Analysis of Algorithms and Partial Algorithms

Andrew MacFie(✉)

School of Mathematics and Statistics, Carleton University, Ottawa, Canada
amacfie@sent.com

Abstract. We present an alternative methodology for the analysis of algorithms, based on the concept of expected discounted reward. This methodology naturally handles algorithms that do not always terminate, so it can (theoretically) be used with partial algorithms for undecidable problems, such as those found in artificial general intelligence (AGI) and automated theorem proving. We mention an approach to self-improving AGI enabled by this methodology.

1 Introduction: Shortcomings of Traditional Analysis of Algorithms

Currently, the (running time) analysis of algorithms takes the following form. Given two algorithms A, B that solve the same problem, we find which is more efficient by asymptotically comparing the running time sequences (a_n), (b_n) [4,15]. This could be using worst-case or average-case running times or even smoothed analysis [16]. We refer to this general method as *traditional analysis of algorithms*.

As with any model, traditional analysis of algorithms is not perfect. Authors have noted [1,9] that comparing sequence tails avoids the arbitrariness of any particular range of input lengths but leads us to say $a_n = n^{100}$ is superior to $b_n = \big(1 + \exp(-10^{10})\big)^n$ which is false for practical purposes.

A further issue with traditional analysis is illustrated by this situation: Say we have a function $F : \{0,1\}^* \to \{0,1\}$ and an algorithm A that computes F such that for $n \geq 0$, A takes $(n!)!$ steps on the input 0^n and n steps on any other input of length n. The algorithm A then has worst-case running time $(n!)!$ and average-case running time slightly greater than $2^{-n}(n!)!$, which are both terrible. However, if the inputs are generated according to a uniform distribution, the probability of taking more than n steps is 2^{-n} which is quickly negligible. We see that A should be considered an excellent algorithm but traditional analysis does not tell us that, unless we add "with high probability".

The same issue arises if A simply does not halt on 0^n, in which case the worst-case and average-case running times are infinite. Indeed, this is not an esoteric phenomenon. For any problem with Turing degree $\mathbf{0}'$ we cannot have an algorithm that halts on every input, but we develop partial solutions that work on a subset of inputs. Such problems include string compression (Kolmogorov complexity), the halting problem in program analysis [2], algebraic simplification [17], program optimization, automated theorem proving, and Solomonoff

B. Steunebrink et al. (Eds.): AGI 2016, LNAI 9782, pp. 284–293, 2016.
DOI: 10.1007/978-3-319-41649-6_29

induction (central to artificial general intelligence [13]). E.g. in the case of automated theorem proving, Buss, describing the main open problems in proof theory [3], states, "Computerized proof search ... is widely used, but almost no mathematical theory is known about the effectiveness or optimality of present-day algorithms."

Definition 1. *An algorithm A is a* partial algorithm *(a.k.a. computational method [12, p. 5]) for a given problem if on all inputs, A either outputs the correct value, or does not terminate.*

Definition 2. *We refer to partial algorithms for problems with Turing degree $\mathbf{0}'$ as $\mathbf{0}'$* algorithms.

To analyze $\mathbf{0}'$ algorithms, and perhaps to better analyze normal terminating algorithms, we need a new approach that is not based on worst-case or average-case running time sequences. In Sect. 2 we present a new method for analyzing algorithms, called expected-reward analysis that avoids some of the issues mentioned above. Then in Sect. 3 we mention how this method can be used in self-improving AI systems. We give directions for further work in Sect. 4.

Notation 1. *Given a (possibly partial) algorithm A and an input ω, we denote the number of steps taken by A on ω by $c_A(\omega)$, which takes the value ∞ if A does not halt on ω.*

2 Expected-Reward Analysis of Algorithms

2.1 Definition

Let A be a (possibly partial) algorithm with inputs in Ω. We say the *score* of A is

$$S(A) = \sum_{\omega \in \Omega} P(\{\omega\}) r(\omega) D(c_A(\omega)) = E(r \cdot (D \circ c_A)),$$

where P is a probability measure on Ω, D is a discount function [7], and $r(\omega)$ is a reward (a.k.a. utility) value associated with obtaining the solution to ω. The expression $S(A)$ may be interpreted as the expected discounted reward that A receives if run on a random input, and the practice of comparing scores among algorithms we call *expected-reward analysis*. A higher score indicates a more efficient algorithm.

The functions D and r are arbitrary and are free to be set in the context of a particular application. E.g. in graphical user interface software we often desire near-instant responses, with utility rapidly dropping off with time. Assuming $0 \leq r \leq 1$, we immediately see that for all A, partial or not, we have

$$0 \leq S(A) \leq 1.$$

For simplicity in this paper we assume $r(\omega) = 1$ and D is an exponential discount function, i.e.

$$D(c_A(\omega)) = \exp(-\lambda \, c_A(\omega)),$$

where $\lambda > 0$ is a discount rate.

The choice of P is also arbitrary; we remark on two special cases. If all inputs of a given length are weighted equally, P is determined by a probability mass function on $\mathbb{Z}_{0+}$. In this case any common discrete probability distribution may be used as appropriate. The measure P is also determined by a probability mass function on $\mathbb{Z}_{0+}$ if we weight equal-length inputs according to Solomonoff's universal distribution m [13], which is a particularly good general model, although computationally difficult.

Expected-reward analysis is non-asymptotic, in the sense that all inputs potentially matter. Thus, while expected-reward analysis can be used on terminating algorithms, we expect it to give different results from traditional analysis, in general. Since particular inputs can make a difference to $S(A)$, it may be advantageous to "hardcode" initial cases into an algorithm. This practice certainly exists, e.g. humans may store the 12×12 multiplication table as well as knowing a general integer multiplication algorithm.

Computational complexity theory often works with classes of problems whose definitions are equivalent for all "reasonable" models of computation [5]. However, even a varying constant factor could arbitrarily change a score. This is simply the price of concreteness, and outside of complexity theory, traditional analysis of algorithms generally selects a particular model of computation and gives precise results that do not necessarily apply to other models [6].

Unlike traditional analysis, experimental data is relevant to score values in a statistical sense. If we are able to generate inputs according to P, either artificially or by sampling inputs found in practice, $S(A)$ is a quantity amenable to statistical estimation. This suggests a form of experimental analysis of algorithms which focuses on a single real number rather than plotting the estimated running time for every input length, which, in the necessary absence of asymptotics in experimental analysis, may not conclusively rank two competing algorithms anyway.

The expected-reward paradigm already appears in the analysis of artificial agents, rather than algorithms [8]. As we see in Sect. 3, however, even in applications to AI, working in the more classical domain of algorithms brings benefits.

2.2 Theory and Practice

Traditional analysis of algorithms has an established literature going back decades which provides a set of techniques for performing traditional analysis on algorithms developed for various problems. We do not significantly develop a mathematical theory of expected-reward analysis here, but we make some very brief initial remarks.

By way of introductory example, we consider expected-reward analysis applied to some well-known sorting algorithms. Let S_n be the set of permutations of $[1..n]$ and let $\mathit{\Pi}_n$ be a uniform random element of S_n. We denote the algorithms mergesort and quicksort by M and Q, as defined in [15], and set

$$m_n = E\left[\exp(-\lambda\, c_M(\mathit{\Pi}_n))\right], \quad q_n = E\left[\exp(-\lambda\, c_Q(\mathit{\Pi}_n))\right],$$

where $c_A(\omega)$ is the number of comparison operations used by an algorithm A to sort an input ω.

Proposition 1. *For $n \geq 1$ we have*

$$m_n = \exp\left(-\lambda(n\lceil \lg(n)\rceil + n - 2^{\lceil \lg(n)\rceil})\right), \qquad m_0 = 1, \tag{1}$$

$$q_n = \frac{e^{-\lambda(n+1)}}{n} \sum_{k=1}^{n} q_{k-1} q_{n-k}, \qquad q_0 = 1.$$

Proof. From [15], M makes the same number of comparisons for all inputs of length $n \geq 1$:

$$c_M(\Pi_n) = n\lceil \lg(n)\rceil + n - 2^{\lceil \lg(n)\rceil},$$

so (1) is immediate.

Now, when Q is called on Π_n, let $\rho(\Pi_n)$ be the pivot element, and let $\underline{\Pi}_n, \overline{\Pi}_n$ be the subarrays constructed for recursive calls to Q, where the elements in $\underline{\Pi}_n$ are less than $\rho(\Pi_n)$, and the elements in $\overline{\Pi}_n$ are greater.

We have

$$\begin{aligned} &E[\exp(-\lambda c_Q(\Pi_n))] \\ &\quad = \frac{1}{n}\sum_{k=1}^{n} E[\exp(\,-\lambda(n+1+c_Q(\underline{\Pi}_n)+c_Q(\overline{\Pi}_n))\,) \mid \rho(\Pi_n)=k] \\ &\quad = \frac{e^{-\lambda(n+1)}}{n}\sum_{k=1}^{n} E[\exp(\,-\lambda(c_Q(\underline{\Pi}_n)+c_Q(\overline{\Pi}_n))\,) \mid \rho(\Pi_n)=k]. \end{aligned}$$

It can be seen that given $\rho(\Pi_n) = k$, $\underline{\Pi}_n$ and $\overline{\Pi}_n$ are independent, thus

$$\begin{aligned} &E[\exp(-\lambda c_Q(\Pi_n))] \\ &\quad = \frac{e^{-\lambda(n+1)}}{n}\sum_{k=1}^{n} E[\exp(-\lambda c_Q(\underline{\Pi}_n)) \mid \rho(\Pi_n)=k]\cdot \\ &\qquad\qquad E[\exp(-\lambda c_Q(\overline{\Pi}_n)) \mid \rho(\Pi_n)=k] \\ &\quad = \frac{e^{-\lambda(n+1)}}{n}\sum_{k=1}^{n} E[\exp(-\lambda c_Q(\Pi_{k-1}))]E[\exp(-\lambda c_Q(\Pi_{n-k}))]. \qquad \square \end{aligned}$$

From examining the best-case performance of Q, it turns out that $c_M(\Pi_n) \leq c_Q(\Pi_n)$ for all n, so the expected-reward comparison of M and Q is easy: $S(M) \geq S(Q)$ for any parameters. However, we may further analyze the absolute scores of M and Q to facilitate comparisons to arbitrary sorting algorithms. When performing expected-reward analysis on an individual algorithm, our main desideratum is a way to quickly compute the score value to within a given precision for each possible parameter value P, λ. Proposition 1 gives a way of computing scores of M and Q for measures P that give equal length inputs equal weight, although it does not immediately suggest an efficient way in all cases. Bounds on scores are also potentially useful and may be faster to compute; in the next proposition, we give bounds on m_n and q_n which are simpler than the exact expressions above.

Proposition 2. *For $n \geq 1$,*

$$\frac{e^{-2\lambda(n-1)}}{(n-1)!^{\lambda/\log(2)}} \leq m_n \leq \frac{e^{-\lambda(n-1)}}{(n-1)!^{\lambda/\log(2)}}. \tag{2}$$

For all $0 < \lambda \leq \log(2)$ and $n \geq 0$,

$$\frac{e^{-2\gamma\lambda(n+1)-\lambda}}{(n+1)!^{2\lambda}}(2\pi(n+1))^{\lambda} < q_n \leq \frac{e^{-2\lambda n}}{(n!)^{\lambda/\log(2)}},$$

where γ is Euler's constant.

Proof. Sedgewick and Flajolet [15] give an alternative expression for the running time of mergesort:

$$c_M(\Pi_n) = \sum_{k=1}^{n-1} \left(\lfloor \lg k \rfloor + 2\right).$$

Statement (2) follows from this because

$$\log(k)/\log(2) + 1 < \lfloor \lg k \rfloor + 2 \leq \log(k)/\log(2) + 2.$$

With $0 < \lambda \leq \log(2)$, we prove the upper bound

$$q_n \leq \frac{e^{-2\lambda n}}{(n!)^{\lambda/\log(2)}} \tag{3}$$

for all $n \geq 0$ by induction. Relation (3) clearly holds for $n = 0$. We show that (3) can be proved for $n = N$ ($N > 0$) on the assumption that (3) holds for $0 \leq n \leq N-1$. Proposition 1 gives

$$\begin{aligned} q_N &= \frac{e^{-\lambda(N+1)}}{N} \sum_{k=1}^{N} q_{k-1} q_{N-k} \\ &\leq \frac{e^{-\lambda(N+1)}}{N} \sum_{k=1}^{N} \frac{e^{-2\lambda(k-1)}}{((k-1)!)^{\lambda/\log(2)}} \frac{e^{-2\lambda(N-k)}}{((N-k)!)^{\lambda/\log(2)}} \end{aligned}$$

(by the assumption)

$$\begin{aligned} &= e^{-3\lambda N+\lambda} \left(\frac{1}{N} \sum_{k=1}^{N} \left(\frac{1}{(k-1)!} \frac{1}{(N-k)!} \right)^{\lambda/\log(2)} \right) \\ &\leq e^{-3\lambda N+\lambda} \left(\frac{1}{N^{\lambda/\log(2)}} \left(\sum_{k=1}^{N} \frac{1}{(k-1)!} \frac{1}{(N-k)!} \right)^{\lambda/\log(2)} \right) \end{aligned}$$

(by Jensen's inequality, since $0 < \lambda/\log(2) \le 1$)

$$= e^{-3\lambda N+\lambda}\left(\frac{(2^{N-1})^{\lambda/\log(2)}}{(N!)^{\lambda/\log(2)}}\right)$$
$$= \frac{e^{-2\lambda N}}{(N!)^{\lambda/\log(2)}}.$$

Thus (3) has been proved for all $n \ge 0$.

For the lower bound on q_n, we use the probabilistic form of Jensen's inequality,

$$q_n = E\left[\exp(-\lambda c_Q(\Pi_n))\right] \ge \exp(-\lambda E\left[c_Q(\Pi_n)\right]),$$

noting that average-case analysis of quicksort [15] yields

$$E\left[c_Q(\Pi_n)\right] = 2(n+1)(H_{n+1}-1), \qquad n \ge 0,$$

where (H_n) is the harmonic sequence. For $n \ge 0$, the bound

$$H_{n+1} < \log(n+1) + \gamma + \frac{1}{2(n+1)}$$

holds [11] (sharper bounds exist), so we have

$$q_n > \exp\left(-2\lambda(n+1)\left(\log(n+1) + \gamma + \frac{1}{2(n+1)} - 1\right)\right)$$
$$= e^{-2(\gamma-1)\lambda(n+1)-\lambda}(n+1)^{-2\lambda(n+1)}.$$

We finish by applying Stirling's inequality

$$(n+1)^{-(n+1)} \ge \sqrt{2\pi(n+1)}\, e^{-(n+1)}/(n+1)!, \qquad n \ge 0. \qquad \square$$

From these results we may get a sense of the tasks involved in expected-reward analysis for typical algorithms. We note that with an exponential discount function, the independence of subproblems in quicksort is required for obtaining a recursive formula, whereas in traditional average-case analysis, linearity of expectation suffices.

We end this section by mentioning an open question relevant to a theory of expected-reward analysis.

Question 1. If we fix a computational problem and parameters P, λ, what is $\sup_A S(A)$, and is it attained?

If $\sup_A S(A)$ is not attained then the situation is similar to that in Blum's speedup theorem. Comparing $\sup_A S(A)$ among problems would be the expected-reward analog of computational complexity theory but because of the sensitivity of S to parameters and the model of computation, this is not useful.

3 Self-improving AI

The generality of $\mathbf{0}'$ problems allows us to view design and analysis of $\mathbf{0}'$ algorithms as a task which itself may be given to a $\mathbf{0}'$ algorithm, bringing about recursive self-improvement. Here we present one possible concrete example of this notion and discuss connections with AI.

Computational problems with Turing degree $\mathbf{0}'$ are Turing-equivalent so without loss of generality in this section we assume $\mathbf{0}'$ algorithms are automated theorem provers. Specifically, we fix a formal logic system, say ZFC (assuming it is consistent), and take the set of inputs to be ZFC sentences, and the possible outputs to be `provable` and `not provable`.

Let a predicate β be such that $\beta(Z)$ holds iff Z is a $\mathbf{0}'$ algorithm which is correct on provable inputs and does not terminate otherwise. In pseudocode we write the instruction to run some Z on input ω as $Z(\omega)$, and if ω contains β or S (the score function), their definitions are implicitly included.

We give an auxiliary procedure Search which takes as input a $\mathbf{0}'$ algorithm Z and a rational number x and uses Z to obtain a $\mathbf{0}'$ algorithm which satisfies β and has score greater than x (if possible). Symbols in bold within a string literal get replaced by the value of the corresponding variable. We assume $\mathbf{0}'$ algorithms are encoded as strings in a binary prefix code.

```
1: procedure Search($x, Z$)
2:     $u \leftarrow$ the empty string
3:     loop
4:         do in parallel until one returns provable:
5:             A: $Z(``\exists v : (Z^* = \mathbf{u}0v \implies \beta(Z^*) \wedge S(Z^*) > \mathbf{x})")$
6:             B: $Z(``\exists v : (Z^* = \mathbf{u}1v \implies \beta(Z^*) \wedge S(Z^*) > \mathbf{x})")$
7:             C: $Z(``Z^* = \mathbf{u} \implies \beta(Z^*) \wedge S(Z^*) > \mathbf{x}")$
8:         if A returned provable then
9:             $u \leftarrow u0$
10:        if B returned provable then
11:            $u \leftarrow u1$
12:        if C returned provable then
13:            return $u$
```

We remark that the mechanism of Search is purely syntactic and does not rely on consistency or completeness of ZFC, or the provability thereof. This would not be the case if we strengthened β to require that $\beta(Z)$ is true only if at most one of $Z(\omega)$ and $Z(\neg\omega)$ returns `provable`. Such a β would never provably hold in ZFC.

The following procedure Improve takes an initial $\mathbf{0}'$ algorithm Z_0 and uses dovetailed calls to Search to output a sequence of $\mathbf{0}'$ algorithms that tend toward optimality.

```
procedure IMPROVE($Z_0$)
    $best \leftarrow Z_0$, $pool \leftarrow \{\}$, $score \leftarrow 0$
    for $n \leftarrow 1$ to $\infty$ do
        $a_n \leftarrow n$th term in Stern-Brocot enumeration of $\mathbb{Q} \cap (0,1]$
        if $a_n > score$ then
            $initialState \leftarrow$ initial state of SEARCH($a_n, best$)
            add $(a_n, best, initialState)$ to $pool$
        $improvementFound \leftarrow$ false
        for $(a, Z, state)$ in $pool$ do
            run SEARCH($a, Z$) one step starting in state $state$
            $newState \leftarrow$ new current state of SEARCH($a, Z$)
            if $state$ is not a terminating state then
                in $pool$, mutate $(a, Z, state)$ into $(a, Z, newState)$
                continue
            $improvementFound \leftarrow$ true
            $best \leftarrow$ output of SEARCH($a, Z$)
            $score \leftarrow a$
            for $(\hat{a}, \hat{Z}, \hat{state})$ in $pool$ where $\hat{a} \leq score$ do
                remove $(\hat{a}, \hat{Z}, \hat{state})$ from $pool$
            print $best$
        if $improvementFound$ then
            for $(a, Z, state)$ in $pool$ do
                $initialState \leftarrow$ initial state of SEARCH($a, best$)
                add $(a, best, initialState)$ to $pool$
```

The procedure IMPROVE has the following basic property.

Proposition 3. *Let (Z_n) be the sequence of* $\mathbf{0}'$ algorithms *printed by* IMPROVE. *If $\beta(Z_0)$ holds, and if there is any* $\mathbf{0}'$ algorithm Y *and* $s \in \mathbb{Q}$ *where* $\beta(Y)$ *and* $S(Y) > s > 0$ *are provable, we have*

$$\lim_{n \to \infty} S(Z_n) \geq s.$$

If (Z_n) is finite, the above limit can be replaced with the last term in (Z_n).

Proof. The value s appears as some value a_n. For $a_n = s$, if $a_n > score$ in line 5, then SEARCH($s, best$) will be run one step for each greater or equal value of n and either terminates (since Y exists) and $score$ is set to s, or is interrupted if we eventually have $score \geq s$ before SEARCH($s, best$) terminates. It suffices to note that when $score$ attains any value $x > 0$, all further outputs Z satisfy $S(Z) > x$ and there is at least one such output. □

The procedure IMPROVE also makes an attempt to use recently printed $\mathbf{0}'$ algorithms in calls to SEARCH. However, it is not true in general that $S(Z_{n+1}) \geq S(Z_n)$. Checking if a particular output Z_n is actually an improvement over Z_0 or Z_{n-1} requires extra work.

In artificial general intelligence (AGI) it is desirable to have intelligent systems with the ability to make autonomous improvements to themselves [14]. If an AGI system such as an AIXI approximation [10] already uses a $\mathbf{0}'$ algorithm Z to compute the universal distribution m, we can give the system the ability to improve Z over time by devoting some of its computational resources to running Improve. This yields a general agent whose environment prediction ability tends toward optimality.

4 Future Work

We would like to be able to practically use expected-reward analysis with various parameter values, probability measures, and discount functions, on both terminating and non-terminating algorithms. Particularly, we would like to know whether $\mathbf{0}'$ algorithms may be practically analyzed. It may be possible to develop general mathematical tools and techniques to enhance the practicality of these methods, such as exist for traditional analysis; this is a broad and open-ended research goal.

Acknowledgements. The author wishes to thank Zhicheng Gao, Nima Hoda, Patrick LaVictoire, Saran Neti, and anonymous referees for helpful comments.

References

1. Aaronson, S.: Why philosophers should care about computational complexity. In: Computability: Gödel, Turing, Church, and Beyond (2012)
2. Burnim, J., Jalbert, N., Stergiou, C., Sen, K.: Looper: lightweight detection of infinite loops at runtime. In: International Conference on Automated Software Engineering (2009)
3. Buss, S.: Re: proof theory on the eve of year 2000 (1999). http://www.ihes.fr/~carbone/papers/proofsurveyFeferman2000.html. Accessed: 04 Oct 2015
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press, Cambridge (2009)
5. van Emde Boas, P.: Handbook of Theoretical Computer Science, vol. A, pp. 1–66. MIT Press, Cambridge (1990)
6. Flajolet, P., Sedgewick, R.: Analytic Combinatorics. Cambridge University Press, Cambridge (2009)
7. Frederick, S., Loewenstein, G., O'Donoghue, T.: Time discounting and time preference: a critical review. J. Econ. Lit. **40**, 351–401 (2002)
8. Goertzel, B.: Toward a formal characterization of real-world general intelligence. In: Proceedings of the 3rd Conference on Artificial General Intelligence, AGI, pp. 19–24 (2010)
9. Gurevich, Y.: Feasible functions. London Math. Soc. Newslett. **206**, 6–7 (1993)
10. Hutter, M.: Universal Artificial Intelligence: Sequential Decisions based on Algorithmic Probability. Springer, Berlin (2005)
11. Julian, H.: Gamma: Exploring Euler's Constant. Princeton University Press, Princeton (2003)
12. Knuth, D.E.: The Art of Computer Programming, vol. 1. Addison-Wesley, Reading (1997)

13. Li, M., Vitányi, P.: An Introduction to Kolmogorov Complexity and its Applications. Springer Science & Business Media, New York (2013)
14. Schmidhuber, J.: Gödel machines: fully self-referential optimal universal self-improvers. In: Goertzel, B., Pennachin, C. (eds.) Artificial General Intelligence, pp. 199–226. Springer, Heidelberg (2007)
15. Sedgewick, R., Flajolet, P.: An Introduction to the Analysis of Algorithms. Addison-Wesley, Reading (2013)
16. Spielman, D.A., Teng, S.H.: Smoothed analysis: an attempt to explain the behavior of algorithms in practice. Commun. ACM **52**(10), 76–84 (2009)
17. Trott, M.: The Mathematica Guidebook for Symbolics. Springer Science & Business Media, New York (2007)