4 AND-OR Tree Search

To grasp the notion of an 'AND-OR tree' we recall how the 'OR-tree' search problem of the previous chapter can be envisaged as an investigation into the truth of a logical expression. In this chapter we address a more general problem than that of the previous one, by redefining the class of logical expressions to be those which include L iff it satisfies one of the following:

1. L is a logical primitive.
2. $L \equiv (X \cup Y)$, where X and Y are both logical expressions.
3. $L \equiv (X \cap Y)$, where X and Y are both logical expressions.
4. $L \equiv X^{\mathrm{c}}$, where X is a logical expression.

The symbol '$\cap$' represents the customary binary Boolean 'AND' operator, and '$^{\mathrm{c}}$' the unary Boolean 'NOT' operator, both defined as usual. Any logical expression L can be written as a tree, with the internal nodes as '$^{\mathrm{c}}$', '$\cap$' and '$\cup$' operators, and the leaves as logical primitives. There are a variety of normal forms for representing logical expressions as defined above, of which I have found one particularly clear, because it emphasises the similarity with the OR-Tree model. It uses the following equivalence:

$$X \ \cap \ Y \equiv (X^{\mathrm{c}} \ \cup \ Y^{\mathrm{c}})^{\mathrm{c}}$$

We shall also exploit the associative property of the '$\cup$' operator, and so all the logical expressions in this chapter will appear in a form without

'$\cap$' operators, and with alternate levels of '$\cup$' and '$^{\mathrm{c}}$' operators. For example, the logical expression $(A \cap B^{\mathrm{c}}) \cap (C \cup D)$ would be transformed into $(A^{\mathrm{c}} \cup B \cup (C \cup D)^{\mathrm{c}})^{\mathrm{c}}$, represented in the below figure.

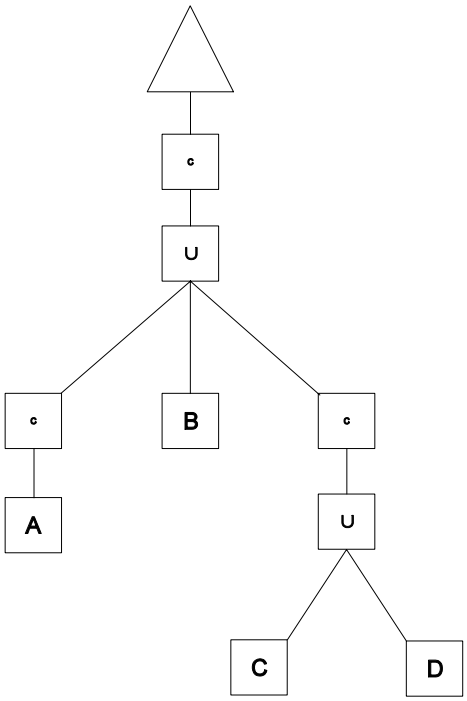

Figure 26: Sample Representation of $(A^{\mathrm{c}} \cup B \cup (C \cup D)^{\mathrm{c}})^{\mathrm{c}}$

Whilst it is well known that any such logical expressions can be transformed to one which has only the single operator 'NAND', this representation is not useful for our purposes. This is because the economy of operators is achieved by duplicating the logical primitives (e. g. $A^{\mathrm{c}} \equiv (A \text{ NAND } A)$, which brings more serious problems of its own[17].

[17]Since the two A's in the above example correspond to a single node, they share a single expansion. This representation would therefore enforce a dependence structure

4.1 Deterministic Case

By way of introduction we first consider the case in which the logical primitives are boxes. This problem was first addressed by Joyce [35]. The optimal policy may be efficiently deduced in time proportional to the overall length of the logical expression by working backwards from the tree which represents it. The leaves are simple logical primitives. Iterative application of the two steps below suffices to transform an arbitrary logical expression into an OR-tree of depth one which can then be solved in the fashion described in the Section 3.1:

1. *Sibling chunking:* This applies to logical expressions of the form $L = l_1 \cup l_2 \cup \ldots \cup l_N$, where each l_i is a simple logical primitive. This is an example of the boxes case of Section 3.1, so the optimal policy is to search the l_i in decreasing order of $\emptyset_i$. Assuming the node types are ordered by $\emptyset$, so that $\emptyset_i \geq \emptyset_j$ for $i < j$, we have the following formulae for t_L, the expected time taken to determine the truth of L, and for p_L, the probability that L is true:

$$t_L = \sum_{i=1}^{N} t_i \prod_{j=1}^{i-1} q_j \qquad\qquad p_L = 1 - \prod_{i=1}^{N} q_i$$

between the logical primitives which violates the fundamental assumption, that each logical primitive is a single search opportunity which may be explored independently of all the others.

Thus, the compound expression, $l_1 \cup l_2 \cup \ldots \cup l_N$ can be represented by a new simple logical primitive, L, with details $d_L = (t_L, p_L)$.

2. *'Complement' Removal:* This applies to logical expressions of the form L^{c}, where L is a simple logical primitive. We replace L^{c} by a new simple logical primitive, $\bar{L}$, with $t_{\bar{\mathrm{L}}} = t_L$, $p_{\bar{\mathrm{L}}} = 1 - p_L$.

4.2 Stochastic Case

We now increase the scope of the model to cater for a more general class of logical primitives. We assume that, upon search, each logical primitive may expand to any logical expression, according to a known distribution. This requires an extension of the notation used for the OR-tree model, since the space of logical expressions which include '$\cup$' as well as '$^{\mathrm{c}}$' cannot be summarised in the same fashion by a finite length vector. As an introduction to this extra notation, consider the node of type A, defined below:

$$d_A \;=\; (\frac{1}{3}, 9, \frac{1}{2} + \frac{1}{4}s_A + \frac{1}{8}s_B^2 + \frac{1}{8}s_A s_B^2)$$

We represent this as follows:

$$A[9] = \begin{pmatrix} \langle \mathrm{T} \rangle & \langle \mathrm{F} \rangle & A & B_1 \cup B_2 & A \cup B_1 \cup B_2 \\ \frac{1}{3} & \frac{1}{3} & \frac{1}{6} & \frac{1}{12} & \frac{1}{12} \end{pmatrix}$$

The special elements, $\langle \mathrm{T} \rangle$ and $\langle \mathrm{F} \rangle$ correspond to 'true' (an object is found), and 'false' (no object is found, and there are no descendants). We

shall use this form of notation wherever necessary — that is, wherever a search may give rise to an expression with a '^{c}' or '$\cap$' constructor.

As defined above, the model encompasses some problems for which the notion of 'optimal policy' is ill-defined. Consider, for example, the node type below:

$$A[1] = \begin{pmatrix} \langle \mathrm{T} \rangle & \langle \mathrm{F} \rangle & (A_1 \cap A_2) \cup (A_3 \cap A_4) \\ \epsilon & \epsilon & 1 - 2\epsilon \end{pmatrix}$$

For $\epsilon = 0$, the truth of A is obviously undecidable, since it will never expand to $\langle \mathrm{T} \rangle$ of $\langle \mathrm{F} \rangle$. For small enough positive ϵ, there is non-zero probability of the expression being undecidable, as can be understood from thinking of the search of A as a branching process. Such expressions, where the expected number of searches of the optimal policy is infinite, we term *intractable*, and do not address further.

Another class of expressions *require* only a finite number of searches, but may nevertheless be searched *ad infinitum*. As an illustration, consider the degenerate node type X, defined below:

$$X = (A \cup B) \qquad A[1] = \begin{pmatrix} A \\ 1 \end{pmatrix} \qquad B[1] = \begin{pmatrix} \langle \mathrm{T} \rangle \\ 1 \end{pmatrix}$$

4.3 Nature of the Optimal Policy

Fundamental to the simplicity of the stochastic OR-tree search model is the fact that results from one part of the search tree have no influence upon how it is optimal to explore other parts of the tree. This might seem to be a simple consequence of the independence of each individual offspring distribution, but this is a necessary, not a sufficient condition for this property. In addition to this there must also be a simplicity of structure of the search model. This is satisfied in the stochastic OR-tree search model, since it is possible to deduce whether or not the goal has been met by looking at each leaf in isolation. (If an object has been discovered at any of them, it has!)

This simplicity of structure means that each part of the tree can be considered in isolation, and implies that there is an optimal policy within the class of pre-empt/resume policies. This class contains policies that dynamically choose *which* parts of the tree to search, but not *how* to search them. An obvious first question when addressing the general stochastic AND-OR tree search problem is whether there is a pre-empt/resume policy which is optimal.

Figure 27 overleaf shows two different search problems which include the sub-search of the expression Y, illustrating why a pre-empt/resume policy is not necessarily optimal. Suppose that there are two alternative policies for searching Y. Policy π_1 allows for a quick and relatively accurate judgement to be made about whether or not Y is true. Policy π_2 by contrast does not, but determines the truth in less time, on average.

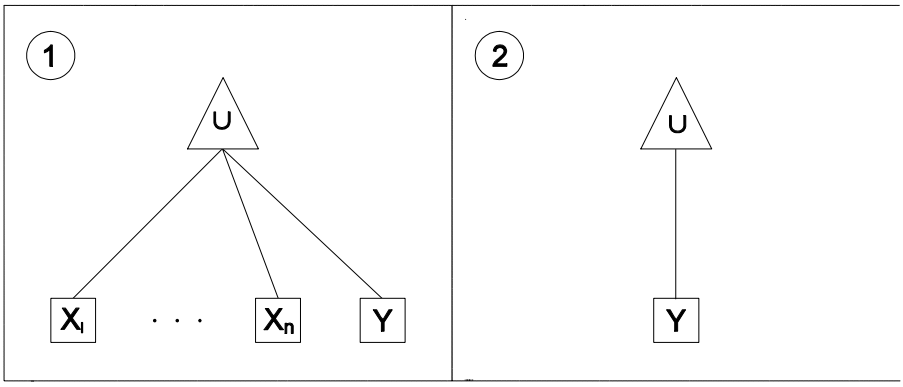

Figure 27: Different Requirements for Searching Y

Policy π_2 is to be preferred in case (2), but not necessarily in case (1). The reason for this is that the 'intermediate information' associated with policy π_1 is of use in case (1), since it may show that switching to search one of the X_i expressions is preferable to further search of Y. In this way, the expected time to search the overall expression can be reduced.

An example of an expression Y for which two such search policies exist is as follows:

$$Y = (A \cup B) \quad X = (Y_1^{\mathrm{c}} \cup Y_2^{\mathrm{c}} \cup \ldots Y_N^{\mathrm{c}})$$

The following node types are defined:

$$d_A = \left(0, 1, \frac{1}{2}s_B + \frac{1}{2}s_C\right) \quad d_B = \left(\frac{99}{100}, 10, 1\right) \quad d_C = \left(\frac{1}{2}, 10, 1\right)$$

We must search one of the identical Y_i, so for definiteness assume we search Y_1. The decision to be made is therefore whether we search A_1 or B_1. A little reflection on the problem will suffice to show that the optimal choice

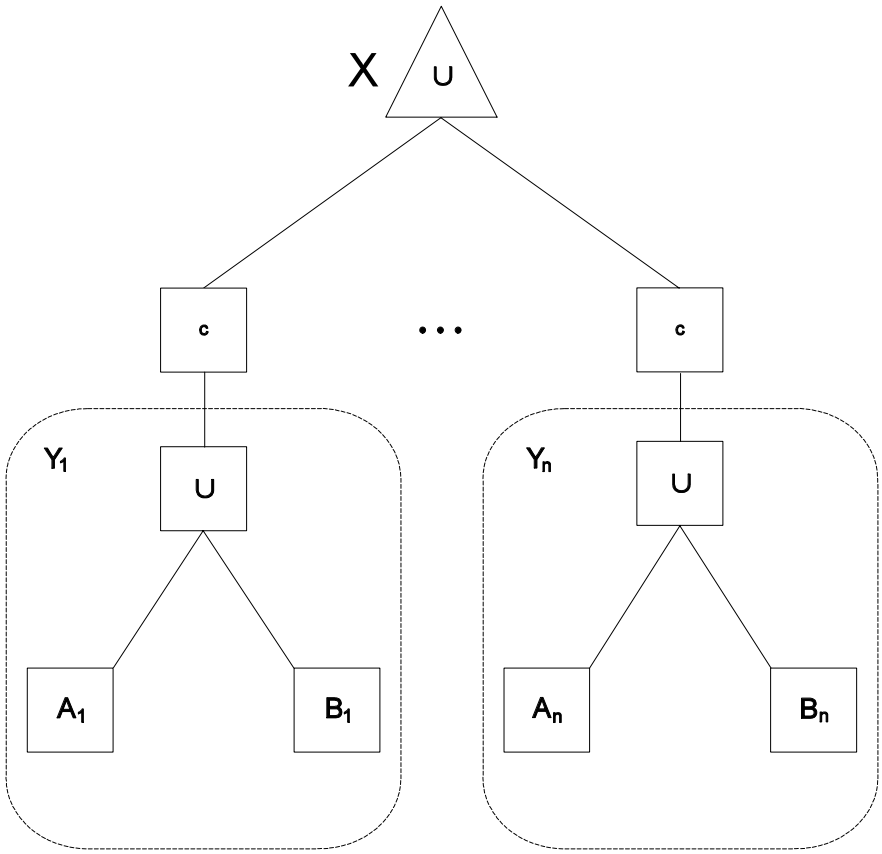

Figure 28: Pre-empt/Resume Counterexample

depends upon the value of N, thus violating non-locality and implying that optimal play cannot in general be a pre-empt/resume policy.

As $N \to \infty$, the probability that X is true tends to 1, and the value of refuting a single Y_i^{c} tends to zero. The aim of search must therefore be to prove a Y_i^{c} to be true, thus proving X to be true and allowing termination of the search. This is equivalent to refuting a Y_i. Bearing this in mind, it is better to start search of a Y_i by an exploratory search of A_i. If this shows the Y_i to be equivalent to $B_{i_1} \cup B_{i_2}$, it is optimal to 'retire' from this Y_i and choose another, until a Y_i is discovered that is equivalent to $B_i \cup C_i$, an

expression which is 50 times more likely to be false, and so more worthy of investigation than $B_{i_1} \cup B_{i_2}$.

Now consider the opposite extreme, $N = 1$. In this case, a refutation of Y_1^{c} will terminate the search, showing X to be false. This can be achieved with probability $\frac{99}{100}$ by search of B_1. Search of A_1, by contrast, offers no immediate chance of terminating the search, and, whatever node it reveals, the best node to search next will be B_1.

To recap, we find that there is no single answer to the question "How is it optimal to search the sub-expression Y_1?"; if it has no siblings, it is optimal to begin by searching B_1, while if it has many siblings, it is optimal to begin by searching A_1.

4.4 Non-optimality of Index Policies

The counterexample described in the previous section demonstrates that the AND-OR tree search problem cannot be solved by an index in the same way that the OR-tree search problem can be solved by calculating $\emptyset()$. We now examine the possibility of treating an AND-OR tree as an OR-tree of depth one with an infinite number of node types.

We see that an AND-OR tree can be written $(Y_1 \cup Y_2 \ldots Y_n)^{\mathrm{c}}$, as shown in Figure 29. The presence or absence of a '$^{\mathrm{c}}$' makes no difference to the fundamental conception, which is to extend the result of the Chapter 3 by dealing with each of the Y_i's as if it were a simple logical primitive as opposed to a compound logical expression. In general, this requires that there be a

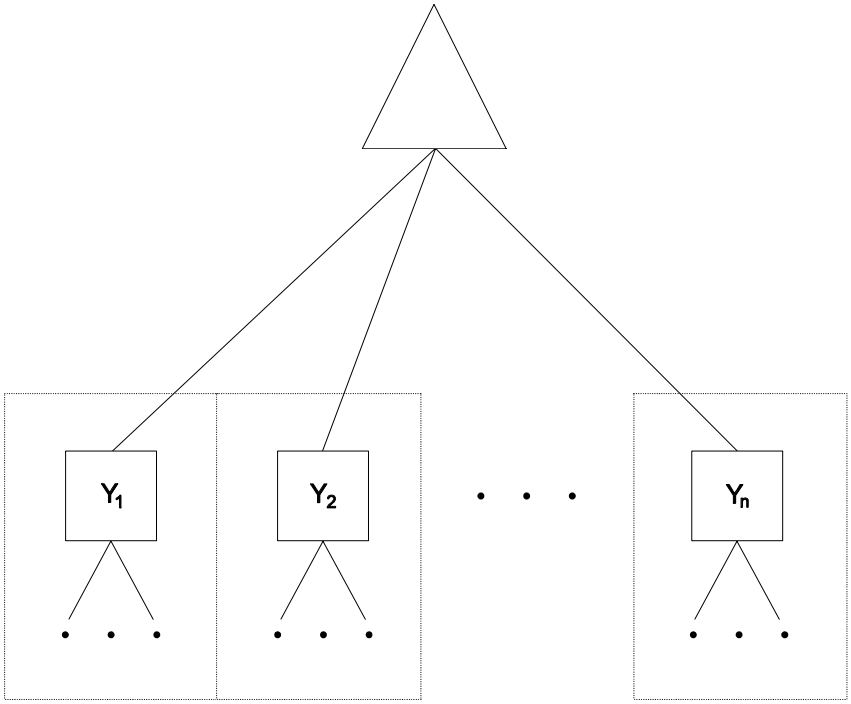

Figure 29: Viewing an AND-OR Tree as an OR Tree

countably infinite number of node types, since each Y_i may be an arbitrarily complex AND-OR expression. Even in the special case in which the Y_i cannot become arbitrarily complicated, but can be transformed by search into one of only a finite number of terms, we show by considering the preempt/resume counterexample of the previous section that this concept still presents problems.

To represent X as an OR expression, we need a state for each logical expression to which search of the Y_i can lead. For clarity, we index these

states numerically:

$$\begin{aligned} s_1 &= C^{\mathrm{c}} & \qquad s_4 &= (B \cup C)^{\mathrm{c}} \\ s_2 &= B^{\mathrm{c}} & \qquad s_5 &= (B_1 \cup B_2)^{\mathrm{c}} \\ s_3 &= A^{\mathrm{c}} & \qquad s_6 &= (A \cup B)^{\mathrm{c}} \end{aligned}$$

The expression Y therefore corresponds to a node of type 6, so state $(0, 0, 0, 0, 0, 1)$, whilst X corresponds to state $(0, 0, 0, 0, 0, n)$. The node types have the following characteristics:

$$\begin{aligned} d_1 &= \left(\tfrac{1}{2}, 10, 1\right) \\ d_2 &= \left(\tfrac{1}{100}, 10, 1\right) \\ d_3 &= \left(0, 1, \tfrac{1}{2}s_1 + \tfrac{1}{2}s_2\right) \\ d_4 &= \left(0, 10, \tfrac{99}{100} + \tfrac{1}{100}s_1\right) \text{ or } \left(0, 1, \tfrac{1}{2}s_5 + \tfrac{1}{2}s_6\right) \\ d_5 &= \left(0, 10, \tfrac{99}{100} + \tfrac{1}{100}s_2\right) \\ d_6 &= \left(0, 10, \tfrac{99}{100} + \tfrac{1}{100}s_3\right) \text{ or } \left(0, 10, \tfrac{1}{2} + \tfrac{1}{2}s_2\right) \end{aligned}$$

This problem is not yet in a form soluble by the OR-tree search model because of node types 4 and 6, which may be expanded in more than one way. In practice, since this example is small, dynamic programming can be used to deduce the optimal policy. Whilst a general solution seems an unlikely prospect, use of points made in Section 3.9 may prove sufficient to allow an ad hoc solution by dynamic programming to some specific classes of problems.

4.5 Summary

The AND-OR model described above is a powerful one, capable of application without modification to the task of (bi-valued) game tree searching. Unfortunately, it does not seem to be an easy task to deduce an optimal policy.

An important observation about the AND-OR tree search case is that the optimum policy is not a pre-empt/resume policy. The non-locality of the optimum policy, together with the observation that no one has yet come close to solving this problem strongly suggest to me that aiming for a method of deducing a strictly optimal policy may, in the general case, be an unrealistic goal. A more fruitful approach may therefore be to pursue methods of deducing a policy which is both easily computable and 'nearly optimal', in some sense.

A final, important, point which deserves to be made is a consequence of the fact that knowledge of whether or not an object exists affects the optimal policy. *Given that an object exists*, a simple Ø-based policy can be used, not to direct all the search, but simply to choose which top level branch it is optimal to search. An admittedly speculative example of how this might be used is the assumption that might be made by a game-playing program that somewhere in the game tree it is searching, a winning variation does indeed exist.