Investigations of Continual Computation

Dafna Shahaf*
Carnegie Mellon University
dshahaf@cs.cmu.edu

Eric Horvitz
Microsoft Research
horvitz@microsoft.com

Abstract

Autonomous agents that sense, reason, and act in real-world environments for extended periods often need to solve streams of incoming problems. Traditionally, effort is applied only to problems that have already arrived and have been noted. We examine *continual computation* methods that allow agents to ideally allocate time to solving current as well as potential future problems under uncertainty. We first review prior work on continual computation. Then, we present new directions and results, including the consideration of shared subtasks and multiple tasks. We present results on the computational complexity of the continual-computation problem and provide approximations for arbitrary models of computational performance. Finally, we review special formulations for addressing uncertainty about the best algorithm to apply, learning about performance, and considering costs associated with delayed use of results.

1 Introduction

Agents immersed in real-world environments over extended periods of time typically face a continuing stream of problems over their lifetimes. In competitive, time-critical situations, they cannot afford the luxury of ceasing their problem solving during periods that might typically be referred to as idle time between challenges. Continual computation ideally partitions all available computation to current as well as to future problems under uncertainty in the nature and arrival time of the future problems [Horvitz, 2001]. The methods consider how best to reason, reflect, and recall solutions to forthcoming challenges. The methods also describe how to shift computational resources from real-time challenges to problems that have not yet arrived.

We shall review prior work on continual computation (CC). Then, we focus on several extensions. We first consider how agents can handle multiple tasks and subtasks, extending the prior work on handling streams of separate, single tasks. Then, we explore continual computation for problems with shared subtasks. The earlier work introduced formulations and families of utility models describing the value achieved with computational refinement that enabled the tractable composition of optimal continual computation policies. The tractability of the approach hinges on proofs that demonstrate that greedy selections are optimal for the utility models. We show how the extensions can fit into the same paradigm for computing tractable policies. Then we relax the constraints on utility models and discuss the complexity of computing continual computation policies for general utility functions. We introduce a polynomial approximation. We move on to introduce special formulations of continual computation and their solutions. We consider situations including the case where the freshness of precomputed results is important and where the value of stored results degrades with time. We also consider methods for learning and reasoning about the performance of algorithms. For several models, we share simulations of studies of continual computation on synthetic datasets.

2 Background

Prior results on continual computation provide ideal policies for allocating time to solving current and future problems, whether the solution procedures are all-or-nothing algorithms (where results have no value unless a problem is solved completely), or flexible, anytime procedures, where partial results of increasing quality are produced with computation. The work work also describes policies that include caching and reuse of results. We shall briefly review several core ideas of continual computation but refer readers to [Horvitz, 2001] for details.

Assume that an agent can infer probabilities, $p(I_i|E)$, of seeing different problem instances $I_i \in \mathbb{I}$ in the next time period within an environment E, based on its experiences. Given probabilities of future challenges, how should an agent spend the time available between its real-time challenges?

The earlier work provides methods for both minimizing the time required for a system to solve problems and for maximizing the expected utility of computational problem solving. If we take T to be the period of time available between real-time challenges, and t_i^p as the idle time allocated to initiating or completing the solution of problem instance I_i ahead of time, then the maximum quantity of time that can be allocated solving a future problem instance is just the time needed to solve the problem completely, $t(I_i)$. Without precomputation, the delay expected for solving future potential prob-

*Dafna Shahaf performed this research during an internship at Microsoft Research.

lem instances under uncertainty is $\sum_i p(I_i|E)t(I_i)$. If the agent begins solving or completely solves one or more future problems in advance of its arrival, the expected delay in the next period is $\sum_i p(I_i|E)(t(I_i) - t_i^p)$, where $t_i^p \leq t(I_i)$ and $\sum_i t_i^p \leq T$.

Theorem 2.1 (Policy for Minimizing Computational Delay [Horvitz, 2001]). *Given an ordering over problem instances by probability $p(I_1|E) \geq p(I_2|E) \geq ... \geq p(I_n|E)$, of being passed to a problem solver in the next period, the idle-time resource policy that minimizes the expected computation time in the next period is to apply all resources to the most likely problem until it is solved, then the next most likely, and so on, until the cessation of idle time or solution of all problems under consideration in the next period.*

The greatest reduction in the expected real-time delay is obtained by the greedy algorithm of applying all resources to the most likely instance until it is solved, removing that instance from consideration and then making the same argument iteratively with the remaining $n-1$ problems and remaining idle time.

The prior work on continual computation also provides results for minimizing the expected cost of delay. We are given a time-dependent cost function for each future problem instance, $Cost(I_i, t)$, that takes as arguments that instance and the period of time required to compute each instance following a challenge (the delayed response). If we apply computation to begin solving (or solve completely) one or more potential problems, the expected delay after the arrival of a challenge is $\sum_i p(I_i|E)(Cost(I_i, t(I_i)) - Cost(I_i, t_i^p))$. Tractable strategies for specific classes of cost functions were introduced. For example, consider the case where costs increase linearly with increases in delay until a solution is available, $Cost(I_i, t) = C_i \cdot t$.

The expected value following the arrival of a challenge is maximized by allocating idle time available in advance of the challenge to commence solving the instance associated with the greatest expected rate of cost of diminishment, $p(I_i|E)C_i$, and to continue until it is solved, and then to solve the instance with the next highest expected rate, and so on, until all of the instances have been solved.

Studies of continual computation also examined policies that trade off the resources applied to continue to solve problems that have already arrived with proactive computation of problems that might arrive in the future. The idea is that an agent might wish to slow down or cease problem solving on a current problem given the expected value that might come from precomputing an answer to a problem that is expected to arrive later with some probability [Horvitz, 1999].

The prior work on continual computation includes policies for allocating time to flexible algorithms . We briefly review these policies. Assume a algorithm S that refines an initial problem instance I or previously computed *partial result*, $\pi(I)$, into a new result, terminating on a final, or *complete result*, $\phi(I)$. As reasoning systems may often be uncertain about the value associated with future computation, agents must consider a probability distribution over results achieved in return for some investment of time, , conditioned on the current partial result, and the procedure selected, resources,

$$S(\pi(I), t) \rightarrow p(\pi'(I)|\pi(I), S, t) \tag{1}$$

Studies of continual computation with flexible procedures make use of *expected value of computation* (EVC) [Horvitz *et al.*, 1989]. The expected value of computation (EVC) is the change in the net expected value of a system's behavior with the refinement of one or more results with the allocation of computational resources, t. The EVC considers the probability distribution over outcomes of computation and the costs of computation. The EVC for refining a single result is,

$$\begin{aligned} \mathrm{EVC}(\pi(I), S_i, t) \quad = \quad & \int_j p(\pi'_j(I)|\pi(I), S_i, t) u_o(\pi'_j(I)) \\ & -u_o(\pi(I)) - \mathrm{Cost}(t) \end{aligned} \tag{2}$$

where $u_o(\pi(I))$ is the value of a previously computed result $\pi(I)$.

A key concept in deriving continual computation policies for flexible procedures is the *expected value of precomputation* (EVP) and its derivative, the *EVP flux*. EVP is the EVC expected with precomputing potential future problem instances I_i with procedures S_i with time t_i^p,

$$\mathrm{EVP} = \sum_i p(I_i|E)\mathrm{EVC}(S_i, I_i, t_i^p) \tag{3}$$

where $t_i^p(I_i) \leq t(I_i)$ and $\sum_i t_i^p \leq T$

The EVP flux, $\psi(S, I, t^p)$, is the instantaneous rate at which the expected value of the future result changes at the point of allocating t^p seconds of precomputation time to solving a problem

$$\psi(S_i, I_i, t_i^p) = \frac{dEVP(S_i, I_i, t_i^p)}{dt_i^p} \tag{4}$$

Continual computation policies for sequencing precomputation effort among potential future problem instances can be composed by considering $\psi(I, t)$. The EVP of policies is computed by integrating over the EVP flux for resources allocated to each instance and summing together the EVP derived from precomputing each problem instance,

$$EVP = \sum_i \int_0^{t_i^p} \psi(I_i, t) dt \tag{5}$$

Theorems are presented in the prior work for several classes of algorithmic that specify how time should be allocated to problems based on a following of the maximal EVP flux.

We note that, although probability distributions over future problems are called out in CC methods, key policies for guiding the allocation of computation time can take as input qualitative orderings over likelihood, EVP, and EVP flux. Thus, coarser probabilistic knowledge can be used to ideally triage idle time.

3 Handling Subtasks and Multiple Tasks

The prior work considered a stream of individual, independent problems. We first consider dependencies among tasks, where portions of solutions can be cached and later shared among tasks. Then, we present policies to handle the cases where multiple tasks can arrive.

3.1 Subtasks

A key consideration is that work performed to solve one instance might be reused to solve another instance. Such a situation occurs when instances share a result computed by a common procedure.

Let $\mathbb{S} = \{S_1, ..., S_k\}$ a set of subtasks, such that all tasks are composed of them ($I_i \subseteq \mathbb{S}$). Suppose the subsets are independent – that we can solve each of the instances separately.

Proposition 3.1 (Policy for minimal expected time to completion with subtasks). *Let $p(S_j|E) = \sum_{i.S_j \in I_i} p(I_i|E)$, the probability that the next problem instance needs subtask S_j. Given an ordering over subproblem instances by probability $p(S_1|E) \geq p(S_2|E) \geq ... \geq p(S_k|E)$, the policy that minimizes the expected computation time in the next period is to apply all resources to the most likely subproblem until it is solved, then the next most likely, and so on, until the cessation of idle time or solution of all subproblems under consideration in the next period.*

When we relax the assumption of independent subtasks, the problem becomes harder. Suppose, for example, that $\mathbb{S} = \{S_1, ..., S_k\}$ has a corresponding partial precedence order, imposed by a directed-acyclic graph (DAG). In other words, a subtask cannot be executed until all the preceding subtasks have been completed.

We now consider the corresponding decision problem: given $\mathbb{S}$, and total precomputation time T, is there a (perhaps fractional) subset of $\mathbb{S}$ we can perform to gain expected time of at least K?

Theorem 3.2. *If the set of subtasks $\mathbb{S} = \{S_1, ..., S_k\}$ can have an arbitrary precedence order, the decision problem is $\mathcal{NP}$-hard.*

Proof sketch: We show a reduction from k-clique. Given a graph $G = (V, E)$, construct the following instance. Let $\mathbb{S} = V \cup E$ – i.e., there is a task for every edge and a task for every vertex. Define the precedence ordering such that an edge task can be picked only after the completion of its two corresponding vertex tasks. Let the task probabilities be $p(v) = |E|^2 + 1$, $p(e) = 1 + \epsilon$, and the total time needed be $t(v) = |E|^2 + 1$,$t(E) = 1$. Let $T = k(|E|^2 + 1) + \binom{k}{2}$, and $K = k(|E|^2 + 1) + (1 + \epsilon)\binom{k}{2}$. Since p's represent probabilities, we need to normalize all of these at the end.

If there is a k-clique in $G = (V, E)$, picking the corresponding vertices and then edges is a valid solution. If there is a valid solution, it is easy to see that it has to include at least k vertices. It cannot include $k + 1$ vertices, since the capacity of T is not large enough. Since the ratio of p to t is lower than $(1 + \epsilon)$ for every v, fractional vertices cannot lead to a valid solution as well.

A valid solution, therefore, must include $\binom{k}{2}$ edges. The precedence constraints imply that those edges form a k clique with the chosen vertices. Therefore, the graph contains a k-clique.

3.2 Multiple tasks

Prior work on continual computation focused on the case of an idle or problem solving period disrupted by the arrival of single task arriving. We can generalize this formulation to the case where multiple tasks can arrive simultaneously. Let $p(S|E)$ be the probability that subset $S \subseteq \mathbb{I}$ appears. We want to maximize the EVP.

Corollary 3.3. *We can consider sets of tasks as single tasks and the elemental components of the larger reformulated task as subtasks. Therefore, the greedy approximation as described in Proposition 3.1 applies, using marginals $p(I_i|E) = \sum_{I_i \in S} p(S|E)$ is optimal.*

Note that the greedy algorithm is not optimal for maximizing many other natural objective functions (e.g., it cannot be used to minimize the total response time, the time since the instance was posed until it was solved). Other useful properties that hold in the single-task model do not hold in this model as well, e.g. we cannot assume that future tasks arriving with the same probability are interchangeable.

We conducted experiments to evaluate the utility of continual computation with multiple tasks. We generated a synthetic dataset of tasks, where each task is assigned a probability of appearing, and a function describing how the utility of partial results increases with computation. The probabilities were modeled based on analyses done on a project centering on the application of CC policies for web page prefetching.

The probabilities and utilities for the study were obtained by crawling a subset of the web every few hours (details about the crawl can be found here [Adar *et al.*, 2009]). We take the task of refreshing a cache of webpages i as a task I_i. If a webpage has changed significantly between two consecutive snapshots (determined by Jaccard similarity), then we assert that I_i is active; intuitively, the task is to refresh a local copy of the web page. We used this data to calculate a probability of activation for every task. We assumed that each task becomes active independently of previous time steps, or other tasks. Rather than assuming that documents provide value to users only when they are downloaded completely, we allow value to be drawn from portions of documents that are immediately available, similar to the approach taken in [Horvitz, 1999].

In other words, we can prefetch *partial* content from a web page. The utility function we used was a nonlinear curve, capturing the notion of decreasing marginal returns with additional fetched content. We employed the class of utility function for all pages, but we scaled the functions for different pages to reflect the page's popularity. In other words, refreshing portions or all of more popular pages have higher utility than doing the same for less popular pages. Due to the diminishing returns property, we may prefer to start fetching less popular pages before completing the popular ones.

We simulated different test sequences of tasks being passed to the solver. We compared the value generated by a traditional reactive algorithm, that initiates problem solving only when a problem appears, versus a CC algorithm, while keeping the size of the web cache constant for all conditions.

Figure 1 shows the results of these experiments. The figure shows two lines, each representing the ratio of the utility generated across the cache for CC and for the default Reactive procedure over time. The dashed line corresponds to a study with tasks arriving with a mean arrival time that is 1.5 times faster than the case represented by the solid line, in order to see how the rate of instance arrival affects the utility ratio. CC shows improvements of 10-30% over the reactive com-

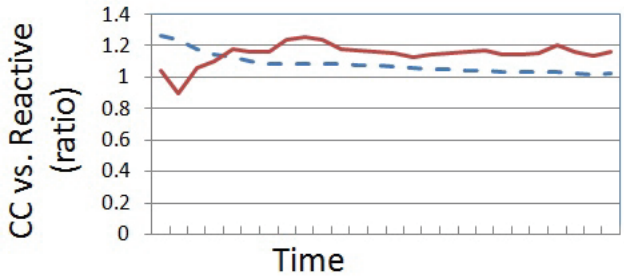

Figure 1: Ratio of the utility of CC and of traditional, reactive computation over time. The mean arrival rate for instances with utility ratio captured by the dashed line is 1.5 times faster than for the utility ratio captured by the solid line.

putation most of the time. However, CC performed worse at some points. At these times, continual computation was solving future instances rather than current ones. The investment in the precomputation proves to be useful over time. Not surprisingly, CC performs better when there are longer idle periods (solid line).

4 Hardness and Approximation for General Utility Models

The prior work on continual computation presented ideal policies for several classes of flexible procedures, as defined by the functional form of the refinement of utility of partial results with computation. These classes include algorithms that refine results with constant flux, piecewise linear flux with decreasing returns, and smoothly increasing utility with decreasing returns [Horvitz, 2001].

A natural question arises: Can we generalize those optimal policies to other classes? We now show that the most general form of the continual-computation problem (CC) is $\mathcal{NP}$-hard, and we give a constant-factor approximation algorithm for it.

4.1 $\mathcal{NP}$-hardness of CC

We start by formally defining CC. We are given $\mathbb{I} = \{I_1, ..., I_n\}$ problem instances with corresponding utility functions.

Definition 4.1 (Utility Functions). *Let $f_1(t), ..., f_n(t)$ be a set of monotonic functions. We say that f_i is the* utility function *of instance I_i if $f_i(t)$ is the utility for partial results if we spend t computation units on instance I_i.*

As we are working with computation units, we can assume they are piecewise-linear functions on $\mathbb{N} \rightarrow \mathbb{R}^+$ (the problem is essentially discrete; we cannot have a temporal grain size that is arbitrarily fine). Also, wlg $f_i(0) = 0$.

The CC problem is finding the $\max \sum_i f_i(t_i^p)$ s.t. $\sum t_i^p \leq T$. We now consider the decision variant of that problem:

Definition 4.2 (CC Decision Problem). *Given utility function f_i and integers R and T, can we find t_i^p s.t. $\sum t_i^p \leq T$ and $\sum_i f_i(t_i^p) \geq R$?*

Standard techniques (i.e., binary search) allow us to switch between the decision and the search problems.

Theorem 4.3. *CC is $\mathcal{NP}$-hard.*

Proof. CC is in $\mathcal{NP}$ (given a solution, checking it requires polynomial time). CC is also $\mathcal{NP}$-hard. We show a reduction from the knapsack problem:

In the knapsack problem, we are given n items. Each item i has a value r_i and a weight w_i. The maximum weight that we can carry in our bag is W. The problem is to find out if we can put items in the bag with a total value $\geq R$.

Given an instance of a knapsack problem, we construct the following CC instance: Let $T = R + W$. Each item i corresponds to a piecewise-linear function f_i, s.t. $f_i(t) = 0$ if $t \leq w_i$, $f_i(t) = t - w_i$ for $t \in [w_i, w_i + r_i]$, and $f_i(t) = r_i$ for $t \geq w_i + r_i$.

A solution to this problem corresponds exactly to a solution to the knapsack problem. Let $S = \{i \mid t_i^p \geq w_i\}$, the items corresponding to instances whose corresponding precomputation time was at least their weight. Without loss of generalization, we can assume that for no instance $t_i^p > w_i + r_i$ (since the utility is constant after that point). This is feasible:

$$\sum_{i \in S} w_i = \sum (t_i^p - f_i(t_i^p)) \leq T - R = W$$

$$\sum r_i \geq \sum f_i(t_i^p) \geq R$$

since $t_i^p \in [w_i, w_i + r_i], f_i(t_i^p) = t_i^p - w_i, f_i(t_i^p) \leq r_i$. Therefore, CC is $\mathcal{NP}$-hard. □

4.2 Approximations for CC

As CC is hard, we cannot hope for an optimal polynomial algorithm. Instead, we investigate an approximation algorithm for general utility functions, and show a constant-factor algorithm. We formulate the problem in terms of set cover. Let $f_1(t), ..., f_n(t)$ be monotonic piecewise-linear functions as before. For every instance I_i, we create items $x_{i,k}$, for $k = 1 ... \min(T, t(I_i))$. $x_{i,k}$ represents the kth computation unit spent on I_i. We define the *reward* of an item as $r(x_{i,k}) = f_i(k) - f_i(k-1)$, the marginal utility from spending an extra unit of I_i after having spent $k-1$ already. We then create sets $S_{i,k} = \cup_{j=1...k} x_{i,j}$. $S_{i,k}$ corresponds to spending k units on I_i. It is associated with cost k, the time it takes to compute all its members. Let $\mathbb{S}$ denote the set of all those sets. The total number of sets is polynomial in the representation size of CC.

Our problem is to find $A \subseteq \mathbb{S}$ that maximizes the total reward of covered elements

$F(A) = \sum_{x \in A^u} r(x)$ (where $A^u = \cup_{S \in A} S$)

s.t. $Cost(A) = \sum_{S \in A} Cost(S) \leq T$.

Theorem 4.4. *Let A_{CB} be the cost-benefit greedy solution, and A_{UC} be the unit-cost greedy solution (ignoring costs). Then $\max F(A_{CB}), F(A_{UC})$ is a $\frac{1}{2}(1 - \frac{1}{e})$ approximation algorithm that runs in $O(Tn)$ time.*

The proof is based on [Leskovec *et al.*, 2007]; note that $F(A)$ is submodular, and $Cost(A)$ is modular. If we are willing to pay $O(Tn^4)$, we can also get an $(1 - \frac{1}{e})$ approximation [Sviridenko, 2004]. Lazy evaluations can also speed up the algorithm.

5 Special Formulations

We now move to special formulations of continual computation that introduce elements that have not been explicitly modeled in prior studies. These formulations include considerations of hard time constraints on the generation of solutions, the use of explicit models of learning and reasoning about the performance of algorithms, and models focused on the value of updating or refreshing aging of results, where we will to such tasks as web caching and crawling and explore special analyses for such tasks.

5.1 Hard Constraints on Solution Time

Previous work considered the problem of reasoning with a procedure that provides smoothly increasing utility with decreasing returns in the context of cost that increases constantly with time (decreasing returns, constant cost scenarios). In such settings the continuous processes of refinement and cost are weighed and computation or precomputation ceases when the cost of computation outweighs the benefit of additional refinement of a result. In contrast to such smooth processes, we shall now consider the case of continual computation with *hard deadlines*.

Let us suppose that each task I_i has an associated deadline $d(I_i)$. If the task is not completed $d(I_i)$ time units after it appears, we cannot complete it, and we gain nothing from the computation. Our task is to maximize the probability of completing a task or maximizing the expected utility associated with a task being completed.

First, we note that, if $t(I_i) \leq d(I_i)$, we do not need to spend any time on I_i; if it appears, we have enough time to solve it on the spot. Without a loss of generalization, we assume $t(I_i) > d(I_i)$ for all I_i. If this is not true, we do not spend any precomuptation on the other tasks. If the inequality relationship is true, it only makes sense to precompute I_i if we spend exactly $d(I_i) - t(I_i)$ units of precomputation time.

This is exactly the knapsack problem. We have a knapsack with capacity T, and each task I_i corresponds to an item of size $d(I_i) - t(I_i)$ and value $p(I_i|E)$.

We can also optimize the total expected value via approximate greedy or semi-greedy algorithms in an attempt to maximize the overall expected value. For example, with a *value-density* approximation, we prioritize partial results in an order dictated by the ratio of their value and cost. The value-density approach is used in an approximation procedure to come within a factor of two of the maximal value storage policy [Sahni, 1975]. Other very efficient FPTAS exist for the knapsack problem; [Lawler, 1977] achieves a running time of $O(n \log \frac{1}{\epsilon} + \frac{1}{\epsilon^4})$ for a $1 + \epsilon$ approximation.

To explore the performance of a knapsack approximation for a real-time CC procedure, we generated synthetic datasets of tasks with hard deadlines and tested the quality of solutions when the algorithm runs for limited amounts of time. Like the experiment described earlier, we base the arrival rates for instances on webpage change rates. The deadlines and the times needed to solve each instance were chosen uniformly at random for each dataset, while making sure that $t(I_i) > d(I_i)$.

We used the algorithm of [Botev and Kroese, 2008] and manipulated the number of iterations. Figure 2 shows the results. Each point represents the ratio of the value of the approximated and the optimal solution—as computed via dynamic programming. Following 10 iterations, the approximation algorithm was achieving very good results. After as little as 20 iterations, the results were near-optimal. The approach took only a fraction of a second to execute, running orders of magnitude faster than the dynamic-programming solution.

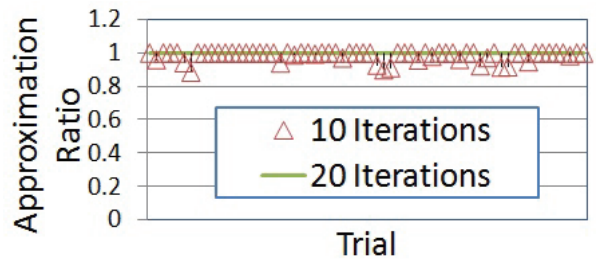

Figure 2: Approximation ratio of knapsack approximation for hard deadline model. Each point represents the ratio for a different trial, with a new random choice of deadlines and completion times of tasks. The triangles represent the approximate solution ratio after 10 iterations, and the solid line after 20. The approximation recovered a near-optimal solution for all datasets after twenty iterations.

5.2 Result Freshness and Time of Use

Most of the formulations of continual computation to date allow for an arbitrary delay before a precomputed solution is used to solve a real-time challenge. Continual computation policies can also be developed that provide special constraints and costs on the timing of usage of a precomputed result. For example, a precomputed result might have to be immediately harnessed by an agent so as to draw any value from the precomputation, lest the result is rendered worthless. If I_i arrives at time t, the reward obtained is $f_i(t_i^p(t))$. That is, the reward is only available for an immediate use and reflects the amount of precomputation that has been performed when the task arrives. In a related setting, the utility of a precomputed solution can diminish with the amount of time that passes before it is used. This situation can be viewed as being forced to pay a rental fee for storing precomputed results in memory. Thus, it is best to produce results so that they are ready near the time when they are needed, rather than producing them far in advance and paying for their storage.

As an example of a real-world scenario of continual computation with these characteristics, consider the case where computer users access web pages via a proxy server. The server has cached copies of web pages. However, page i can change with probability c_i at every step. Although we do not know when a page changes, we can fetch a new copy of it, guaranteeing that it will be up to date in the next step. In addition, some websites are more popular than others; the expected number of requests arriving for a page every step is r_i. If a reward is received for responding to user requests with an updated page, what is the ideal fetching policy? We wish to maximize

$$\sum_t \sum_i r_i \cdot \mathbb{1}(\textit{page } i \textit{ is fresh at time } t)$$

The optimal CC policy is periodic, since the (discrete) number of states is finite. A natural extension is to seek an op-

timal stochastic-periodic policy. We refer the reader to [Shahaf *et al.*, 2009] for the derivation and studies of the operation of an algorithm for finding an optimal stochastic periodic policy that maximizes the expected number of requests fulfilled in polynomial time. The algorithm refomulates the problem as a convex optimization problem.

5.3 Probabilistic Performance Model

Many real-world problems involve uncertainty about the outcome of actions. Such uncertainty can apply to problem solving. Let us assume that the result of spending one unit of computation on I_i is not completely under a system's control. In addition, when the system is targeted at running indefinitely in an environment, we may want to minimize the *discounted* reward.

We assume known action models, full observability (i.e., we know the result of our past actions), and the Markov property: given the state of the system, transition probabilities to the next state are independent of all previous states or actions. Markov decision processes (MDPs) provide a useful mathematical framework for modeling this type of situation. We briefly review the MDP framework, and then show how to model a CC problem as an MDP.

An MDP is defined as a 4-tuple (X, A, R, P) where X is a finite set of N states, A is a finite set of actions, $R : X \times A \rightarrow \mathbb{R}$ is a reward function , such that $R(x, a)$ represents the reward obtained by the agent in state x after taking action a. P is a Markovian transition model where $P(x'|x, a)$ represents the probability of going from state x to state x' with action a.

Let us assume that problem instances can appear at any time along a timeline. At every time step, we can either spend computation on an instance or return a solution to an active instance. The reward we receive is based on the amount of refinement of that instance. In addition, we have a storage limit on the amount of refinement that can be stored, M.

This can easily be modeled as an MDP: X is the state of the system (how many computation units have been applied to each instance I_i, which instances are currently active). We have three types of actions: *Add*$_i$ (spend the next time-step on computation for I_i), *Free*$_i$ (free from memory the last time-step on computation you did for I_i), and *Solve*$_i$ (submit the best current refinement of I_i). The rewards are $-\infty$ for any illegal action (trying to add something when the memory is full, freeing unit i when there is none is memory, or solving a problem that is not active). The reward for solving an active task I_i is based on its utility function f_i and the number of computation units have been applied to refine I_i in X.

There exist efficient algorithms for MDPs that enable us to compute an optimal update policy. However, the MDP above has an exponential number of states, and thus is untractable for even small ns. Fortunately, this MDP has significant internal structure, and can be modeled compactly if this structure is exploited in the representation.

Factored MDPs [Boutilier *et al.*, 2000] is one approach to representing large, structured MDPs compactly. In this framework, a state is implicitly described by an assignment to some set of state variables. A dynamic Bayesian network (DBN) can then allow a compact representation of the transition model, by exploiting the fact that the transition of a variable often depends only on a small number of other variables. Furthermore, the momentary rewards can often also be decomposed as a sum of rewards related to individual variables or small clusters of variables.

We represent CC as a factored MDP. Our set of states is defined via a set of random variables $\{X_1, ..., X_n, Y_1, ..., Y_n, K\}$ where X_i denotes the number of computation units we have cached for task I_i, Y_i binary variables denoting whether there is an active instance of I_i passed to the solver, and K is $\sum_i X_i$, the number of cache units used.

Actions and rewards are the same as in the MDP version. The transitions for the deterministic case are straightforward: *Add*$_i$ causes K and X_i to increase (if possible), *Free*$_i$ causes them to decrease. *Solve*$_i$ resets Y_i. Any action causes Y_j to switch from 0 to 1 with probability $p(I_j|E)$, and never the other way around. In the case of non-deterministic transitions, the formulation is slightly different:X_i denotes the current point of I_i's utility function, and the possible outcomes are modeled in the transition model of *Add*$_i$.

In general, the factored MDP framework is very flexible. Many other plausible scenarios can be modeled – e.g., the case of immediate response (with or without cost of delay), or the case of non-reusable computation (where *Solve*$_i$ resets Y_i and X_i). We used the techniques of [Guestrin *et al.*, 2003] to efficiently solve MDPs for a large set of artificial tasks.

5.4 Learning about Performance

So far, we have assumed that agents have some knowledge about the performance of algorithms. We now focus on methods that can support continual computation in agents by endowing agents with the ability to learn about performance over time. Several classes of learning procedures can be used to update distributions over future streams of instances and the performance of algorithms on each instance. On the latter, predictive models of performance can be conditioned on attributes of the instances as well as observations provided by the solution procedure as prior and/or the current instance is solved [Horvitz *et al.*, 2001]. We focus here on CC methods that manage the *exploration-exploitation* challenge. We would like our agents to balance exploration in pursuit of new data on performance with exploitation that applies the learned policy to refine results.

Assume we have a set of tasks $\mathbb{I}$ which might be passed to the solver in the next instance. In addition, we have $\mathbb{A}$, a set of solver procedures we can apply. For example, in the case of prefetching webpages, a procedure may be the name of the server to use. Each task I_i is associated with a subset of $\mathbb{A}$ that can be applied to it. Each algorithm has its utility function f_A.

Theorem 5.1 (Optimal Policy for Multiple Algorithms with Known Utility Functions). *If the reward profiles of applying every algorithm $A \in \mathbb{A}$ are known in advance, the greedy policy is optimal.*

What happens if the algorithms in $\mathbb{A}$ are *probabilistic* procedures with a linear utility function of unknown slope? As before, we assume full observability: every time we apply an algorithm to an instance, we know the outcome. Choosing an algorithm provides some information about its utility function, but it is partial.

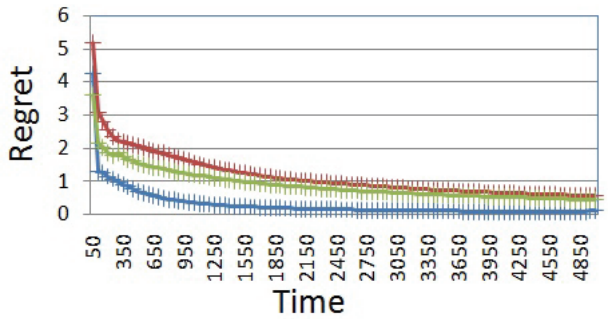

Figure 3: Regret over time with learning. Each line represents a different choice of algorithms (modeled as a noisy process with a random mean), and their association to tasks. The regret decreases as the gambler learns more.

This problem can be solved using techniques from the multi-armed bandit problem. This problem is based on an analogy with a traditional slot machine (one-armed bandit) but with more than one lever. When pulled, each lever provides a reward drawn from a distribution associated with that specific lever. The gambler wants to maximize the collected reward sum through iterative pulls. Again, the gambler faces a tradeoff at each trial between exploitation of the lever that has the highest expected payoff and exploration to get more information about the expected payoffs of the other levers.

In the case at hand, the algorithms correspond to levers. At every turn we choose I_i with highest probability that is not solved yet, and then concentrate on the subset of algorithms that can solve it, and apply bandits techniques until the instance is solved. We can use knowledge gained from previous iterations when solving the next instance.

A typical goal in bandit settings is to minimize the *regret*. The regret after T rounds is defined as the difference between the average reward associated with an optimal strategy and the average of the collected rewards. Applying a simple weighted-majority algorithm, we can achieve regret of order $O(\sqrt{n \log n/T})$.

In order to evaluate the bandit framework for CC we generated a synthetic dataset. We considered a set of 500 tasks and 20 algorithms. The tasks are again modeled after the web-page fetching scenario. Each task was associated with a randomly selected subset of algorithms. Each algorithm was modeled as a noisy process with a different mean, chosen uniformly at random. We used the EXP3 algorithm [Auer *et al.*, 1995] in the evaluation. Figure 3 shows the regret over time for three different sets of randomly selected algorithms. As expected, the regret decreases as the gambler learns more. Examining the results revealed that the reduction of regret comes with the diminishing of the empirical variance of the algorithms' performance.

6 Summary

We reviewed key aspects of prior work on continual computation and then provided new analyses that extend the earlier work. The analyses provide agents which face streams of challenges with new capabilities for optimizing how they allocate their time to current and future problems. The methods extend deliberation to the handling of multiple and interrelated tasks. We provide complexity results on continual computation for general utility models and then offer polynomial approximation procedures. Finally, we described methods for handling special case situations, including models of the uncertainty in the performance of algorithms and hard time constraints on computing problem solutions. We believe that continual computational procedures can endow agents immersed in the open world with principles of intelligence for allocating their computational resources. We hope these extensions and analyses will be harnessed to increase the value of computational efforts in a variety of real-world environments.

References

[Adar *et al.*, 2009] E. Adar, J. Teevan, and S. T. Dumais. Resonance on the web: web dynamics and revisitation patterns. In Dan R. Olsen Jr., Richard B. Arthur, Ken Hinckley, Meredith Ringel Morris, Scott E. Hudson, and Saul Greenberg, editors, *CHI*, pages 1381–1390. ACM, 2009.

[Auer *et al.*, 1995] P. Auer, N. Cesa-bianchi, Y. Freund, and R. E. Schapire. Gambling in a rigged casino: The adversarial multi-armed bandit problem. In *FoCS 95*, 1995.

[Botev and Kroese, 2008] Z. I. Botev and D. P. Kroese. An efficient algorithm for rare-event probability estimation, combinatorial optimization, and counting. *Methodology and Computing in Applied Probability*, 2008.

[Boutilier *et al.*, 2000] C. Boutilier, R. Dearden, and M. Goldszmidt. Stochastic dynamic programming with factored representations. *Artificial Intelligence*, 2000.

[Guestrin *et al.*, 2003] Carlos Guestrin, Ronald Parr, and Shobha Venkataraman. Efficient solution algorithms for factored MDPs. *JAIR*, 19, 2003.

[Horvitz *et al.*, 1989] E.J. Horvitz, G.F. Cooper, and D.E. Heckerman. Reflection and action under scarce resources: Theoretical principles and empirical study. In *IJCAI 89*, 1989.

[Horvitz *et al.*, 2001] E. Horvitz, Y. Ruan, C. Gomes, H. Kautz, B. Selman, and M. Chickering. A Bayesian approach to tackling hard computational problems. In *UAI*, pages 235–244, 2001.

[Horvitz, 1999] E.J. Horvitz. Thinking ahead: Continual computation policies for allocating offline and real-time resources. In *IJCAI 99*, 1999.

[Horvitz, 2001] E. Horvitz. Principles and applications of continual computation. *Artificial Intelligence*, 126:159–196, 2001.

[Lawler, 1977] E. L. Lawler. Fast approximation algorithms for knapsack problems. *FoCS 77*, 0, 1977.

[Leskovec *et al.*, 2007] J. Leskovec, A. Krause, C. Guestrin, C. Faloutsos, J. Vanbriesen, and N. Glance. Cost-effective outbreak detection in networks. In *KDD '07*, 2007.

[Sahni, 1975] S. Sahni. Approximate algorithms for the 0/1 knapsack problme. *J. ACM*, 1975.

[Shahaf *et al.*, 2009] D. Shahaf, Y. Azar, E. Lubetzky, and E. Horvitz. Utility-directed policies for continual caching and crawling. Technical report, 2009.

[Sviridenko, 2004] M. Sviridenko. A note on maximizing a submodular set function subject to knapsack constraint. *Operations Research Letters*, 2004.