Chapter 2

Evolution of Computer Game Playing Techniques

In order to understand the recent developments in decision theory and its application to computer game playing as a testbed AI problem, it will help us to review some of the past development of computer game-playing techniques. But before we begin describing the classical game playing algorithms, we will first take a look at the motivating task of *computer game playing* itself, and explicate some of the fundamental simplifying assumptions about this task that the field of computational game theory has traditionally made.

People often simplify problems in order to solve them. However, as AI researchers we need to simplify more than usual, because it's our not-so-bright computer programs that must do the solving. One classic AI simplification of a problem domain, exemplified in the early game-playing work discussed in §2.2 (p. 20), is to characterize the domain in terms of a well-defined set of possible states and state transitions. For those problems in which this simplification is reasonably appropriate, it facilitates automated reasoning by giving our programs a well defined, formalized space within which to search for a solution to the problem.

Unfortunately, if the domain is at all interesting, we often find that the sheer size of the state-space overwhelms the computational resources available to us. But in such cases, we'd prefer not to altogether abandon the comfortable simplicity of state space search. Instead, we compromise by searching only a part of the space. (See §2.3.2, p. 26.)

Partial search raises two obvious issues: what part of the space do we search, and how do we use information about only a part of the space when an ideal solution would require searching the whole space? The traditional answers to these questions have been kept as simple as possible; for example, by searching all and only those states within d

transitions of the initial state, or by making crude assumptions that just use the partial space as if it were the whole space (cf. minimax, §2.4).

Recently, however, researchers have been becoming more brave. They are beginning to abandon the simplest methods, and instead look for ones that can somehow be justified as being *right*. To do this, many are turning to the methods of probability and decision theory. The former can tell us what to believe, the latter tells us what to do. Thus we can answer the questions of what part of the space to search, what to believe as a result, and what external action, finally, to do. In §2.5 and §2.7 we discuss the advent of the probabilistic and decision-theoretic methods, respectively, in computer game playing.

Progress in being made steadily along these lines in computer game-playing, but researchers are beginning to be forced to confront problems with both decision theory and with the old partial state-space search paradigm. Some of the innovative ideas that are being developed to solve these problems will be discussed and expanded upon in chapters 3 and 4.

2.1 Why Games?

This thesis claims to be about developments in decision-theoretic AI in general. So why are we spending a large chapter studying DT methods in the context of game-playing, in particular? Perhaps we should concentrate instead on efforts to write decision-theoretic programs that actually do something useful. CGP researchers are constantly faced with such questions. Playing games is such a frivolous pursuit that it would seem that any research whose overt aim is to write programs to play games must be frivolous as well. Indeed, if the only aim of a piece of CGP research is to write a game playing program, then I would agree that that piece of research was frivolous.

Serious CGP research, in my opinion, must demonstrate an intent and desire to eventually develop general methods that can be applied to other problems besides games, or at the very least, to gain practice and experience in the use of general tools and techniques (for example, probability, decision theory, and statistics) which will then help us when we later attack more practical problems. Still, this begs the question of why we should pick game playing as the domain in which we will develop new general methods or exercise our skills.

To me, the reason is that a game is a clear, well-defined domain that may yet involve subtle issues of reasoning, and pose a real challenge for AI programs. Unlike "toy" domains made up especially for the programs built to solve them, games played by humans were not devised with the strengths and weaknesses of current AI methods in mind, so the game may involve more hidden subtleties and difficult problems than would the toy domain.[1]

But, unlike more general "real-world" situations, most games have a limited, finite specification in terms of their rules, which allows AI programs to at least have a fair chance at addressing them adequately, whereas in less formal real-world domains it is much harder to describe and reason about all the factors that might be relevant in reasoning about the domain. The domain may be too open ended for this to be feasible.

And yet, in games there are clear performance criteria: does our program win the game? What score does it obtain on average? How good is it compared to other programs that play the same game? These criteria help to tie the research to reality; if the program does not play well, then this suggests that the theory behind the program might be inadequate,[2] and the degree to which this is true can be quantitatively measured by, for example, the fraction of games won against a standard opponent. Games allow us to quantify the success of our ideas about automated reasoning.

Finally, the time limits imposed in some games encourage us to make sure that our algorithms not only reason well, but also that they reason efficiently. Efficiency is a vital concern for AI that hopes to be useful in the real world, because in the real world we do not have unlimited computing resources, and we can not necessarily afford to wait a long time to get an optimal answer, when instead we might be able to get a good enough answer from a less ambitious but much faster program.

---

1. However, we must be careful; if we have to work long and hard to find a game played by humans that our reasoning algorithm can handle, then perhaps this is no different than just making up our own "toy problem" after all—especially if we have to simplify the rules of the game before our method will work.

2. Although of course the program may also play poorly due to bugs or bad design in the implementation. Also, the converse is not necessarily true: if the program plays well at a particular game, this may be due to raw "brute-force" ability, or special-purpose engineering for the particular game, rather than being a result of general theoretical soundness. The degree to which this is true is not so easily quantifiable. However, this problem can be fixed. Later in this chapter we will discuss some new directions in CGP that aim to tie success in games more closely to general intelligence.

In summary, games, in general, are more approachable than other types of domains, and yet the right kinds of games can be very challenging from an AI standpoint, and offer many opportunities for the application of new, sophisticated reasoning techniques. Whether the existing CGP work has fulfilled the potential of games as a domain for AI is a separate issue that I will defer until later in this chapter.

2.2 Early Foundations

The formal analysis of games has had a long history. Games of chance have been studied using the mathematics of probability (and have historically been one of the major motivations for its development) for several centuries, but a general, systematic *game theory* really first arose with John von Neumann [vonNeumann-28], [vonNeumann-Morgenstern-44]. Von Neumann focused his attention on "games of strategy." His theory is most easily applied to games that have a formal, mathematical nature, as opposed to games having vague or ill-defined rules.

Von Neumann's work started a couple of different threads of further research. The book [vonNeumann-Morgenstern-44] was aimed primarily at game theory's relevance to problems in economics, and von Neumann's decision theory fit well into the economics tradition of attempting to model rational choices by individuals and organizations. Many of von Neumann's ideas concerning general decision theory were adopted and expanded upon by economists and by the newly inspired game theorists in the decades to come. Game theory was used in politics, economics, psychology, and other fields; the literature is too large to survey here.

Von Neumann himself had a significant background in computer science, and his game theory work was noticed and read with interest by computer scientists who were interested in demonstrating the power of computers to solve symbolic problems (e.g., [Shannon-50] and [Prinz-52]). Von Neumann's focus on strategic games was perfect for this, and his game theory provided guidance as to how to tackle that task in a systematic way that could facilitate implementation on a computer.

Thus, Claude Shannon set forth in the late 1940s to program a computer to play chess [Shannon-50]. Although Shannon mentioned von Neumann's basic idea of the *game-theoretic* value of a position, he did not make use of von Neumann's deeper idea of using maximum expected utility to make decisions in the presence of uncertainty. Perhaps this was because he felt that the kind of uncertainty present in chess was different from the kinds of uncertainty dealt with in von Neumann's decision theory. Chess is a deterministic game, having no chance element, so on the surface, decision theory's fundamental notion of expected utility does not apply. However, in chess, uncertainty about outcomes does exist, despite the determinism of the rules, due to a player's lack of complete knowledge of his opponent's strategy. Additionally, even if we assume a particular opponent strategy, such as perfect play, we lack sufficient computing power to calculate the outcome of a chess position under any such model. But regardless of the reason, Shannon did not draw any correspondence between his "approximate evaluating functions" and von Neumann's "expected utilities," a correspondence that only cropped up later in the development of computer game-playing, as we shall see in §2.5 (p. 38).

However, Shannon's work was paradigmatic, and has provided much of the context for the computer game-playing work since his time. Although his approach was more pragmatically oriented than von Neumann's theoretical treatment of games, Shannon's work depended to an equal degree on the well-defined nature of the domain (games of strategy) with which he dealt. Shannon used the concept of a branching tree of possible future positions linked by moves. The concept of searching such a tree, (or, more generally, a graph), of primitive states of the domain "world" was established, largely due to Shannon's work and the further computer-chess efforts by AI researchers that followed it, as a fundamental paradigm for CGP, which we will discuss in the next section.

Shannon's main contribution, utilized with by most of the game-playing programs to date with much success, was the idea of using a heuristic function that approximated the true (game-theoretic) value for positions a few moves ahead of the current position, and then propagating those values backwards, as if they were game-theoretic values, to compute approximate game-theoretic values of preceding positions. (We will see how this works in more detail in §2.4). Much criticism has been levied against this approach by later researchers, who point out that a function of approximate inputs is not necessarily a good approximation of the function with actual inputs ([Pearl-83], [Abramson-85]). We will go into more detail on these and other criticisms of minimax in §2.4 (p. 31).

Finally, largely overlooked for many years were some salient suggestions by Shannon as to how his algorithm might be improved by intelligently selecting which lines of play to consider, and how deeply to pursue them. Shannon's ideas in this area were preliminary, but he did have some very concrete suggestions that could have been implemented in the chess programs of that era, but were not (to my knowledge). The idea of *selective search* was finally developed in detail and incorporated into some actual programs several decades later; this work will be discussed further in §2.6 (p. 66).

But again, practically all of the later CGP work (to date), including the selective-search work, has been based on what I will call the *Shannon paradigm,* of so-called[1] "brute-force" search through a partial graph of primitive world-states. The next section will delve a little more deeply into the assumptions inherent in this way of attacking the problem, assumptions which we will consider later in the thesis (in chapter 4) how to abandon or weaken.

Other early work includes a chess program devised and hand-simulated by Turing around 1951 (described in [Bates-et-al-53]), a program at Manchester University for solving mate-in-two problems [Prinz-52], and Samuel's early checkers learning program [Samuel-59].

2.3 Partial State-Graph Search

In this section we examine in more detail the nature of Shannon's approach to game-playing and its impact on AI research in general. We break down the approach into its ontology (the way it views the world), its epistemology (the way it handles knowledge about the world), and its methodology (what it actually does, in general terms).

2.3.1 State-graph ontology

An intelligent agent, real or artificial, generally performs some sort of reasoning about a domain, a part of the world that is relevant to it or to its creator. An important characteristic of an agent is the *ontological stance* it exhibits towards the domain, by which I

1. Even by Shannon, in [Shannon-50].

mean the agent's conceptual model of how the world is structured and what kinds of things exist in it.

The ontological stance in the Shannon paradigm, one that a very large number of AI systems take towards possible circumstances in their domains, is to view the world in terms of its possible *states*. Typically, not much is said about what is meant by a world state; the term is usually left undefined. Here we will take a more careful look at what a world state typically is, and how that differs from what it could be.

The things that are usually called "states" in AI might be more precisely termed *concrete microworld states*. By "microworld" I just mean the domain, the part of the world that the agent cares about, or is built to deal with. A "microworld state" is a configuration of that microworld at some point in time. Finally, "concrete" means that the meaning of a given configuration is fully specific; it is complete and precise as to every relevant detail of the microworld at the given time; it is not at all abstract, nor can it be viewed as encompassing numerous possibilities. It is a single, primitive possible configuration. This is to be contrasted with the view of *abstract* states taken in chapter 4. If we choose to think of our microworld as being described by a definite set of variables, then a concrete microworld state is a complete assignment of values to those variables, rather than a partial one.

Hereafter, when I use the word "state," I will mean concrete microworld state, except where indicated otherwise. Although "concrete microworld state" is more descriptive, it is longer and less standard; thus we adopt the shorter term.

Not only does the use of states imply some sort of concept of time, with the microworld typically thought of as occupying a definite state at any given time,[1] but also, time is typically discretized to a sequence of relevant *moments*, with the world changing from one state to another between them. Additionally, there is often a notion of a set of the legal, possible, or consistent states that the world could ever occupy; states in the set are "possible" while there might be other describable but "impossible" states that are not in the set. This set is referred to as the microworld's *state-space*. (For example, in chess, we might

---

1. This can be contrasted with, for example, the Copenhagen interpretation of quantum physics, in which the world may occupy a probabilistic "superposition" of states. Interestingly, however, there is a lesser-known interpretation of quantum physics, Bohm's interpretation, that retains all the same experimental predictions while preserving the idea of the existence of an actual (but unknowable) world-state (see [Albert-94]).

consider unreachable positions to be states, but not to be in the chess state space. However, the idea of states that are not in the state space differs from some standard usages of the term "state space," and so I will not make use of this distinction much.)

Additionally, there is typically a concept of possible changes of state, expressed in terms of primitive *state transitions* (or "operators" or "actions") that are imagined to transform the world from one state to another between the discretized time steps. There is imagined to be a definite set of allowed or possible transitions from any given state.

The notion of a state space, together with the notion of state transitions, naturally leads to a conceptualization of domain structure in terms of a *state graph*, in which the nodes represent concrete microworld states, and the arcs represent the allowed state transitions. (See figure 1.)

Many of these elements were present in von Neumann's treatment of games, and indeed, they formed a natural analytical basis for the sorts of "strategic games" he was considering. Shannon, in developing computer algorithms for a typical strategic game, chess, continued to make use of this natural ontology for thinking about the structure of the game-playing problem.

And indeed, the vast majority of the AI programs written for game-playing domains, for similar types of domains such as puzzle-solving, and even for some supposedly more practical planning and problem-solving domains, have adopted the state graph, or its specialization the *state tree*, as their primary conceptualization of their domain's structure. State-graph ontology has proved to be a natural and fruitful framework in which

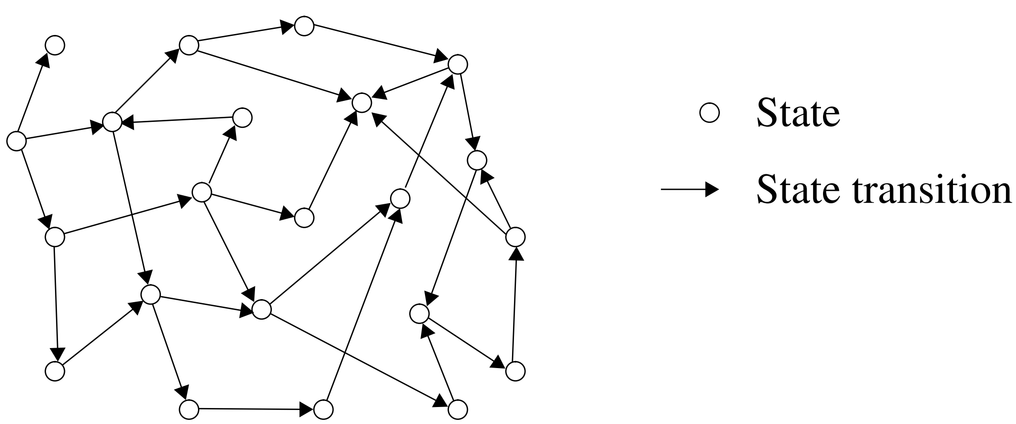


Figure 1. A state-transition graph

to develop well-defined and well-understood algorithms for reasoning in such domains. This success is no accident, since many of these domains are artificial ones that were defined by humans in terms of well-defined states and state-transitions in the first place. Humans, too, seem to make use of the state-graph conceptualization when performing deliberative decision-making in these domains.

However, I will argue in chapter 4 that, even within these state-graph-oriented domains, the state-graph conceptualization is not the only framework that humans use to support their reasoning, and that certain other models permit a greater variety and flexibility of reasoning methods. I propose that the development of AI programs capable of more humanlike and general reasoning might be facilitated by research on programs that demonstrate how to use such alternative conceptual models to exhibit improved, more humanlike performance, even when applied to simple game and puzzle domains. I am not the first to promote this idea; we will mention some of the existing efforts along these lines in chapter 4.

In any case, this "state-graph ontology" is the part of the Shannon paradigm that has perhaps had the most influence on computer game-playing, and on AI in general. It is so pervasive that it is rarely mentioned, and yet it shapes our ideas about AI so strongly that I believe we would be well-advised to occasionally explicate and reconsider this part of our common background, as we are doing here, and at least take a brief look to see if we can find useful ideas *outside* of the state-graph paradigm, ideas that we may have previously missed due to our total immersion in the language of states and transitions. For example, in some of the theoretical directions considered in chapter 3, we will refrain from adopting the state-graph ontology, in favor of a more general view.

Although state-graph ontology gives us a structure within which we can analytically describe what the problem is that we are trying to solve, and what an ideal solution would consist of, it still does not tell us what facts about the domain would be known by a real agent that we design. This is the role of an epistemology, and specifically, we now examine the *partial state-graph* epistemology, another major aspect of the Shannon paradigm.

2.3.2 Partial state-graph epistemology

Besides an ontology, another important characteristic of any agent is the scope of its knowledge about the domain in which it is working. In particular, if we are told that an agent models a domain in terms of a certain kind of state-graph, this still does not tell us very much regarding what the agent does or does not know about the possible circumstances that might occur in its microworld.

Given the state-graph ontology described in §2.3.1 (p. 22), it is natural to describe agents' states of knowledge about particular domain circumstances in terms of an *examined partial graph*, that is, a subgraph of the domain's state-graph expressing that set of nodes and arcs that the agent has *examined*. We will say a node or arc has been examined, or "searched," when an agent constructs a representation for it. Generally, an agent's purpose in examining part of a state-graph is to infer useful information from it. An agent might know things about some of the nodes that it has not yet examined; for example, it may have proven that all nodes in an entire unexamined region of the graph have a certain property.[1] Nevertheless, we will still treat the examined partial graph as a special component of an agent's knowledge, because there is usually a lot of information about a state

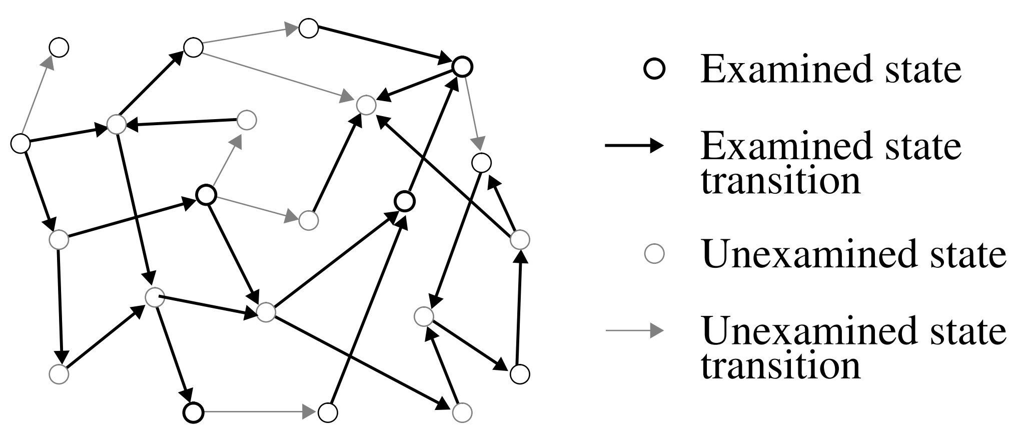


Figure 2. A partial state-transition graph. In general an arbitrary subset of the states and transitions may have been examined (internally represented at some time).

1. A concrete example: the alpha-beta algorithm performs game-tree pruning by proving (under certain assumptions) that none of the nodes in an entire unexamined subtree will occur in optimal play.

that an agent can only obtain, as far as we know, by examining a corresponding region of the state-graph.[1]

For some small game and puzzle domains (e.g., tic-tac-toe and the 8-puzzle) the domain's state-graph is so small that a program can examine it exhaustively in a short time. In these domains, we can process all the information needed to determine an ideal solution or move choice very quickly, and so there is not much to say about epistemology.

More interesting are the larger games, puzzles, and strategic situations, in which the state-graph may be far too large to examine in a feasible time. It may even be infinite. In such domains, the particular state-graph epistemology we apply to our agents becomes an important consideration in their design, and this epistemology can have a major influence on the agent's performance. If performance is to be at all acceptable, the examined partial graph cannot be the full graph, so the knowledge that will be available to our agents at a time when they must make a decision will depend strongly on the agent's earlier choice of which of the many possible partial graphs to examine.[2]

Shannon's answer to this question for chess was the subsequently-popular *full-width search policy,* which says to look exactly d moves ahead along every line of play, for some depth d. However, Shannon realized that this was an arbitrary policy, and not, in any sense, the "right" way to answer the question, and so he proposed a couple of more sophisticated approaches. Later developments in CGP that expanded upon these suggestions are discussed in §2.6 and §2.7. However, Shannon's suggestions and the later developments were still largely *ad hoc*. More recently, partial-search epistemology has been tackled from a decision-theoretic standpoint; these efforts are discussed in §2.8 (p. 69).

So, once we have decided on this sort of epistemology, that our agent's knowledge will consist (at least in part) of knowledge about portions of the state-graph representing the domain's structure (as viewed in our ontology), we have a general guiding framework for the design of our reasoning algorithms, at least ones intended for strategic games and

---

1. For example, the only way we know to obtain the true game-theoretic value of a general chess position is to examine a large tree of successor positions.

2. Moreover, we note that this choice of partial graph obviously cannot be made through explicit consideration of all possible partial graphs—for that would take exponentially longer than examining the entire original graph! Some higher-level policy for choosing the partial graph is needed.

similar problems. But there remains the question of our agent's more detailed, pragmatic behavior; exactly how will its decision-making proceed?

2.3.3 Partial state-graph methodology

We are led to two crucial methodological questions for agents that are built under the presumption of a partial state-graph epistemology. Each agent should, either in its design or at run-time, answer these questions:[1]

- What partial state-graph should I examine?
- How shall I derive useful knowledge from my examination of this partial graph?

I submit that the ways a program answers the above two questions are useful dimensions along which to categorize AI programs that use the partial state-graph paradigm. I suspect that the computational effort expended by these programs can be naturally broken down into computations that are directed towards answering one or the other of these questions.

Many programs answer these questions incrementally, that is, they start with a small partial graph, and examine more nodes and arcs slowly, while simultaneously deriving information from the intermediate graphs, and using this information to guide the further growth of the graph. This can be contrasted with the alternative of creating the whole partial graph in one large atomic step, and only then stopping to derive information from it.

Often the goal behind the incremental approach is to provide an *anytime algorithm* [Dean-Boddy-88] that can be stopped at any time and asked to immediately provide an answer based on the partial graph examined so far.[2] (Anytime algorithms have been investigated as a general solution to some of the problems of limited rationality, for example, see [Zilberstein-Russell-93, Zilberstein-93]. This work will be discussed briefly in chapter 3.) Another reason for the incremental approach is that if an algorithm processes states as it examines them, then it may be able to save memory by expunging most of the examined

1. [Newell-et-al-58] provides a similar characterization of the Shannon paradigm.
2. For example, the iterative deepening approach in game-tree search, see [Newborn-89].

nodes, and only keeping the useful information that is derived from them.[1] In my terminology, I will consider expunged nodes to still have the status of "examined" nodes, unless all information gained from examining them is thrown away as well. For example, if, in game-playing, we throw away all the results of our search after making a move, then the nodes we had searched are now "unexamined" again, until and unless we revisit them in the course of choosing our next move.

Another rarely-mentioned but important characteristic of most partial state-graph search programs is that the examined partial graph is usually a connected graph (undirectedly). This connectedness is a consequence of the natural graph-examination scheme of always examining nodes that are connected to ones that have already been examined and stored in memory. The connectedness property also tends to make it easier to extract useful information from the partial graph, by using rules for propagating information along state transitions and combining information at nodes in a semantics-preserving fashion. We will see several examples of such algorithms. However, one can imagine a partial graph-search scheme that examines, for example, scattered randomly-chosen nodes, or that expands the graph outwards from a small selection of unconnected search "islands." (See figure 3.)

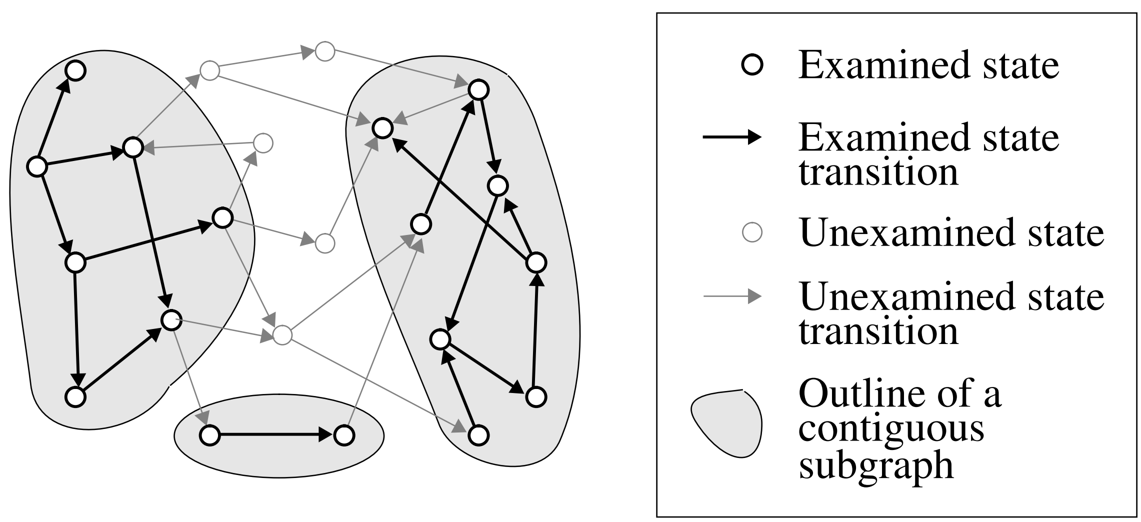


Figure 3. A partial state-transition graph with a few contiguous examined regions, or search islands. The examined graph has more structure than in figure 2.

1. This is done by alpha-beta, see §2.4.3, p. 37.

2.3.3.1 Choosing a partial graph to examine

The first fundamental methodological question is how to determine exactly what partial state-space graph we are going to produce. As mentioned above, this determination is usually done incrementally. However, methods exist that examine a predetermined set of nodes without regard to information obtained from examining them, for example, a depth-*n* full-width minimax search.[1] Most methods are incremental to some degree; for example, alpha-beta search uses the values of some subtrees to choose which other subtrees are worth searching. A* ([Hart-et-al-68], [Hart-et-al-72]) is a single-agent example of an incremental search.

2.3.3.2 Deriving useful information from a partial graph

The second fundamental question is how to derive information from a partial graph that will be useful in making a final decision. The most common technique is to obtain some information at the nodes located on the outer edge of the partial graph (i.e., at the *leaves*, if the graph is a tree), and then to propagate that information toward the current state, along the arcs of the graph, transforming the information along the way to determine how it bears on the overall decision or problem faced in the current state. We will see many examples of this sort of gather-and-propagate algorithm in the CGP research discussed in the sections ahead.[2]

The information at the leaves is usually obtained by some heuristic function, although the heuristic may be obtained from relevant statistics about similar past nodes, rather than being handmade by a human.

Certain propagation algorithms (e.g., minimax) can be interpreted as implicitly assuming that a piece of heuristic leaf information is a correct representation of some function of the entire unexplored part of the graph below that leaf (for example, the leaf's game-theoretic value). The probabilistic and decision-theoretic algorithms we will see later in this chapter attempt to weaken or abandon such assumptions.

1. Here I am not counting the set of children as information obtained from examining the parent.

2. However, not all state-space CGP techniques use information solely from nodes on the periphery of the examined partial graph—some also utilize information from interior nodes; for example, see [Delcher-Kasif-92] and [Hansson-Mayer-89]. Whether this is the "right thing" to do is still unclear.

A final simplifying assumption is that many algorithms for extracting information from partial graphs assume that the graph has a restricted structure, such as that of a tree or a DAG. The reason for this assumption is often that the correct algorithm would be too slow or too hard to describe. Such assumptions may even be used in cases when they are known to be false, under the hope that the inaccuracies caused by the false assumptions will not degrade decision quality very much. In practice this may be true, but it makes the theoretical performance of the algorithm much harder to analyze.

2.3.4 Is partial state-graph search really what we want?

In the above subsections we have revealed some of the simplifying assumptions about the world, and how our agents should deal with it, that are implicit in the paradigm established by Shannon and much early AI work. To summarize:

- The history of the world is broken into a discrete sequence of times.
- At any given time, the world is fully characterized by the unique state it occupies at that time.
- There is a definite space of possible states the world may be in.
- There is a definite mapping from states to possible next states.
- Our agents will represent the world by a graph of nodes representing these world-states, with arcs representing possible transitions.
- Our agents will reason about the world by examining a subgraph of this state-graph.
- Our agents will explore contiguous regions of the state graph.
- Our agents will make approximate guesses about properties of the graph beyond the part that has been examined.
- Our agents will infer what to do by "propagating" information through the graph, typically from the edges to a known "current" state.

This is really quite a large number of things to assume without justification. And yet, much research using these assumptions, such as most of CGP work, proceeds without even a mention of any justifications, either because the researchers are not even realizing they are making these assumptions, or because they presume that the reasons for making these assumptions have already been adequately established in the literature, and they are just following along in the tradition of the field.

However, it is my opinion that the field has *not* adequately established why these assumptions should be adopted, and that this paradigm is being followed mostly just due to its own momentum. The fact that it is used so universally means that it doesn't even occur to most researchers to question it.

This thesis, however, intends to question some of these assumptions (and any others encountered), and either find satisfactory justifications for them or present better alternatives. The later chapters will dive more deeply into this enterprise of questioning, but for now, let us just accept that partial state-graph search is the basis for much of the current work in AI, and CGP especially, and see what we can learn by examining in detail a number of the existing CGP techniques, while revealing and questioning some of their additional assumptions along the way.

2.4 The Minimax Algorithm

The previous section described the crucially important, but rarely-mentioned epistemological assumptions that are pervasive throughout all computer game-playing research, and much of the rest of AI, from Shannon up to the latest work. Now we shall go into more specific detail about Shannon's algorithm, and some of the particular assumptions under which it can be justified, even beyond the basic assumptions and limitations of the partial state-graph search paradigm. Together all these assumptions make up what I call "the Shannon paradigm."

The Minimax algorithm has been described and explained informally too many times to repeat that effort here. Readers unfamiliar with the algorithm should refer to any introductory AI textbook (such as [Rich-Knight-91] or [Winston-92]) for a tutorial explanation. Instead, my contribution here will be a more formal treatment that will be suited to our later discussions.

2.4.1 Formalization of the minimax propagation rule

We assume the game we are considering has two players, which we will refer to as "Max" and "min," and can be represented as $G = (\Omega, \delta, M, v)$, where Ω is the space (set) of possible states the game may be in, $\delta: \Omega \to 2^{\Omega}$ is the *successor* or *transition* function mapping each state in Ω to the set of its possible successor states, $M \subseteq \Omega$ is the set of states in which Max gets to determine the next state (whereas min gets to determine the next state in $\Omega - M$), and $v: \{s \in \Omega: \delta(s) = \emptyset\} \to \mathbf{R}$ is a function that maps each terminal game state (where a terminal game state is one with no successors) to a real number, with the intention that the number expresses the state's degree of desirability from Max's point of view. It is also assumed that $-v(s)$ expresses the degree of desirability of s from min's point of view.[1] (Later in this section we will question some of the definitions and assumptions in this framework; for now we just present it.)

Then, assuming the range of δ includes only finite sets, we can define the *game-theoretic value* $V(s)$ of any $s \in \Omega$ recursively as follows:

$$V(s) = v(s), \text{ if } \delta(s) = \emptyset\text{; otherwise,} \tag{1}$$

$$V(s) = \max_{t \in \delta(s)} V(t), \text{ if } s \in M\text{; and} \tag{2}$$

$$V(s) = \min_{t \in \delta(s)} V(t), \text{ if } s \notin M. \tag{3}$$

We note that if there are directed paths of infinite length in the graph (Ω, δ) (in other words, if the game may go on forever), then there may not be a unique solution to these equations for all nodes.[2] However, in practice, for example in a program for calculating $V(\cdot)$ of all nodes, we could handle this case also, by declaring infinite length games to have value 0, corresponding to a "draw," so long as the game graph (number of states) is not infinite. A detailed algorithm for this is provided in §2.5.4.4 (p. 60). But for now, we will, for simplicity's sake, restrict our attention to only those cases where a unique solution does exist.

---

1. In other words, we are assuming that the game is "zero-sum."

2. It is often said that the "50-move rule" in chess implies that the game is finite. But actually, the rules of chess do not actually require either player to claim the draw; so in principle a legal chess game could indeed continue forever.

Clearly, if we have access to only a partial graph, then we cannot in general compute $V(s)$ even for s in the partial graph. If all the successors of a node s are in the partial graph, and we can compute $v(t)$ for all successors t that are terminal, then we can compute $V(s)$. But what do we do for nodes whose successors have not all been examined? Shannon's answer is to compute, for each "leaf" node l (a leaf is an examined node whose successors are all unexamined), an *estimate* or *approximation* $\hat{v}(l)$ of the game-theoretic value $V(l)$ of l. Then, if we assume that all children of non-leaf examined nodes are also examined, we can just use (1)-(3), putting hats on all the v's and V's, to compute $\hat{V}(e)$ "estimates" for all examined nodes e. If the leaf estimates are correct, that is, if $\hat{v}(l) = V(l)$ for all leaves l, then it will also be the case that $\hat{V}(e) = V(e)$ for all e; i.e., the estimates of all non-leaf examined nodes will be correct as well. Analogously, it is hoped that if the $\hat{v}(l)$ are, in some sense, "good" approximations to the game-theoretic values, then the derived estimates will be good approximations as well. (Whether this hope is borne out will be examined in the next few subsections.)

So, this is how the minimax rule propagates information through the examined partial graph. In order to actually use minimax to decide on a move when it is our turn, we assume that we have determined the set $\delta(c)$ of immediate successors of our current state c, and that we have examined a partial graph including those successors and have computed their minimax values according to the above propagation rule. Then, if we are taking the role of player Max, we choose the successor (move) s that has the highest $\hat{V}(s)$, and if min, we choose s so as to minimize $\hat{V}(s)$.

2.4.2 Basic assumptions

Minimax involves assumptions that go far beyond the basic elements of the state-space paradigm described in §2.3. First, there are the basic assumptions having to do with the domain's being a strategic game: the state-space contains terminal states, and the desirability of the entire episode of interaction between the agent and its environment is reduced to the desirability of the final game state reached. Moreover, the game is assumed to be a zero-sum, two-player game. The game is deterministic, with the choice of state-transition always being completely made by one of the two players (and not involving any nondeterministic external factor).[1] Minimax requires that the successor function and the value of terminal states should be easily calculable, and that the number of successors of nodes not be too large to enumerate. Finally, using minimax requires that the game be one of "perfect information" where we can identify, whenever it is our turn, which state the world actually occupies.

However, even apart from these assumptions, one fundamental problem with minimax is that there is no theoretical justification for why it should work! Although the minimax rule is, by definition, the correct propagation rule for game-theoretic values, there is no theoretical reason to believe that, when given just "approximations" to game-theoretic values of leaf nodes, minimax will produce any reasonable approximation to the game-theoretic values of nodes higher in the partial tree. In fact, as we will see in §2.4.4 (p. 38), under some theoretical models of what kind of "approximation" the evaluation function $\hat{v}(\cdot)$ is providing, we can show that minimax can actually *degrade* the quality of the approximation, more so with each additional level of propagation. This is termed "pathological" behavior.

Nevertheless, we can attempt to justify minimax by showing that it does the right thing under certain sets of assumptions, and arguing informally that those assumptions are likely to be "almost" correct "most of the time" in practice, or that if the assumptions are categorically *not* correct, then the error incurred by this incorrectness is likely not to matter much. However, it should be kept in mind that such arguments are informal and inconclusive, and would probably be ignored if it were not for the fact that the Shannon paradigm has been empirically so successful.[1] One hopes that some of these informal arguments could be turned into formal approximation theorems that show, for realistic models of games and evaluation functions, that minimax is close to being correct most of the time, in a formalizable sense. However, no such analyses have been performed, to my knowledge.

Let us look at some of the sets of assumptions under which minimax can be shown not to incur errors. The first and most obvious one is just the assumption that the evalua-

---

1. If the desirabilities behave like utilities (i.e., if our goal is to maximize our mathematical expectation of them) then it is easy to add a rule to minimax propagation to accommodate games having nondeterministic "chance" moves as well; but the large body of more sophisticated algorithms (such as alpha-beta pruning) that are based on the Shannon paradigm will not so easily accommodate this addition.

1. Again, see practically any general AI textbook or general survey of computer game-playing, such as [Newborn-89].

tion-function values are the correct game-theoretic values. Then, of course, minimax yields correct values for all nodes. However, if we really want to assume that our evaluation function gives game-theoretic values, then we need not search at all, and so although it yields a correct answer, doing a deep minimax evaluation does not really make sense under this assumption. One can try to save the argument by saying that the evaluation function is assumed to be correct deep in the tree, but may be prone to errors closer to the root, but if one is not careful in one's model of this error gradient, pathological behavior may still occur ([Pearl-83]).

An alternative set of assumptions goes as follows. We interpret evaluation function values as giving our current expected utility (e.g., in a 2-outcome game, just the probability of winning) if play were to proceed through that node, and then we assume that our opponent shares those same assessments of expected utility, and that both he and ourselves will continue to have those same assessments while play proceeds, until one of the leaf nodes is reached. Assuming that both sides will choose moves they perceive as having maximum expected utility, this allows us to predict, with certainty, the value of the move that would be made at each node (although not the move itself, since there might be more than one move that yields maximum expected utility). I claim that under these assumptions, the minimax propagation rule yields consistent assessments for our expectation of the utility of the examined nodes.

This set of assumptions is an instance of a case where we might persuasively argue that the assumptions are close to being correct, or that their incorrectness does not matter so much. In an informal sort of way, they are conservative assumptions, in the sense that they assume things are worse for us than is actually the case. How is this so?

First, if we assume that our assessments of leaf values will not change as play proceeds, that is a conservative assumption, because it means we will not gain any new information about the leaf values beyond what we have so far, which is worse than the truth. If we assume that the opponent knows everything that we know, then that, too, is a conservative assumption. More conservative would be to assume the opponent knows a lot more than we do, but we could not hope to model such knowledge,[1] so we might as well pro-

1. Not everyone would agree with this argument. [Carmel-Markovitch-93] attempts to do just this sort of modeling.

ceed on the assumption that the opponent knows everything we know, but no more.[1] Similarly, if we assumed that the opponent were to gain more information about leaf values during play (and that we would not), this would also eliminates all hope of success, so we stick with the assumption that he will continue to just know everything that we know now. Finally, since the game is zero-sum, assuming that the opponent will act to maximize his expected utility is a conservative assumption, whereas the assumption that *we* will continue to attempt to maximize our utility, while not as conservative as assuming we won't, is at least known to be true.

So, although this argument is completely informal, it perhaps lends some insight into how to explain the high level of performance of minimax and its derivatives in practice: in a sense, minimax makes *conservative* assumptions, and thus serves as a good "engineering approximation." However, the assumptions are still rather questionable, and the fact remains that the only existing theoretical justifications of minimax depend on these assumptions. It would be better to have a propagation algorithm that did the right thing under assumptions that were more clearly realistic. Some attempts to do this are described in sections §2.5 and §2.7.

2.4.3 Alpha-beta pruning

Alpha-beta (α-β*) pruning* is a well-known technique for speeding up minimax search. (See [Rich-Knight-91] for a tutorial explanation.) Its exact origin is unknown; [Newell-et-al-58] briefly mention a technique they use that sounds similar, but do not give sufficient detail to tell if it is precisely the same algorithm. The first known full description of the algorithm is [Hart-Edwards-61], where one form of it is attributed to McCarthy but no reference is given. Some experiments on the algorithm's performance are described in [Slagle-Dixon-69], and some formal analyses of the effectiveness of the algorithm appear in [Fuller-et-al-73] and [Knuth-Moore-75]. (α-β is sometimes wrongly attributed to Knuth in the CGP literature; we wished to avoid repeating that mistake here.)

1. This is an instance of the general observation made by Doyle and others that it can be rational to adopt a belief, or at least act as if you believed it, even if (probabilistically speaking), you don't think it's likely to be true.

The α-β algorithm has received much attention in the computer game-playing literature. However, it is not of very much interest to us here, for several reasons. First, it does not lend us any new insights as to the meaning of node values or how to propagate them; it is merely a speed-up algorithm that computes the same value for a game tree that minimax would, but without examining all of its nodes. α-β does, however, serve as an example of how to incrementally utilize results from a partial search to guide the choice of nodes to examine in further search, a concept we will see more examples of later.

Second, although some authors have made much ado about how alpha-beta is "provably correct," what we should try to keep in mind is that, although the values it computes are indeed correct minimax values for the equivalent full-breadth partial tree, they are only correct, in any deeper sense, in the same sense that minimax is correct, i.e., only under the rather questionable assumptions mentioned in §2.4.2. There is still not much that can be said, theoretically speaking, about why a deep alpha-beta search (equivalent in its output to a full-width minimax search of the same depth) should actually lead to good game-playing decisions.

2.4.4 Pathology

Earlier (§2.4.2) I mentioned that under some attempts to model exactly what it is that an "approximate" evaluation function is doing, it is possible to prove that propagation via the minimax rule can actually degrade the quality of the approximation, rather than improve it. This indicates that, in some cases, deep searches may be futile. In this section I discuss these analyses in more detail, because they provide good examples of how to assign a clear semantics to node values. The move towards providing a clear node-value semantics is a vital step in the attempt to make sense out of game-playing algorithms and synthesize a "correct" approach. The probabilistic and decision-theoretic techniques discussed in §2.5, §2.7, and §2.8 similarly require some sort of well-defined semantics for node values.

The first exposition of pathology was in Ph.D. work by Dana Nau (see [Nau-80] for a summary). In Nau's model, evaluation function (EF) values are interpreted as being independent noisy measurements of nodes' true game-theoretic (GT) values. In other words, for each possible GT value, we have a probability distribution over what the EF value of a node with that GT value would be. We assume that the EF values for different leaf nodes are chosen independently according to these distributions.

Additionally, in Nau's original model, the distribution for a win node was symmetrical to that for a loss node, and the game trees had a restricted structure, but later it turned out that these properties of the model were not important for the pathology results.

Nau proved, within his model, that as search depth increases, the probability of choosing a game-theoretically optimal move using minimax (at nodes where both optimal and suboptimal moves exist) approaches the fraction of moves that are optimal, i.e., it approaches the probability that a random choice would yield an optimal move. These results were rather surprising, because in actual game-playing programs, increasing search depth has been empirically found to achieve higher move quality.

Nau's results were generalized in several ways by Pearl in [Pearl-83]. Pearl showed that for game trees of uniform structure (but less restricted in structure than in [Nau-80]), a very high rate of improvement of evaluation function accuracy with depth was needed to avoid pathology. But under a certain model of nonuniformity in the game tree, in which terminal positions are distributed randomly, pathology is more easily avoided, possibly explaining why real games do not seem to exhibit the phenomenon.

In [Nau-82], Nau investigates another way of explaining the lack of pathology in real games, via considering dependencies between sibling nodes in the game tree. In contrast, the model of evaluation function error used in the previous pathology results assumed that EF values were independent noisy measurements of GT values, even for nodes very close together in the game tree. One might suspect that in real games like chess, where most characteristics of a position only change incrementally, EF and GT values of neighboring nodes would be more closely related. Thus, Nau investigated a couple of "actual games," one in which the GT values exhibited such correlations, and one where they did not, and showed that pathology was avoided in the case where dependencies existed. However, the "board-splitting" games investigated were somewhat unrealistic. Not only did they *have* perfectly uniform branching-factor, uniform-depth game trees (unlike, e.g., chess) but also the game consisted *only* of a game tree—there were no high-level "rules" giving structure to the arrangement of game outcomes, other than the correlations Nau introduced at random. Moreover, the evaluation functions "cheated," in that they essentially examined the entire game tree below a node to determine its value, whereas in real games, the whole point of the EF is that the game tree too large to examine. A final problem with the board-splitting model is that it precludes investigating games having large numbers of moves, because the representation of a game position is essentially just a game tree, and so it is exponentially large in the number of moves left in the game.

Thus, the board-splitting game really is no more of an "actual game" than are the abstract, idealized game-tree models to which it is isomorphic. So it is somewhat misleading to draw conclusions about typical real games from results in the board-splitting domain. Further investigations after Nau's (e.g., [Abramson-85]) have continued to draw such conclusions, but I propose that in the future it would be better to just openly admit that the board-splitting "game" is really just an idealized game tree, and reserve the term "actual game" for games having more structure. It would be interesting to see pathology demonstrated in such games. In §2.5.4.5, I study one real game exhaustively, and show that pathology does occur in it (up to a point).

An interesting recent paper on pathology is [Delcher-Kasif-92]. They use the same model of evaluation function error, but show that if interior node values as well as leaf values are taken into consideration, the effects of pathology are reduced.

All the work in pathology has in common a certain semantics for evaluation function values, namely, they are treated as if they were noisy probabilistic measurements of the node's true game-theoretic value. However, for most evaluation functions, this semantics does not fit the actual situation exactly; the evaluation function is rather just some arbitrary function of position, and so there may be hidden interrelationships between the EF and GT values that the "noisy measurement" model fails to capture. But, as we will see later, it is possible to construct statistical evaluation functions that fit the probabilistic semantics more precisely.

2.5 Probabilistic evaluation

Partly spurred by the results on minimax pathology, researchers went on to consider alternatives to minimax that would abandon some of its assumptions. One intriguing interpretation of minimax node values we encountered in §2.4.2 was that node values represented our *expected* values for nodes, under our uncertain state of knowledge about what would happen next if the given position were actually reached. For a two-outcome (win/loss) game, the expected value is just equal to our probability of winning, under, at most, a linear transformation, so hereafter we will just talk about the probability of winning instead. However, recall that in order for minimax to make sense under this interpretation of node value, we had to make some rather implausible additional assumptions. In particular, we had to assume that our opponent shared our assessments, and that both players would continue to maintain the same assessments until one of the leaf nodes was reached. Could this assumption be changed to something more realistic while still maintaining a relatively simple propagation rule?

2.5.1 Modifying minimax's assumptions

One obvious way to change the assumption is to go to the opposite extreme: instead of assuming that neither player gains any new knowledge until a leaf is reached, one can assume that instantly after the current move is made, both players will gain complete knowledge of the true outcome that will occur at each leaf node,[1] thus enabling both players to play optimally (with respect to that knowledge) after the current move choice, until a leaf is reached. This means that the quantity we need to calculate at each child of the root (current) node is the probability that it is a win with optimal play down to the leaves, given the probability of winning at each leaf.

Unfortunately, a probability distribution over each of several variables (in this case the outcome of each leaf) is not sufficient information to give a distribution over the joint combination of the variables, unless we know whether and how the variables depend upon one another. A parent of a leaf will be a win (for the player to move from the parent) if and only if one of the children is a win (since we are assuming that that player will know the win/loss status of each child, and thus will make a correct move from the parent). But this condition is a property of the entire configuration of all the children, not of the children

1. This is not necessarily the game-theoretic value of the node; it is just an outcome that we somehow know will occur (perhaps, for example, because we discover that we will reach a certain kind of end-game that we know one of the players does not know how to solve).

taken individually. So without some sort of model of the dependencies between siblings, we cannot properly propagate probabilities from children to parent.

However, we could conceivably calculate bounds on the probability of the parent being a win, where the bounds would encompass the entire range of possible joint distributions we might have for the children that would be consistent with our individual distributions over each of them. How would this be done? We can derive this as follows.

First, we note that the minimum win probability at the parent (for the player to move there) is equal to the win probability of the child with the highest win probability. This is because we know that we will definitely win if that child is a win (remember, we are assuming we will learn the true values of the leaves before any future moves). But our joint distribution could be such that if the child is a loss, all the other children are losses as well, so the minimum probability for the parent is no more than for that child. Thus, the lower bound on the parent is the max of the children.

What about the upper bound? Well, it could be that all the children win in disjoint circumstances, so that the total probability of any one of the children being a win is the sum of the probabilities of each of them being a win individually—up to, of course, the maximum probability of 1.

Thus, we can accommodate our lack of a model of the dependencies between the value of leaf nodes by propagating probability bounds up the tree in this way. However, we will still have to make a rather arbitrary decision about how to choose between moves with overlapping probability bounds at the children of the root. So this solution is rather unsatisfactory.

Instead, we can make a new assumption to model dependencies between leaf values, so that we can derive a joint distribution over the children of a node from our individual distributions for each of them. Perhaps the simplest and most obvious model is just to assume that there *is* no dependence; the outcomes at the leaves are independent. Is this a good assumption? We will attempt to answer that question in §2.5.4 (p. 47). In any case, this assumption yields a propagation rule that has received much attention in the literature on CGP in AI: the probability product rule.

2.5.2 The product rule

In the previous subsection we assumed that the evaluation function values of a leaf represents our expected utility if we were to go to that leaf, or equivalently (with scaling) our probability of winning (in a 2-outcome game). We also assumed, in direct opposition to minimax's assumptions, that as soon as we pick a child of the root to move to, both ourselves and our opponent will gain complete knowledge of the "true" value of each leaf (say, by expanding the leaves completely), i.e., each player would know whether or not we would win if we went there. This allows both players (assuming basic competence on their part) to play optimally down to the leaves, after which the player certain to win from the leaf eventually does. Under this assumption, the probability of winning from a child of the root is just the probability that a random assignment of leaf outcomes (chosen according to the probabilities at the leaves) would make that child a win.

Now, suppose we also assume that our joint distribution over all leaves is just the independent combination of our distributions for the individual leaves' outcomes. We want a propagation rule that will tell us, for each child of the root, what our probability of winning is if we go there.

The correct propagation rule for this model was first stated by Pearl ([Pearl-81], §3.5, [Pearl-84], §10.2.4), although he was not as explicit as this about what "probability of winning" meant, and seemed to be referring to a node's probability of being a game-theoretic win, rather than just being a node that the current players would happen to win from. (However, the two models are equivalent in terms of their propagation rule.) Pearl's probability propagation rule is as follows.

We assume we have an examined partial game graph $G = (\Omega, \delta, M, p)$, similar to the G we used in §2.4.1 (p. 32), but with the $v(\cdot)$ function replaced by a function $p\colon \{s \in \Omega : \delta(\Omega) = \varnothing\} \to [0, 1]$ mapping leaves of the partial graph to our conditional probability of Max winning, given that play proceeds to that leaf. Now, we define the function $P\colon \Omega \to [0, 1]$ (in place of our earlier $V(\cdot)$) recursively as follows,

$$P(s) = p(s), \text{ if } \delta(s) = \varnothing; \text{ else} \tag{4}$$

$$P(s) = 1 - \prod_{t \in \delta(s)} 1 - P(t), \text{ if } s \in M, \text{ and} \tag{5}$$

$$P(s) = \prod_{t \in \delta(s)} P(t), \text{ if } s \notin M. \tag{6}$$

Together, these formulae are referred to, for obvious reasons, as the "product rule." Although these equations work only for the simple case of a two-outcome game, they can be extended straightforwardly to a game with several outcomes.

Pearl introduced the product rule almost just as an aside to a discussion about why minimax works, and about the phenomenon of pathology. Pearl originally hypothesized that the product rule might not show pathological behavior due to its increased "statistical validity" over minimax. But it is important to keep in mind that the product rule is only valid *under the assumptions* described above, assumptions that do not seem very close to the conditions that obtain in a real game. As a result, the product rule displays artifacts of its own: win probabilities may fluctuate between very high and very low values as search depth increases. (I replicated these results in experiments on several simple games, in which leaf win probabilities were not derived from an ad-hoc hand-tuned evaluation function, or even from the results of learning from many games, but rather from the true counts of the number of nodes in various classes that were wins or losses, out of the entire game tree. Details of these experiments are provided in §2.5.4, p. 50.)

Despite the product rule's flawed assumptions, researchers took Pearl's proposal seriously and proceeded to perform various experiments comparing the product rule to minimax and checking for pathologic behavior (e.g., see [Chi-Nau-87]). In general, it was indeed found to avoid pathology.

But, as noted above, the assumptions necessary for the product rule to make theoretical sense, although different from those needed for minimax, fail to be self-evidently more realistic than minimax's. The product rule, in a sense, takes the opposite extreme from minimax, in terms of its model of the players' future beliefs. So perhaps a more realistic model would tend to interpolate between the two models. The model introduced by Baum in [Baum-92] and explored further in [Baum-Smith-93] and [Baum-93] attempts to do just that. Baum and Smith develop workable assumptions that *are* pretty clearly more realistic than those used by either minimax or the product rule, but the result is a significantly more complex model and algorithm, requiring more time to develop to the point where satisfactory amounts of empirical data can be collected to validate the model's accuracy and effectiveness. Their technique has many interesting features that are relevant to our current efforts to understand how to use decision-theoretic principles in control of reasoning, so we will delve into their work in great detail in §2.8 (p. 69).

But for now, let us step back and take a more general look at the issues surrounding all these different ideas about how to "correctly" propagate node values through a state graph.

2.5.3 Node value semantics

If we wish to be able to make claims about the "correctness" of a particular propagation rule for game trees, we must first define a clear semantics for node values. Having a node value semantics also gives us a standard by which to judge leaf evaluation functions; namely, we can ask how closely they approximate the intended meaning—and we can even, for some meanings of node value, use statistical models to directly implement an evaluation function that embodies the desired meaning (for example, see the models I used in my experiments in §2.5.4 (p. 47), and the model I propose in §2.11, p. 96).

For games, the simplest node value semantics is just that a node value represents the game-theoretic value of the position. We will call this interpretation *GTV-semantics.* Minimax is the correct propagation rule for GTV-semantics. If you feed game-theoretic values into minimax, you get game-theoretic values out. However, in real games evaluation functions necessarily fail to capture GTV-semantics; they instead provide some sort of crude approximation to a position's game-theoretic value. Nothing can be said about the quality of the output of minimax given this sort of input, unless we have some sort of model of the relationship between the true game-theoretic value of a position and its evaluation-function value. We saw some examples of such models, and the results about minimax that can be derived from them, in §2.4.4 (p. 37).

Another simple semantics for two-player, two-outcome games, which we will call *P(GTW)-semantics*, is that the node value represents the (reasoner's) probability that the position is a game-theoretic win for (say) Max (for multiple-outcome cases we extend this to a complete distribution over all possible game-theoretic values). If we assume that the probabilities at all leaves are independent, then the correct propagation rule for this semantics is Pearl's "probability product rule," discussed above (although in that earlier discussion, we justified the product rule using a different semantics that we will call P(W|A) semantics and will define in a moment). If the independence assumption is abandoned, then we would require a model of the dependencies to derive a correct propagation rule. No one has done such modeling as of yet, to my knowledge.

P(GTW) semantics and the product rule extend easily to games with more than two outcomes—we replace the single probabilities with distributions, and propagate then by taking products of cumulative distribution functions, as in [Baum-Smith-93].

One nice feature of P(GTW) semantics is that, in principle, we can produce evaluation functions that embody the semantics exactly, by using statistical modeling based on real data about the game-theoretic value of positions similar to the one we are evaluating. (We actually do this in the experiments reported in the next subsection.) However, it is impossible to collect data on game-theoretic values for certain classes of positions (such as most positions in chess). So, often people substitute statistics about whether a game was *actually* won from a given position in a real game. But if one does this, one is no longer obeying P(GTW) semantics; another semantics must be defined to capture what one is doing with such statistics, and we must reconsider what the correct propagation rule would be for such an evaluation function.

Another problem with P(GTW) semantics, pointed out in [Baum-Smith-93], is that it is not really the right basis for choosing a move. Just because a position has a high probability of being a game-theoretic win, that does not necessarily mean that a real player would have a high probability of winning in that position, because a real player does not have infinite computational capabilities. So P(GTW) semantics is not what we want if we wish to use node values as a basis for move choice.

Instead, suppose we say that the value of a position is our probability that, in the current game with its current real players, the Max player would proceed to win in play from that position. But let us be more precise. In particular, is this probability intended to be conditioned on the information that the position is *reached* in actual play between our typical players? Or, on the other hand, are we supposed to have no information about how we got into the position (e.g., we assume we have been plunked down into a position chosen at random)? This is an important semantic difference. To see why, consider how you would collect statistics for an evaluation function. Suppose your statistical model works by partitioning the set of board-positions into many classes, each containing many possible board-positions, and statistics are collected by counting wins and losses for each class. For the first semantics, you would collect statistics by observing many games between the two players, and for each position reached in each game, tallying the game's final outcome in the position's class. Whereas, in the second ("no information") case, you would enumerate all possible board positions, and for each position, have the players play from that position, and tally the final outcome in the class of that position. We would not expect the statistics to come out the same in the two cases, because in the first case, some of the board positions in a given class may happen to come up in real play more often than others, and the ones that happen to come up may be either more or less easily winnable for Max than the others. Thus, the two versions of the semantics are different. We will call them *P(W|A) semantics* (for Probability that max would Win in actual play from the position, given that the position is Arrived at in an actual game) and *P(W) semantics* (for Probability that max would Win in actual play from the position, conditioning on no other information).

Let us now consider what the valid propagation rules might be for P(W|A) and P(W) semantics. First, P(W|A) semantics. Suppose we have an evaluation function *f* that embodies P(W|A) semantics. Suppose the game tree looks as in figure 4, with indicated

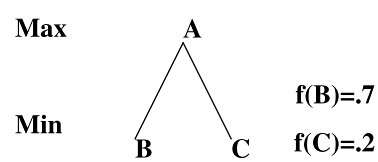
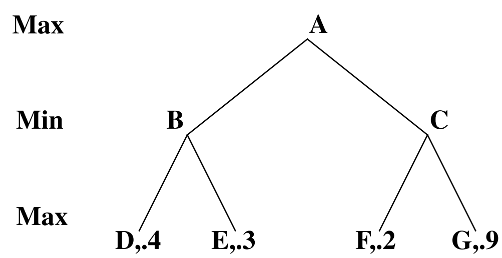


Figure 4. A very simple game tree.

values for *f*, and we are playing Max, and ourselves and our opponent are the Max and Min player(s) used in collecting the statistics for *f*, and play is currently at node A. Then clearly, we should move to B, because then the condition of arrival at B would be satisfied, and our probability of winning would be 0.7. Since we know we will definitely move to B from A, we know that our current probability of winning is also 0.7. So our propagation rule should give A a value of 0.7 in this case.

Now we consider a slightly more complicated case (figure 5). Clearly, to compute values for B and C, we will need to have some kind of model of what the opponent would do in those positions. One simplistic model would be to assume that the opponent will have exactly the same information available at position B or C as we have now. This allows us to predict exactly what min would do, (namely, go to E from B and to F from C), and the correct propagation rule for this tree would just be to use minimax (this sort of assumption was discussed in §2.4.2, p. 34).

Figure 5. A slightly more complicated game tree.



However, let us be a little more general, and just say that we have *some* kind of model of both players that yields, for each node *n*, a probability distribution $\vec{p}$ that gives, for each child of the node, our probability p_i that the player whose turn it is would move to that child. Then, given the vector $\vec{v}$ of the values of *n*'s children, the propagation rule for P(W|A)-semantics is just the expectation $\vec{p} \cdot \vec{v}$.[1] This is a sort of all-encompassing propagation rule that can be instantiated in many different ways, depending on how the $\vec{p}$ distributions are computed. In practice, we will want the $\vec{p}$'s to be computed using information from node values. For example, the $\vec{p} \cdot \vec{v}$ rule reduces to minimax if we assume that all players will definitely move to the child whose value appears best for that player to us right now. (In which case each $\vec{p}$ is a unit vector pointing to the highest or lowest child.)

What if we were to assume that each player were to play in a game-theoretic optimal fashion after the current move? Well, then, our idea of a "typical, real" player had better be that of an optimum player, and so the P(W|A) semantics reduces to P(GTW|A) semantics—values are probabilities that the nodes are game-theoretic wins, given that optimal players arrive at them. This is not quite the same as the P(GTW) semantics described earlier (again, consider the difference in how evaluation function statistics

1. I first encountered this idea in [Baum-Smith-93].

would be collected—in P(GTW|A) your sample set for collecting statistics would only include nodes in principle variations, because other nodes would never be arrived at by optimal players). However, if we make a (false) independence assumption, then the expectation rule $\vec{p} \cdot \vec{v}$ will reduce to the product rule, as with P(GT).

The technique described in [Baum-Smith-93] (discussed in §2.8 of the current document), can be seen as using P(W|A) semantics, but with a more sophisticated player model that takes advantage of extra information that is propagated along with the node values. Alternatively, we can view Baum and Smith's technique as embodying a new, non-scalar semantics for node value. This semantics, which we call P(V)-semantics, is that a node value is a distribution over the P(W|A) expected-utility value that a player would assess for the node after performing a typical amount of additional tree expansion below the node, beyond what has already been done. The mean of this distribution is taken to be our current P(W|A) value for the node. Baum and Smith's technique, which we will describe later, essentially assumes that all distributions are independent, and then propagates the P(V) distributions correctly, under this assumption.

Now, let us consider P(W) semantics. P(W) semantics is not really the right way to choose moves in a real game, because we are throwing away the information that real players actually *did* arrive at the position, which should tell us *something*. It is also harder to collect statistics for a P(W) evaluation function, because we have to enumerate all possible positions, or generate them randomly according to a flat distribution (which could be nontrivial). Because of these problems, nobody ever really uses P(W) semantics; instead P(W|A) semantics is almost always intended when someone says "the probability of winning from this node." Propagation of P(W) values is just like that of P(W|A) values.

Now, the node semantics aside, we note that several times in our discussions so far, the possibility has arisen of combining probabilistic information about child nodes to derive consistent probabilistic information about parent nodes. However, we noted that to perform such a combination required some model of the dependencies between our distributions for the different children. By far the simplest kind of model of dependency is just to assume there is no dependence, and indeed, that is just the assumption that Pearl, Palay, Nau, and others have made.

But the question remains: is this assumption valid, or even close to being valid? The next section presents the results of some original investigations on this matter.

2.5.4 Experiments on sibling interdependence

As discussed above, researchers have extensively studied the product propagation rule as an alternative to minimax. We saw that the product rule involved some unrealistic assumptions about players' future knowledge of node values. But even putting those assumptions aside, the product rule involves another fundamental assumption, one that is of special interest because a similar assumption is made in an interesting piece of later research that attempts to improve on both minimax and the product rule, that is, Baum and Smith's work [Baum-Smith-93], discussed in §2.8.

In the earlier work by Pearl and others, the independence assumption was just made, without much comment. But Baum and Smith, on the other hand, explicitly stated that they expected that in real games, the amount of dependence between siblings would be small, or that in any case that any dependencies that do exist are not expected to greatly affect the quality of play. Just in case, they incorporated a tuning parameter into their algorithm that was supposed to compensate for unexpected dependencies. However, in some preliminary experiments with the Baum-Smith algorithm, the empirically-determined optimal setting of this parameter was to perform a large amount of compensation, indicating that perhaps dependencies did exist. This was a rough, preliminary result that could be interpreted in other ways, and the Baum-Smith dependence assumption is not, on the surface, exactly the same as the product rule independence assumption.

Nevertheless, in my mind, this result made me question the heretofore mostly unquestioned independence assumption used by the product rule. So I decided to determine, once and for all, if the values of sibling nodes in some real games (as opposed to the random, structureless "board-splitting" games mentioned in §2.4.4, p. 38), were interdependent. In this subsection I report on my results.

2.5.4.1 A quantitative measure of dependence

In order to make our discussion of independence more quantitative, we will wish to define, given random variables X and Y, a quantitative measure $D(X;Y)$ called the *degree of dependence* of X on Y. We will wish that $D(X;Y) = 0$ if X and Y are independent, that $D(X;Y) = 1$ if X's value is fully determined by Y's value, and that $D(X;Y)$ should have some meaningful value in between 0 and 1, otherwise. The fairly obvious answer I came up with was to define $D(X;Y)$ to be the fraction of our uncertainty about the value of X that we would expect to be eliminated upon learning the value of Y. The standard information-theoretic measure of our uncertainty about a variable X is the *entropy* $H(X)$, defined as (see [Hamming-80]):

$$H(X) = -\sum_x p(x) \lg p(x), \tag{7}$$

where x ranges over the possible values of X, and $p(x)$ represents our probability for the event that $X = x$.

The entropy is simply our expectation for the amount of information we would expect to gain upon learning the value of X. It is always positive, and is measured in bits, if the logarithm is taken *base 2* (as "lg" indicates it is here).

Now, we can calculate the expected reduction of entropy of X upon learning the value of Y, by considering the probabilities of the different values for Y, and the entropies of the resulting distributions $p(X|\, Y{=}y)$. The result is the information-theoretic measure called the *system mutual information* of X and Y, $I(X;Y)$ defined as follows:

$$I(X;Y) = H(X) - H(X|\, Y), \tag{8}$$

i.e.,

$$I(X;Y) = H(X) - \sum_y p(y) H(X|\, y), \tag{9}$$

which, it turns out, can be computed by

$$I(X;Y) = \sum_x \sum_y p(x, y) \lg \frac{p(x, y)}{p(x)p(y)}. \tag{10}$$

This is always be a positive quantity. See [Hamming-80] for a more detailed exposition of entropy and system mutual information.

Now, armed with the concepts of entropy and system mutual information in (7) and (10), we give a definition of dependence that exhibits the desired properties mentioned above:

$$D(X;Y) = \frac{I(X;Y)}{H(X)}. \tag{11}$$

Note that this "degree of dependence" is a very general concept that can be applied to any system of statistical variables whatsoever. However, for now we will not explore applications other than our current concern with node values.

So now, armed with this measure, we are at least prepared to explain what we mean when we say that sibling nodes are more or less independent. But now, how would we go about measuring the degree of dependence of siblings in real games?

2.5.4.2 Measuring degrees of dependence between siblings

First of all, let us state what node value semantics we will be considering. As seen earlier, the values manipulated by the product rule can be interpreted in at least two ways: the more sensible one is that they represent our probability that, with the current players, Max would win from the given node, assuming that both players immediately discover, for each leaf of the current partial graph, whether the current Max player would win from that leaf. (This does not necessarily mean knowing whether the leaf is a game-theoretic win. The discovery of the outcome of a leaf might instead involve expanding a leaf to find that all its children introduce endgames that we know from experience to be very difficult for the person playing Max, but we might not know the true value of the leaf.)

However, it is difficult to work with this sort of semantics in general, because we would need models of the players. So let us consider the other (simpler but possibly less realistic) interpretation of product-rule node values, namely that they are probabilities that the given node is a game-theoretic win. This is more satisfactory for our purposes, since we can actually determine the game-theoretic values of nodes in small games, and there is no question about how to model some vague thing like players' knowledge of each other.

To find degrees of dependence, we need a joint probability distribution over two variables, from which we can compute the value of equation (11). In our case, the two variables will be the game-theoretic value of a node, and that of its siblings. If the game is small, we can find out exactly what the ideal distribution would be, by counting, for each joint (x, y) pair, how many ways we can have a node with a value x having a sibling with value y.

But this is still incomplete. We need to specify what these "ways" are, and how they will be counted. We need to specify which nodes we are examining, and which siblings, and how many of each we are considering. The "real" joint distribution we are seeking is a meaningless concept, except when it is relative to some definite method of counting these things.

I considered several possibilities methods of counting. One was to count once for each possible node-sibling pair out of a full game-tree.[1] But then, I thought, this way of counting is biased towards nodes having lots of siblings, because the more siblings a node has, the more ways it would be counted. So, I considered normalizing the count, so that if a node had 4 siblings (apart from itself), each pair of the node and one of its siblings would count 1/4, so that we would have a total count of "1" for the node. (Actually, "2", in my program, because all the pairs would be counted in the opposite order as well, but this constant factor over all nodes makes no difference to the probability distribution obtained.)

But then, I reflected that I was being biased towards nodes deep in the tree, because there are many more deep nodes than shallow nodes. This did not seem to make sense for evaluating partial-search algorithms, because when we are playing through a game, we will spend roughly as much time early in the game doing computations on partial trees located near the top of the tree as we will spend late in the game doing computations on partial trees located near the bottom. So instead, our statistics should reflect equally nodes that might occur throughout all stages of the game.

So finally, I settled on weighting each node by its probability of occurring in a game—but that requires a model of the players again. Realistic player models were not really feasible, so as a rough approximation, I weighted each node by its probability of occurring in the game, given random moves by both players (i.e., flat distributions over the set of legal moves), or rather, the expected number of times it would occur. This can also be viewed as counting the number of times we would encounter a node in an infinite sequence of randomly-played games. (Of course, this number is infinite for all reachable nodes, but we can still define the relative counts, in the limit.)

1. Note, by exploring a *tree*, not a DAG, we mean there is no merging of transpositions. That is, our nodes correspond to histories of game-positions, rather than to game-positions themselves. In other experiments we used DAGs instead.

In the end, it turned out not to matter much—as we will see in a moment, for all these ways of counting, and for all of the games I explored, the degrees of dependence turned out to be very small; in fact, in some cases they were not significantly larger than you would get by accident in identically-structured games having random leaf assignments.

2.5.4.3 Experiments on tic-tac-toe

For my experiments I required a game whose game-graph would be sufficiently small to explore in its entirety, so that I would know for certain that the statistics were not just an artifact of the particular subset of the positions that I was exploring. Tic-tac-toe is such a game, and yet it is a game interesting enough to actually be played with interest by human players (although, admittedly, usually very young ones). For coding simplicity, I did not bother with symmetry reductions or transpositions; the game tree was still a very manageable size. Here are some basic statistics of the game tree, as computed by my program:

Table 1: Basic statistics of the tic-tac-toe game tree.

Number of nodes	549,946
Number of terminal nodes	209,088
Number of (ordered) pairs of siblings (metric 1)	740,160
Number of nodes having siblings (metric 2)	422,073
Expected number of nodes with siblings that would be encountered in a randomly-played game (metric 3)	7.273810

The last three items correspond to the three different ways I mentioned above of counting statistics for the dependency computations. For example, table 2 shows, under the second way of counting, how many of the nodes with siblings have each of the three possible game-theoretic values.

However, what we are really interested in, for each of these ways of counting, are the joint distributions between the nodes and their siblings. So, for example, table 3 shows

Table 2: Distribution of game-theoretic values for tic-tac-toe nodes having siblings. The entropy of this distribution is 1.518 bits.

	Win for "O"	Draw	Win for "X"	TOTAL
Number of nodes	148,708	86,101	187,264	422,073
Probability	0.352	0.204	0.444	1.0

Table 3: Joint distribution for pairs of siblings, weighted to equally represent all nodes that have siblings.

	O Wins	Draw	X Wins
O Wins	0.143	0.050	0.159
Draw	0.050	0.066	0.088
X Wins	0.159	0.088	0.197

the resulting joint probability distribution over game-theoretic values of nodes having siblings and their siblings. (Again, collected using the second way of counting.)

To illustrate what we mean by the entropy of one variable given another $H(X|\,Y)$, mentioned in equation (8), table 4 shows the distributions over node value we would get if

Table 4: Distributions of values for a node, given the value of a sibling. (Normalizing the rows of table 3.)

Sibling value	Distribution over node value			Entropy
	O Wins	Draw	X Wins	
O Wins	0.359	0.198	0.443	1.51
Draw	0.244	0.326	0.431	1.55
X Wins	0.407	0.141	0.452	1.44

a sibling value were given. Note that in the case where the sibling is a win for X, which we saw above was the most likely case, the entropy of our distribution for the node's value is decreased to 1.44 from the 1.52 entropy of the original distribution, so we are gaining some information. (If the sibling is a draw, that is unexpected, and our uncertainty about the node's value goes up; however, that case is low probability. The expected increase in information is always positive.)

The entropy of the distribution in table 2 is 1.518 bits, whereas the system mutual information of the joint distribution in table 3 is 0.0223 bits. Thus, according to our definition of dependence in (11), we get 0.0147 for the average dependence between siblings in tic-tac-toe. This does not intuitively seem to be a very high degree of dependence, and indeed, we will see that comparisons with the degree of dependence of game trees having purely random leaf outcomes bear out this perception somewhat.

But first, let us summarize the results for our different ways of counting. In table 5, X is the value of a node, and Y is the value of its sibling.

Table 5: The calculation of the degree of dependence in tic-tac-toe, for each of the three different statistical counting methods used.

Method of Counting	**Mutual Information I(X;Y)**	**Entropy H(X)**	**Degree of Dependence D(X;Y)**
Pairs of Siblings	0.04222	1.497	0.02819
Nodes with Siblings	0.02230	1.518	0.01469
Nodes w. sibs in random game	0.1875	1.503	0.12470

We can see that the degree of dependence was highest (0.12) in the case when nodes were weighted based on their probability of occurring (assuming random play), which was the counting method that we decided above was the fairest. However, that degree of dependence still seems rather low.

It occurred to me that dependencies might be highest deep in the game tree, where outcomes might be largely predetermined by moves made earlier, so I separated the statistics by depth. Table 6 summarizes the results. (Note that in tic-tac-toe it happens that all three ways I mentioned of counting statistics are identical for any given depth, because every node at a given depth has the same branching factor and the same probability of being reached in a random game. Thus we only need to give one measure of the degree of dependence here.)

Table 6: Summary of statistics for nodes at different depths. N/A means the quantity was not applicable because there were no nodes that had siblings.

Depth	# nodes	nodes w. siblings	Entropy	Mutual Information	Degree of Dependence
0	1	0	N/A	N/A	N/A
1	9	9	0	0	undefined
2	72	72	0.9183	0.0009	0.0010
3	504	504	1.4409	0.0242	0.0168
4	3,024	3,024	0.9420	0.0214	0.0227
5	15,120	15,120	1.5270	0.1358	0.0889
6	54,720	54,720	1.1887	0.0487	0.0410
7	148,176	148,176	1.5161	0.0933	0.0615
8	200,448	200,448	1.5459	0.0371	0.0240
9	127,872	0	N/A	N/A	N/A

We can see that although the degree of dependence is different at different depths, it is still never larger than 0.0889 (at depth 5).

One factor we should perhaps take into consideration, when interpreting these results, is that these joint probability distributions over the values of nodes and their siblings are conditioned on no information, other than the general method of counting nodes. But in a real program using the product rule, node win probabilities would in practice be estimated by observing some feature of the node (for example, in Nau's board-splitting research that was mentioned earlier, the feature used was the number of winning leaves below the node), and the corresponding statistics would count nodes having identical values for that feature. I considered it possible that, within some such restricted classes of nodes, there would actually be higher levels of dependence; for example, if the class picked out nodes that tended to be in regions of the tree where large contiguous regions of nodes tended to all have the same game-theoretic value.

So, I devised a simple 4-valued classifier, which classified each tic-tac-toe position in terms of whether each player had a threat (i.e., two-in-a-row with the third space empty), and collected statistics separately for each of the 4 classes of nodes. (This classifier would be a reasonable board feature to use in an evaluation function.) See table 7 for results using this classifier.

Table 7: Degrees of dependence for classified nodes. Metric 1 counts pairs of siblings, metric 2 counts nodes having siblings, and metric 3 counts nodes having siblings in a random game

Classification	# nodes with siblings	Degrees of Dependence		
		Metric 1	Metric 2	Metric 3
no threat	34,121	0.02570	0.01960	0.14140
X threat	101,968	0.05120	0.03360	0.07170
O threat	65,536	0.01640	0.02900	0.05130
X and O threat	220,448	0.00018	0.00580	0.03250
(All Classes)	422,073	0.02819	0.01469	0.12470

We can see that, even among these specific classes, the highest level of dependence is still only 0.14. Is even this level of dependence significant at all? A natural way to test this is to see is whether it is higher than we would expect to get out of a random game tree. To make the comparison as close as possible, I took the identical tic-tac-toe game tree but substituted randomly-chosen values (out of the set {X wins, draw, O wins}) at the terminal positions, propagating those instead of the real tic-tac-toe values. Results are summarized in table 6 and table 9.

Table 8: Sibling dependence for different depths in a tic-tac-toe-shaped tree with random terminal node values. Compare with table 6.

Depth	nodes w. siblings	Degree of Dependence
0	0	N/A
1	9	undefined
2	72	0.00267
3	504	0.00156
4	3,024	0.00635
5	15,120	0.00044

Table 8: Sibling dependence for different depths in a tic-tac-toe-shaped tree with random terminal node values. Compare with table 6.

Depth	nodes w. siblings	Degree of Dependence
6	54,720	0.00043
7	148,176	0.00003
8	200,448	0.00001
9	0	N/A

Table 9: Sibling dependence for different node classes and ways of counting, in a tree with random terminal node values. Compare with table 7.

Classification	# nodes with siblings	Degrees of Dependence		
		Metric 1	Metric 2	Metric 3
no threat	34,121	0.00909	0.00246	0.09967
X threat	101,968	0.00631	0.00309	0.03973
O threat	65,536	0.00271	0.00064	0.02371
X and O threat	220,448	0.00610	0.00249	0.01629
(All Classes)	422,073	0.00575	0.00232	0.07230

The first thing to note is that the degree of dependence is extremely low for nodes deep in the tree, which is what we would expect. It in interesting, however, to note that the degree of dependence increases as one nears the top of the tree. This is presumably just an artifact of the smaller sample size at the shallower depths. One disturbing thing is that we node that at depth two, the degree of dependence is actually higher than in the real tic-tac-toe game tree. This indicates that there is basically no measurable dependence at that depth in the real tree.

Let us now look at the classified-node results in table 9. We note that they are generally somewhat lower than the dependencies measured in the real tree. However, note that in metric 3, which we had decided was the fairest metric, the differences are less than a factor of two. The largest amount of dependence we have measured, e.g., 0.14 in the no-threat metric-3 case, is not much larger than the corresponding dependence, 0.10, in the random tree. This seems to indicate that most of dependency we are seeing is an artifact of our way of measuring it, and is not an important aspect of the game of tic-tac-toe.

On reflection, this is not too surprising. Tic-tac-toe does not have much of a "positional" aspect; most positions are not clearly "strong" or "weak" positions, and the number of positions is small in which one side clearly dominates the position in the sense that most of the different moves that might be made from the position had the same value. It is more often true that there are moves with many different game-theoretic values available. But we might expect that if we were to look at a more positional game where one player could get far "ahead," then at a typical node, one player would have a winning position, no matter what moves was made next; and thus a node's value will typically be correlated with who is ahead at its parent, and thus with its siblings' values, as well.

So, on the theory that perhaps the large apparent amount of independence was a special feature of tic-tac-toe, I investigated another small game played by human players that was thought to be more positional.

2.5.4.4 Experiments on Dodgem

In [Nau-82], Nau suggests an "incremental" version of his board-splitting game (called an "N-game") as an example of a game with dependencies, and shows that it eliminates minimax pathology. However, as I stated in §2.4.4, I do not consider the board-splitting model to be a good substitute for actual games; experiments on it would not satisfy me in my quest to learn about dependencies in real games.

So instead, I chose to experiment with the game of Dodgem [Berlecamp-et-al-82], which was suggested by Carl Witty[1] as an example of an incremental game that is still small enough to be amenable to a full search for purposes of collecting dependency statistics.

The rules of the game are as follows. The game is played on a 3x3 board with 2 pieces per side. The initial position is shown in figure 6. The goal for each player is to move both his pieces across the board and off the opposite side. White moves first. On his

1. In personal discussions. The version he described differs slightly from the one analyzed in [Berlecamp-et-al-82] in that a position with no legal moves is considered a draw in his version, but is a win in theirs.

turn, a player must move one of his pieces one square forward or to the side (as shown), either moving to an unoccupied square, or moving off the edge of his goal side of the board. The game ends when one player gets both of his pieces off the board, or the player to move has no legal moves. If the player to move has no legal moves, or if the game goes on forever, then the game is a draw; otherwise whichever side moves both his pieces off the board first is the winner.

In this game, the game tree is infinite (both sides can just move a piece back and forth forever), so the full tree-search method used for collecting statistics in tic-tac-toe will not work in this case. However, we note that the number of game *states* is finite and small (in fact, it is 1,963), so if we can figure out how to extract our dependency information from the game graph instead of the game tree, we can still obtain exact results.

The game-theoretic values for all positions can be computed by the algorithm shown in figure 7, which was worked out by Carl Witty and myself. The algorithm assumes that all terminal nodes have already had values out of the set {win for white, draw, win for black} assigned to them, and that all non-terminal nodes are originally marked as "draw."

The algorithm works as follows: it makes a number of passes through the graph. On each pass, it looks at each nonterminal node that is still labeled "draw." If *any* of the node's children are labeled "win" (for the player to move at the parent node), then we update the parent to "win." If *all* of the node's children are labeled "loss" (again, for the player to move at the parent), then we update the parent to "loss." If no nodes change

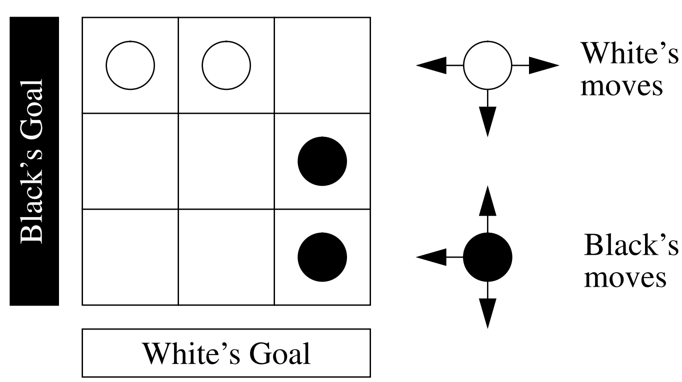


Figure 6. Initial position in Dodgem.

labels on a pass, then we are done. Why is this final labeling correct? For nodes labeled "win" and "lose," their correctness follows obviously from the definition of minimax in equations (1)-(3). For nonterminal nodes labeled "draw," the proof is still easy, but is omitted here, for brevity.

The worst-case running-time for the algorithm is $O(n^2)$ in the number of nodes n. The worst case occurs, for example, when the graph is one long chain or when the graph is fully-connected. For intermediate cases such as this game, the running time is less; in general it is $O(ndb)$, where b is the maximum branching factor and d is defined as the maximum length of any directed path not involving repeated positions that ends at a terminal node. (For this game, $b = 6$ and $d = 21$, so less than $1963 \times 6 \times 21 = 247,338$ iterations of the inner loop are required, in fact, the actual number of iterations is only 18,574.)

Now, to collect statistics on sibling dependence, I need not only data on node values but also a counting method. (This was discussed in the previous section.) In Dodgem I cannot count pairs of siblings in the game tree, or numbers of game-tree nodes having siblings, as I did in tic-tac-toe, because in this game there are an infinite number of them. However, I can iterate over the game-positions that have at least two children, and count over pairs of their children, or over their children. It even turns out I can calculate the number of times we would expect each position to be reached in a random game, just as

```
anyChanges? = true
while anyChanges?
    anyChanges? = false
    for i = 0 to nNodes - 1
        if value[i] = draw and nSuccessors[i] > 0
            allKidsLose? = true
            for s = 0 to nSuccessors[i] - 1
                j = successor[i, s]
                if value[j] ≠ loseFor(playerToMoveAt[i])
                    allKidsLose? = false
                    if value[j] = winFor(playerToMoveAt[i])
                        value[i] = winFor(playerToMoveAt[i])
                        anyChanges? = true
            if allKidsLose? then
                value[i] = loseFor(playerToMoveAt[i])
                anyChanges? = true
```

Figure 7. Pseudocode for game-graph labeling procedure for 3-outcome games.

we did for game-tree nodes in tic-tac-toe. The resulting dependence statistics in this third case are identical to what they would be if we searched the infinite tree instead of the graph, facilitating direct comparison with the tic-tac-toe results.

The algorithm for computing these “weights” of positions, also due to Carl Witty with details worked out by him and myself, goes as follows. As before, we conduct a number of passes through the graph. On each pass, for each node, we set its weight to be the sum of weights propagated from its parents, except that for the initial position, we add 1, because that position is also reached from the “outside” (i.e., not from any predecessor position) exactly once in any game (at the start).[1] If a parent has b children, $1/b$ of its weight is propagated to each child. If the graph contains cycles (as it does in Dodgem), then some of the weight from a node will eventually cycle back to it, increasing its value still further. If the cycle has no outlet, then the weight will increase indefinitely; however, if there a path from the cycle to a terminal position (as is always the case in Dodgem), then the weight of each node will converge on a finite value (the expected value of an exponential-decay distribution). On each pass, a constant number of additional bits of the correct result will be computed, and computation will halt when all bits have been computed down to the limit of precision of our floating-point numbers. In practice, in Dodgem, only 55 passes were required for all node weights to converge to definite double-precision values.

Based on the computations, it turns out that the expected length of a random game of Dodgem is 19.358081 half-moves (moves by one side or the other), and the expected number of nodes encountered that have siblings (including the initial node) is 17.972556. When nodes having siblings are weighted according to the number of times they are expected to be encountered in a random game, the joint distribution of node and sibling values is as shown in table 10.

Next we will show the overall results for all three counting methods (pairs of children of position nodes, children of position nodes, and children of nodes weighted by their probability of occurring in a random game). Only the third measure is comparable to the data obtained for tic-tac-toe, because in the tic-tac-toe game tree the first two measures

1. In Dodgem the initial node actually happens not to have any predecessors, but in general game-graphs, the initial node may generally have predecessors.

Table 10: Joint distribution of node and sibling game-theoretic values in Dodgem, when node-counting is based on the expected number of occurrences in a random game.

Node Value	Sibling Value: Black Wins	Sibling Value: Draw	Sibling Value: White Wins	Total
Black Wins	0.337564	0.009809	0.141333	0.488706
Draw	0.009809	0.001341	0.019723	0.030873
White Wins	0.141333	0.019723	0.319366	0.480421

could be seen as biased towards positions that could be reached in more than one way, relative to the case in Dodgem, where the first two measures are biased against such nodes (since they only occur once in the game graph). However the third measure is measuring exactly the same thing in both cases, namely this: for a position chosen at random out of an infinite concatenation of randomly-played games, if there were alternatives to the move that arrived at that position at that point in the game, and one of those alternatives is chosen at random, what is the joint distribution over the game-theoretic value of the original move, versus the value of the alternative move? The dependence results are shown in table 11.

Table 11: For Dodgem, total counts and degrees of dependence for three counting measures. Only the third measure can be compared directly to the results for tic-tac-toe shown in table 5.

Counting Method	Total Count	Degree of Dependence
Pairs of kids of position nodes	12,464	0.284458
Kids of position nodes	5,196	0.270072
Nodes w. sibs in rand. games	17.97256	0.099197

We note that although the first two degrees of dependence are much higher than anything we have seen so far, the third dependence measure, which is considered the “fairest” one, and is the only one that can be compared directly to the corresponding result for tic-tac-toe, is, surprisingly, actually somewhat less than it was in tic-tac-toe.

Clearly, more experiments are required to understand what is going on. Some possibilities:

- Construct a game-graph version of the tic-tac-toe analyzer, and see if the degrees of dependence for the first two counting methods are higher than for the third method, as they are in Dodgem.
- We note that neither game provides us with a really large sample size, since there are not very many positions in the entire game tree. Indeed, the measured dependencies could even be partly artifacts of the small sample size, as suggested by the data in table 6 where the smaller sample sets have larger dependencies. Games where the initial position is chosen at random could offer larger data sets and more certainty that the dependencies are not artifacts.
- To begin to connect these results with efforts in more "serious" games such as Chess, Charles Isbell[1] suggests analyzing a variety of simple Chess endgames, such as the KPK class of endgames.
- To connect these results to the results in the theory of minimax pathology, we should measure dependencies of the N-type game trees studied by Nau.
- Finally, to see if these dependencies are significant, and to connect these results with the efforts to defeat pathology by using the product rule, we should analyze these games to look for evidence of pathology using both propagation rules.

The last of these items has already been done, for Dodgem; results are reported below.

2.5.4.5 Comparing minimax to the product rule in Dodgem

The degrees of dependence reported above, 0.12 for tic-tac-toe and 0.09 for Dodgem, seem small when compared to a "complete" dependence of 1.0. And yet, we saw that among only 72 nodes at depth 2 in tic-tac-toe, the degree of dependence is fifty times less—0.00267—when the leaf assignments are random. Perhaps our intuitive perceptions

1. Personal discussion.

are amiss, and these dependencies such as 0.09 are actually very large, in terms of their effect on the performance of game-playing algorithms.

To find out, I have started to perform experiments to directly measure the performance of the game-playing algorithms in the game of Dodgem. To do this, I defined a simple position-classifier, like I did for tic-tac-toe. Dodgem positions were classified according to the relative total distance of each side's pieces from the goal (13 classes ranging from white-ahead-by-6 to black-ahead-by-6). Within each class I counted win-draw-loss positions, weighted by their expected number of occurrences in a random game. This yielded a good evaluation function of positions that would map positions to the probability distribution over their outcomes (assuming they are positions out of randomly-played games). Then I used a dynamic programming method, similar to the propagation methods for game-theoretic values described above, to compute the backed-up minimax and product-rule values for all nodes and all depths. Then I collected statistics on what fraction of nodes (weighted by the expected number of occurrences in a random game) a correct move choice was made by each algorithm, as a function of depth. These data are shown in figure 8. Unfortunately, my evaluation function is too good; a correct move choice is made

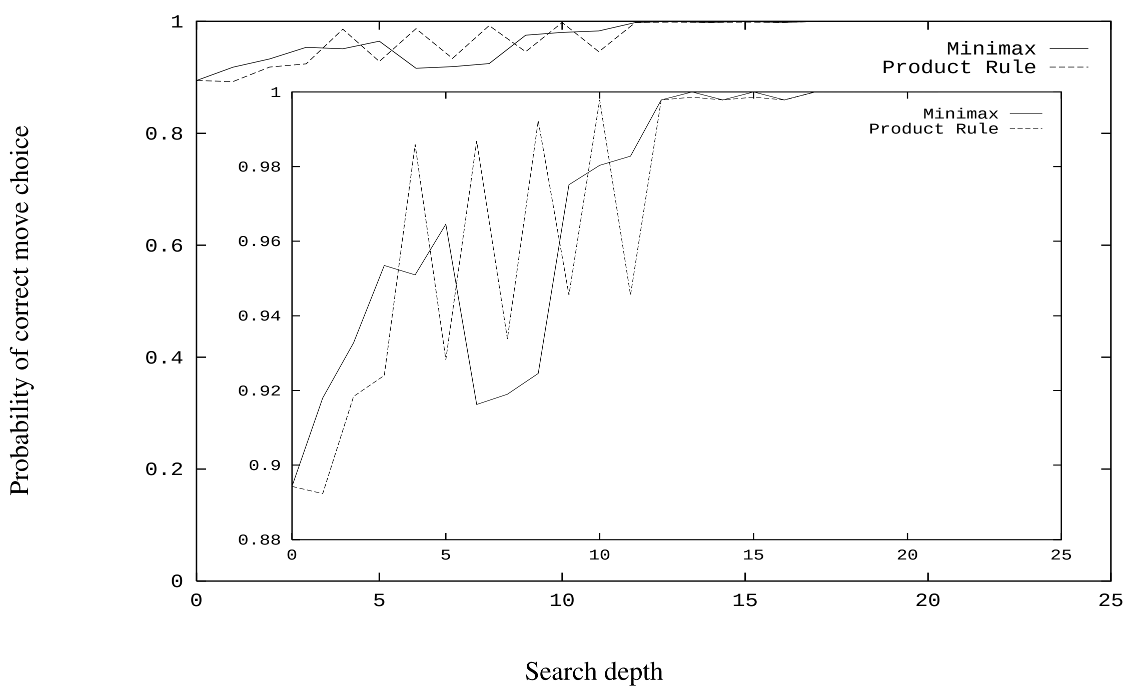


Figure 8. Probability of a correct move being made at a node for different depths of minimax and product-rule propagation. Inset shows expanded vertical scale.

at over 89% of nodes even when their children are only searched 0 levels deep (i.e., using the straight evaluation-function value of the child). Thus there was not much room for much improvement with further search. However, looking at the graph, we can see several interesting features:

- The minimax algorithm does exhibit a form of pathology, in that, for example, with a depth-6 minimax search, correct moves are made at several percent fewer nodes than with depth-5 search.[1]
- The product rule seems to be fluctuating between high and low values for many nodes. More detailed studies are needed to see exactly what is going on here.
- Both propagation rules reach 100% decision accuracy by depth 21, because in this game all nodes happen to have a solution tree of no more than that depth.
- Minimax and the product rule seem to improve, very roughly speaking, at the same rate, within the bounds of the fluctuations of the product rule curve.

Thus, we can see that perhaps the 0.09 degree of dependency in this game is, indeed, *not* sufficient to avoid pathology, and thus perhaps *is* a small value, validating our intuitive perceptions and justifying the product rule's independence assumption. However, we do not here see evidence that the product rule is doing significantly better than minimax, and the fluctuations are an annoying artifact. Perhaps further experiments on larger sample spaces will be better able to resolve the question of the importance of dependency and its bearing on the performance of the two propagation rules.

2.5.4.6 Conclusions regarding the independence assumption

What can we conclude from the above experiments? First, there are several qualifications we must make when trying to interpret these results:

- In these experiments I am measuring dependence between true game-theoretic values. However, for playing a real game with real players, we should be more concerned with the *actual winnabilities* of nodes, i.e., whether the actual cur-

1. Ordinarily one might question whether a few percent is statistically significant; however, note that in these experiments we are examining *all* nodes, not taking a random sample.

rent Max player really would win from a given node, and whether these node statuses are correlated between siblings.

- Also, the real independence issue is not just whether the actual outcomes of two nodes are independent, but also whether the evaluation-function assessments for those nodes are conditionally independent, given the outcomes.
- These statistics about nodes are collected under the assumption that the node in question is chosen randomly out of an infinite concatenation of games between random players. If we consider nodes chosen according to a different distribution, such as a flat distribution over game-tree nodes or game-graph nodes, we will get different results. None of these models of node distributions are actually *correct*; if we are concerned with the performance of an actual player in games against other players, we should really be picking nodes that would arise in the actual player's actual games. This could not really be done for the simple games we were considering, because a reasonable "actual" player would just be a perfect one, and all its moves would be to game-theoretic win nodes.
- Finally, and perhaps most importantly, I should point all that all the above dependency measures are collected between only *pairs* of nodes. But *pairwise independence does not imply mutual independence* between a set of variables. In other words, even if pairs of nodes are completely independent, there might still remain strong dependencies among sets of three or more siblings. This could be measured easily enough by taking the second variable in our joint distribution to be the vector of values of all of a node's siblings, instead of just the value of a randomly-chosen sibling. (Or, if we wish to ignore ordering information, a we could take this variable to be triple giving the number of wins, draws, and losses among the siblings.)

In summary, the results so far are very inconclusive, and more research is needed. The degree-of-dependence measure defined in §2.5.4.1 (p. 50) has not yet been firmly established as a good, solid measure that can accurately predict phenomena such as pathology, and we have not yet firmly established that real games do or do not have significantly high levels of dependence. However, these experiments at least seem to provide circumstantial evidence that perhaps the independence assumption is not too bad, and leaves open the possibility that future generations of game-playing programs may make fruitful use of probabilistic evaluation techniques that utilize this assumption.

Even if, despite these results, the independence assumption does prove to be a fatal flaw in the existing probabilistic algorithms (for some games), there does remain the possibility of developing probabilistic methods that actually *avoid* the independence assumption, via some sort of probabilistic modeling of the dependencies between pairs or sets of siblings. However, what form such a model would take remains unclear.

2.6 Selective search

So far in this chapter we have examined a variety of methods for evaluating partial state-graphs, but all under the assumption that the partial graph we were considering consisted of exactly all states that are reachable from the current state within some constant number of transitions. This is a rather arbitrary choice of partial graph. Even Shannon [Shannon-50] observed that human players generally search farther down some lines of play than others, and suggested that future versions of his computer chess player might do the same.

Early in the history of CGP, there were attempts to use "forward pruning," which avoided exploring moves that were judged heuristically to be poor choices. But these attempts were largely unsuccessful ([Smith-Nau-93a] attempt to analyze why this happened).

Later, Berliner developed a selective search procedure, B* [Berliner-79]. The B* algorithm depended on evaluation functions providing bounds to indicate a range of likely values, and propagated these bounds up the tree. This allowed search to focus on parts of the tree where tightening the bounds would help most in demonstrating that one move at the top level was better than all others. However, Baum and Smith have pointed out [Baum-Smith-93] that Berliner's attempt to discriminate one move from all the rest is not necessarily the right thing to do (if there are several equivalent best moves, B* would waste time trying to prove that one is better than the other). We will look at the question of the “right thing to do” in search control in more detail when we discuss Baum and Smith’s methods in a later section.

Another general but somewhat *ad hoc* technique for selective search was developed by Rivest [Rivest-88]. His method used differentiable approximations to the min and max functions in minimax search to allow a searcher to expand the node upon which the root value depends most sensitively.

McAllester [McAllester-88] proposed a technique for selective search based on counting the number of nodes below a node whose evaluations would have to change (in “conspiracy” together) to change the node’s value. Smaller number of changes are considered more likely, and thus, counting these “conspiracy numbers” allows search to be focused towards parts of the tree most likely to change the choice of best move. Some experiments with the technique are reported in [Schaeffer-91], and [Allis-et-al-91a] devises a version that uses a pruning technique similar to alpha-beta. [Allis-et-al-91] describes a specialized version of conspiracy search called proof-number search that is optimized for finding solution trees; it has been used in solving several games (proving their game-theoretic values).

All the above selective search methods are general, in that they can be applied to any number of different games. However, it turns out that special-case methods for particular games have been more popular and successful in competitive game-playing programs, so far. The method of “singular extensions,” a relatively specialized technique for choosing places in the chess tree to search more deeply, was developed by the team working on the Deep Thought chess playing machine. (See [Anantharaman-et-al-90] and [Hsu-et-al-90].) Singular extensions remain the dominant technique for chess.

However, I believe that as CGP researchers focus on a wider variety of games, the more general varieties of selective search techniques will gain in popularity, since they do not have to be reinvented for each game.

But still, it is difficult to know how to compare the different selective search techniques. There does not seem to be one outstanding, obviously “right” method. And it is hard to say what the deeper principles are behind these techniques that would lead to their application outside the realm of chess-like, two-player, zero-sum games. It would be desirable to have selective search methods that taught us general lessons about control of search, and of reasoning, in a more general context. This is the motivation for the more recently developed decision-theoretic techniques which we will now describe.

2.7 Decision-theoretic search control

All of the methods for selective search discussed in the previous section seem to be trying to approximately "do the right thing" with regard to tree growth—but what exactly *is* the ideal "right thing," how can we formalize it, and how can we then implement something that more explicitly tries to approximate that formalized ideal? The tools of decision theory offer a general approach for making "right" decisions, but those tools had not even been referred to by the selective-search CGP community.

Eventually, however, these ideas began to find purchase. A method for game-tree growth based on decision-theoretic principles was described by Stuart Russell and Eric Wefald and implemented in their program MGSS*. ([Russell-Wefald-89], [Russell-90]). Russell and Wefald had a number of new fundamental insights. One was to characterize the problem of tree growth in terms of a meta-level search of possible sequences of computational actions, just as base-level game-playing characterizes the problem in terms of a search of sequences of game actions (moves).

Another insight was that the utility of a computational action can be defined in a principled way, by estimating how that computational action (such as a leaf expansion) will improve our expected utility for the overall, externally-directed decision we will make. In other words, computational utility should be tied to real-world utility. Russell and Wefald formalized this idea fairly well, at least for state-space domains with small numbers of actions. Another fundamental insight was that in order to compute computational utilities defined in this way, it was necessary to assign nodes distributions that describe what our estimate of the node's value will be in the future (after we expand it), as opposed to the distribution over game-outcomes used by product-rule tree evaluation. This constituted a fundamentally new node-value semantics, one that explicitly encompasses knowledge about not just the game-position itself, but also about our epistemology, our knowledge about the game position. This is an example of why it can be useful to be explicit about epistemology, because our programs might need to model or reason about their own knowledge in order to do well. We will delve into this matter more in chapter 3.

A third basic contribution of Russell and Wefald's work was to define the first-ever principled criterion for deciding when to stop searching: stop when the expected utility of further search becomes less than the cost of the time required to do it. (Of course, to make use of this criterion required a relatively simple model of the cost of time, one that might be appropriate for games but not for other domains.)

MGSS* contained many fundamental advancements, but it had its problems. It assumed that the error distributions at nodes were normal distributions, and this required a lot of messy approximations to properly propagate these distributions up the tree. However, at least they were trying to approximate the "right thing," instead of just devising a more *ad hoc* selective-search algorithm like singular extensions or conspiracy theory that you couldn't really say, theoretically speaking, why it would or wouldn't work. A more fundamental problem with MGSS* was the "single-step assumption," which defined the utility of a node expansion as just the utility of expanding the node and then immediately moving. This meant that MGSS* could only search until the tree reached what McAllester would call "conspiracy depth 1." Also, the asymptotic running time of their algorithm was a superlinear function of the size of the tree produced, which meant MGSS* was really not competitive with traditional (alpha-beta) search in terms of runtime. Finally, after growing the tree, MGSS* just performed an ordinary minimax evaluation of that tree, adopting the flawed assumptions of all minimax-based approaches. It seems strange to spend so much effort modeling our knowledge for purposes of tree growth only to throw away that model when considering what variation in the resulting game-tree might actually be pursued.

However, despite these drawbacks, Russell and Wefald demonstrated that their program performed much better than alpha-beta for a given number of nodes searched, in the game of Othello. But note that this does not imply that it performed better for a given amount of computation time.

MGSS* could be seen as merely a step towards more significant future developments in decision-theoretically-based game-tree search. However, Russell and Wefald's work was a milestone in CGP research in that it attempted not just to play games but to develop important concepts of meta-level reasoning and limited rationality and frame them in general terms that could be applied to areas other than game-playing as well. This attempt to be general, as I have said, is vital for any CGP research that is justified from an

AI perspective. In their pursuit of ideas applicable to general reasoning, Russell and Wefald have set a new standard to which I think all future CGP research in AI should aspire. I discuss Russell and Wefald's general limited rationality efforts in more detail in chapter 3.

Other researchers besides Russell and Wefald have worked on decision-theoretic search control for general search problems, for example Hansson and Mayer in [Hansson-Mayer-89a]. And also, others such Baum and Smith have worked on more refined methods for decision-theoretic search control in games. We will describe Baum and Smith's work in more detail in the next section, because although it does not attempt to be general, it contains many valuable ideas and tools that I believe could be generalized and combined with ideas from other work such as Russell and Wefald's, and used in other kinds of reasoning systems, such as, for example, decision-analysis systems. Some initial work to do just that sort of reapplication to decision analysis is described in the next chapter.

2.8 Baum-Smith Search

Although so far we have seen a variety of game-playing algorithms, some of which draw upon the very useful, general tool of probability theory, none of them are really very sophisticated from the point of view of someone well-acquainted with statistical decision theory.[1] Even Russell and Wefald's work, although decision-theoretically inspired, does not involve detailed, realistic statistical models, since they were more concerned with the high-level concepts they were working on. It is my view that the recent work by Baum and Smith, described here in some detail, contributes greatly to the statistical sophistication of CGP, although there is room for more exploration of the relevant statistical models.

Baum and Smith's approach to partial-tree construction and evaluation combines many of the best aspects of the earlier methods, and improves upon all of them. Their method applies decision theory to both the evaluation and construction of partial search trees, as opposed to Hansson & Mayer's BPS, which only does evaluation rationally, and MGSS*, which only does exploration rationally.

1. This is not to say that the author claims to be such himself—but he is conversant enough in that field to be able to perceive its absence in the earlier CGP work.

Baum and Smith's method evaluates partial trees by backing up probability distributions from the leaves, similarly to Russell & Wefald's MGSS*. However, the Baum-Smith method gives the distributions a different semantics that facilitates the removal of Russell & Wefald's single-step assumption from the tree-growth procedure. Moreover, Baum and Smith's method doesn't require normal distributions, and so they are able to back up their distributions with no loss of information. They represent their distributions in a form that allows for extremely simple exact propagation up the tree.

Baum and Smith produce their partial trees by incremental expansion from the root state for as long as time permits. Their trees are grown by greedily expanding those leaves having the highest expected utility, as in MGSS*. However, Baum and Smith abandon Russell & Wefald's single-step assumption in their definition of the utility of leaf expansion, thus allowing the tree growth procedure to expand the tree beyond conspiracy depth one.[1] Using several mathematical and computational tricks, they approximate their ideal tree-growth procedure in an efficient algorithm.

The next subsection describes Baum-Smith tree evaluation, while §2.8.2 describes tree growth.

2.8.1 Evaluating partial trees using BPIP-DFISA

Baum and Smith's method for extracting information from partial trees is called BPIP-DFISA,[2] for "Best Play for Imperfect Players, by Depth-Free Independent Staircase Approximation." BPIP-DFISA, like most partial-tree processors, works by *evaluating* the tree, deriving *values* (expected utilities) for nodes from the values of their children; values of leaves are obtained from examining only the leaf itself. However, Baum and Smith deem the ancient minimax definition of the value of interior game-tree nodes to be unsatisfactory, due to its assumptions of perfect play by both players and of the correctness of leaf values. They propose an alternative definition which is based on probabilistic modelling of the future value assessments of the agents involved, and describe a technique for doing this modelling.

1. Meaning, we can continue expanding even if no single node expansion can possibly change our choice of best move. MGSS* was not able to do this.

2. Best Play for Imperfect Players, by Depth-Free Independent Staircase Approximation.

My description of BPIP-DFISA is broken down into several parts. First, the basis for the technique is a simple conceptualization of the problem called "the ensemble view" (§2.8.1.1). From this is derived a recursive strategy for node valuation called "Best Play for Imperfect Players," or BPIP (§2.8.1.2). However, BPIP requires a probabilistic model of future information in order to be concrete. For this, the authors provide the method called DFISA, for "Depth-First Independent Staircase Approximation" (§2.8.1.3). DFISA, in turn, requires an evaluation function that returns good probability distributions; Baum and Smith describe a statistics-gathering technique for generating such (§2.8.1.4). Finally, information gathered at the leaves must be propagated up the tree; a few simple operations are described for doing this (§2.8.1.5).

2.8.1.1 The ensemble view

Baum and Smith's simple conception of partial-tree valuation is as follows. We have a partial tree T. There is presumed to be some unknown *assignment* A of true game-theoretic values to all the leaf nodes in T. We imagine that for each leaf node λ of T we have a probability distribution $\Phi_\lambda(x)$ over the value that A assigns to λ.

Assuming that all the Φ_λ's are independent, we can derive from them a distribution over the possible assignments a, as follows:

$$\Phi(a) = \text{Prob}\,\{A{=}a\} = \prod_{\lambda} \Phi_\lambda(a(\lambda)), \tag{12}$$

where λ ranges over all the leaves of T, and $a(\lambda)$ is the value that a assigns to λ. This distribution is called the *ensemble*, and it allows us to define a distribution over possible values for each node ν in T, as follows:

$$\Phi_\nu(x) = \sum_{a:\,(\upsilon_a(\nu) = x)} \Phi(a), \tag{13}$$

where $\upsilon_a(\nu)$ is defined as the game-theoretic value of ν, given the leaf assignment a. This formula says that our probability of a node ν having the value υ is just the sum of the probabilities of all leaf assignments in which ν has that value.

The problem with this ensemble method of defining node value is that it's pretty far removed from anything approaching an actual algorithm. Summing over all possible trees is not something we can do in practice; we need a simple recursive definition of node value that we can efficiently apply. This is the role of the BPIP definition; it is equivalent to the ensemble view but is expressed in terms that don't require large constructions like the ensemble.

2.8.1.2 Best Play for Imperfect Players

BPIP is a node-evaluation strategy by which an agent or *player* in a 1-agent (or 2-agent zero-sum) strategic situation[1] can determine the child of the root that maximizes his expectation of the value[2] of the final outcome of the interaction, given only a partial *game-tree* of reachable states. The BPIP strategy assumes, as minimax does, that both players are using the strategy and that this is common knowledge. However, BPIP abandons minimax's incorrect assumption that the leaves of the partial tree are terminal states and that their static values are exact and correct.

The following definition of BPIP is a formalization of the definition that Baum & Smith informally describe.

First, some notational conventions:

- T means a partial search tree.
- ν_T indicates the root node of T.
- T_ν means a partial search tree that is rooted at the node ν.
- α is an *agent.*
- α_T is the agent α in a state of knowledge where it has direct knowledge of only those nodes that are in T.
- We write α^ν for the agent whose turn it is to move at node ν.[3]
- Υ_ν is a random variable for the desirability of the final outcome of game-play if the current players were to play from node ν.

1. There may be chance events, but actions by the two agents and by nature must be non-simultaneous, and instantly known to both players. Work remains to be done on how to generalize BPIP to strategic interactions with simultaneous play or with more than 2 agents, as has been done for minimax (see [Bodkin-92]).

2. Values can be thought of as expected utilities. I use the term "value" instead of "expected utility" in this discussion because Baum and Smith do so, and it is shorter.

3. We assume that there is a definite agent whose turn it is at each node. For chance nodes, "nature" is treated as an agent. This was done in [vonNeumann-Morganstern-44].

- $\wp_\alpha\{P\}$ is α's subjective probability that the proposition P is true.
- $E_\alpha[X]$ means agent α's expectation for the value of the random variable X:

$$E_\alpha[X] = \sum_x x\,\wp_\alpha\{X{=}x\}\,. \tag{14}$$

- An agent α_T*'s subjective value* $\upsilon_{\alpha_T}(\nu)$ is defined over nodes ν in T as:

$$\upsilon_{\alpha_T}(\nu) = E_{\alpha_T}[\Upsilon_\nu]\,, \tag{15}$$

  in other words, the expected payoffs in the final outcome of play between the current players from ν (at a time when the players will have access to a tree rooted at ν).

- Finally, α_T's *best play* $\beta_{\alpha_T}(\nu)$ from a node ν in T is that child ν_i of ν having the best subjective value $\upsilon_{\alpha_T}(\nu_i)$, from α_T's point of view. α_T may be minimizing, maximizing, or choosing randomly (e.g., if α is "nature").

The definition of the BPIP strategy is a consistency constraint on an agent's subjective values for nodes. Agents must be obey this constraint to qualify as obeying the BPIP strategy. This constraint is expressed in the form of a recursive definition of the values of a non-leaf tree node in terms of the values of its children.

Definition. Best play for imperfect players (BPIP):

An agent α is obeying the BPIP strategy if and only if, when given only a partial tree T of nodes to access, α's subjective value function υ_α obeys the following recurrence for each interior (non-leaf) node ν in T:

$$\upsilon_\alpha(\nu) = E_\alpha[\upsilon_\alpha(\beta_{\underline{\alpha}}(\nu))] = \sum_i \upsilon_\alpha(\nu_i)\,\wp_\alpha(\beta_{\underline{\alpha}}(\nu){=}\nu_i) \tag{16}$$

where $\underline{\alpha} = \alpha^\nu_{T_\nu}$, that is, the agent α^ν whose turn it will be at node ν, at a time when the progress of the game has actually reached node ν, and α^ν is in a knowledge state wherein he can access a different partial tree T_ν of nodes below ν.[1]

1. α need not know exactly what this tree will be since it may extend beyond T, the limit of α's knowledge. α's lack of knowledge of the identity of T_ν is encompassed in α's overall uncertainty about the value of $\beta_{\underline{\alpha}}$.

Additionally, if $\nu = \nu_T$ (the root of T) so that this time is now and $\alpha^\nu = \alpha$, BPIP requires α to realize that T_ν is just the current tree T, and $\underline{\alpha}$ is just α. Then (16) simplifies to:

$$\upsilon_\alpha(\nu) = \upsilon_\alpha(\beta_\alpha(\nu)). \tag{17}$$

To sum up (16), BPIP requires that an agent define its subjective value for interior non-root nodes to be its expectation of the value of the move that would be made from that node. Equation (17) says that the value of the root is the value of its best child.

Now, the key difference between BPIP and minimax is that BPIP does not require explicit maximization except at the root node; the value of an interior node is instead the sum of the children's values, weighted by the probability that the child will be chosen by α^ν. Thus BPIP handles future actions by both players as uncertain, chance events. This is the right perspective, given the fact that, even if we know exactly how the player α^ν works, we do not know exactly what information will be available to him then that isn't available to us now, and thus, his choice of move is uncertain.

Now, BPIP is an incomplete strategy in the same way that minimax is. Like minimax, it neglects to specify how an agent gets its leaf-node values υ. However, unlike minimax, BPIP also depends on personal probability distributions over the $\beta_{\underline{\alpha}}(\nu)$ variables indicating which moves will be made at future decision points. Minimax assumes that all future decisions will be made just the way that we would make them now.

So, a BPIP agent is assumed, unlike one using minimax, to have some degree of knowledge about the agents involved in the game, including itself; knowledge that lets it assess how these agents will play in future situations in which different information is available. This knowledge is expressed not only by the distributions over β's but also through the υ's of the leaf nodes. BPIP, unlike minimax, imposes a particular interpretation on leaf node values, namely, that they represent an expected value for the desirability of the final game outcome, given that the current players are continuing to play from a given node but with more information about the tree below that node.

The authors note,

> In order to make BPIP concrete, one needs a *probabilistic model of extra information* so that one can estimate the probabilities in [(16)]. If one chooses a

> sufficiently tractable such model, then it becomes possible to actually compute BPIP ([1], p. 7).

They go on to describe their chosen probabilistic model of extra information, which they call "Depth-free independent staircase approximation (DFISA)." This is described in the next section.

2.8.1.3 Depth-free independent staircase approximation

We have seen that BPIP does not constrain the values of leaf nodes. Of course, to instantiate BPIP we must come up with a way to derive leaf values, just as in other partial-tree-evaluation paradigms. However, in BPIP we must provide something extra as well.

Recall that (16) required, for each interior node, a probability distribution over the moves that a player might make at that node. A BPIP agent must provide this distribution. Can we derive this distribution solely from the values of the child nodes? If one were willing to assume that the player's future assessment of the value of the children would be the same as our current value for them, then yes, we could just declare the child that gives the best value to that player to be the one that is certain to be chosen. This is ordinary minimax. However, the point of BPIP is to move beyond this assumption, and model the fact that the valuation of a node will generally have changed by the time that node is reached in a real game. How can we model this possibility of change?

Baum and Smith's answer is to model it probabilistically. They broaden the notion of node evaluation to include not just the expected value of a node, but also a distribution over the possible future expected values for the node, that is, the values we might get for it after we do "more expansion" below it. In DFISA, the precise meaning of "more expansion" is intentionally left undefined; DFISA makes the fundamental assumption (the "depth-free approximation") that *any* amount of further expansion would have approximately the same effect. The idea is that the important difference is the difference between some expansion and no expansion at all. In practice, "more expansion" has a fairly definite meaning, namely, the expansion that would be done by Baum and Smith's tree-growth algorithm in a typical amount of remaining time. This is still not precise, but it is assumed by Baum and Smith to be good enough.

In any case, armed with these node-value distributions, let's look again at the problem of finding the probability that a player would make a certain move in some situation. If we knew what new values the player will have assigned to the node's children by the time the node is reached, we could predict which child the player would pick as his move. We already have distributions over the values that will be assigned to individual children, and so if we just assume independence (the "I" in DFISA), this allows us to get distributions over entire assignments of values to all the children—like in the ensemble view of §2.8.1.1. (Note that here we are talking about distributions over future subjective expected values, rather than distributions over true game-theoretic values, as in the ensemble view. However, the means of these two kinds of distributions are assumed to be equivalent.) Given these distributions over assignments, we can predict, in theory, the probability that a certain move will be made by summing over the assignments in which that move is the best one.

In practice this summing would be infeasible, and Baum and Smith's actual proposal for the implementation of DFISA avoids this work, and without approximation, by taking advantage of a special "staircase" (the "S" in DFISA) representation for the distributions. This will be discussed more in §2.8.1.5.

2.8.1.4 Leaf evaluation

BPIP-DFISA requires a value for each leaf, which is the mean of a probability distribution over the values that would be obtained by future search at that leaf. Thus, we need an *evaluation function* that, given a leaf, can provide a reasonable value and distribution. Baum and Smith propose a specific method for doing this for games like chess. It would work by collecting statistics from a large database of games and their outcomes. They describe the method briefly, but do not go into detail; it is not presented as a main contribution of their work.

The leaf evaluation task is broken into two functions: a function E_1 that produces an expected value for the node, and another function that returns an error distribution relative to this value. In Baum and Smith's proposal, functions for the two parts are learned separately, from different data. E_1 is just like an ordinary evaluation function, and can be learned in the normal way, but learning the error distributions requires a separate, special method.

To produce E_1, they manually define a classification tree for chess positions, that classifies any position into a class or "bin." For each position in each game in a large library of well-played games, they categorize the position into one of these bins. Within each bin, they fit a multilinear function of positional features $x_1, \ldots, x_n$ to the utility of the game's outcome. Thus, to predict the value of a new position, they simply look up the position's bin, find its feature vector, and run it through the linear function learned for that bin.

To produce the distribution-returning function, they produce another (possibly different) classification of positions into bins. They play a large number of games using E_1 with ordinary minimax search, and for positions searched in these games, they record the change in the node's E_1 value obtained by deeper search (for example, depth 6). Recall that they assume the exact amount of deeper search not to be very important. They compress the set of data-points in each bin into a simpler representation of the distribution of points; the exact method for this is described in a separate paper [Smith-92].

2.8.1.5 Backing up distributions

Baum and Smith's method for quickly backing up distributions depends heavily on the particular "staircase" (the "S" in DFISA) representation that they use.

The ideal of a continuous probability density function is approximated instead by a distribution consisting of a sum of delta functions, i.e., a set of spikes. This can be thought of in mechanical terms as a set of point probability masses p_i at different positions x_i, where "position" means the value in the domain of the distribution.

Second, this representation can simultaneously represent the "jumps" in a pair of staircase *cumulative distribution functions* that give, for each value x in the domain, the amount of probability mass located above (below) x. Formally, the upper and lower CDF's $\bar{c}^{(\nu)}$ and $\underline{c}^{(\nu)}$ of a node ν are defined by:

$$\bar{c}^{(\nu)}(x) = \sum_{i:\, x_i^{(\nu)} \geq x} p_i^{(\nu)} \tag{18}$$

and

$$\underline{c}^{(\nu)}(x) = \sum_{i:x_i^{(\nu)} \le x} p_i^{(\nu)} \tag{19}$$

where $p_i^{(\nu)}$ and $x_i^{(\nu)}$ are the probability masses and abscissae (positions) of our spikes i for the distribution of ν.

The advantage of the CDF functions is that they permit the exact computation of a distribution for a node ν in terms of the distributions for its children ν_i, in a way that is consistent with BPIP but skips the step in (16) of computing probabilities of moves.

The formula for the lower CDF of a node ν in which the agent to move at that node is attempting to maximize his value is:

$$\underline{c}^{(\nu)}(x) = \prod_i \underline{c}^{(\nu_i)}(x). \tag{20}$$

The formula for the upper CDF is similar. If we wish to have a minimax (as opposed to minimax) representation of value, we can substitute $\bar{c}$ for $\underline{c}$ and $-x$ for x on the right side of (20).

If the player at ν is "nature," then the formula is instead:

$$\underline{c}^{(\nu)}(x) = \sum_i \wp_i^{(\nu)} \underline{c}^{(\nu_i)}(x), \tag{21}$$

where the $\wp_i(\nu)$ are current agent's assessments of the probabilities of the different children given the parent.

Baum and Smith say that (20) can be computed quickly and exactly by something like a binary list merge on the set-of-spikes representations of the children, taking only $n \log n$ time in the number of spikes. With this technique, no information is lost as distributions are propagated up the tree.

2.8.2 Constructing partial trees

Baum and Smith apply the decision-theoretic approach to the construction as well as the evaluation of partial search trees. They grow their partial trees incrementally, expanding the tree a little at a time, with the choice of what part of the tree to expand being guided by information obtained during the evaluation of the result of the previous tree.

However, BPIP-DFISA tree evaluation takes time proportional to the number of leaves in the tree, so if the tree were deepened one leaf at a time in this way, the running time would be quadratic in the size of the final partial tree produced. Baum and Smith avoid this problem using the "Gulp trick," in which a constant fraction f of the leaves in the tree are expanded before each reevaluation. This ensures that the running time is dominated by the time taken by the final tree evaluation, like in traditional iterative deepening. The gulping trick is described in more detail in §2.8.2.1.

So with gulping, the goal in each stage of tree expansion is to expand that set of k leaves whose expansion will have the highest expected utility. In order to make this feasible, Baum and Smith do two things: (1) they assume that there is an additive measure of leaf expansion importance such that the value of expanding a set of leaves is the sum of the importances of the leaves in the set. Although they provide a counter-example that disproves this assumption, they hope it will work well enough in practice. (2) They assume that expanding a leaf always makes its distribution collapse to a single point, so that expanding the most important leaves amounts to expanding those leaves whose true values we would most like to know. Baum and Smith note that making wrong assumptions such as these in tree construction is less critical than making them in tree evaluation, since they are only being used to guide tree growth and not to choose actual moves; they affect efficiency of search rather than quality of play. Also, even with these simplifications, Baum and Smith expect their search to dominate both flat search and MGSS* search.

The measure of leaf expansion importance that they define is different from Russell & Wefald's. Recall that MGSS* assumed that the utility of expanding a leaf was equal to the utility of expanding it and then immediately moving: the "single-step assumption." The single-step assumption is bad, because it means that when the tree's conspiracy depth becomes greater than 1 (i.e., no single leaf expansion can change our choice of best move), which happens very quickly in practice, MGSS* will conclude that no expansion has utility higher than the utility of just moving without further search, so it will terminate tree-growth.

Baum and Smith instead define leaf expansion importance with a measure they call "Expected Step Size," or ESS. The ESS of a leaf-expansion is essentially the absolute amount that we expect it will change our expected utility for all possible future expansion.

In other words, if we think that expanding a certain leaf will clarify, one way or the other, the value of future expansion, then that leaf is a good one to expand. If it lowers our utility for future expansion, it prevents us from wasting time doing more expansion, and if it raises our utility for expansion, it prevents us from quitting and making a move too early. Baum and Smith's quitting criteria are discussed in §2.8.2.5.

Interestingly, ESS can be thought of as an instance of a more general principle of limited rationality, which is that learning more about the value of actions is a very valuable action in itself. When you are starting to do a round of expansion, first consider what you will do afterwards. There are two possible actions: you can either move right away after this round, or you can continue doing more expansion. You have some current utility for each of these actions, and some current best action. However, these utilities are uncertain. You might have a distribution over how your utility for each of these actions will change after this round of expansion. If these distributions overlap, then you would like for your next round of expansion to tighten up those distributions, so as to tell you for certain whether or not to go on and do another round.

Baum and Smith don't have a distribution over the utility of continuing to compute, or a model of how much certain leaf expansions will tighten it. Instead, they estimate how much a particular leaf expansion will perturb the mean of the distribution; the assumption being that large perturbations correspond large tightenings of the distribution.

ESS can be computed using the concept of "influence functions," which express the amount that some quantity at the root node will change if some quantity at one root changes while other roots remain fixed. These will be described in §2.8.2.4.

2.8.2.1 Gulp trick

The "gulp trick" is a cost-amortization technique that enables Baum and Smith's tree-growth algorithm to run efficiently despite the requirements of ESS-controlled tree growth. The trick, as mentioned above, is to expand not just the single most important leaf at each step, but instead the $\lceil nf \rceil$ most important leaves, where n is the current number of leaves and $0 < f \leq 1$ is a constant fraction determined empirically. Thus the cost of reevaluating the tree after each growth step is amortized over a large number of new nodes produced, and the total cost of all steps is dominated by the time for the final tree evaluation.

Baum and Smith describe the efficiency of gulping in terms of a constant *slow-down factor* C which describes, given f and the average branching factor b, how much time the tree growth (with gulping) takes compared to the time to evaluate the final tree produced—assuming that the average branching factor is larger than 1. The slowdown factor for chess is computed to be 2.2 for $f \cong 0.02$.

Now, the gulp trick is an approximation; more ideal would be to reevaluate the tree after each node expansion. In some circumstances, gulping might grow much worse trees than the ideal. To illustrate, imagine a chess position in which there is a single, long, clearly-forced sequence of moves following that position, where the ideal would be to trace down that forced sequence to see how it comes out, before examining other possibilities. With gulping, however, we would expand other nodes in parallel with each step down the forced path; a larger and larger set of them each time. So we would not get very far down the forced path.

Still, we expect to get farther down the forced path than flat alpha-beta would. And we don't usually expect there to be only one forced line like this; in general we expect that the number of reasonable variations increases exponentially with tree size; if $f+1$ is tailored to fit this exponent, then gulping should usually do well. We cannot make f too small, however, because then the slowdown factor will increase.

In summary, gulping seems like it ought to do fairly well for chess. To handle a more general range of problems, we might want the tree-growth algorithm to try to adapt the value of f to fit the needs of the problem at hand, but Baum and Smith don't discuss this.

2.8.2.2 Leaf expansion importance

Given our "gulping" policy of expanding a certain number k of the leaves of the current tree before proceeding, the problem within that policy is how to find the set of k leaves whose expansion would give us the highest expected utility. Baum and Smith describe a solution that gives the ideal decision-theoretic way to derive the utility of expanding any subset of leaves, given the CDF's for the children of the root, and the assumption mentioned earlier about distributions collapsing to a point. The definition essentially sums over all possible moves that might be found to be better than our current best move, and over the amounts that it might be better, if that set of leaves were expanded. (This is similar to one of Russell & Wefald's definitions.) However, to apply this definition to the problem of choosing the right subset of leaves, it seems necessary to apply it to all the subsets, which is clearly infeasible.

So instead, Baum and Smith simplify the problem by assuming that the utility of expanding any set of leaves is just the sum of a measure, called the *leaf expansion importance,* that is evaluated for each leaf in the set. They in fact prove, with a counter-example, that no such measure can exist. However, they proceed in the hope that it will "often be approximately true," as with their other simplifying assumptions.

One possible importance measure, which Baum and Smith call "Single-Leaf Expansion Utility" (SLEU) would incorporate Russell & Wefald's single-step assumption. However, Baum and Smith dismiss this measure for several reasons, including its limitation to conspiracy-depth-1 trees.

Instead, the authors do the following: they evaluate the above-mentioned true utility of expansion for the set of all leaves, and consider this to be the *expected utility of all future expansion (EUAFE).* In other words, it would be the expected utility of knowing the true values of all the leaves in the tree. To make sense of this, think of it as the expected cost of proceeding with our current guesses for the leaf values instead—in other words, how likely is it that our current guesses are wrong.

This EUAFE measure is then used to define the ESS leaf-importance measure, in a way that implicitly seems to identify the EUAFE with the expected utility U_{continue} of the action "continue to expand the tree" (as opposed to "stop and make a move now"), although Baum and Smith do not themselves make this identification. Since, in continuing to expand the tree, we will probably not actually expand all the leaves, this interpretation of EUAFE is not exactly correct.

2.8.2.3 Expected Step Size

The "Expected step size," or ESS, of a leaf is defined to be the expected absolute value of the change in the EUAFE that would be caused by expanding that leaf (reducing its distribution to a point) while keeping other leaves' distributions the same. As I described in the introduction to §2.8.2, this can be seen as an approximate measure of the expected improvement in the quality of our decision about whether to continue doing more expansion after the current round of it is over.[1] Unfortunately, Baum and Smith do not attempt to justify ESS in these terms, and instead put forth a number of less satisfying arguments of why ESS is a good measure. They do, however, provide an approximation theorem that describes ESS in utilitarian terms.

One nice thing about the ESS measure is that it can be computed quickly from a node's influence function. An influence function describes how changes in the node will affect the root's value. The influence function can, in turn, be computed in time only linear in the number of nodes in the tree. Baum and Smith provide fairly detailed pseudo-code for the computation of ESS and influence functions in an appendix, so I will not attempt to reinvent their algorithm here.

2.8.2.4 Influence functions

In Baum and Smith's algorithm, influence functions are computed top down, with each node's influence function computed from its parent's. This algorithm, like the BPIP-DFISA algorithm, depends strongly on the representation of distributions as a set of spikes. This representation allows a node's influence function to be represented as a vector giving the partial derivative of the root value with respect to the height of each spike in the node's distribution. Baum and Smith observe that this partial derivative is just a constant, because their propagation functions ((20) and (21)) are just linear in the CDF's of root nodes. These constants can be computed for a child based on the distributions in its siblings and parent and on the parent's influence function.

Thus, Baum and Smith's influence-function computation works by traversing the tree downward from the root, accumulating factors into these constants on the way down the tree. The total work is proportional to the number of nodes.

2.8.2.5 T ermination of growth

Finally, Baum and Smith suggest a natural termination condition for search in games, which is to consider that there is some additive utility cost for CPU time, and,

1. As mentioned in §2.8.2.2, I interpret EUAFS as an attempt to approximate U_{continue}, the utility of continuing to search instead of stopping after this round of expansion.

assuming we can estimate the time required to do the next round of expansion, searching should cease when the EUAFE becomes less than the current utility minus the cost of that time, similarly to how Russell and Wefald terminate search (note that this assumes that we will get most of the benefit from future expansion in the very next round of expansion). For a timed game like chess, Baum and Smith suggest time-cost could be proportional to the estimated number of moves left in the game divided by the time remaining, using a constant factor determined by experiment.

However, this way of handling time is not very theoretically satisfying. There is very little that can be said about exactly why, and when, and how well this technique will really approximate maximization of our probability that we will win the game, or find the solution on time. What will happen if the rules of the game depend on time in a more complicated way than in chess? Even for chess, this treatment of time is weak: for example, suppose the opponent is running low on time and we are not. To maximize the probability of winning, we ought to try to steer play into complicated situations, where careful thinking is required in order to avoid making a blunder, in the hope that the opponent, in his rush, will make a mistake. Treating time as external to the domain, and only using it to cap computation time, does not address this kind of reasoning at all. I will discuss the issue of time some more in a later section.

2.8.3 Problems

Baum-Smith search has not yet, to my knowledge, been implemented by anyone, for any domain, so its methods have yet to withstand the rigors of empirical study. The authors are working on implementing it for the domain of chess, using a handmade classification space and a large library of well-played chess games to train up an initial evaluation function.

In doing away with some of the assumptions of traditional minimax, Baum and Smith have to make quite a few new, untested simplifying assumptions and approximations. Many of these are fairly well-justified by Baum and Smith's theoretical arguments, and it seems that the problems are not too serious, but it remains to be seen how well they will work in practice. A list of some of the assumptions and approximations:

- Depth-free approximation (§2.8.1.3).
- Gulp trick (§2.8.2.1).
- Additive leaf expansion importance (§2.8.2.2).
- Additive time cost (§2.8.2.5).

Additionally, Baum and Smith's method is currently not very approachable; parts of their description are very difficult to understand. However, it is a draft paper, and I expect that their presentation of their ideas will improve with time. I hope that this thesis will, as a supplement to the Baum and Smith draft paper, serve to clarify some of their ideas.

2.9 What' s Next for CGP?

Baum and Smith close their paper with the following insightful comment on the lack of generality of the entire partial-tree-search paradigm for game-playing:

> We conclude with the remark that this still has little to do with how humans play games. The computer science approach (which we are attempting to perfect) has since Shannon basically regarded a game as defined by its game tree. But what makes a game interesting is that is has a low complexity, algorithmically efficient definition apart from the game tree.... Any procedure which only accesses the underlying simplicity of a game in the form of an evaluation function is inherently doing the wrong thing.... The main open question is how to go beyond the evaluation function picture of games.

I heartily agree with this sentiment, and I would like to propose that it applies not just to game-playing but to the similar search methods often used in puzzle- and problem-solving domains. I also propose that in questioning the current approach, we ought to try to reveal all of its hidden assumptions (which I attempted to enumerate earlier), and determine which are most necessary for future progress, and which would be beneficial to try working without. Sections §2.9.1 and §2.9.2 reexamine a few of these assumptions and suggest how they might be relaxed. Sections §2.9.3 and §2.9.4 examine some unrelated methodological issues.

2.9.1 Generalizing the notion of games

In the typical notion of game-playing dealt with in AI, there are many implicit assumptions that remain invisible until pointed out. Games are assumed to have only two players. Moves are made in a definite sequence, rather than simultaneously. There is always a definite player whose turn it is to move. Work by Ronald Bodkin has shown how to loosen these assumptions and generalize some of the traditional game-playing algorithms [Bodkin-92]. Can this be done for Baum and Smith's approach as well? It seems likely that work can be done along these lines; generalizing scalar outcome-values to payoff vectors, generalizing move lists to multi-player move matrices, etc. This is one way that the Baum and Smith approach might readily be generalized.

2.9.2 Abstract search

A fundamental issue in treating any domain as a state space is the choice of which properties of the world will be included in your notion of a state (or concrete microworld state, in the terminology of section §2.3). The normal game-tree notion of state includes things relevant to the rules, like where the pieces are, and whose turn it is. However, normal game-tree states often miss a couple of things that may well be relevant: the history of moves, and the current time. In chess, the history of moves is important because of the rule forbidding thrice-repeated sequences of positions, and the time is important because you lose if you take too much time on your moves. Ignoring these properties leads to problems in chess, and proposed solutions (such as in Baum and Smith's appendix G) usually have an unsatisfying, *ad hoc* flavor.

So why don't people include these properties, which are, after all, relevant to the game, in the concept of a game state? The reason is that CGP researchers realize that including these properties would ruin the performance of their game-search algorithms. Why? Because all of a sudden there would be many more states than before, many of them differing only in these two properties. It multiplies the size of the problem a thousandfold.

But since these properties are important to they game, how can game-programs play well while ignoring them? After all, games researchers don't ignore other properties, such as the positions of pieces.

The answer is that the games researchers have encoded, in their definition of state, domain-specific knowledge to the effect that these properties *usually aren't very important* to things we care about, such as the expected utility of a state. Usually, in a game like chess, the utility of a board position is the same no matter how you got there, or exactly what time it is. In other words, the program designers have built in an abstract notion of state, not a concrete one at all, because they realize that the abstraction is useful; it provides a *good approximation* to the more computation-intensive ideal of considering the position history and game time.

This raises a number of very interesting questions:

- If we can abstract over these properties, would it perhaps be useful to consider abstracting over others? For instance, wouldn't it be nice to realize that my planned fork will work no matter where my king is, so that I don't have to invent it again when considering moving my king? In solitaire, wouldn't it be nice to be able to think about moves I can do after revealing some hidden card, regardless of the identity of that new card? (We will discuss this case in more detail later.)
- Could the kind of abstraction done be different depending on the situation being reasoned about? After all, sometimes time and history *are* important. Wouldn't it be nice to do the right thing in such cases, without requiring a separate ad-hoc mechanism for that?
- The existing kind of abstraction just assumes complete equivalence between states having only the abstracted-over feature(s) different between them. Without any abstraction, we traditionally assume all states are distinct. Is there perhaps a middle ground, where you take the value of one complete state as probabilistic evidence for a similar state's having a similar value? And have the degree of strength of these relationships be based on real, statistical data about the correlation?
- How do you define the value of an abstract state, not knowing how that state will be instantiated? (My proposed answer: use the *expected* value if it were to be instantiated.)

- Perhaps this issue of abstraction is the reason for Baum and Smith's observation that humans play games very differently—perhaps humans are abstracting over states in many ways, all the time, rather than always considering complete states.
- Could we perhaps, in moving to this idea of abstract states, bring with us some of the useful ideas from concrete-state-space search? E.g.: state graphs, partial search, incremental refinement, propagation of values through the graph, distributions over future expected values, rational evaluation, rational control of search.

Most of these questions remain unanswered, and even rather poorly-defined. At first they only seem to hint at a future direction. However, in the research discussed in chapter 4, I have begun to make some progress towards clarifying and answering some of these important questions.

One preliminary conclusion is that transplanting rational techniques from concrete state-space search to abstract state-space search is probably not so hard after all. It is made easier by the observation above that existing game-state-spaces are really abstract spaces already, and thinking of them in those terms makes it easier to see how the current methods might make sense with other kinds of abstraction as well.

One might think that it would be easier to first try transplanting simple techniques like minimax before attempting to generalize all of Baum-Smith search to a world of abstract states. On the contrary, I think that a decision-theoretic approach is vital for properly handling all of the uncertainties and all of the questions of usefulness that are involved in any sort of abstract reasoning. I think that some of the previous attempts in AI to do related things such as "chunking" and "reasoning by analogy" have been relatively unsuccessful because they did *not* take a proper decision-theoretic stance—of plausible inference and cost-effective action—towards the selection and use of the abstract generalizations that they were making.

In chapter 4, I begin illustrating the use of simple rational abstraction in the game of spider solitaire, which offers some good opportunities for rational abstraction and search-control to show themselves off. Spider also serves as an example of a very large single-agent decision-analysis problem. I will also discuss how to generalize rational abstraction methods to be of some use in other large decision problems as well.

2.9.3 More automated training of evaluation functions

In asking Smith for more details on how he was his producing his evaluation function for his chess program, I was surprised to discover that he was doing a large amount of work by hand: defining a carefully-constructed classification-tree for separating his chess data into bins of approximately equal size, and defining an extremely large set of chess features on which to perform linear fits.

This raises an issue of methodology: as computer game-playing/problem-solving researchers, should we focus a lot of our effort on encoding lots of domain-specific knowledge into programs that introduce new, supposedly general techniques?

I would argue that the answer is no. Groveling over an evaluation function might be the wrong place to focus attention; we should be looking at methods to reduce the amount of human effort required to tailor the evaluation function, and make the computer do more of the work. Besides saving us time, this might give us experience in the use of general machine-learning techniques, which we could reapply when transplanting our rational search methods to other domains.

For example, in the next section I describe a method for constructing good classification spaces automatically from a set of features, by explicitly measuring how well a classification space performs at predicting the data of interest (here, a position's value). We would do well, I think, to use such automated methods when possible, so as to minimize the amount of time we spend writing hand-tailored domain-specific code.

Also, as Matt Ginsberg has argued [Ginsberg-Geddis-91], there are generally domain-independent reasons why domain-specific tricks work. If we expend a little extra effort to find those more-general forms for our computational tricks and techniques, we stand the chance of gaining great benefits when we learn how we, and our agents the computers, can apply that knowledge in other areas.

2.9.4 A note on domains

I would suggest that researchers currently working in typical game-playing and puzzle-solving domains ought to focus more on applying their work to a wider variety of such domains, and even to other, apparently very different "real-world" domains. It's very easy to get lost in the intricacies of tricks for playing one specific game or another, so much so that we often forget to think about how our tricks might be generalizable. Often they are.

Many people like to work on well-established AI problems, so that the relevance of their work can be easily demonstrated by comparing their new program's performance with that of well-established, earlier programs. However, I feel that comparison with existing programs in traditional domains can be extremely misleading. These programs usually win not because they have better general intelligence, but because they're more narrowly focused and specialized to that one small domain.

This is especially true for chess. I think that, in particular, the computer chess community has gone much too far in the direction of domain-specific hacking, for chess competitiveness to be a good measure of the worthiness of, say, a new general theory of search. It's possible that someday, smart AI programs will beat the brute-force chess engines, but the chess hackers have too much of a head start for chess to be a good place, in my opinion, for an AI researcher to start.

So, if we abandon comparison against programs in well-established narrow domains as a good basis for evaluating AI techniques, how can we evaluate our methods? One solution, proposed by Barney Pell in [Pell-93], is to try to establish a new, broader competitive problem domain; one for which even a program that was very domain-specific would still need lots of general reasoning mechanisms.

Another idea, which is easier because it can be implemented on the individual level, is as follows:

- Strive for generality. Think about several specific domains when designing your method, ones that are as different as you can possibly handle. Make sure that everything you do makes sense when instantiated in each of the domains. The proof of your method will come in showing that it does well in a variety of domains.

- Don't worry about direct competition against other people's programs. This means, don't hack in C to gain competitiveness. Don't spend a lot of time adding domain-specific knowledge to improve competitiveness. If you're working in a well-established domain like chess, ignore the good brute-force programs. You're not going to beat them unless you spend most of your time on domain-specific stuff.
- Efficiency is still important! The realities of limited computational resources should not be ignored. Pay attention to asymptotic running times. Write a variety of simple, brute-force programs for the domain to compare your smart technique against. Show that your smart program does better than the brute-force method even (especially!) when time is taken into account. Be careful though: don't make the brute-force programs asymptotically slower than they need to be, and don't optimize your smart program for efficiency more than you do for the brute-force one. Write your smart program in the same language as the brute-force one. Use the same evaluation function (when possible) for the programs being compared. Don't spend much time on the evaluation function or other domain-specific performance-improvement—unless you think that having more domain knowledge would help the brute-force program more than the smart one, in which case you're on the wrong track. More knowledge ought not to hurt.

2.10 T owards new advances

The traditional state-graph-search approach to game-playing and problem-solving appears to have reached a new pinnacle of achievement in Baum and Smith's technique for rationally and *efficiently* creating and evaluating a partial search tree. Their method remains untested, but theoretically speaking, it seems to nicely wrap of many of the major issues. A number of further refinements of the technique are undoubtedly possible, but it is beginning to look to this researcher as if the larger part of future progress in computer game-playing and problem-solving will be in broadening the scope of the old state-graph-search paradigm. Specifically, I propose broadening it to reason about states at different levels of abstraction. Decision-theoretic tools such as those used by Baum and Smith seem ideally suited for handling the many uncertainties and control issues involved in reasoning at a more abstract level, but many questions remain unanswered. Only time, and research, will tell.

2.11 Automatic Evaluation Functions

This section describes an alternative way to get a good evaluation function from game-playing data, which is proposed as an alternative to Baum and Smith's method of a handmade classification tree and a linear fitting function. This method tries to be as general as possible, and applies to machine learning tasks of all kinds, not just evaluation-function learning.

2.11.1 The Problem

Suppose we are making a game-playing program. We have a game search algorithm that can do the job, if it's given a fast source of a certain kind of probability information. For example, given a game position, we may want to quickly find our personal probability $\wp(Win)$ that we would win if we were in that game position. Or, if it's a game with several possible outcomes $o_1, \ldots, o_n$, we may want a probability distribution $\wp(o_i)$ over these outcomes. Or, for control of search, we may want, for each o_i, some approximate representation of a continuous distribution over the amount that our $\wp(o_i)$ for the position will change after some more computation. In any of these cases, the problem is the same: how do we do a quick, yet high-quality assessment of these probabilities?

One good answer is to get these probabilities by collecting statistics. Unfortunately, in real games there are far too many positions to feasibly collect statistics for each of them individually. The standard solution to this problem is to collect statistics for whole classes of positions at once, without remembering the statistics for the individual positions within a class. The *classification space* for positions is often defined as the cross-product of a number of *feature spaces* $\phi_1, \ldots, \phi_k$, each of which is the range of a *feature* f_i, which is some easily-computed function of game positions (see figure 9, p. 98). We generally try to choose a classification space that we think will provide a lot of information about the variable we are interested in (the game outcome, or our future probability assessment).

Once we've defined our classification space, or set of bins, we can begin gathering statistics from our data. Each data point is assumed to be in the form of a game position P, together with the value of the variable y we are interested in. We classify P, look up a data storage bin for its class, and add y to the data stored in that bin. We might also associate y with some additional information about P that we know relates to y in a certain special way; for example, if y is numeric, we might expect its value to be correlated with the values of certain numeric features $x_1, \ldots, x_k$; these features would then be left out of the feature set used to define the classification space, and instead their values would be stored along with the value of y. (As Baum and Smith do in their position-classifier.)

Then, for each bin, we form a model of the data in that bin, in the form of a probability distribution over y (or, in the case where additional information is stored in the bin, a function from $x_1, \ldots, x_k$ to distributions over y). We might get the distribution by choosing the maximum-likelihood distribution given the data, out of some simple space of approximate representations of distributions.

So then we are done; when we see a game position, we simply look up its bin and use the stored probability distribution as our quick first assessment of our belief about which outcome will actually occur.

Unfortunately, we often find that it's hard to come up with one simple feature that does at well at predicting the outcome. Even combinations of 2 or 3 simple features might not be enough. We generally find that we need a lot of features in order for the statistics to discriminate the different outcomes well. However, this leads to another problem: the more features we have, the larger the classification space becomes (exponentially), and so a huge amount of data is required in order to get a statistically significant sample in each class. This section will briefly describe one possible solution to this problem.

2.11.2 The Proposed Solution

The basic idea is this: even though it may be true that no single feature, or combination of just a few features, discriminates outcomes very well, it may be the case that lots of these small classification spaces do *okay*. If only we could combine the recommenda-

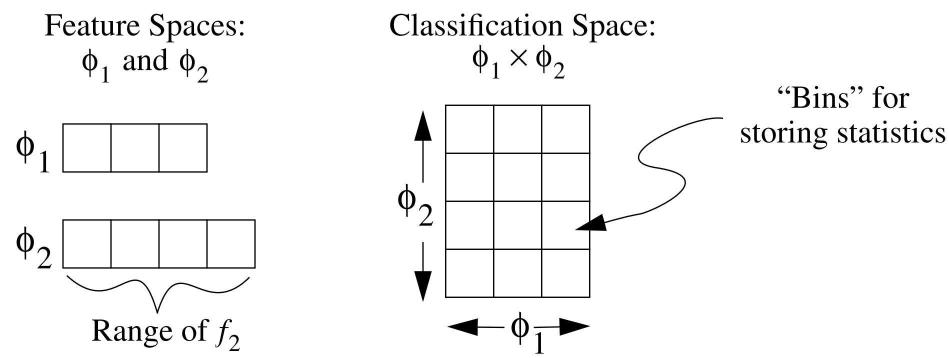


Figure 9. Classification Spaces.

tions of these small classification spaces in a principled way, we might get a good overall answer without having to resort to collecting statistics on the exponentially-large classification space that combines all the features.

For example, we could form a classification space for each combination of 1, 2, or 3 different features. Each data point can be classified within each space, so we only need to collect enough data to get a good sample in each bin of the largest such space.

But when it comes time to use the results, how do we combine the probability distributions produced by a position's classification in all the different spaces? We don't want to just add them together, because some of them will be much better predictors of the outcome than others; we want them to contribute to the answer proportionately. Is there any principled way to weight the distributions?

One possibility is to think of each classification space, with probability distributions in all of its bins, as a probabilistic model of the data, i.e., as a hypothesis that a position's outcome is determined by selecting a value from its class's distribution. Thought of in this way, we can say that each classification space, each hypothesis, has some probability of being the "correct" one. If we could only compute this probability, it would make sense to use *that* as the space's weight. (Assuming that all these hypotheses are mutually exclusive—the damage of assuming this in practice is hoped to be small.)

We can compute model probabilities using Bayes' rule. Since these are probabilistic models, it's trivial to compute the likelihood of the data given the model. So the problem is reduced to one of finding sensible priors for all the models. This is not too critical since, with a good statistical sample in all of a model's bins, the differences in the data-likelihood between the different spaces ought to be the deciding factor.

In any case, the prior for a classification-space model is just the product of the priors for the distributions in each of its bins, since the classification-space can be thought of as a conjunctive hypothesis of the form "in this bin, the distribution is this; and in this other bin, it's this; etc..."

How we compute a prior for the distributions in the bins depends on how we represent the distributions, and whether we have additional stored features (Baum and Smith's $x_1, \ldots, x_k$) complicating our model. However, one answer in the simple case is that the prior ought to be just e^H, where H is the entropy of the probability distribution in the bin; a high-entropy distribution carries less information and so should be *a priori* more probable.

2.11.2.1 Application of the Technique

A version of the above-described technique has been tried in God, a simple card-playing game; the results were very successful.

In God, one player is the "god," and makes up a rule determining what stacks of cards are "allowed." For example, the rule might say that a stack is allowed if and only if the top card has a different suit and than either of the next two cards. The other player starts with all the cards, and he attempts to play them one at a time on a growing stack; after each attempt the god says "yes," or "no;" if "no," the player must take back the card and try again. The player's goal is to get rid of the cards as quickly as possible. Assuming a greedy strategy, his goal on each play is to play the card that forms a stack having the highest probability of being allowed, based on the answers received so far. (Other rules prevent the god from making the rule too restrictive or complicated; I won't go into those here.)

One thing to note is that the data is very scant; in a well-played game the player receives only a few bits per card. Thus my collaborator Mark Torrance and I required a technique that used the data very efficiently. We defined about ten features of card stacks, for example, whether the top two cards were the same suit. We then formed classification spaces for every combination of one or two of these features, and derived predictions from past data as described above. The technique proved very effective in generating good predictions, our player was the best out of about a half-dozen God-playing programs written for a class.

2.11.2.2 Problems With this Approach

I already mentioned the problem that several classification spaces (with bins filled in) might turn out to embody the same probabilistic model of the data, and thus it will be improper to count them as mutually-exclusive hypotheses. This can happen, for instance, with the spaces ϕ_i and $\phi_i \times \phi_j$, if the outcome is totally independent of $f_j(P)$ given $f_i(P)$, because $\phi_i \times \phi_j$ will then represent the same hypothesis as ϕ_i. Thus ϕ_i's contribution to the total distribution will be unfairly magnified. This particular case should not be a serious problem, since $\phi_i \times \phi_j$ will have a negligible probability compared to ϕ_i, as a result of it being a much larger space and so having a much smaller prior. But if two identical features are included by accident, their contribution will be counted twice; and any two features that are not independent will degrade the prediction quality, to a degree that is still unclear.

Another problem is that we are considering only models that contain maximum-likelihood distributions for the data. What if we also considered, in each space, a continuum of nearby models that contained distributions that were slightly different? Might that yield different weights for our classification spaces?

To avoid this problem, we need to make sure that our distributions are not just "most likely" distributions but rather express the sum of all possible distributions, weighted by their probability given the data. If we do this, then we can be sure that when we calculate the probability of a classification space, we are truly calculating the probability of just the hypothesis that it's the right space, not the hypothesis that it's the right space *and* that the distribution in each node is such-and-such particular distribution.[1]

---

1. But if the hypothesis is just that the space is correct, shouldn't the space's prior depend only on the space itself, and not on the distributions stored in its bins? Maybe its probability should just be proportional to $e^{-\#\text{bins}}$. This needs more work.

If the variable y has a small, discrete range (e.g., $\{\text{Win}, \text{Lose}, \text{Draw}\}$), and we accept a normal flat prior over the space of possible distributions over y, then there is no problem; a standard formula[1] for the exact distribution is given by:

$$\wp(y_i) = \frac{m_i + 1}{M + n_y} \tag{22}$$

where m_i is the number of data points where $y = y_i$, M is the total number of data points $\sum_{j=1}^{n_y} m_j$, and n_y is the number of possible values of the variable y. This formula works well even on small amounts of data. Note that when there is no data, we get a flat distribution; this is the correct weighted sum of all possible distributions if all are equally likely. If there are two outcomes $\{\text{Win},\text{Lose}\}$ and the data has 1 win and no losses in the bin, when we get $\wp(\text{Win}) = \frac{2}{3}$.

However, if the range of y is continuous (or just too large to store all the m_i's), as is the case if y represents a future probability assessment, then we cannot use (22) directly. In any case, we probably don't want our prior over distributions over such a large space to be flat; smooth, simple distributions seem more likely *a priori* than complex, jagged ones with high spatial frequencies.

One simple method that ignores high spatial frequencies but still allows multimodal distributions is to discretize y into evenly-spaced sections, and use (22) on those. However, this loses the distinction between a needle-sharp peak and a slightly smoother one that's still all in one discrete section.

2.11.3 A More Refined Version

I recently thought of a refinement of this technique that seems like it would produce even more accurate results; however, this new method has not been examined in detail or tested.

In the previous proposal, we weighted a classification space's prediction of a game-position's outcome by the probability that the entire classification space was the "right" model of the data. However, each game-position lies in only one bin in each classi-

1. (22) is the mean of the beta-multinomial distribution, for a uniform prior. The beta-multinomial distribution generalizes the binomial distribution to more than 2 outcomes.

fication space. So perhaps, the probability of that individual bin, rather than that of the entire space, would be the more appropriate quantity for weighting the bin's prediction.

However, this raises a couple of new problems. It complicates the question of whether two alternative hypotheses are independent. Also, it complicates the comparison of the probability of the two hypotheses, since in general they will have been based on different data.

Further work remains to be done on this alternative.

2.12 Towards CGP as a True Testbed for AI

Computer game playing has been called "the drosophila of AI" [McCarthy-90], but have the CGP developments to date really fulfilled that promise? I would say that until recently, they have not. Too much time and effort has been spent on optimizations for particular games (especially chess), optimizations that do not really teach us anything about how to write programs to solve more practical problems. In fact, the entire Shannon paradigm is really a chess optimization. The fact that humans play a number of other games amenable to minimax search is more of an historical accident than anything else. If we broaden our sights a bit to look at other kinds of games, such as games with huge numbers of possible actions, we see that partial state-space search no longer seems so appropriate—especially if the search is unguided.

We can see two things that are fundamentally needed for further progress in automated reasoning: First, we need to have a solid, principled, general understanding of the problem of limited computational resources, instead of just blindly trying arbitrary techniques for control of reasoning and settling on whatever seems to work in one context. Second, we need to question our ontologies, such as the insistence on defining some fixed level of world-states, when it seems that humans shift around, reasoning about game-positions and other problematic situations at many different levels of abstraction.

In the last few sections we have seen that CGP research is beginning to turn to the general tools of decision theory, which at least show promise for understanding the issues of limited rationality. In the next chapter we will look more carefully at these issues.

Nobody, however, has come up with good, general algorithms for game-playing at different levels of abstraction. The problem is that the kind of abstractions needed vary from game to game. In chapter 4 we will begin to explore a general framework for reasoning about abstractions.

However, what is really needed is to combine these two advances: limited rationality and abstraction. If CGP research truly achieves a synthesis of principled management of their limitations with a flexible way of viewing the world, I believe that it both reach new heights of game-playing excellence, and teach us many useful things about reasoning in general.

CGP can still be the drosophila of AI, but it must begin to try consciously to be relevant and general, rather than focusing on next week's chess tournament. Efforts like Barney Pell's Metagame project [Pell-93], which attempt to force CGP work into a more general context, and away from its past focus on the idiosyncracies of particular games, are the right kind of step. Russell and Wefald's attempts to extract general principles of limited rationality from their search algorithms are also good, as are Baum and Smith's application of standard statistical tools. However, whether the masses of CGP researchers will turn to follow some of these new directions remains to be seen. I hope that they will.

Decision theory got its start from von Neumann's investigations of games. In the next chapter we will see how it has since struggled to reinvent itself in limited-rationality terms.