Chapter VI Memory Limited Strategy Optimization

Let us start with a big picture to describe the relationship between this chapter and the previous chapters. We focus on how to solve a simulation-based strategy optimization problem[1]. Conceptually, we need three components: a program which implements a strategy γ, a performance evaluation technique to calculate $J(\gamma)$ when γ is applied to the system, and an optimization algorithm to find the optimal or good enough strategy. The relationship among these three components is shown in Fig. 6.1.

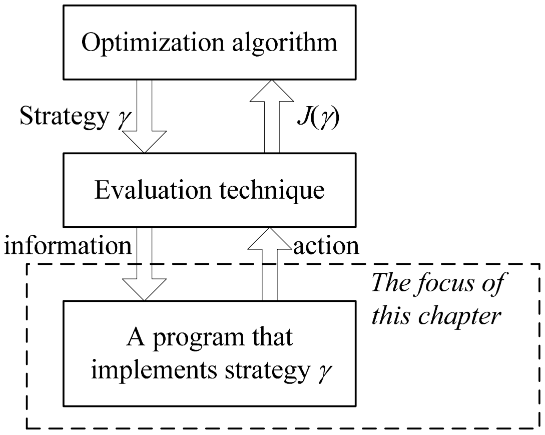

Fig. 6.1. A big picture for simulation-based strategy optimization

Note that the optimization algorithm only requires the evaluation technique to return the performance of a strategy, but does not care about how this evaluation is obtained; the evaluation technique only requires the program to return the corresponding action of a strategy when the information is the input. When implementation is considered, we have to make

[1] A strategy is simply a mapping from input information to output decision or control. Strategy is also known as decision rule, if-then table, fuzzy logic, learning and adaptation algorithm, and host of other names. However, nothing can be more general than the definition of a function that maps all available information into decision or action.

sure the strategy is simple enough so that it can be stored within given limited memory, in other words, we require a strategy is simple enough to be implementable. Furthermore, when ordinal optimization is considered, one challenge will be sampling of design space. Sampling is easy when we have a natural representation of a design as a number or a vector as we have done for previous chapters. In the context of strategy optimization, representing all implementable strategies so that sampling can be taken seems nontrivial. The focus of this chapter is to provide a systematic representation of strategy space so that the optimization algorithms developed earlier can be applied.

1 Motivation (the need to find good enough and simple strategies)

It can be argued that the Holy Grail of control theory is the determination of the optimal feedback control law or simply the feedback control law. This is understandable, given the huge success of the Linear-Quadratic-Gaussian (LQG) theory and applications in the past half-century. It is not an exaggeration to say that the entire aerospace industry from the Apollo moon landing to the latest GPS owes a debt to this control-theoretic development in the late 1950s and early 1960s. As a result, the curse of dimensionality notwithstanding, it remains an idealized goal for all problem solvers to find the optimal control law for more general dynamic systems. Similar statements can be made for the subject of decision theory, adaptation and learning, etc. We continue to hope that, with each advancement in computer hardware and mathematical theory; we will move one step closer to this ultimate goal. Efforts such as feedback linearization and multimode adaptive control (Kokotovic 1992; Chen and Narendra 2001) can be viewed as such successful attempts.

The theme of this chapter is to argue that this idealized goal of control theory is somewhat misplaced. We have been seduced by our early successes with the LQG theory and its extensions. There is a simple but always neglected fact that ***it is extremely difficult to specify and impossible to implement a general multivariable function even if the function is known.***

Generally speaking, a one variable function is a two-column table; a two-variable function is then a book of tables; a three-variable function, a library of books; four-variable, a universe of libraries; and so on. Thus, how can one store or specify a general arbitrary 100-variable function never mind implementing it even if the function is God given? No hardware advancement will overcome this fundamental impossibility, even if mathematical advancements provide the general solution. This is also clear from the following simple calculation. Suppose there are *n*-bit input information and *m*-bit output action for a strategy. To describe such a strategy as a lookup table, we need to store all the (information, action) pairs. There are 2^n such pairs in total, and we need $(n + m)$ bits to store each pair. Thus we need $(n + m)2^n$ bits to store a strategy. When $n = 100$, $m = 1$, this number is 101×2^{100}bits $\approx 2^{107}$bits $= 2^{74}$ Gega Bytes (GB), which exceeds the memory space of any digital computer known nowadays or the foreseeable future. Exponential growth is one law that cannot be overcome in general. Our earlier successes with the Linear-Quadratic-Guassian control theory and its extensions are enabled by the fact that the functions involved have very a special form, namely, they decompose into sums or products of functions of single variable or low dimensions. As we move from the control of continuous variable dynamic systems to discrete event systems or the more complex human-made systems discussed in this book, there is no prior reason to expect that the optimal control law for such system will have the convenient additive or multiplicative form. Even if in the unlikely scenario that we are lucky to have such simple functional form for the control law of the systems under study, our efforts should be to concentrate on searches for actual implementation of such systems, as oppose to finding the more general form of control law.

In this light, it is not surprising that “Divide and Conquer” or “Hierarchy” is a time-tested method that has successfully evolved over human history to tackle many complex problems. It is the only known antidote to exponential growth. Furthermore, by breaking down a large problem into ever-smaller problems, many useful tools that do not scale up well can be used on these smaller problems. Decomposing a multivariable function into a weighted sum of one-variable functions is a simple example of this principle. In addition, Nature has also appreciated this fundamental difficulty of multivariable dependence. There are many examples of adaptation, using simple strategies based on local information and neighboring interactions to achieve successful global results abound (Think globally but act locally), such as ants, bees, germs, and viruses (Vertosick 2002). Recent research on the No-Free-Lunch theorem (Ho and Pepyne 2004) also points to the importance and feasibility of “simple” control laws for complex systems. And as we venture into the sociological and psychological realm, there are even more evidences showing that it only leads to unhappiness and non-optimality to strive for the “best” (Schwartz 2004).

The purpose of this chapter is to discuss systematic and computationally feasible ways to find “good enough” AND “simple” strategies. Since we will focus on simulation-based strategy optimization, many difficulties mentioned in earlier chapters remain, such as the time-consuming performance evaluation and the large design space. In addition we have one more difficulty, that is the constraint on the limited memory space to store strategies.

2 Good enough simple strategy search based on OO

2.1 Building crude model

It is important to understand that lookup table or brute force storage and representation is usually not an efficient way to implement a strategy and is infeasible and impractical for almost all large scale problems. Recall that a strategy is a mapping from the information space to the action space. In other words, a strategy determines what to do when specific information is obtained. As long as we find a clever way (such as using a program) to generate the output for any given input, we can represent the strategy. The size of memory we use may be much less than the lookup table. To identify simple strategies (or strategies that need less memory than a certain given limit), we need to introduce the concept of descriptive complexity (also known as the Kolmogorov complexity (KC) (Li and Vitányi 1997)) which mathematically quantifies the minimal memory space that is needed to store a function. A. N. Kolmogorov developed this concept in 1965 (Kolmogorov 1965). The Kolmogorov complexity of a binary string s is defined as the length of the shortest program for a given universal Turing machine *U* (explanation follows) to output the string, i.e.,

$$C_U(s) = \min_p \{|p| : \psi_U(p) = s\},$$

where $\psi_U(p)$ represents the output of the universal Turing machine *U*, when program *p* is executed on *U*. Roughly speaking, a universal Turing machine (UTM) is a mathematical model of the computers we are using nowadays, which consists of the hardware (e.g., the hard drive and the memory chip to store the program, and equipment to read from and write on the hard drive and the memory chip) and the software (including low level system software such as operating systems, e.g., Microsoft Windows and Mac OS, and application software *p* developed for a specific task). Obviously KC depends on which UTM is used. This is reasonable and practical when we use computers to search for a good simple strategy, the hardware and the software in the computer are already given, so the *U* is fixed. In the following discussion, we will omit the subscript *U*, when there is no confusion. Giving the concept of KC, in principle, we can judge whether the KC of a strategy is within the given memory limit. Using the terminology of OO, KC is the true model to determine whether a strategy is simple. Unfortunately, it is a basic result in the theory of Kolmogorov complexity that the KC cannot be computed by any computers precisely in general (Theorem 2.3.2, p. 121, (Li and Vitányi 1997)). From an engineering viewpoint, this means that it is extremely time-consuming to find out the true Kolmogorov performance of the proposed strategy, if it is not impossible. Thus, the methodology of OO naturally leads us to consider the usage of approximation, which is computationally fast to replace it[2,3], and to sample simple strategies. In the rest of this chapter, we will formulate this idea of simple strategy generation based on estimated descriptive complexity, which can then be utilized even if the user has little knowledge or experience of what a simple strategy might look like. This is in contrast to existing efforts where no quantification on the descriptive complexity for the strategies is explored. Examples include threshold type of strategies, Neuro-dynamic programming (NDP) (Bertsekas and Tsitsiklis 1996) which uses neural networks to parameterize strategy space, State aggregation (Ren and Krogh 2002), time aggregation (Cao et al. 2002), action aggregation (Xia et al. 2004), and event-based optimization (Cao 2005).

The crude model of the KC for a strategy we would like to introduce here is the size of a program based on the reduced ordered binary decision diagram (ROBDD, or simply OBDD) representation for the strategy which will be introduced below. ROBDD regards each strategy as a (high-dimensional) Boolean function. (For simplicity we let the output decision variable be binary. This can be generalized in obvious ways. See Exercises 6.1 and 6.2 below.) The observation behind this is that reduced ordered binary decision diagrams (ROBDDs) usually supply a succinct description for a Boolean function (Andersen 1997). Let us first describe how ROBDD can be obtained for a Boolean function and furthermore for a strategy through an example.

[2] In the same spirit of OO with constraints in Chapter V but different in that we are using an upper bound estimation for memory used for describing a strategy, so we never include infeasible strategies in our selected set, nor do we know when a strategy is not estimated as simple, what is the probability for it to be truly simple.

[3]It should be noted that although KC in general cannot be calculated by computers, there are extensions of KC that can be calculated by computers, such as the Levin complexity (Levin 1973, 1984), which considers both the length of the program and the time for the program to be executed. It is still an open question how to combine Levin complexity with ordinal optimization to find simple and good enough strategies.

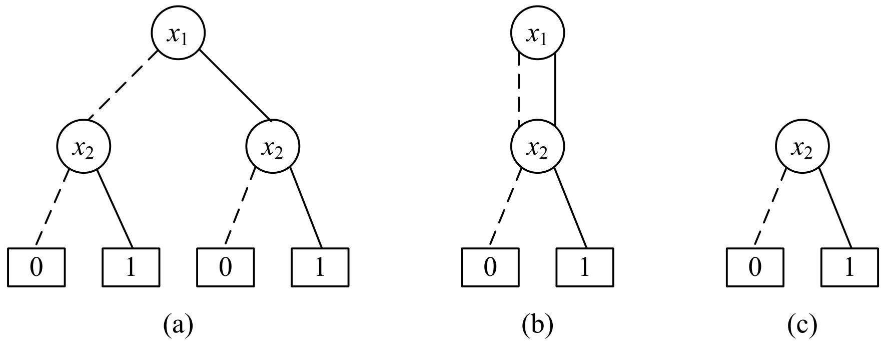

Fig. 6.2. Reduction process from BDD to OBDD for the function $f(x_1,x_2) = (x_1 \wedge x_2) \vee (\neg x_1 \wedge x_2)$

In Fig. 6.2(a), we use a BDD (Binary Decision Diagram) to describe the Boolean function $f(x_1,x_2) = (x_1 \wedge x_2) \vee (\neg x_1 \wedge x_2)$, where $\wedge$ is AND, $\vee$ is OR, and $\neg$ is NOT. To construct a BDD for f, we start from a root node (circle) representing a Boolean variable (x_1 for this example). We connect to this node a dotted line and a solid line representing "if the variable takes value "0" or "1", respectively. For each branch (dotted line or solid line), we add a new node (circle) by choosing a new Boolean variable (x_2 for this example). We keep this branching procedure until all Boolean variables have been added and two lines added. Note that any path from the root node to a latest added node is corresponding to an assignment of 0 or 1 to all Boolean variables. For example, in Fig. 6.2(a), the path in which both node x_1 and x_2 take the dotted branch is corresponding to the assignment $(x_1, x_2) = (0,0)$. As the last step to construct a BDD, we add at the end of each path a box labeled by the evaluation of f under the assignment corresponding to the path. For the assignment $(x_1, x_2) = (0,0)$, we attach a box labeled "0" because $f(0,0) = 0$. Before doing any reduction, BDD will have an exponentially (in n) large number of nodes. One way to reduce BDD is to introduce order when adding Boolean variables. If, in a BDD, all paths choose the same order when adding new Boolean variables, we get an OBDD, where the first "O" stands for "ordered". Fig. 6.2(a) is in fact an OBDD.

OBDDs allow us to find simpler representation for Boolean functions. We can combine the redundant nodes, i.e., the nodes with identical subgraphs. For example, Fig. 6.2(b) gives a more compact OBDD than Fig. 6.2(a). Obviously, if both of the two lines connected to a node are connected to the same successor on the other end (e.g., the lines connected to node x_1 in Fig. 6.2(b)), this means the input value of this Boolean variable does not affect the output of the function. So this node can be removed to make the OBDD more compact. The OBDD in Fig. 6.2(b) can be further simplified to the one in Fig. 6.2(c), where node x_1 is removed. By eliminating redundancies in an OBDD, a unique OBDD can be obtained which is called ROBDD[4] for the Boolean function. In the rest, when we mention an OBDD of a strategy, we will always refer to the ROBDD for the strategy.

Exercise 6.1: How can we encode a strategy from a finite information space to a finite action space with a high dimensional Boolean function?

The readers may consult Chapter VIII. 4 for such an example.

Exercise 6.2: How can we generalize the above OBDD to represent more than one-bit outputs, say two bits? In other words, there are totally four actions, 00, 01, 10, and 11.

Exercise 6.3: Currently there is no randomness in OBDDs. Is it possible to introduce any randomness in OBDDs? In other words, instead of deterministically selecting either the dotted line or the solid line and thus deterministically outputting 0 or 1 finally, can we generalize the OBDD to randomly output 0 or 1? How? If possible, please show the example when the OBDD outputs two-bit actions. What is the advantage of these random OBDDs comparing with the deterministic OBDDs?

Once we have an OBDD for a Boolean function describing a strategy, we can follow a natural way to convert the OBDD to a program that can represent the strategy. For a given input to the strategy, the purpose of the program is to generate the output (either 0 or 1 if the strategy has only two actions to choose) for the strategy. We start from the top node of the OBDD, considering which line to choose (and thus which successor to go to) according to the input values of the Boolean variables until arriving at the bottom box (either 0 box or 1 box), and then output the value in the box. This procedure can be described by a sequence of rules. Each rule looks like

(*state*, *input*, *action*, *next state*),

where *state* represents which node the program is currently at, *input* represents the input value of the Boolean variable associated with that node, *action* describes what the program is going to do (such as to choose

[4] The ROBDD depends on the order of variables.

either of the lines if the program is staying at a node; or to output either "0" or "1", if the program is at one of the bottom boxes; or simply to end the program if the output is already done), and the *next state* represents the node that the program is going to (either the low- or the high-successor of the current node, if the program is now staying at a node; or an END state which describes the end of the program, if the program is now staying at one of the bottom boxes.). For example, the rules to describe the OBDD in Fig. 6.2(c) are:

(node x_2, 0, choose the dotted line, box 0),
(node x_2, 1, choose the solid line, box 1),
(box 0, э, output 0, END),
(box 1, э, output 1, END),

where э means that no input is needed.

Based on this program representation of a strategy, we can estimate its KC as $\hat{C}(\gamma) = 4(2b+2)\lceil \log_2(b{+}3{+}4) \rceil$ by calculating number of bits to implement the strategy, where b is the number of nodes (excluding the bottom boxes) of the OBDD and $\lceil a \rceil$ represents the minimal integer no less than a. In fact, we have the following observations. In general, there are 2 rules associated with each of the nodes (excluding the bottom boxes), and there is a rule associated with each bottom box. Then the number of rules is $r = 2b + 2$. To describe each such rule, we need to encode each of the four elements in a rule by binary sequences. Since we need to distinguish all the b nodes, 2 bottom boxes, the END state, and the 4 possible actions to take (choose either the dotted line or the solid line, output either 0 or 1), we need $d = \lceil \log_2(b{+}3{+}4) \rceil$ bits to describe each element. Thus, in total, we need $4rd$ bits to implement an OBDD. Note that $4rd$ is only an estimate on the minimal number of bits to describe an OBDD. First, different order in Boolean variables may lead to OBDD with different size. Unfortunately it is too time-consuming to find the simplest OBDD to describe a strategy in general (which has been proven to be NP-hard (Bollig and Wegener 1996)). Second, there may be different requirements on the rules in different computer systems. For example, some computer systems may allow us to encode four elements separately, which means the computer knows which one of the four elements it is reading, then we can further save the number of bits to represent a rule. In some other computer systems, the value of r and d are required to be clearly explained to the computer. r and d need to be encoded in specific ways to ensure the computer understands them. Considering the different requirements in different computer systems, we may have a more detailed and more specific estimate of the number of bits to represent a strategy γ. Examples can be found in (Jia et al. 2006b).

In summary, to simplify the discussion, we use $\hat{C}(\gamma)$ to represent the number of bits given by whatever approximation. The users are free to use either $\hat{C}(\gamma) = 4rd$ or any other problem-specific estimates.

Exercise 6.4: How can we modify $\hat{C}(\gamma) = 4rd$ when there are *m*-bit outputs?

2.2 Random sampling in the design space of simple strategies

Once we have a way to estimate the descriptive complexity (KC) for a strategy as above, to take advantage of OO in searching a small set of strategies that contains given number of good enough simple strategies with high probability, we have to find a way to do random sampling in the set of strategies describable within the given memory limit[5]. Our idea is to sample only the estimated simple strategies. More specifically, we randomly generate OBDDs so that the estimated number of bits to describe this OBDD does not exceed the given memory space C_0, i.e., $\hat{C}(\gamma) \leq C_0$. The strategy described by this OBDD is by definition an estimated simple strategy[6]. By sampling these OBDDs, we are sampling simple strategies. One question of this is, as we explained earlier, there might be several OBDDs representing the same strategy, uniformly sampling the OBDDs

[5] Note in general, it is impossible to enumerate all simple strategies since the total number of simple strategies is still large.

[6] Since we are using estimation, some truly simple strategies may be excluded. Some readers might be curious to know how many true simple strategies may be excluded. Honestly, this is a difficult question. One reason is that this difference depends on which UTM is used, i.e., the hardware and the software in the computer that we use to do the optimization. Although the difference between the KC of a given string s in different UTM can be bounded by a constant, which is independent from s and only depends on the two UTMs (Li and Vitányi 1997), this constant might be large. This means the same estimate of KC might exclude different numbers of true simple strategies when different UTMs are used. However, how to choose the UTM, i.e., which software or hardware to use, is also an optimization problem, which is probably not easy. It is still an open question to study how many true simple strategies are excluded by a given estimate of KC. Thus, ultimately we must still let the end result justify our approach. See Chapter VIII for an example.

might not mean uniformly sampling the simple strategies. After introducing some restrictions, say we fix the order of the variables from the top node to the bottom box, and combine all the redundant nodes, the sampling redundancy can be sufficiently reduced. To distinguish from the usual OBDD, we call such an OBDD a partially reduced OBDD (PROBDD).

The definition of PROBDD ensures the uniqueness of the nodes in each level (the top node is in level 1 and there are at most n levels), which allows us to say: no two PROBDDs with the same number of levels represent the same Boolean function. Astute reader might notice that although n-level PROBDDs can represent all the 2^{2^n} strategies using n-bit information, some strategies that do not use all the n bit information can be represented by simpler PROBDDs. However, since there are 2^{2^i} different i-level PROBDDs, and all the Boolean functions are represented by an i-level PROBDD (where the order of the variables is $x_1 \ldots x_i$) can be represented by exactly an (i+1)-level PROBDD (where the order of the variables is $x_1 \ldots x_i x_{i+1}$), among all the 2^{2^n} strategies using n-bit information, 2^{2^i} strategies can be represented by i-level PROBDDs, $i = 1,2,\ldots n$. This result brings us two advantages. First, suppose we start from 1-level PROBDDs and incrementally increase the number of levels, until we generate all the 2^{2^n} strategies. We generate at most $\sum_{i=1}^{n} 2^{2^i}$ PROBDDs in total. The redundancy is

$$\sum_{i=1}^{n-1} 2^{2^i} \Big/ 2^{2^n} \approx 2^{-2^{n-1}},$$

which reduces to zero faster than an exponent when n increases. This shows the high efficiency of the aforementioned sampling method of simple strategies. The redundancy is ignorable. As an example, for n = 1,2,3, and 4, we test the redundancy numerically and show in Table 6.1, where the Redundancy = (Total PROBDD # − Total Strategy #)/Total Strategy # × 100%. For n = 4, the redundancy has already been very small (less than 1%). The implication is that, for large n and a given memory space, it is sufficient to uniformly sample PROBDDs for obtaining uniform samples from the estimated simple strategy space defined by $\left\{\gamma : \hat{C}(\gamma) \le C_0\right\}$.

Table 6.1. The small redundancy of the sampling method of simple strategies (Jia et al. 2006b) © 2006 IEEE

n	Total Strategy # (2^{2^n})	Total PROBDD #	Redundancy (%)
1	4	4	0
2	16	18	12.5
3	256	272	6.25
4	65536	65806	0.412

To uniformly sample PROBDDs, we first fix the order of the variables in all levels of the PROBDD, say x_1, x_2, $\ldots x_n$. Then we estimate what is the largest number of nodes that can be stored in the given memory space, denoted as $b_{\max}$. We randomly pick an integer b between 0 and $b_{\max}$, where 0 means that the PROBDD does not use any input information and always outputs 0 (or 1). Based on b, we then determine the number of the levels in the PROBDD and the number of nodes in each PROBDD. After that we randomly determine the type of the connections between the nodes in two adjacent levels (including the connections between the nodes in the last level and the bottom boxes), i.e., whether a line between two nodes is dotted or solid. In this way, we can randomly generate a PROBDD that is estimated simple.

Recall the big picture in Fig. 6.1. Once the PROBDDs representing simple strategies are randomly sampled, we remove the constraint on limited memory space from the original simulation-based strategy optimization problem. In the OO procedure, this means we have the N sampled designs from the entire design space now, i.e., Θ_N. Then we can use standard OO to find strategies in Θ_N with good enough performances as described in Chapter II. In this way, we can find simple and good enough strategies with high probability. We show an example to illustrate this procedure in details in Section VIII.4.2.

Exercise 6.5: Besides saving the memory space, what are the other advantages of simple strategies?

3 Conclusion

In summary, this chapter discusses the importance of considering the constraint of limited memory space when applying computer-based control and optimization in large scale simulation-based strategy optimization. This constraint is one of the important reasons why we can only search within the simple strategies in practice. We use multivariate Boolean functions to represent a strategy. OBDD is an efficient conceptual way to represent n-variable Boolean functions. We have developed a method to systematically explore the n-variable Boolean functions that can be captured by i-variable ($i<n$) Boolean functions for $i = 1,2,\ldots$. This exploration can be easily combined with OO to find a strategy with good enough performance and i-variable Boolean function representation for an optimization problem. In Chapter VIII Section 4, we demonstrate this on the well known Witsenhausen problem and obtain a 40-fold decrease in strategy complexity with minor (within 5%) degradation of performance.