it is a goal node. When a goal node has been found the process of expanding nodes finishes and we trace back the solution path through the pointers.

These are the basic elements of a problem-solving process, but the order in which nodes are to be expanded is still left open. We may choose, for instance, to search one entire branch of the tree before examining nodes in the other branches. Alternatively, we may decide to expand all nodes that are on the same level, in different branches. The first option would result in what is called a *depth-first* search: the most recently generated node gets expanded first. In a *breadth-first* search, the second option, nodes are expanded in the order in which they are generated.

2.4 Finding an optimal solution, uniform-cost search

It should be noted that for some problems we are not interested in finding *any* solution, but rather the *optimal* or *best* one. What 'best' means depends on the problem at hand, but for now we will call a solution path optimal if it contains the least possible number of nodes leading to the goal node. Later on we will refine our definition to include a different, but related type of problems. In the case of a depth-first search we cannot guarantee that the best solution will be found. This is because every branch is examined seperately, so if the search process finds a goal node in one branch it will terminate. But it may well be that a better solution is located in a different branch. Breadth-first search on the other hand is guaranteed to find the shortest path, because it expands all nodes on one level before advancing to the next.

For some problems, however, finding the best solution does not mean finding the shortest path, but rather the *cheapest* path. This is true for instance when we need to find the shortest path from one city to another. In this case there may be several routes that can be used to get from city A to city B, visiting other cities along the way. Now suppose the cities are nodes in a search tree, clearly what we want is not the smallest number of nodes (cities) that make up a path from A to B, but the shortest route. To solve problems like this one we need to associate *costs* with the arcs in the tree (in this case the costs will represent the distances between the cities). The object is to find a path having the least cost.

A more general version of the breadth-first method, called the *uniform-cost search method* is guaranteed to find a path of minimal cost from the start node to a goal node. Instead of expanding paths of equal length like the breadth-first method, this method expands paths of equal cost. To compute the cost of a path s to a node n we will use the function *g(n)*. The cost associated with a node will consist of the cost associated with its parent plus the cost of getting from the parent to this node. Using this method to order the set of nodes we are sure the uniform-cost method expands nodes in order of increasing g(n).

One should note that the graph search technique that we described earlier must be modified when we are looking for an optimal solution. We said that during a graph search every node that is generated twice can simply be thrown away. If we would use this technique in a uniform-cost search we could never guarantee to find the cheapest path. As said, in a graph there may be multiple paths leading to the same node. But each of these has its own (possibly different) cost. So if we throw away a node without paying attention to this fact we may be missing a better solution without ever noticing. Therefore, the graph search procedure must be modified in the following way: every time a new node is generated we check whether it already exists in the graph. If not, we add it. If it does, we compare the cost of the old node and the cost of the newly generated node. If the old node is better (cheaper) nothing has to be done, if it is worse we change its cost and direct its pointer to the parent on the least costly path that has just been found.

2.5 Blind search vs heuristic search, best-first search

All of the methods described so far are called *blind-search procedures* because they do not use any specific information about the problem to be solved, i.e, the search process just continues until it happens on a solution: it is not directed in some way to the goal. The advantage of blind-search procedures is that they are easy to implement and may find a solution quickly for small problems. The obvious disadvantage of these methods is that they may be led astray, expanding a lot of nodes that are not part of the solution path. For example, when traversing a tree in depth-first order we dive into the left most branch, this is fine if we happen to find a solution there. But suppose the goal node is located in a different part of the tree, e.g., at the right most branch. In this case we will have searched the entire tree before getting on the right track. It would have been much better if we had known in advance which way to go. But of course, this is impossible; if we had known this we would never have to *search* for a solution. Still, there may be situations where we do not know exactly where to go, but can give an estimate of how far a node is removed from the goal node and hence determine if it is on the (best) solution path. Using this information it is possible to improve the efficiency of the search process. Here, we have introduced the idea of using a *heuristic*.

A heuristic is a rule of thumb, a technique that improves the efficiency of a search process, possibly by sacrificing claims to completeness [Rich 1983, 35]. This means that, like all rules of thumb, heuristics may lead the search in the most promising way, finding a solution quickly, but also that they may take a wrong turn (but still leading to the goal) or lead to deadends. A good way to use heuristic information is by means of a *heuristic function* that evaluates every node that is being generated, i.e. that determines the goodness or badness of a node. Using a heuristic function it will be possible to conduct the search in the most profitable direction, by suggesting which path to follow first when more than one is available.

We will define a heuristic function *f(n)* being the sum of two components *g(n)* and *h(n)*:

$$f(n) = g(n) + h(n)$$

Function g(n) is the same as the one described in section 2.4: a measure of the cost of getting from the initial state to the current node. The function h(n) is an estimate of the additional cost of getting from the current node to a goal state. Put differently, h(n) is the function in which the real heuristic knowledge is imbedded. Using f(n) we are able to order the set of nodes waiting for expansion, by convention this will be done this in *increasing* order. An algorithm can then be used to select the node having the smallest f(n) value next for expansion. One of the methods that uses this technique is the *best-first search method* or *A^* algorithm*.

It is important to keep in mind that most heuristics are imperfect and that, inevitably, the search process will be affected by this. In general, a search algorithm is called *admissible* if for any graph it terminates in an optimal path to a goal whenever a path exists. Using a heuristic function to conduct the search process we cannot always make this claim because the behaviour of the search will depend on how accurately the function evaluates nodes. If we use a perfect heuristic function we are guaranteed to find an optimal solution, but heuristics having this property are hard to find. Furthermore, the difficulty of computing the function's result affects the total computational effort of the search process. Also, it may be less important to find a solution whose cost is absolutely minimal than to find a solution of reasonable cost within a reasonable amount of time. In this case one may prefer a heuristic function that evaluates nodes more accurately in most cases, but sometimes overestimates the distance to the goal, thus resulting in an inadmissible algorithm. Most of the time we need to make this sort of compromise: the efficiency of the search process needs to be improved at the sacrifice of admissibility (see Barr(1981), p.65/66, Nilsson(1971), p.59ff.)

3 The search class library

3.1 Why C^{++}?

One of the things to be explained will be the reason why we decided to use C^{++} for programming an AI-type problem while so many specialized AI programming languages are available. For one thing, we wanted to know how much more effort it would be to use a lower level programming language (lower compared to AI programming languages like Prolog, that is) to program this type of problem. C^{++} seemed excellent for this job because it is based on C, a third generation programming language, and also because it follows the object oriented paradigm, meaning that it supports a higher level of abstraction, in the case of OOP: the combination of procedural and data abstraction. But the main reason why we decided to use C^{++} is that it supports *inheritance*. This feature makes it possible to easily make use of existing software when developing new applications. Combined with the possibility to define *virtual* functions this makes it possible to design foundation classes that are of no use in themselves, but can be easily extended for real applications.

The main objective was to seperate the problem-solving process that we described above and the representation of the problem itself. In chapter 2 we showed that a lot of problems can be solved using standard techniques, e.g., the state space representation and search. It seemed useful therefore, to develop some basic routines offering a number of search methods that could easily be used when designing problem-solving software. And this is exactly what the foundation classes are for. Each of them implements a particular search algorithm while leaving open the exact nature of the problem to be solved. So, using these classes it will be possible, just like in, e.g., Prolog, to concentrate on the representation of the problem at hand without worrying about how it has to be solved. But unlike Prolog the user need not make use of a fixed search method (in Prolog: depth-first), but may choose one that suits the problem best.

3.2 Techniques used, hierarchy of the search classes.

In this section we describe a basic technique used in the different search classes and explain how these classes relate to each other.

In writing the search engine we followed the procedure outlined in section 2.2 except that we do not build an actual tree-like structure. Instead, we use two lists, called OPEN and CLOSED. OPEN is a list containing nodes ready for expansion and CLOSED a list of those nodes that already have had the expansion procedure applied to them. The algorithm forming the search procedure consists of the following steps:

1. Put the start node on OPEN.
2. Get the first node from OPEN. If OPEN is empty exit with failure, otherwise continue.
3. Remove this node from OPEN and put it on CLOSED; call this node n.
4. Expand node n, generating all of its successors. This is done by calling function do_operator() or expand().
5. Provide every successor with a pointer back to n and pass it to function add() that may or may not put the node on OPEN.
6. Check each successor to see if it is a goal node. If so exit, otherwise go to 2.[5]

The algorithm that we just described can be found in function solve() which is a member-function of class *SEARCH_* and also of class *BISEARCH_* and of class *AOSEARCH_*. These three classes are the most important classes in the search class hierarchy, each implements basic routines needed by different search algorithms, as follows:

- SEARCH_ : basic class for uni-directional search routines.
- BISEARCH_ : basic class for bi-directional search routines.
- AOSEARCH_ : basic class for AND/OR search routines.

[5]This step is not done in the bidirectional and AND/OR search algorithms. They have a different way of determining whether the problem is solved or not.

The names of these three classes correspond to three different *kinds* of search techniques that are offered to the user to solve different kinds of problems: uni-directional, i.e., normal search, bi-directional search and AND/OR search. However, these classes should never be used for direct derivation, they must be thought of as skeleton classes that outline the overall search method. Other classes, derived from these three basic classes, implement the actual search algorithms, but before we are going to describe these classes we must first introduce another important class, class *NODE_,*.

Class NODE_ specifies a general structure that is to be processed by class SEARCH_ (and by class BISEARCH_). Class NODE_ is itself derived from class SVOBJECT_ (sortable object), which, in turn, is derived from class VOBJECT_. Class NODE_ may be thought of as an abstraction of the nodes in a search tree or, equivalently, of the states in a state space. When designing problem-solving software most time will be spent in finding a good representation of these nodes/states. Once this representation is found it must be turned into a class that is derived from class NODE_ (or from one of its derivatives, depending on the search algorithm that is used, see later), like this:

```
class PNODE_ : public NODE_
{
    ...
};
```

As said, class SEARCH_, BISEARCH_ and AOSEARCH_ are the most fundamental search classes and it should never be necessary to derive directly from these classes, but rather from one of the following classes, each derived from class SEARCH_:

- DEPTH_TREE_ and DEPTH_GRAPH_. These two classes implement a depth-first search, by creating a search tree or graph, respectively.
- BREADTH_TREE_ and BREADTH_GRAPH_. These two classes implement a breadth-first search, by creating a search tree or graph, respectively.
- UNICOST_TREE_ and UNICOST_GRAPH_. These two classes implement a uniform-cost search, by creating a search tree or graph, respectively.
- BEST_. This class implements a best-first search.

Or from one of the following classes, derived from class BISEARCH_:

- BIDEPTH_TREE_ and BIDEPTH_GRAPH_. These two classes implement a depth-first bidirectional search, by creating two search trees or graphs, respectively.
- BIBREADTH_TREE_ and BIBREADTH_GRAPH_. These two classes implement a bidirectional breadth-first search, by creating two search trees or graphs, respectively.

Or from one of the classes derived from class AOSEARCH_:

- AODEPTH_TREE_. This class implements a depth-first AND/OR search, by creating a depth-first AND/OR tree.
- AOBREADTH_TREE_. This class implements a breadth-first AND/OR search, by creating a breadth-first AND/OR tree.

To make use of any of the search algorithms that the search class library offers the user must derive a class from one of the classes above, for instance:

```
class PUZZLE_ : public DEPTH_GRAPH_
{
   ...
};
```

The first four sets of classes, DEPTH_TREE_, DEPTH_GRAPH_, BREADTH_TREE_ and BREADTH_GRAPH_ and also the the classes derived from BISEARCH_, i.e., BIDEPTH_TREE_, BIDEPTH_GRAPH_, BIBREADTH_TREE_ and BIBREADTH_TREE_ must be used in conjuntion with class NODE_. This means that when performing, for instance, a depth-first search the class that is used to represent the nodes in the search tree must be derived from NODE_ (see first example above and also demo one and demo two).

Class UNICOST_TREE_, UNICOST_GRAPH_ and BEST_ require the nodes to have some special features. They must be used in conjunction with a derivative of class NODE_, class UNI_NODE_ or class BEST_NODE_, respectively (see demo four and five for classes that are derived from one of these two).

Lastly, class AODEPTH_TREE_ and AOBREADTH_TREE_ must be used in conjunction with classes ORNODE_, a derivative from class AONODE_, which is derived from class NODE_ (see demo seven for a class derived from class ORNODE_). Another derivative from class AONODE_ is class ANDNODE_, but this class should never be used for derivation, its use will be explained later.

To summarize:

- The depth-first and breadth-first search routines (both uni-directional and bi-directional) must be used in combination with class NODE_.
- The uniform-cost search routines must be used in combination with class UNI_NODE_.
- The best-first search routine must be used in combination with class BEST_NODE_.
- The AND/OR search routines must be used in combination with class ORNODE_.

3.3 How to use the search class library.

Now that we have introduced the search classes and have outlined their hierarchy we will explain how they can and should be used. As said, class SEARCH_[6] is one of the most important and most basic classes. One of the tasks of class SEARCH_ is to keep track of the number of operators that may be applied to the nodes (in the expansion procedure). It receives this information through its constructor, therefore, every class that is derived from SEARCH_ must call SEARCH_'s constructor, passing to it the number of operators, as an integer. The node representing the initial state and the node representing the goal state must also be passed to this constructor, except when deriving from class AOSEARCH_, in this case only the start node and number of operators should be passed[7]. But as we never derive directly from class SEARCH_ we do not pass this information to SEARCH_ directly, but through one of its derivatives, using the constructor of the derived class. Suppose we build a class called PUZZLE_, derived from class DEPTH_GRAPH_, then this could be a constructor of PUZZLE_:

[6] we will take this class as an example, what we tell here applies also to class BISEARCH_ and class AOSEARCH_; important differences will be discussed later.

[7] A problem that is to be solved using the problem reduction representation, i.e., using an AND/OR search algorithm, does not have a goal state, because to solve the problem we do not look for a goal state, but need to divide the problem into sub-problems that may or may not be solvable.

```
PUZZLE_::PUZZLE_(PNODE_ *start, PNODE_ *goal)
    :DEPTH_GRAPH_(start, goal, 4)
{               // pass start node, goal node and number of
}               // of operators to DEPTH_GRAPH_ 's construc-
                // tor (that will pass them on to SEARCH_)
```

As PUZZLE_ is ultimately derived from SEARCH_ it must implement all virtual functions (in SEARCH_) that are still left undefined (i.e., that are not instantiated by one of SEARCH_'s derivatives). In the case of the DEPTH_ and BREADTH_ classes there are none. But some of the other search classes have a couple of virtual functions that must implemented by the user. Class UNICOST_TREE_ and UNICOST_GRAPH_ require the implementation of funcion compute_g() and class BEST_ of both this function and of function compute_h(). Both of these functions serve to compute a cost associated with a node: compute_g() computes the cost of getting from a node's parent to the node itself, i.e. the cost associated with the arc connecting both nodes, and compute_h() computes the heuristic value of a node (see section 2.4 and section 2.5, but note that function compute_g() must only compute the second half of g(n)):

```
int compute_g(const NODE_ &)    // computes cost of getting from
                                // node's parent to node itself
int compute_h(const NODE_ &)    // computes heuristic value of
                                // a node
```

The search classes derived from BISEARCH_ do not have any non-defined virtual functions left. But the last set of classes, AODEPTH_TREE_ and AOBREADTH_TREE_, require the implementation of function is_terminal() which checks whether a node represents a terminal node:[8]

```
int is_terminal(const AONODE_ &)    // node is terminal node?
                                    // 1 : yes, 0 : no
```

Just like class SEARCH_ class NODE_ also has a number of virtual functions that must be implemented by the user. One of the most important of these is do_operator() that is used for node expansion (step 4 in the algorithm).

```
NODE_ *do_operator(int) const    // apply operator n and
                                 // return new node or NULL
```

[8]a terminal node is a node that represents a primivite problem in a problem reduction presentation, see section 2.1

Note that the object returned by do_operator() must have been allocated in memory. Function do_operator() is called by SEARCH_::solve(), that passes do_operator() an integer representing one of the operators. This way the operators are numbered, starting at 0 and ending at number of operators minus 1. If the operator can be applied do_operator() should return a new node (allocated by new), if not, it should return NULL. This is one way a node may be expanded. Another possibility is by use of funtion expand(). This function is similar to do_operator(), except that it returns a linked list of all of its successors, instead of one successor at a time. This is useful when dealing with problems that do not use operators or that have a variable number of operators.

```
NODE_ *expand(int) const  // expand node and return all of
                          // its successors in a linked list
```

The linked list that expand() returns must be built using the next-pointer field in NODE_ (NODE_ *next). An example of how funtion expand() may be used and how to build the linked list is given in demo five.

Apart from do_operator() there are two other functions, which are virtual in class NODE_, that must be implemented. One of these, equal(), tests whether two nodes are the same, it must return 1 if true and 0 if not. The other one, display() is used to display a node:

```
int equal(const VOBJECT_ &) const  // nodes are the same node?
                                   // 1 : true, 0 : false
void display() const               // display the node
```

Note that the argument in equal() is VOBJECT_ and not NODE_, this is because equal() is inherited by NODE_ from VOBJECT_.

Using these funtions the implementation of user-defined problems should be straightforward. Still, it will be helpful to make some remarks concerning the the AND/OR search classes. In section 2.2 we explained how an AND/OR graph may be created and we introduced the concept of AND-nodes and OR-nodes. The meaning of these terms is slightly different in the implementation of the AND/OR search classes. Here we will call all nodes OR-nodes, except those nodes that connect a set of sub-problems (nodes N and M in fig. 4), which are to be called AND-nodes. This relates to the AND/OR search classes in the following way. As said, the user-definable objects that serve to represent the nodes in the search tree must be derived from class ORNODE_. But now suppose that some node A may be reduced to the set of sub-problems {B,C,D}. Normally we would call B, C and D AND-nodes, but here, as said, they are called OR-nodes. The next step is to create a node that connects nodes B, C and D and this node is called an AND-node. This AND-node must created by calling **new ANDNODE_()** and adding to this node all of its successors, in this case node B, C and D, by calling addsucc(), like this:

```
ANDNODE_ *andnode;

andnode = new ANDNODE_;       // it must be allocated
andnode->addsucc(node_a);
andnode->addsucc(node_b);
andnode->addsucc(node_c);

return(andnode);
```

When the number of successors is known in advance a different possibility is to pass this number to the constructor of ANDNODE_ and then using setsucc() to pass the successors, like this:

```
AND_NODE_ *andnode;

andnode = new ANDNODE_(3);       // we will add 3 nodes
andnode->setsucc(0, node_a);     // we start counting at 0
andnode->setsucc(1, node_b);
andnode->setsucc(2, node_c);

return(andnode);
```

The only problem that may, and most of the time will, arise is that we don't know before hand what kind of node will be produced, an AND-node or and OR-node, so we do not know if we need and AND_NODE_ or an OR_NODE_ pointer (this is especially true when building a linked list of nodes as in expand()). But this is easily solved when we use a AONODE_ pointer, because both AND_NODE_ and OR_NODE_ are derived from this class, and next casting the result if needed. An example of this technique is given in demo seven.

One thing has not been mentioned yet: how must the search be started? This is done by a call to function generate(), a member function of class SEARCH_. For example:

```
PUZZLE_
    puzzle;
....
puzzle.generate();      // start looking for a solution
```

3.4 Include and library files.

In this section we describe which files must be included and which files must be linked when developing problem solving software using the search class library.

All include files needed by the search class library are in directory /include. Most of these are used internally and the only two files the user needs to consult are tree.h and graph.h:

- tree.h. Include this file when using one of the tree search algorithms.
- graph.h. Include this file when using one of the graph search algorithms.

Library files are located in directory /lib which contains two library files (only one in UNIX):

- searchs.lib. Library containing all objects needed by the different search classes, compiled in the small memory model.
- searchc.lib. Idem, but compiled in the compact memory model.

4 Bibliography

Winston, P.H. (1984), *Artificial Intelligence* (2nd ed.), London: Addison-Wesley.
Barr, A., Feigenbaum, E.A. (1983), *The Handbook of Artificial Intelligence*, Los Altos: Kaufmann.
Nillson, N.J. (1971), *Problem Solving Methods in Artificial Intelligence*, New York: McGraw-Hill.
Nillson, N.J. (1986), *Principles of Artifial Intelligence*, Los Altos: Kaufmann.
Knuth, D.E. (1979), *The Art of Computer Programs* (2nd ed.). London:

Addison-Wesley.
Pearl, J. (1984), *Heuristics: Intelligent Strategies for Computer Problem Solving*, London: Addison-Wesley.
Rich, E. (1983), *Artificial Intelligence*, New York: McGraw-Hill.