16

Foundations of Search

A Perspective from Computer Science

James A. R. Marshall and Frank Neumann

Abstract

Since Alan Turing, computer scientists have been interested in understanding natural intelligence by reproducing it in machine form. The field of artificial intelligence is characterized, to a large extent, by search algorithms. As search is a computational process, this too has been well studied as part of theoretical computer science, leading to famous results on the computational hardness of problems. This chapter provides an overview of why most search problems are known to be hard and why general search strategies are impossible. It then discusses various heuristic approaches to computational search. The fundamental message intended is that any intelligent system of sufficient complexity, using search to guide its behavior, should be expected to find solutions that are good enough, rather than the best. In other words, it is argued that natural and artificial brains should *satisfice* rather than *optimize*.

Introduction

Almost since the first digital computers were created, computer scientists have speculated on their potential capacity for intelligent behavior (Turing 1950). Such thinking prompted the creation of the modern discipline of "artificial intelligence" (AI), which seeks to reproduce animal or human-level intelligence. Classic problem domains for AI include formal games such as chess (Shannon 1950). Other forms of richer interaction are even more interesting, the most stringent of which is probably Turing's famous "imitation game" (typically referred to now as the "Turing test") or variants thereof, in which a computer attempts to fool a human interrogator that it is itself human, by maintaining a conversation on any topic (Turing 1950). AI techniques are also applied to solve computationally hard constraint satisfaction and optimization problems,

From "Cognitive Search: Evolution, Algorithms, and the Brain," edited by Peter M. Todd, Thomas T. Hills, and Trevor W. Robbins. 2012. Strüngmann Forum Report, vol. 9, J. Lupp, series ed. Cambridge, MA: MIT Press. ISBN 978-0-262-01809-8.

such as the well-known traveling salesman problem, in which a salesman must find the shortest circular route visiting all cities on his itinerary exactly once. In all these areas of AI, computational search plays a key role.

The earliest AI approaches to chess used search methods to choose promising moves (Shannon 1950), and contemporary chess computers have managed to beat human grandmasters by deploying massive computational power alongside dictionaries of expert-provided openings and gambits. Similarly, computational search can be used in developing a conversational program to play Turing's imitation game. Finally, AI approaches to traveling salesman problems and their like use computational search to find a good quality solution.

Thus, search appears to be a mainstay of AI. In fact, it could be claimed that all intelligence is search (natural as well as artificial). This is likely to be an overstatement; a human chess expert can only evaluate a fraction of the moves considered by a computer, yet can still reliably beat their artificial opponent; similarly it is not obvious that we perform a search process analogous to the computer's when we talk to each other, yet our conversational ability is much greater. Still, it seems likely that search processes are involved in many important aspects of behavior and intelligence, both human and animal. By search process, we mean an internal search process over different representations of a problem within a brain. In this chapter, we explore what is known about computational search and its limitations, and speculate about how the study of natural intelligence might benefit from this information.

The Problem with Computational Search

Given their pervasiveness and general importance, much research effort in theoretical computer science has been invested in analyzing search problems and algorithms. A *search problem* is defined as considering a set of alternative *solutions X*, where the quality of each solution x in X can be evaluated using an *objective function f(x)*. The search problem is then usually to find the solution x with the "best" value $f(x)$ (typically by minimizing or maximizing f). In this case, the search problem is one of *optimization* according to the objective function; however, the search could also be to *satisfice* by finding a solution whose objective value satisfies some minimum requirement. The definition just given is very general; there are almost no constraints on what the solutions in X and the objective function can represent. A *search algorithm* is then an automatic procedure for attempting to find the required solution to a given problem (i.e., for *solving* the problem), whether that requirement is to find the best available solution or simply one of sufficient quality. As we will discuss, some very powerful results have been derived showing that finding a good algorithm for search problems of the kind just described can present considerable difficulties.

From "Cognitive Search: Evolution, Algorithms, and the Brain," edited by Peter M. Todd, Thomas T. Hills, and Trevor W. Robbins. 2012. Strüngmann Forum Report, vol. 9, J. Lupp, series ed. Cambridge, MA: MIT Press. ISBN 978-0-262-01809-8.

Most Interesting Search Problems Are Hard

Before considering if search problems are hard, it is necessary to define what is meant by "hard." In theoretical computer science, this is done in terms of the efficiency of algorithms to solve problems. Briefly, an *efficient algorithm* is one that runs in *polynomial time*; in our search terms, this means that the time to find the best solution in a set having size n is no longer than a polynomial function of n, such as $\log n$, n, n^2, etc. This upper bound is denoted with "big-oh" notation, such as $O(n^2)$, and one says that such an algorithm is in $O(n^2)$. In contrast, an *inefficient algorithm* is one that runs in exponential time; in other words, the upper bound is an exponential function of the size of the problem set n, such as 2^n, n^n, *etc*. As before, this upper bound is notated as $O(n^n)$, and we say that an algorithm is in $O(n^n)$, for example. This approach to studying algorithms neglects a lot of detail and focuses instead on what is really important: how the running time of the algorithm increases with the size of the solution set to which it is applied. The approach also ignores the detail of the computational device on which the algorithm is running, since all discrete computers of a certain complexity are able to do the same kinds of computation (Church 1936; Turing 1936).

Now consider the following apparently simple problem. The problem is from Boolean logic, where all formulae are written in terms of *variables*, which can be TRUE or FALSE, logical AND (TRUE if *both* arguments are TRUE, denoted $\wedge$), logical OR (TRUE if *either* argument is TRUE, denoted $\vee$), and logical NOT (TRUE if the argument is FALSE, and FALSE if the argument is TRUE, denoted $\neg$). The problem is then to find a *satisfying assignment* of truth values to variables, such that a formula of the following kind evaluates to TRUE:

$$\left(A \vee C \vee \neg D\right) \wedge \left(B \vee \neg B \vee C\right). \tag{16.1}$$

The formula above is *satisfiable* if values of the variables can be found such that the first clause (in brackets) *and* the second clause are both true. Each clause is true if *any* of its *literals* (variables or their negation) are true. Since this is a satisfiability problem (i.e., is the formula satisfiable or not?), and since each clause has three literals, it is referred to as 3-SAT.

The above may seem rather technical and arcane, but 3-SAT has some fascinating and very useful properties. First, there is no known algorithm that can solve 3-SAT in less than exponential time, so as the number of clauses in the formula grows, the search time grows exponentially. This effectively means that in the worst case, the search for a solution might need to include every member of the solution set, which is clearly bad. However 2-SAT, in which the number of literals per clause is 2 instead of 3, can be solved with a polynomial-time algorithm; this transition in hardness is also seen in other kinds of problems. The particularly interesting thing about 3-SAT, however, is that it is representative of the difficulty of all interesting search problems. 3-SAT

From "Cognitive Search: Evolution, Algorithms, and the Brain," edited by Peter M. Todd, Thomas T. Hills, and Trevor W. Robbins. 2012. Strüngmann Forum Report, vol. 9, J. Lupp, series ed. Cambridge, MA: MIT Press. ISBN 978-0-262-01809-8.

can be converted by a polynomial-time algorithm to all other problems for which no efficient algorithm is known, such as the traveling salesman problem mentioned earlier. This equivalence may not be immediately apparent, since 3-SAT is a yes/no problem, whereas the traveling salesman problem is one of optimization. A traveling salesman problem can, however, be turned into a decision problem by phrasing it as such: Is there a tour of length no more than k in this graph? Actually, the perceptive reader will see that this formulation is closer to a description of a satisficing problem than an optimization problem; the link to optimization further requires that k be reduced progressively until the answer to the preceding question is "no," at which point the optimal solution is that most recently found in determining the answer to the question for a larger value of k. Thus, finding an efficient algorithm for 3-SAT, or any of these other equivalent problems, would result in an efficient algorithm for *all* such problems. Computer scientists refer to these problems as belonging to the *complexity class* NP-Complete (NP-C). For a problem to be a member of NP-C, it means that no efficient algorithm for it is known to exist, so the only algorithms known for them run in exponential time. In contrast, those problems for which efficient algorithms *are* known belong to the complexity class P (for polynomial time). It is not known whether efficient algorithms for problems in NP-C do not exist, or whether they simply have not yet been discovered. However, given the decades of research into such problems, the consensus among computer scientists is that efficient algorithms for them really do not exist. This could be significant for brains, if they use computational search procedures for certain problems of that nature.

Efficient yet General Search Algorithms Are Impossible

The computational hardness of most interesting search problems, described in the last section, has led computer scientists to consider a number of heuristic approaches to finding solutions that are of good quality. Some of this research has been biologically inspired, although much mathematically grounded heuristics work has also been done. Some heuristics researchers, primarily in evolution-inspired algorithms, began to make claims that their heuristics were generally applicable to all problems, or even superior to alternative heuristics. In response to this, the *no-free-lunch* theorems for search were developed (Wolpert and Macready 1997). These results appear to prove that, across all possible search problems, and for any objective function, all search algorithms have equivalent performance. Given the strength of their results, these theorems require very stringent and unrealistic assumptions about the nature of the set of problems and the behavior of the algorithms. Critics countered that since these assumptions do not correspond to real-world search problems and algorithms, the no-free-lunch theorems do not offer useful results, and indeed some algorithms *can* be generally superior. A more recent extension of the

From "Cognitive Search: Evolution, Algorithms, and the Brain," edited by Peter M. Todd, Thomas T. Hills, and Trevor W. Robbins. 2012. Strüngmann Forum Report, vol. 9, J. Lupp, series ed. Cambridge, MA: MIT Press. ISBN 978-0-262-01809-8.

no-free-lunch framework relaxes these assumptions and concludes that in fact there is a generally superior search algorithm which maximizes the expected quality of the solutions it finds in any finite amount of time, but it is blind enumeration of the set of possible solutions (Marshall and Hinton 2010). Less formally, the intuitive message is that when playing a game where the aim is to draw numbered balls out of a bag to find as high a value as possible in a given number of attempts, the best strategy is not to put balls you have already seen back in the bag before drawing again! Of course, such results do not mean that a particular search algorithm cannot be designed to perform well on a particular set of related problems. However, in conjunction with the computational hardness of most interesting problems, no-free-lunch theorems do seem to rule out general, efficient search algorithms, that are able to find the optimal solution to any problem in less time than it takes to enumerate all the solutions to that problem.

Why Things Might Not Be So Bad

Our discussion thus far has shown how most interesting search problems are computationally hard and that there is no generally superior search heuristic other than blind enumeration of all possible solutions. While algorithms can still be designed on a problem-by-problem basis to find good quality solutions, these results suggest that finding general search principles in the brain might be futile, and that search processes in the brain might only be applied to comparatively simple problems. In practice, however, computational search might be easier than it first appears.

Average Case versus Worst Case

The first way in which computational search might not be so difficult is that problems which are hard in the *worst case* might be easy *on average*. In describing the class of NP-C problems above, we defined it in terms of problems whose best-known algorithms run in exponential time. However, for exponential-time algorithms, such as those in $O(2^n)$, remember that the 2^n is an *upper bound* on the running time, so the running time can actually be lower on a particular instance of a problem. This upper bound is derived in terms of *worst-case* difficulty of a problem. However, the *average-case* difficulty of a problem is much more relevant, for brains as well as for computer scientists. Returning to the 3-SAT problem, it is easy to see that many instances of the problem can quickly be discovered to be satisfiable or unsatisfiable, such as the formula:

$$\left(A \vee A \vee \neg A\right) \wedge \left(A \vee A \vee \neg A\right), \tag{16.2}$$

where *any* value of A makes the formula true, or

From "Cognitive Search: Evolution, Algorithms, and the Brain," edited by Peter M. Todd, Thomas T. Hills, and Trevor W. Robbins. 2012. Strüngmann Forum Report, vol. 9, J. Lupp, series ed. Cambridge, MA: MIT Press. ISBN 978-0-262-01809-8.

$$(A \vee A \vee A) \wedge (\neg A \vee \neg A \vee \neg A), \tag{16.3}$$

where *no* value of A makes the formula true. In fact, it is interesting to know that for 3-SAT, there is a *phase transition* in the difficulty of problem instances. Below a critical threshold of the ratio of number of clauses to number of variables, almost all instances are satisfiable; above that threshold, almost all instances are unsatisfiable (Kirkpatrick and Selman 1994). The computationally difficult instances of 3-SAT are clustered around this critical threshold. Since 3-SAT is a representative computationally hard problem, this could represent a general characteristic of other hard search problems. Although a brain could be tackling a search problem that is, in theory, computationally infeasible, in practice, for most of the instances of that problem, it either readily encounters an optimal solution or it will be quickly established that such a solution does not exist. What matters for the brain is the kinds of problems that it has encountered over evolutionary time; since these problems are unlikely to all be hard instances, effective shortcuts in the search process might evolve.

Performance of Simple Heuristics

Another way in which shortcuts might be taken by brains doing computational search is in the use of heuristics. Although earlier we pointed out that the only general search heuristic is blind enumeration, for limited classes of problems certain very simple classes of heuristic can be shown to have better-than-average performance. One particularly simple local-search heuristic is *gradient descent*. In this framework, some local structure over the set of solutions is induced by defining a neighborhood operator, such as assignments to variables that differ in only one truth-value in the 3-SAT example discussed earlier. Local search then iterates a simple procedure: evaluate the quality of the current solution (in the 3-SAT example, this could be number of true clauses), evaluate the qualities of all neighboring solutions, then move to the neighbor giving the best improvement. If no improvement is possible, the search procedure terminates and the solution arrived at is chosen as the best found during the search. It is hard to imagine a more simple-minded search procedure, yet it has been shown that, for a variety of NP-C problems (e.g., the traveling salesman problem), choosing a neighborhood function having certain properties results in some interesting performance guarantees for the search process (Grover 1992). In particular, it can be shown that the local search process always converges on a solution whose quality is better than the average quality of the solution set, and that it will do so efficiently, in time proportional to the size of the solution set. Thus, even if faced with a hard search problem, in which the solution is neither easily found nor easily shown not to exist, a brain could efficiently arrive at a better-than-average solution, by following a very simple heuristic.

From "Cognitive Search: Evolution, Algorithms, and the Brain," edited by Peter M. Todd, Thomas T. Hills, and Trevor W. Robbins. 2012. Strüngmann Forum Report, vol. 9, J. Lupp, series ed. Cambridge, MA: MIT Press. ISBN 978-0-262-01809-8.

Learning and Evolvability

Thus far we have considered the optimization of a given target function; here we turn to another relevant topic related to search: *learning*. Humans learn all the time and develop new skills. Essentially, the brain is solving a lot of classification problems. Considering such classification problems, one tries to learn a function that gives a good classification for a given set of examples. In the simplest case, consider a function f that takes elements from a given set X and classifies them as either positive or negative. The field of machine learning (Mitchell 1997) is an integral part of computer science, whose goal is to design algorithms that make use of empirical data to do classification. Based on given observed examples (training data) and their classification, a learning algorithm has to generalize this classification to unknown examples coming from the same domain.

Again, theoretical computer scientists are interested in which classes of functions can be learned in polynomial time and which classes require exponential time to be learned. The field of computational learning theory (Kearns and Vazirani 1994) studies learning from a theoretical point of view and classifies which functions can be learned efficiently. *Efficient* always means in polynomial time with respect to the given input size n and the inverse of a tolerance error ε. The goal is to determine which classes of functions can be learned in polynomial time and which ones require more computational effort.

Recently, these techniques have been used to gain new insight into evolutionary learning by considering which classes of functions are learnable through an evolutionary algorithm. This is done under the term *evolvability*. Below, we introduce the most popular models in computational learning and relate them to the notion of evolvability. Later we will introduce a kind of evolutionary algorithm that is used for learning unknown functions in practice.

PAC Learning

The most popular model of learning in computational learning theory is the *probably approximately correct* (PAC) learning model developed by Valiant (2009). This model introduces complexity theory concepts, of the kind described in earlier sections, to machine learning and thus allows one to determine which classes of functions are learnable in an efficient way.

To make the task precise, the goal is to learn an unknown function, $f: X \rightarrow Y$, mapping elements from some input space X to their corresponding *class values* in Y. The function f comes from a known concept class C which contains functions of similar structure.

In the PAC learning model, the algorithm is given random examples of X according to an unknown distribution D and their corresponding class values. The goal is to compute a hypothesis h which approximates f in the following sense: Whenever a new example x from X is drawn according to the

From "Cognitive Search: Evolution, Algorithms, and the Brain," edited by Peter M. Todd, Thomas T. Hills, and Trevor W. Robbins. 2012. Strüngmann Forum Report, vol. 9, J. Lupp, series ed. Cambridge, MA: MIT Press. ISBN 978-0-262-01809-8.

distribution D, h makes with probability $1 - \delta$ (where δ is positive but close to zero) an error of at most ε; that is, $|h(x) - f(x)| < \varepsilon$ holds with probability at least $1 - \delta$. A concept class C is *learnable* if an algorithm exists to solve the given task for every function in C in polynomial time. Note that there is no restriction on how an algorithm learns the given class of functions.

Often $Y = \{+1, -1\}$ holds; that is, the function can only take on these two class values. The reader may think of examples that are classified either as positive or negative, such as "there is a predator in the grass" or "there is not a predator in the grass" to give a biologically relevant example.

The basic PAC learning model is also referred to as distribution-independent learning as it works for any fixed distribution D. In the distribution-specific PAC learning model, the algorithm is required to learn the function f with respect to a distribution D that is known in advance. A more restricted model of PAC learning is *statistical query* (SQ) learning (Kearns 1998). SQ learning is motivated by random noise in the learning process. SQ learning is a natural restriction of PAC learning, where the algorithms are only allowed to use statistical properties of the data set rather than the individual examples.

Evolvability

Using these formal models of learning, theoretical insights can be gained on how an evolutionary process is able to learn. Recently, the learnability of evolutionary algorithms has been studied under the term evolvability. These studies provide insights into the process of evolution from the perspective of theoretical computer science. Feldman and Valiant (2008) motivate their studies with the following example: Consider the human genome of roughly 20,000 genes. For each gene, the condition under which the protein corresponding to it is expressed, in terms of all the other proteins, is encoded in its regulatory region. This means that each protein is controlled by a function f of the other 20,000 proteins. The question is: How expressive can this function be, such that it is able to perform the complex tasks of biology as well as be efficiently learnable by evolution?

Considering how difficult it is for an evolutionary process to discover a particular learning or classification algorithm and the representations used, evolution *is* relevant here. Since individual learning can also be thought of as an evolutionary process, such results can also be relevant for understanding the discovery of search and learning methods *within* individuals.

As in the PAC learning model, classes of functions are considered and examined as to whether they are evolvable (i.e., learnable by an evolutionary algorithm). Thus, similar to PAC learning—given a target function f from a concept class C of ideal functions, a class of *representations* R, and a probability distribution D over the input space—the task is to compute a representation r from R which with high probability outputs the same function value as f when choosing an input element according to D.

From "Cognitive Search: Evolution, Algorithms, and the Brain," edited by Peter M. Todd, Thomas T. Hills, and Trevor W. Robbins. 2012. Strüngmann Forum Report, vol. 9, J. Lupp, series ed. Cambridge, MA: MIT Press. ISBN 978-0-262-01809-8.

Evolvability (Feldman and Valiant 2008) considers mechanisms that can evaluate many argument functions. New functions are explored by mutation. Here it is crucial that only a small amount of the whole function set can be explored in each iteration of the evolution process, and that the evolution process only takes a limited number of generations. Furthermore, it is assumed that the performance of a function can be measured. This is crucial for evolution since better functions should have a higher chance of being transferred to the next generation. Selection of which functions to transfer to the next generation is based on this performance measure.

Valiant shows that his model of evolvability is a constrained form of PAC learning (Valiant 2009). The main difference between evolvability and PAC learning is that the general PAC learning framework allows the update of a hypothesis in an arbitrary way, depending on the examples that have been considered so far. In evolution, the update only depends on the aggregated knowledge that has been obtained during the process. This knowledge is given by the set of functions of the current generation. In contrast to the general PAC learning framework, one cannot look at a particular example presented in the past.

Valiant showed that parity functions which are learnable in the PAC framework are not evolvable (Valiant 2009). Furthermore, Feldman has given a new characterization of SQ learning and shown that if a function is SQ learnable, then it is also evolvable (Feldman 2008). This shows that a broad class of functions is learnable by evolutionary algorithms that run in polynomial time.

Having examined learning and its relation to evolvability from a theoretical point of view, we now introduce a class of evolutionary algorithms used to learn functions for classification and learning.

Genetic Programming

After having stated some theoretical results on learning, we want to examine how computer scientists make use of mechanisms in nature to come up with computer algorithms. Many scientists are acquainted with evolutionary algorithms, such as genetic algorithms. Genetic programming, developed by Koza (1991), is a type of evolutionary algorithm designed to learn certain types of functions; one evolves functions to solve the given task. Individuals are therefore functions, usually represented as trees describing mathematical expressions. Similar to the other evolutionary approaches, a set of individuals constitutes a population. A parent population creates an offspring population using *crossover* and *mutation*. The *fitness* of a function is measured in terms of its performance with respect to some test cases, possibly with penalties for unduly complicated functions to avoid overfitting. Based on their fitness, individuals from the combined populations of parents and children are selected to build the new parent population. The process is iterated until some stopping criterion is satisfied. In the case of crossover, two trees are combined to construct a new tree, which represents a new function. Mutation usually changes

From "Cognitive Search: Evolution, Algorithms, and the Brain," edited by Peter M. Todd, Thomas T. Hills, and Trevor W. Robbins. 2012. Strüngmann Forum Report, vol. 9, J. Lupp, series ed. Cambridge, MA: MIT Press. ISBN 978-0-262-01809-8.

the tree slightly such that a similar tree is obtained. A crucial difference to Valiant's notion of evolvability is that genetic programming uses less powerful operators for constructing new solutions to the given problem. Valiant only uses mutation, but allows a much more powerful mutation operator. Here, a new function can be constructed by any algorithm; the only restriction is that it must run in polynomial time. Thus, a basic question is: How powerful are operators actually for evolution, and which setting is realistic to explain the evolutionary learning process?

Genetic programming has had success in the fields of symbolic regression, financial trading, medicine, biology, and bioinformatics. A particularly relevant application of genetic programming is that of Trimmer (2010, chapter 5) in an attempt to learn the well-known Rescorla-Wagner rule (Rescorla and Wagner 1972), which describes classical conditioning:

$$V \leftarrow V + \alpha\beta(\lambda - V). \tag{16.4}$$

This rule specifies an update mechanism for learning the value of a stimulus (V) based on experienced rewards (λ) and learning rate parameters (α and β). The work of Trimmer is interesting in that it takes a *fitness landscape* approach and considers the relative performance of Rescorla-Wagner, as well as other rules which could plausibly be discovered by the evolutionary process, to examine how hard learning that particular rule might be.

As discussed above, there are many applications of genetic programming but the theoretical foundations of this type of algorithm are still in their infancy. This is due to the complex stochastic processes of such algorithms, which are hard to analyze. Different approaches have been applied to understand the behavior of genetic programming in a theoretical way, such as Markov chain analyses, convergence, and computational complexity analyses (see Poli et al. 2010). The goal of these approaches is to understand the learning process and determine which structural properties make the learning task hard or easy.

Don't Optimize, Satisfice

The predominant approach in theoretical studies of behavior is to consider the optimal solution to a particular problem as a benchmark against which observed behavior is assessed (Parker and Maynard Smith 1990). This approach is possible because typically the problem under consideration is sufficiently simple so that an optimal solution can be derived. Classic examples relevant to behavioral ecology include bandit problems, secretary problems, and statistical decision problems. As soon as sufficiently complex search problems are considered, however, the optimality approach becomes impossible, because optimal solutions to these kinds of problems are unknown and may not even exist. In this new field, therefore, the problem is no longer one of *optimization* but of *satisficing*; that is, finding solutions that are *good enough*. Herbert

From "Cognitive Search: Evolution, Algorithms, and the Brain," edited by Peter M. Todd, Thomas T. Hills, and Trevor W. Robbins. 2012. Strüngmann Forum Report, vol. 9, J. Lupp, series ed. Cambridge, MA: MIT Press. ISBN 978-0-262-01809-8.

A. Simon (1996) referred to this as procedural rationality, as opposed to substantive rationality. Animals have been argued to satisfice, even when optimal solutions are possible (Gigerenzer et al. 1999); yet in computational search, satisficing is typically not a shortcut to the best known solution, it *is* the best known solution. It might appear that satisficing is no more computationally feasible than optimization, given its link with NP-C problems, such as 3-SAT (outlined above). The important point is, provided that the search criteria are set appropriately for the search problem, that the average case complexity of satisficing is much lower that optimization. If search criteria are set too high for the distribution of solution values in some class of search problems, satisficing will be akin to optimization; however, if the criteria are set low enough, but not so low as to accept anything, a useful satisfactory solution will usually be found with a relatively small amount of searching.

In conclusion, we hope that our discussion has explained why optimal search is so uniformly impossible in computational search problems. We suggest that the results on heuristic search from the computational science community offer a rich store of ideas for those interested in behavior and cognition.

Acknowledgments

We are grateful to Nathaniel D. Daw and Thomas T. Hills for helpful comments on an earlier draft and to the Ernst Strüngmann Forum for the invitation to participate in this extended discussion.

From "Cognitive Search: Evolution, Algorithms, and the Brain," edited by Peter M. Todd, Thomas T. Hills, and Trevor W. Robbins. 2012. Strüngmann Forum Report, vol. 9, J. Lupp, series ed. Cambridge, MA: MIT Press. ISBN 978-0-262-01809-8.

From “Cognitive Search: Evolution, Algorithms, and the Brain,” edited by Peter M. Todd, Thomas T. Hills, and Trevor W. Robbins. 2012. Strüngmann Forum Report, vol. 9, J. Lupp, series ed. Cambridge, MA: MIT Press. ISBN 978-0-262-01809-8.