From Planning to Searching for the Shortest Plan: An Optimal Transition

R. Trejo, J. Galloway
C. Sachar, V. Kreinovich

Department of Computer Science
University of Texas at El Paso
El Paso, TX 79968, USA
{rtrejo,vladik}@cs.utep.edu

C. Baral
Le Chi Tuan

Dept. Computer Science & Eng.
Arizona State University
Tempe, AZ 85287-5406, USA

Abstract

If we want to find the shortest plan, then usually, we try plans of length 1, 2, ..., until we find the first length for which such a plan exists. When the planning problem is difficult and the shortest plan is of a reasonable length, this linear search can take a long time; to speed up the process, it has been proposed to use binary search instead. Binary search for the value of a certain parameter x is optimal when for each tested value x, we need the same amount of computation time; in planning, the computation time increases with the size of the plan and, as a result, binary search is no longer optimal. We describe an optimal way of combining planning algorithms into a search for the shortest plan – optimal in the sense of worst-case complexity. We also describe an algorithm which is asymptotically optimal in the sense of average complexity.

Keywords: planning, optimal planning

1 Introduction

Since Kautz and Selman's paper [9] on satisfiability based planning there has been several papers [10, 15, 4, 5, 16, 13, 14, 11] on planning through finding models of a logical theory. Most of these papers focus on finding a plan of a given length; see also [17, 7, 2, 6, 12]. If we want to find *the shortest plan*, then usually, we try plans of length 1, 2, ..., until we find the first length for which such a plan exists. When the planning problem is difficult and the shortest plan is of a reasonable length, this *linear search* can take a long time; to speed up the process, it has been often mentioned – as in [13], that *binary search* should be used instead.

Binary search for the value of a certain parameter p is optimal when for each tested value p, we need the same amount of computation time (see, e.g., [3]). In planning, the computation time drastically increases with the size of the plan. As a result, as we will see, binary search is no longer optimal.

In this paper, we describe an optimal way of combining planning algorithms into a search for the shortest plan – optimal in the sense of worst-case complexity. We also describe an algorithm which is asymptotically optimal in the sense of average complexity.

2 Towards Formalization of the Problem

We assume that we have a sequence of planning algorithms, i.e., that for every integer k from 1 to some upper bound N, we have an algorithm A_k which, given a planning problem, looks for a plan of length k and returns either such a plan or, if a plan of this length does not exist, a message that no such plan is possible.

We also assume that if for some planning problem, there exists a plan p of length k, and $l > k$, then, by adding "dummy" actions, we can turn a plan p into a new plan of length l; as a result, for every $l > k$, the algorithm A_l will discover some plan for this problem. (If we want to avoid dummy actions, then, alternatively, we can assume that an algorithm A_k looks for a plan of length $\leq k$.)

We want to combine the algorithms $A_1, \ldots, A_N$ into a single algorithm which would either find the shortest plan of length $\leq N$, or return a message saying that no plan of length $\leq N$ can solve the given problem. In this combination, we start with one of the algorithms A_{k_1}, then, depending on the outcome of A_{k_1}, apply some other algorithm A_{k_2}, etc. Two examples of such combinations are linear search and binary search:

- In linear search, we start with A_1 ($k_1 = 1$); then, if A_1 did not find a plan, we proceed to A_2, etc.
- In binary search, we start with applying $A_{\lfloor N/2 \rfloor}$; if this algorithm $A_{\lfloor N/2 \rfloor}$ finds a plan, we then try $A_{\lfloor N/4 \rfloor}$, otherwise, we apply $A_{\lfloor 3N/4 \rfloor}$, etc.

We want to find a combination for which the worst-case computation time is the smallest possible. Let us fix a class of planning problems, and let c_k denote the worst-case computation time of an algorithm A_k on problems from this class. For each combination algorithm C, and for each actual size s, we can define the worst-case computation time $T(C, s)$ of the combination C on such planning problems by adding the times c_k for all the algorithms A_k which are used for the planning problems with the shortest plan of size s. Then, we can define the worst-case complexity of the combination C as the largest of the values $T(C, 1), \ldots, T(C, N)$. For example, for linear search $C_{\rm lin}$, when $s = 1$, we only apply A_1, so we need time $T(C_{\rm lin}, 1) = c_1$; when $s = 2$, we apply both A_1 and A_2, so the time is $T(C_{\rm lin}, 2) = c_1 + c_2$, etc. The worst-case computation time is when $s = N$, then this time is equal to $T(C_{\rm lin}) = T(C_{\rm lin}, N) = c_1 + \ldots + c_N$. For binary search, we will have to similarly compare $c_{\lfloor N/2 \rfloor} + c_{\lfloor N/4 \rfloor} + \ldots$ with $c_{\lfloor N/2 \rfloor} + c_{\lfloor 3N/4 \rfloor} + \ldots$, etc.

Which combination C is optimal depends on the values c_k. If $c_k = \text{const}$, then binary search is the best combination algorithm [3]. In general, however, c_k grows with k. Planning is known to be an NP-hard problem (see, e.g., [6, 1]), and so we would expect the time c_k to grow at least exponentially with k (i.e., that $c_k \geq 2^k$ and $c_{k+1}/c_k \geq 2$). Our preliminary experiments show that this is probably indeed the case: when we computed c_k as the largest time on some small set of problems, we got an exponential growth.

For exponentially growing c_k, the following algorithm is optimal:

3 Optimal Combination of Planning Algorithms Which Search for Plans of Different Sizes

Combination Algorithm C_1.

- *First, apply the algorithm A_{N-1}.*
- *The second step depends on whether there exists a plan of size $N-1$. If no such plan exists, we then apply A_N:*
  - *if A_N finds a plan, this is the shortest plan, so we stop;*
  - *if A_N does not find a plan, then no plan of size $\leq N$ is possible, so we stop.*

  If there is a plan of size $N-1$, then we apply A_{N-3}.
- *The third step depends on whether there exists a plan of size $N-3$. If no such plan exists, we then apply A_{N-2}:*
  - *if A_{N-2} finds a plan, this is the shortest plan, so we stop.*
  - *if A_{N-2} does not find a plan, then the plan produced by A_{N-1} is the shortest possible, so we stop.*

  If there is a plan of size $N-3$, then we apply A_{N-5}.
- *So, if after applying A_{N-3} we have not yet found the shortest plan, we apply A_{N-5}, and then, depending on the result, A_{N-4} or A_{N-6}. If we are not done yet, we apply A_{N-7}, and then, depending on the result, A_{N-6} or A_{N-8}. etc.*

We will prove that this combination algorithm C_1 is optimal, and that it is optimal not only when the growth is *exactly* exponential ($c_k = 2^k$), but also for the case when the growth is *approximately* exponential:

Definition 1. *Let $q = 1.325\ldots$ denote a solution to the equation $q^3 = q + 1$. We say that a sequence c_k is approximately exponential if for all k, we have $c_{k+1}/c_k \geq q$.*

Theorem 1. *If a sequence c_k is approximately exponential, then, of all possible combination algorithms, the Algorithm C_1 has the smallest possible worst-case computation time: $T(C_1) = \min_C T(C)$.*

Proof. To prove this theorem, we will prove two statements: first, that for C_1, we have $T(C_1) = T(C_1, N) = c_N + c_{N-1}$, and second, that for every other combination algorithm C, we have $T(C) \geq c_N + c_{N-1}$. From these two statements, we can conclude that $T(C) \geq T(C_1)$ for all C and therefore, that C_1 is indeed optimal.

The second statement is the easiest to prove: if the actual size is $s = N$, then the combination algorithm C has to produce a plan of size N, and the only way to produce such a plan is to apply the algorithm A_N. To prove that this is indeed the shortest plan, we need to prove that no plan of size $N-1$ exists; for this, we must apply the algorithm A_{N-1}. Thus, for $s = N$, we must apply at least two algorithms A_N and A_{N-1}. As a result, the corresponding value $T(C,N)$ has to be $\geq c_N + c_{N-1}$. Therefore, $T(C) = \max_s T(C,s) \geq T(C,N) \geq c_N + c_{N-1}$. The statement is proven.

Let us now prove the first statement. From the description of the algorithm C_1, we see that for $s = N$ and $s > N$, we have $T(C_1, s) = c_{N-1} + c_N$; for $s = N-1$ and $s = N-2$, we have $T(C_1,s) = c_{N-1} + c_{N-3} + c_{N-2}$, etc. In general, for $s = N-(2k-1)$ and $s = N-2k$, we have $T(C_1,s) = c_{N-1} + c_{N-3} + \ldots + c_{N-(2k+1)} + c_{N-2k}$. To prove that $T(C_1,N)$ is the largest of these values, we will prove that for every $k \geq 0$,

$$T(C_1, N-(2k+2)) \leq T(C_1, N-2k); \qquad (1)$$

then, by induction, we will be able to conclude that $T(C_1,s) \leq T(C_1,N) = c_N + c_{N-1}$. Substituting the above explicit expressions for $T(C_1,s)$ instead of both sides of the desired inequality (1), and canceling equal terms on both sides, we conclude that this inequality is equivalent to the following one: $c_{N-(2k+3)} + c_{N-(2k+2)} \leq c_{N-2k}$. If we divide both sides of this inequality by $c_{N-(2k+3)}$, we get an equivalent inequality $1 + q_1 \leq c_{N-2k}/c_{N-(2k+3)}$, where we denoted $q_1 = c_{N-(2k+2)}/c_{N-(2k+3)}$. Similarly, we can represent $c_{N-2k}/c_{N-(2k+3)}$ as $q_1 \cdot q_2 \cdot q_3$, where we denoted $q_2 = c_{N-(2k+1)}/c_{N-(2k+2)}$ and $q_3 = c_{N-2k}/c_{N-(2k+1)}$. Since c_k is approximately exponential, we have $q_i \geq q$ for all i. In terms of q_i, the desired inequality takes the form $1+q_1 \leq q_1 \cdot q_2 \cdot q_3$, or, equivalently, $1 \leq q_1 \cdot (q_2 \cdot q_3 - 1)$. This inequality can already be proven. Namely, since $q_i \geq q > 1$, and $q^3 + q = 1$, we have $q_1 \cdot (q_2 \cdot q_3 - 1) \geq q \cdot (q^2 - 1) = q^3 - q = 1$. The inequality is proven, and hence, the first statement is also true. Thus, the theorem is proven.

Comment. In our considerations, we estimated the computation time of each algorithm A_k by its worst-case computation time c_k. The actual computation time of applying A_k depends on the actual size of the plan: e.g., if we apply a program A_N to the planning situation in which the size of the actual plan is equal to 1, we expect A_N to run much faster than on the cases when the actual size of the plan is equal to N. To take this difference into account, let us denote, by $c_{k,s}$, the worst-case computation time of the program A_k on planning problems in which the actual size of the plan is s. It is reasonable to assume that the value $c_{k,s}$ is monotonically non-decreasing with s. We also assume that for all values $s > k$ (for which A_k cannot find the plan), the value $c_{k,s}$ is the same and equal to $c_{k,k}$; then, $c_k = \max_s c_{k,s} = c_{k,k}$.

In this new model, for each combination algorithm C and for each actual plan size s, we define $\widetilde{T}(C)$ as the largest of the values $\widetilde{T}(C,s)$, where $\widetilde{T}(C,s)$ is defined as the sum of the values $c_{k,s}$ corresponding to all the algorithms A_k which were called in the process of running C.

One can see that, in effect, the above proof of Theorem 1 also proves that the algorithm C_1 is optimal under the new definition as well, i.e., that $\widetilde{T}(C_1) \leq \widetilde{T}(C)$ for any other combination algorithm C.

4 Average-Case Computation Time: Towards Formalization of the Problem

In the previous section, we described a combination algorithm for which the worst-case computation time is the smallest possible. In this combination algorithm, we start with checking for a plan of size $N-1$. In some real-life situations, we know that most probably, the actual plan is not long; in such situations, starting with plans of (almost) maximum possible size $(N-1)$ would be, in most cases, a waste of computation time. In general, if we know the probabilities $P(s)$ of different values of the actual size, then, instead of looking for a combination algorithm C with the smallest *worst-case* computation time $T(C) = \max_s T(C,s)$, it is reasonable to look for a combination algorithm C with the smallest possible value of the *average-case* computation time $T_{\rm av}(C)$.

This average-case computation time can be defined as $T_{\rm av}(C) = \sum_s P(s) \cdot T_{\rm av}(C,s)$, where the average time $T_{\rm av}(C,s)$ can be defined as a sum of average-case computation times $c_k^{\rm av}$ of applying algorithms A_k which are called in the process of applying the combination algorithm C to the case when the actual plan size is s.

For small N, we can find the best possible combination algorithm by exhaustive search: namely, we can enumerate all possible combination algorithms, compute the value $T_{\rm av}(C)$ for all of them, and find the combination algorithm C for which this value is the smallest possible. When N increases, the number of all possible combination algorithms increases exponentially and therefore, we cannot try them all. It is therefore desirable to find a method for optimal combination when N is large.

In this paper, we present an *asymptotically optimal* algorithm C for solving this problem, i.e., an algorithm for which, as $N \to \infty$, the value $T_{\rm av}(C)$ gets closer and closer to the optimal value $\min_C T_{\rm av}(C)$. To find this algorithm, let us describe this problem in an asymptotic form.

We want to find a plan of size s which can take any value from 0 to N (0 means that the initial condition already satisfies the goal, so no actions are necessary). Let us describe the size s by a *normalized* *size* $x = s/N$ which takes $N + 1$ possible values $0, 1/N, \ldots, 1$ from the interval $[0, 1]$. For each value k from 0 to N, we know the average-case computation time $c_k^{\rm av}$ of running the algorithm A_k which looks for a plan of size k (i.e., of normalized size k/N). We will describe this computation time as function of a normalized size: $c(k/N)$ is defined as $c_k^{\rm av}$. A typical behavior is when this computation time grows exponentially with k, i.e., when $c_k^{\rm av} = a \cdot b^k$ for some constants a and b; in this case, for $x = k/N$, we have $k = x \cdot N$ and $c(x) = a \cdot b^{N \cdot x}$. This expression can be naturally defined for arbitrary values $x \in [0, 1]$; as a result, we get a monotonically non-decreasing function $c : [0, 1] \to R$ which extrapolates the values $c(k/N)$.

Similarly, we can reformulate the probabilities $P(s)$ in terms of normalized size s/N, as $P_n(s/N) = P(s)$. For the interval $[0, 1]$, it is reasonable to talk about the *probability density* $\rho(x)$ such that the probability $P_n(s/N)$ to have a normalized size from the interval $[s/N, (s+1)/N)$ is equal to $\int_{s/N}^{(s+1)/N} \rho(x)\,{\rm d}x$. In principle, we can define $\rho(x)$ as $P_n(s/N)/(1/N)$ for all $x \in [s/N, (s+1)/N)$, but more reasonably, if we have an analytical expression for $P_n(x)$, then we can extrapolate it to an analytical expression for the density function $\rho(x)$.

Originally, we have no information about the size or the normalized size of the desired plan; this normalized size can take any value from the interval $[0, 1]$, or any value > 1. Since by using algorithms A_k with $k \le N$, we cannot determine the actual size of the plan if this size is larger than N (i.e., if $s/N > 1$), we will describe this situation by saying that the actual size of the shortest plan belongs to the "extended" interval $[0, 1+]$, where "1+" indicates that values > 1 are also possible.

We want to combine the algorithms A_k corresponding to different values of normalized size $x = k/N$ into a single algorithm which would either find the shortest plan of normalized length ≤ 1, or return a message saying that no plan of normalized length ≤ 1 can solve the given problem. In this combination C, we start with some normalized size $x \in [0, 1]$ and check whether there exists a plan of this normalized size. If such a plan exists, then we know that the actual normalized size belongs to the interval $[0, x]$; if a plan of normalized size x does not exist, then we know that the actual size of the shortest plan belongs to the interval $[x, 1+]$. In each of these cases, at the next step, we select a value within the corresponding interval and check whether there exists a plan with this particular value of normalized size, etc.

For example, in binary search, we first check $x = 1/2$; then, depending on the result of this first check, we check for the values $x = 1/4$ or $x = 3/4$, etc.

Our goal is to find the actual size s of the shortest plan. In terms of the normalized size $x = s/N$, we want to find x with the accuracy of $1/N$, i.e., we want to find an interval of width $< 1/N$ which contains the desired value $x = s/N$. In the following text, we will denote $1/N$ by ε; then, the asymptotic of $N \to \infty$ corresponds to $\varepsilon \to 0$.

Now, we are ready to formulate the problem in formal terms:

- We have a monotonically non-decreasing function $c(x)$ from $[0, 1]$ to real numbers; this function describes the cost of checking for a plan of normalized size x.

- We also have a probability density $\rho(x)$; this probability density describes relative frequencies of plans of different size.

The goal of a combination algorithm is to find the actual normalized size x with the given accuracy $\varepsilon > 0$. For each such algorithm, and for each actual value x of normalized size, we can define the computation time $t(C, x)$ as the sum of all the values $c(x)$ for all points x for which the corresponding algorithm A_k was invoked. Then, we use our knowledge of the probabilities of different sizes to define the *average-case* computation time $t(C)$ as the average value of $t(C, x)$: $t(C) = \int_0^1 t(C, x) \cdot \rho(x)\,{\rm d}x$. We want to choose the combination algorithm C for which the average computation time $t(C)$ is (asymptotically, when $\varepsilon \to 0$) the smallest possible.

5 Average-Case Computation Time: Asymptotically Optimal Combination of Planning Algorithms

We will prove that the following algorithm C_2 is asymptotically optimal:

Combination Algorithm C_2**.** *In the beginning, we only know that the normalized size x of the shortest plan is somewhere in the interval $[x^-, x^+]$, where $x^- = 0$ and $x^+ = 1$.*

On each step of this algorithm, whenever we have the interval $[x^-, x^+]$, we select a checking point $x \in (x^-, x^+)$ which divides the integral $\int_{x^-}^{x^+} \rho(z) \cdot c(z)\,{\rm d}z$ into two equal halves:

$$\int_{x^-}^{x} \rho(z) \cdot c(z)\,{\rm d}z = \int_{x}^{x^+} \rho(z) \cdot c(z)\,{\rm d}z.$$

Then:

- *if there is a plan of normalized size x, then we replace x^+ by x: $[x^-, x^+] := [x^-, x]$;*

- *otherwise, we replace x^- by x: $[x^-, x^+] := [x, x^+]$.*

We continue these iterations until we get $x^+ - x^- < \varepsilon$.

Theorem 2. *When $\varepsilon \to 0$, the algorithm C_2 has (asymptotically) the smallest possible average-case computation time $t(C_2)$.*

Proof: Main Idea. Let us start with estimating the average-case computation time for the binary search algorithm C_b.

- At the first step of binary search, we check for $x = 0.5$. As a result of this first step, the interval $[x^-, x^+]$ containing the actual normalized size x is equal either to $[0, 0.5]$, or to $[0.5, 1+]$. The corresponding computation time is equal to $P([0,1]) \cdot c(0.5)$, where, for arbitrary a and b, $P([a,b])$ denotes $\int_a^b \rho(x)\,\mathrm{d}x$.
- At the second step, we check $x = 0.25$ if $s \in [0, 0.5]$ (the probability of which is $P([0, 0.5])$, and we check $x = 0.75$ if $s \in [0.5, 1]$. As a result, we get an interval $[x^-, x^+]$ of width 2^{-2}. The resulting contribution of the second step to the average computation time is $P([0, 0.5]) \cdot c(0.25) + P([0.5, 1]) \cdot c(0.75)$.
- ...
- On k-th step, the corresponding contribution is equal to

$$P([0, 2^{-(k-1)}]) \cdot c(0.5 \cdot 2^{-(k-1)}) +$$

$$P([2^{-(k-1)}, 2 \cdot 2^{-(k-1)}]) \cdot c(1.5 \cdot 2^{-(k-1)}) + \ldots$$

After k steps, we end up with an interval of width 2^{-k}; so, the number n of steps necessary to get an interval of desired width ε is determined by the equality $2^{-k} \approx \varepsilon$, i.e., $k \sim \log_2(1/\varepsilon)$.

The sum corresponding to each of these n steps is an integral sum for the integral $\int_0^1 \rho(x) \cdot c(x)\,\mathrm{d}x$, and the larger k, the closer this sum is to the integral. Thus, this integral is an asymptotic expression for the sum, and asymptotically, the total computation time is equal to the product of this integral and the total number of steps: $t(C_b) \sim (\int \rho(x) \cdot c(x)\,\mathrm{d}x) \cdot \log_2(1/\varepsilon)$.

Let us now use this estimate to find the (asymptotic) average computation time $t(C)$ for an arbitrary combination algorithm C. An arbitrary combination algorithm can be characterized as follows. Let us map the first point $x \in (0, 1)$ which is checked according to the algorithm C into a point $f(x) = 0.5$ which is first checked in binary search C_b. Then, let us map the points $x' \in (0, x)$ and $x'' \in (x, 1)$ which are checked next in C (depending on the result of checking for x) into, correspondingly, $f(x') = 0.25$ and $f(x'') = 0.75$. In general, let us map every point x which is checked in C into the value $f(x)$ which is checked at the corresponding step of the binary search algorithm C_b. As a result, we get a monotonic function $f : [0, 1] \to [0, 1]$; let $g(x)$ denote an inverse function to f, i.e., $g = f^{-1}$.

Due to our choice of this function, with respect to the new variable $y = f(x)$, the algorithm C is simply a binary search. In this binary search, the computation time $\widetilde{c}(y)$ corresponding to a value y is equal to $c(x)$ for $x = f^{-1}(y)$, i.e., $\widetilde{c}(y) = c(g(y))$. Similarly, the probability density $\widetilde{\rho}(y)$ for y can be determined from the fact that when $y = f(x)$, the probability $\widetilde{\rho}(y) \cdot dy$ that y is between y and $y + dy$ is equal to the probability $\rho(x) \cdot dx$ for $x = g(y)$ to be between $x = f^{-1}(y)$ and $x + dx = f^{-1}(y + dy) = g(y) + g'(y) \cdot dy$, where $g'(y)$ denotes the derivative of the function $g(y)$. To get an interval of y-width ε, we need the number of steps which asymptotically equals to $\log_2(1/\varepsilon) \cdot I$, where $I = \int \widetilde{\rho}(y) \cdot \widetilde{c}(y)\,\mathrm{d}y$. Substituting the above values for $\widetilde{\rho}(y)$ and $\widetilde{c}(y)$, we conclude that $I = \int \rho(x) \cdot c(x)\,\mathrm{d}x$ for the new variable $x = g(y)$. In short, to get an interval of y-width ε, we need exactly the same average computation time as binary search.

However, this computation time only leads to an interval of y-width $\Delta y = \varepsilon$, and we need an interval of x-width $\Delta x = \varepsilon$. Thus, we need to narrow down from $\Delta y = \varepsilon$ to $\Delta x = \varepsilon$. Within a narrow interval of width $\Delta y = \varepsilon$, the functions $c(x)$ and $\rho(x)$ do not change much and are asymptotically constant as $\varepsilon \to 0$. Thus, within such an interval, binary search is the best possible strategy. We know that for a binary search to narrow down an interval k times, we need $\log_2(k)$ steps. Hence, for each x, the average number of steps needed for this narrowing is equal to $\log_2(1/f'(x)) = -\log_2(f'(x))$. Therefore, the average computation time for this x is equal to $-\log_2(f'(x)) \cdot c(x)$, and the average over all possible values $x \in [0, 1]$ is equal to

$$J = -\int_0^1 \log_2(f'(x)) \cdot \rho(x) \cdot c(x)\,\mathrm{d}x. \quad (2)$$

The total average computation time is equal to the binary search time (which does not depend on the choice of the algorithm C) plus this time J; so, to minimize the total average computation time, it is sufficient to minimize this expression J.

We are describing each combination algorithm C by the function $f(x)$, so we must find the function $f(x)$ which minimizes J. The only restriction on $f(x)$ is that it should be a mapping from the entire interval $[0, 1]$ into the entire interval $[0, 1]$, i.e., that we should have $f(0) = 0$ and $f(1) = 1$. Since

$$\int_0^1 f'(x)\,\mathrm{d}x = f(1) - f(0),$$

we can represent this restriction in terms of $f'(x)$ as follows:

$$\int_0^1 f'(x)\,\mathrm{d}x = 1. \quad (3)$$

We are, therefore, minimizing (2) under the condition (3). Lagrange multiplier method enables us to reduce this conditional optimization problem to an unconditional optimization problem

$$-\int_0^1 \log_2(F(x)) \cdot \rho(x) \cdot c(x)\,\mathrm{d}x +$$

$$\lambda \cdot \int_0^1 F(x)\,\mathrm{d}x \to \min, \quad (4)$$

where λ is a constant (Lagrange multiplier) and $F(x)$ denotes $f'(x)$. Differentiating the left-hand side of (4) with respect to $F(x)$ (see, e.g., [8]), and taking into consideration that $\log_2(z) = \ln(z)/\ln(2)$, we get the equation $-\rho(x) \cdot c(x)/(\ln(2) \cdot F(x)) + \lambda = 0$, i.e., $F(x) = f'(x) = \mathrm{const} \cdot c(x) \cdot \rho(x)$. Thus, $f(x) = \mathrm{const} \cdot \int_0^x \rho(z) \cdot c(z)\,\mathrm{d}z$.

In the algorithm C, when we make the first division, we select a point x for which $f(x) = 0.5$, i.e., which divides the integral $\int_0^1 \rho(z) \cdot c(z)\,\mathrm{d}z$ into two equal halves $\int_0^x \rho(z) \cdot c(z)\,\mathrm{d}z = \int_x^1 \rho(z) \cdot c(z)\,\mathrm{d}z = 0.5$. Similarly, on each step, when we have reached an interval $[x^-, x^+]$, we select a point $x \in (x^-, x^+)$ which divides the integral $\int_{x^-}^{x^+} \rho(z) \cdot c(z)\,\mathrm{d}z$ into two equal halves. In short, the optimal algorithm is exactly algorithm C_2. The theorem is proven.

Comment. From the proof, we can extract the explicit (asymptotic) analytical expression for the computation time of the (asymptotically) optimal combination algorithm C_2.

Indeed, the integral in the binary-search expression is simply the average computation time $\langle c \rangle$ of checking for a plan of random size. We can determine the constant in the formula for $f'(x)$ by using the fact that $\int_0^1 f'(x)\,\mathrm{d}x = 1$, so this constant is equal to $1/\langle c \rangle$. Thus, the above expression for $f'(x)$ for the optimal combination algorithm C_2 takes the form $f'(x) = c_r(x) \cdot \rho(x)$, where $c_r(x) = c(x)/\langle c \rangle$ denotes the "relative" computation time of checking for plans of different normalized size x. Thus, the total average-case computation time takes the form

$$t(C_2) = \langle c \rangle \cdot \log_2\left(\frac{1}{\varepsilon}\right) -$$

$$\langle c \rangle \cdot \int_0^1 \rho(x) \cdot c_r(x) \cdot \log_2(\rho(x) \cdot c_r(x))\,\mathrm{d}x, \quad (5)$$

where we denote $\varepsilon = 1/N$, $\langle c \rangle = \int_0^1 \rho(x) \cdot c(x)\,\mathrm{d}x$, and $c_r(x) = c(x)/\langle c \rangle$.

In particular, when $c(x) = \mathrm{const}$, we get $c_r(x) = 1$, and the second term in the expression (5) turns into the *entropy* $-\int_0^1 \rho(x) \cdot \log_2(\rho(x))\,\mathrm{d}x$ of the probability distribution $\rho(x)$. It is known that the entropy can be defined as, crudely speaking, the smallest number of binary questions which we need to ask to get the exact value. We can therefore view the expression (5) as a natural generalization of the classical notion of an entropy to the case when different questions have different costs (in our case, computation time), and so, instead of simply counting the questions, we count the total cost.

Acknowledgments

This work was supported in part by NASA under cooperative agreement NCC5-209, by NSF grants No. DUE-9750858 and CDA-9522207, by United Space Alliance, grant No. NAS 9-20000 (PWO C0C67713A6), by the Future Aerospace Science and Technology Program (FAST) Center for Structural Integrity of Aerospace Systems, effort sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number F49620-95-1-0518, and by the National Security Agency under Grant No. MDA904-98-1-0561.

The authors are thankful to Vladimir Lifschitz for valuable discussions, and to James Maddox for helpful suggestions.

References

[1] Baral, C., Kreinovich, V., and Trejo, R., "Computational Complexity of Planning and Approximate Planning in Presence of Incompleteness", *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence IJCAI'99*, Stockholm, Sweden, July 31 – August 6, 1999, Vol. 2, pp. 948–953.

[2] Bylander, T., "The computational complexity of propositional STRIPS planning", *Artificial Intelligence*, 1994, Vol. 69, pp. 161–204.

[3] Cormen, T. H., Leiserson, C. E., and Rivest, R. L., *Introduction to Algorithms*, MIT Press, 1990.

[4] Dimopoulos, Y., Nebel, B., and Koehler, J., "Encoding planning problems in non-monotonic logic programs", *Proceedings of European Conference on Planning 1997 ECP-97*, Springer-Verlag, 1997, pp. 169–181.

[5] Ernst, M., Millstein, T., and Weld, D., "Automatic SAT-Compilation of planning problems", *Proc. of IJCAI 97*, 1997, pp. 1169-1176.

[6] Erol, K., Nau, D., and Subrahmanian, V. S., "Complexity, decidability and undecidability results for domain-independent planning", *Artificial Intelligence*, 1995, Vol. 76, No. 1–2, pp. 75–88.

[7] Georgeff, M. P., "Planning", *Annual Review of Computer Science*, 1987, Vol. 2, pp. 359–400.

[8] Hermann, R., *Differential geometry and the calculus of variations*, Math. Sci. Press, Brookline, MA, 1977.

[9] Kautz, H., and Selman, B., "Planning as satisfiability", *Proc. of ECAI-92*, 1992, pp. 359-363.

[10] Kautz, H., and Selman, B. "Pushing the envelop: planning, propositional logic and stochastic search", *Proc. of AAAI-96*,1996, pp. 1194-1201.

[11] Kautz, H., and Selman, B. "Unifying SAT-based and Graph-based planning", *Proc. of IJCAI-99*, 1999, pp. 318-325.

[12] Liberatore, P. *Algorithms and experiments of finding minimal models*, Technical Report 09-99, Dipartimento di Informatica e Sistemistica, Universita di Roma "La Sapienza", 1999.

[13] Lifschitz, V., "Answer set planning," *Proceedings of 1999 International Conference on Logic Programming ICLP'99*, Las Cruces, NM, 1999.

[14] Lifschitz, V., "Action languages, answer sets and planning", In: *The Logic Programming Paradigm*, Springer-Verlag, N.Y., 1999.

[15] Niemela, I., and Simons, P. "Efficient implementation of the well-founded and stable model semantics", *Proceedings of Joint International Conference and Symposium on Logic Programming*, 1996, pp. 289–303.

[16] Simons, P., *Toward constraint satisfaction through logic programs and the stable model semantics*, Technical Report 47, Helsinki University of Technology, 1997.

[17] Wilkins, D. E., "Domain independent planning: representation and plan generation", *Artificial Intelligence*, 1984, Vol. 22, pp. 269–301.