Chapter summaries

This thesis is organized into five chapters. Chapter 1, "Online Algorithms for Maximizing Submodular Functions", develops algorithms for solving a very general online resource allocation problem. These algorithms form the basis for many of the experimental and theoretical results in chapters 2 and 3. Chapter 2, "Combining Multiple Heuristics Online", presents techniques for combining multiple problem-solving algorithms into an improved algorithm by interleaving the execution of the algorithms and, if the algorithms are randomized, periodically restarting them with a fresh random seed. An important feature of the work presented in this chapter is that a schedule for interleaving and restarting the algorithms can be learned online while solving a sequence of problems.

The remaining three chapters present techniques that can be used to improve the performance of specific classes of optimization algorithms. Chapter 3, "Fine-Grained Algorithm Portfolios", presents online algorithms for improving the performance of algorithms that decompose a problem into a number of subproblems. Chapter 4, "Using Decision Procedures Efficiently for Optimization", presents techniques for improving the performance of algorithms that solve an optimization problem by making a sequence of calls to an algorithm for the corresponding decision problem. Chapter 5, "The Max k-Armed Bandit Problem", studies a variant of the classical multi-armed bandit problem in which the goal is to maximize the *maximum* payoff received, rather than the sum of the payoffs. Algorithms for solving the max k-armed bandit problem can be used to improve the performance of *multi-start* heuristics, which obtain a solution to an optimization problem by performing a number of independent runs of a randomized heuristic and returning the best solution obtained.

The results in this thesis are based on five conference papers [23, 24, 25, 26, 27] and a conference submission that is currently under review [11].

Online algorithms for maximizing submodular functions

In this chapter we develop algorithms for solving a very general online resource allocation problem, which can be described formally as follows. We are given as input a set $\mathcal{V}$ of activities. A pair $(v, \tau) \in \mathcal{V} \times \mathbb{Z}_{>0}$ is called an *action*, and specifies that time τ is to be invested in activity v. A *schedule* is a sequence of actions. We denote by $\mathcal{S}$ the set of all schedules. A *job* is a function $f : \mathcal{S} \to [0, 1]$, where for any $S \in \mathcal{S}$, $f(S)$ equals the proportion of some task that is accomplished after performing the sequence of actions S. We require that a job f satisfy the following conditions (here $\oplus$ is the concatenation operator):

1. (monotonicity) for any $S_1, S_2 \in \mathcal{S}$, we have $f(S_1) \leq f(S_1 \oplus S_2)$ and $f(S_2) \leq f(S_1 \oplus S_2)$.
2. (submodularity) for any $S_1, S_2 \in \mathcal{S}$ and any $a \in \mathcal{A}$,

$$f(S_1 \oplus S_2 \oplus \langle a \rangle) - f(S_1 \oplus S_2) \leq f(S_1 \oplus \langle a \rangle) - f(S_1) \ .$$

We will evaluate schedules in terms of two objectives. The first objective is to minimize

$$c\left(f, S\right) = \int_{t=0}^{\infty} 1 - f\left(S_{\langle t \rangle}\right) dt \tag{1}$$

where $S_{\langle t \rangle}$ is the truncation of S at time t. For example if $S = \langle (h_1, 3), (h_2, 3) \rangle$ then $S_{\langle 5 \rangle} = \langle (h_1, 3), (h_2, 2) \rangle$. The second objective is to maximize $f\left(S_{\langle T \rangle}\right)$ for some fixed $T > 0$.

In the online version of the problem, an arbitrary sequence $\langle f_1, f_2, \ldots, f_n \rangle$ of jobs arrive one at a time, and we must finish each job (via some schedule) before moving on to the next. In this setting our goal is to develop schedule-selection strategies that minimize *regret*, which equals the difference between the average performance of the schedules selected by our online algorithm and that of the best fixed schedule (in hindsight) for the given sequence of jobs.

Under each of the two objectives just defined, the problem introduced in this chapter generalizes a number of previously-studied problems. The problem of minimizing $c\left(f, S\right)$ generalizes MIN-SUM SET COVER [8], PIPELINED SET COVER [15, 19], the problem of constructing efficient sequences of trials [5], the problem of constructing task-switching schedules [22, 23], and the problem of constructing restart schedules [12, 18, 24]. The problem of maximizing $f(S_{\langle T \rangle})$ for some fixed $T > 0$ generalizes of the problem of maximizing a monotone submodular set function subject to a knapsack constraint [17, 28], which in turns generalizes BUDGETED MAXIMUM COVERAGE [16] and MAX k-COVERAGE [20]. Prior to our work, many of these problems had only been considered in an offline setting. For the problems that had been considered in an online setting, the online algorithms presented in this chapter provide new and stronger guarantees.

We now give a concrete example of the problem considered in this chapter.

Example 1. Let each $v \in \mathcal{V}$ be a deterministic algorithm for solving some decision problem, and let an action $a = (v, \tau)$ represent running algorithm v with a time limit of τ. (Note that each action represents an independent run, and no work is shared across actions.) Given a set of n instances of the decision problem, one can represent the i^{th}

instance as a job f_i as follows: let $f_i(S) = 1$ if some action in S, when performed on the i^{th} instance, yields a solution to the problem; and let $f_i(S) = 0$ otherwise. It is easy to check that f_i satisfies the monotonicity and submodularity conditions required of a job. By construction, $c\,(f_i, S)$ equals the time required to solve the i^{th} problem instance using schedule S. Because a convex combination of jobs is also a job, the overall task of solving the n instances can also be represented as a single job $f = \frac{1}{n}\sum_{i=1}^{n} f_i$. Then $c\,(f, S)$ is the average time required to solve an instance using S, and $f(S_{\langle T\rangle})$ is the fraction of instances that are solved in time $\leq T$.

Example 1 can be generalized to allow for the use of randomized algorithms. In this case, $f_i(S)$ becomes the *probability* that performing the actions in S yields a solution to the i^{th} problem instance, and $c\,(f_i, S)$ becomes the *expected* time required to solve i^{th} problem instance using schedule S.

In some cases of practical interest, f will not satisfy the submodularity condition. As it turns out, all of our results still hold so long as f satisfies the following (provably) weaker property: for any $S_1, S \in \mathcal{S}$ and any $t > 0$,

$$\frac{f(S_1 \oplus S_{\langle t\rangle}) - f(S_1)}{t} \leq \max_{a=(v,\tau)\in\mathcal{A}} \left\{ \frac{f(S_1 \oplus \langle a\rangle) - f(S_1)}{\tau} \right\} . \tag{2}$$

This fact allows us to address a richer class of online problems. Returning to Example 1, suppose that we allow separate runs of each $v \in \mathcal{V}$ to be kept in memory simultaneously, and let an action $a = (v, \tau)$ represent investing *additional* time τ in the run of algorithm v. Although f_i now violates the submodularity condition, it can be shown that it still satisfies (2).

We now summarize the main technical contributions of this chapter. We first considers the problem of computing an optimal schedule in an offline setting, given black-box access to the job f. As immediate corollaries of existing results [7, 8], we obtain that for any $\epsilon > 0$, (*i*) achieving an approximation ratio of $4 - \epsilon$ for the problem of minimizing $c\,(f, S)$ is **NP**-hard and (*ii*) achieving an approximation ratio of $1 - \frac{1}{e} + \epsilon$ for the problem of maximizing $f(S_{\langle T\rangle})$ is **NP**-hard. Building on and generalizing previous work [8, 28], we then present an offline greedy approximation algorithm that simultaneously achieves the optimal approximation ratios (of 4 and $1 - \frac{1}{e}$, respectively) for each of these two problems.

We then consider the online setting. Assuming $\mathbf{P} \neq \mathbf{NP}$, the computational complexity of the offline optimization problems implies that we cannot expect to minimize regret directly using any online algorithm that makes decisions in polynomial time. Accordingly we seek to minimize α-regret (where $\alpha = 4$ or $\alpha = 1 - \frac{1}{e}$, as appropriate), which is defined in the same way as regret except that the performance of the optimal schedule in hindsight is multiplied by α. We give an online version of the greedy approximation algorithm whose worst-case α-regret approaches zero as the number of jobs approaches infinity. In other words, the performance of the online algorithm is guaranteed to converge to that of the offline greedy approximation algorithm.

Our online algorithms can be used in several different feedback settings. We first consider the feedback setting in which, after using schedule S_i to complete job f_i, we receive black-box access to f_i. We then consider more limited feedback settings in which: (*i*) to receive access to f_i we must pay a price C, which is added to the regret, (*ii*) we only observe $f_i\left(S_{i\langle t\rangle}\right)$ for each $t \geq 0$, and (*iii*) we only observe $f_i\left(S_i\right)$. These limited feedback settings arise naturally in the applications discussed in chapters 2 and 3.

Combining multiple heuristics online

Many important computational problems are NP-hard and thus seem unlikely to admit algorithms with provably good worst-case performance, yet must be solved as a matter of practical necessity. For many of these problems, heuristics have been developed that perform much better in practice than a worst-case analysis would guarantee. Nevertheless, the behavior of a heuristic on a previously unseen problem instance can be difficult to predict in advance. The running time of a heuristic may vary by orders of magnitude across seemingly similar problem instances or, if the heuristic is randomized, across multiple runs on a single instance that use different random seeds [14, 13]. For this reason, after running a heuristic unsuccessfully for some time one might decide to suspend the execution of that heuristic and start running a different heuristic (or the same heuristic with a different random seed).

In this chapter we consider the problem of allocating CPU time to various heuristics so to minimize the time required to solve one or more instances of a decision problem. We consider the problem of selecting an appropriate schedule in three settings: *offline*, *learning-theoretic*, and *online*. The results in this chapter significantly generalize and extend previous work on *algorithm portfolios* [13, 14, 21, 22, 29] and *restart schedules* [18, 10, 12].

The problem considered in this chapter can be described formally as follows. We are given as input a set $\mathcal{H}$ of (randomized) algorithms for solving some decision problem. Given a problem instance as input, each $h \in \mathcal{H}$ is capable of returning a provably correct "yes" or "no" answer to the problem, however the time required for a given $h \in \mathcal{H}$ to return an answer depends both on the problem instance and on the random seed (and may be infinite). For the purposes of discussion, we will imagine that we have access to an infinite number of (equivalent) copies of each heuristic h, each with a different random seed. We use $h^{(i)}$ to denote the i^{th} copy of heuristic h. We will solve each problem instance by interleaving the execution of the heuristics according to some schedule. Consistent with the framework of chapter 1, we consider schedules of the form

$$S = \langle (h_1^{(c_1)}, \tau_1), (h_2^{(c_2)}, \tau_2), \ldots \rangle$$

where each pair $(h_i^{(c_i)}, \tau_i)$ specifies that *additional* time τ_i is to be invested in copy c_i of heuristic h_i.

This class of schedules is very general, and includes both *task-switching schedules* [22] and *restart schedules* [18] as special cases. A *task-switching schedule* is a schedule that only makes use of a single copy of each heuristic (i.e., $c_i = 1$ for all i). If all heuristics in $\mathcal{H}$ are deterministic, then the optimal schedule is a task-switching schedule. A *restart schedule* is a schedule for a single randomized heuristic (i.e., $|\mathcal{H}| = 1$) that never runs the same copy twice (i.e., all c_i are distinct).[1]

Table 1: Behavior of two solvers on instances from the 2007 SAT competition.

Instance	`Rsat` **CPU (s)**	`picosat` **CPU (s)**
industrial/anbulagan/medium-sat/dated-10-13-s.cnf	45	28
industrial/babic/dspam/dspam_dump_vc1081.cnf	3	≥ 10000
industrial/grieu/vmpc_31.cnf	≥ 10000	238

To appreciate the power of task-switching schedules, consider Table 1, which shows the behavior of the top two solvers from the industrial track of the 2007 SAT competition on three of the competition benchmarks. On these benchmarks, interleaving the execution of the solvers according to an appropriate schedule can dramatically improve average-case running time. Indeed, in this case simply running the two solvers in parallel (e.g., at equal strength on a single processor) would reduce the average-case running time by orders of magnitude.

To appreciate the power of restart schedules, consider Figure 1, which depicts the run length distribution of the SAT solver satz-rand on a Boolean formula derived from a

[1]A restart schedule can be written more concisely as a sequence $S = \langle \tau_1, \tau_2, \ldots \rangle$ of positive integers, whose meaning is "run h for τ_1 time units; if this does not yield a solution then restart h with a fresh sequence of random bits and run it for τ_2 time units, …", where h is the single heuristic in $\mathcal{H}$.

logistics planning benchmark. When run on this formula, satz-rand exhibits a heavy-tailed run length distribution. There is about a 20% chance of solving the problem after running for 2 seconds, but also a 20% chance that a run will not terminate after having run for 1000 seconds. Restarting the solver every 2 seconds reduces the mean running time by more than an order of magnitude.

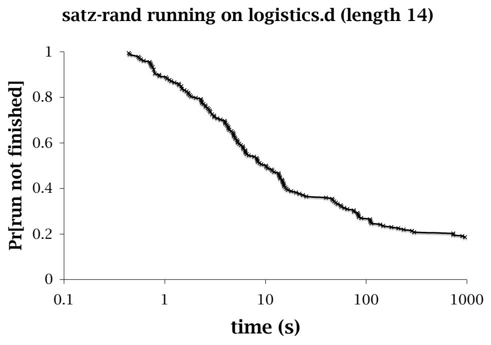

Figure 1: Run length distribution of satz-rand on a formulae derived from a logistics planning benchmark.

We now summarize the main technical contributions of this chapter. As already mentioned, this chapter considers the schedule-selection problem in three settings: *offline*, *learning-theoretic*, and *online*.

In the offline setting we are given as input the run length distribution of each $h \in \mathcal{H}$ for each problem instance in a set of instances, and wish to compute a schedule with minimum average (expected) running time over the instances in the set. In this setting we show that the greedy algorithm from chapter 1 gives a 4-approximation to the optimal schedule and that, for any $\epsilon > 0$, computing an $4 - \epsilon$ approximation is NP-hard. We also give exact and approximation algorithms based on shortest paths that are able to compute an α-approximation to the optimal schedule for any $\alpha \geq 1$, but whose running time is exponential as a function of $|\mathcal{H}|$.

In the learning-theoretic setting, we draw training instances from a fixed distribution, compute an (approximately) optimal schedule for the training instances, and then use that schedule to solve additional test instances drawn from the same distribution. In this setting, we give bounds on the number of training instances required to learn a schedule that is *probably approximately correct.*

In the online setting we are fed a sequence of problem instances one at a time and must obtain a solution to each instance before moving on to the next. In this setting we show that the online greedy algorithm from chapter 1 converges to a 4-approximation to the best fixed schedule for the instance sequence, and requires decision-making time polynomial in $|\mathcal{H}|$. We also present online shortest paths algorithms that, for any $\alpha \geq 1$, can be guaranteed to converge to an α-approximation to the best fixed schedule, but these online algorithms require decision-making time exponential in $|\mathcal{H}|$.

This chapter concludes with an extensive experimental evaluation of the techniques developed for each of these three problem settings. The main results of our experimental evaluation can be summarized as follows.

1. Using data from recent solver competitions, we show that schedules computed by our algorithms can be used improve the performance of state-of-the-art solvers in several problem domains, including Boolean satisfiability, A. I. planning, constraint satisfaction, and theorem proving.

2. We use our offline algorithms to construct a restart schedule for the SAT solver satz-rand that improves its performance on an ensemble of problem instances derived from logistics planning benchmarks.

3. We show how our algorithms can be applied to optimization problems (as opposed to decision problems), and demonstrate that they can be used to improve the performance of state-of-the-art algorithms for pseudo-Boolean optimization.

4. We demonstrate that additional performance improvements can be obtained by using instance-specific features to tailor the choice of schedule to a particular problem instance.

Fine-grained algorithm portfolios

(To be completed.)

Using decision procedures efficiently for optimization

Optimization problems are often solved by making repeated calls to a decision procedure that answers questions of the form “Does there exist a solution with cost at most k?”. Each query to the decision procedure can be represented as a pair $\langle k, t \rangle$, where t is a bound on the CPU time the decision procedure may consume in answering the question. The result of a query is either a (provably correct) “yes” or “no” answer or a timeout. A *query strategy* is a rule for determining the next query $\langle k, t \rangle$ as a function of the responses to previous queries.

One optimization algorithm of this form is SatPlan, a state-of-the-art algorithm for classical planning. SatPlan finds a minimum-length plan by making a series of calls to a SAT solver, where each call determines whether there exists a feasible plan of makespan $\leq k$ (where the value of k varies across calls). The original version of SatPlan uses the *ramp-up* query strategy, which simply executes the queries $\langle 1, \infty \rangle, \langle 2, \infty \rangle, \langle 3, \infty \rangle, \ldots$ in sequence (stopping as soon as a "yes" answer is obtained). Figure 2 shows the CPU time required by the query $\langle k, \infty \rangle$ as a function of k, on a particular planning benchmark instance.

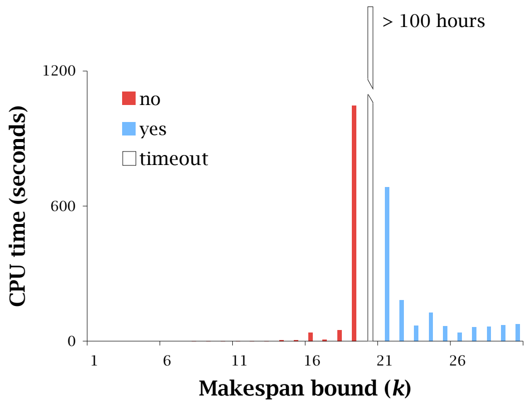

Figure 2: Behavior of the SAT solver `siege` running on formulae generated by SatPlan to solve instance `p17` from the `pathways` domain of the 2006 International Planning Competition.

The motivation for the work in this chapter is that the choice of query strategy often has a dramatic effect on the time required to obtain a (provably) approximately optimal solution. Figure 2 illustrates this fact: on this instance, using the ramp-up query strategy requires one to invest over 100 hours of CPU time before obtaining a feasible plan. On the other hand, executing the queries $\langle 18, \infty \rangle$ and $\langle 23, \infty \rangle$ takes less than two minutes and yields a plan whose makespan is provably at most $\frac{23}{18+1} \approx 1.21$ times optimal.

This chapter presents both a theoretical and an experimental study of query strategies. We consider the problem of devising query strategies in two settings. In the *single-instance* setting, we are confronted with a single optimization problem, and wish to obtain an (approximately) optimal solution as quickly as possible. In the *multiple-instance* setting, we use the same decision procedure to solve a number of optimization problems, and our goal is to learn from experience in order to improve performance.

In the single-instance setting, we are interested in minimizing the CPU time required before the ratio of the upper bound (i.e., the smallest k for which some query $\langle k, t\rangle$ elicited a "yes" answer) to the lower bound (i.e., one plus the largest k for which some query $\langle k, t\rangle$ elicited a "no" answer) is at most α, for some fixed $\alpha \geq 1$. The competitive ratio of a query strategy on a particular problem instance is the ratio of the CPU time it requires to that required by the optimal query strategy for the instance. We analyze query strategies in terms of their competitive ratio on the worst-case instance.

Let $\tau(k)$ denote the CPU time required by the decision procedure when run on input k. The performance of our query strategies will depend on the behavior of the function τ. Let OPT denote the minimum cost of any solution. For most decision procedures used in practice, we expect $\tau(k)$ to be an increasing function for $k \leq \mathsf{OPT}$ and a decreasing function for $k \geq \mathsf{OPT}$ (e.g., see Figure 2), and our query strategies are designed to take advantage of this behavior. More specifically, our query strategies are designed to work well when τ is close to its *hull*, which is the function

$$\mathrm{hull}(k) = \min\left\{\max_{k_0 \leq k} \tau(k_0), \max_{k_1 \geq k} \tau(k_1)\right\} .$$

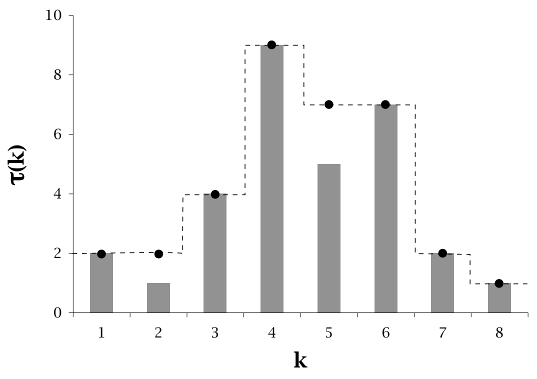

Figure 3: A function τ (gray bars) and its hull (dots).

Figure 3 gives an example of a function τ (gray bars) and its hull (dots). Note that τ and hull are identical if τ is monotonically increasing (or monotonically decreasing), or if there exists a K such that τ is monotonically increasing for $k \leq K$ and monotonically decreasing for $k > K$.

We measure the discrepancy between τ and its hull in terms of the quantity

$$\Delta = \max_k \frac{\mathrm{hull}(k)}{\tau(k)}$$

which we refer to as the *stretch* of τ. The instance depicted in Figure 3 has a stretch of 2 because $\tau(2) = 1$ while $\mathrm{hull}(2) = 2$.

In the single-instance setting, our main result is a query strategy S_2 whose worst-case competitive ratio is $O(\Delta \log U)$, where U is the difference between the initial upper and lower bounds on OPT. S_2 makes use of a form of guessing-and-doubling in combination with a two-sided binary search. We prove a matching lower bound, showing that any other query strategy has a competitive ratio $\Omega(\Delta \log U)$ on some instance. We also show that, in the absence of any assumptions about Δ, a trivial query strategy S_1 based on guessing-and-doubling obtains a worst-case competitive ratio that is $O(U)$, and we prove a matching $\Omega(U)$ lower bound.

In the multiple-instance setting, we prove that computing an optimal query strategy is NP-hard, and discuss how algorithms from machine learning theory can be used to select query strategies online.

In the experimental section of this chapter, we use the query strategy S_2 to create a modified version of SatPlan that finds (provably) approximately optimal plans more quickly than the original version of SatPlan. We also create a modified version of a branch and bound algorithm for job shop scheduling, which yields improved upper and lower bounds relative to the original algorithm. In the course of the latter experiments we develop a simple method for applying query strategies to branch and bound algorithms, which seems likely to be useful in other domains besides job shop scheduling.

The max k-armed bandit problem

The max k-armed bandit problem can be described as follows. Imagine that you find yourself in the following unusual casino. The casino contains k slot machines. Each machine has an arm that, when pulled, yields a payoff drawn from a fixed (but unknown) distribution. You are given n tokens to use in playing the machines, and you may decide how to spend these tokens adaptively based on the payoffs you receive from playing the various machines. The catch is that, when you leave the casino, you only get to keep the *maximum* of the payoffs you received on any individual pull. The max k-armed bandit problem differs from the well-studied classical k-armed bandit problem in that one seeks to optimize the maximum payoff received, rather than the sum of the payoffs.

Our motivation for studying this problem is to boost the performance of *multi-start* heuristics, which obtain a solution to an optimization problem by performing a number of independent runs of a randomized heuristic and returning the best solution obtained. Despite their simplicity, multi-start heuristics are used widely in practice, and represent the state of the art in a number of domains [1, 3, 9]. A max k-armed bandit strategy can be used to distribute trials among different multi-start heuristics or among different parameter settings for the same multi-start heuristic. Previous work has demonstrated the effectiveness of such an approach on the RCPSP/max, a difficult real-world scheduling problem [2, 4, 26].

In this chapter our goal is to develop strategies for the max k-armed bandit problem that minimize *regret*, which we define to be the difference between the (expected) maximum payoff our strategy receives and that of the best pure strategy, where a pure strategy is one that plays the same arm every time. It is not difficult to see regret-minimization is hopeless in the absence of any assumptions about the payoff distributions. As a simple example, imagine that all payoffs are either 0 or 1, that $k-1$ of the arms always return a payoff of 0, and that one randomly-selected "good" arm returns a payoff of 1 with probability $\frac{1}{n}$. In this case, one cannot obtain an expected maximum payoff larger than $\frac{1}{k}$ after n pulls, whereas the pure strategy that invests all n pulls on the "good" arm obtains expected maximum payoff $1-(1-\frac{1}{n})^n \approx 1-\frac{1}{e}$.

We present two strategies for solving the max k-armed bandit problem. The first strategy, Threshold Ascent, is designed to work well when the payoff distributions have certain characteristics which we expect to be present in cases of practical interest. Roughly speaking, Threshold Ascent will work best when the following two criteria are satisfied.

1. There is a (relatively low) threshold $t_{critical}$ such that, for all $t > t_{critical}$, the arm that is most likely to yield a payoff $> t$ is the same as the arm most likely to yield a payoff $> t_{critical}$. Call this arm i^*.

2. As t increases beyond $t_{critical}$, there is a growing gap between the probability that arm i^* yields a payoff $> t$ and the corresponding probability for other arms. Specifically, if we let $p_i(t)$ denote the probability that the i^{th} arm returns a payoff $> t$, the ratio $\frac{p_{i^*}(t)}{p_i(t)}$ should increase as a function of t for $t > t_{critical}$, for any $i \neq i^*$.

Figure 4 illustrates a set of two payoff distributions that satisfy these assumptions.

The idea of Threshold Ascent is very simple. Threshold Ascent attempts to maximize the number of payoffs $> T$ that it receives, where T is gradually increased over time. For any fixed T, this goal is accomplished by mapping payoffs $> T$ to 1 and mapping payoffs $\leq T$ to zero, then treating the problem as an instance of the classical k-armed

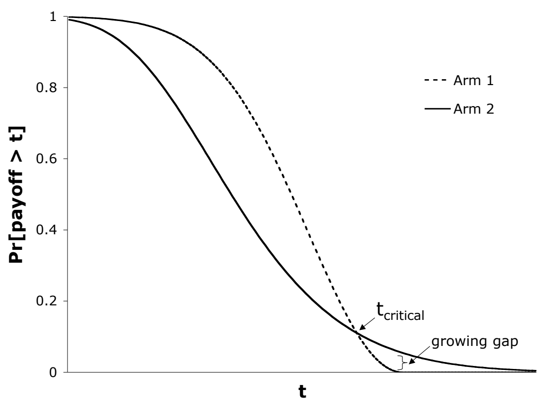

Figure 4: A max k-armed bandit instance on which Threshold Ascent should perform well.

bandit problem (where the goal is to maximize the sum of the payoffs received). As T increases, non-zero payoffs become increasingly rare, and thus we would like to have an algorithm for solving the classical k-armed bandit problem that works well when the mean payoff of each arm is very small. Toward this end, we design and analyze a new algorithm for the classical k-armed bandit problem called Chernoff Interval Estimation, which yields improved regret bounds when each arm has a small mean payoff.

In the experimental section of this chapter, we demonstrate the effectiveness of Threshold Ascent by using it to select among multi-start heuristics for the RCPSP/max, a difficult real-world scheduling problem. We find that Threshold Ascent (a) performs better than any of the multi-start heuristics performs in isolation, and (b) outperforms the recent QD-BEACON max k-armed bandit algorithm of Cicirello and Smith [2, 4].

Following the lead of Cicirello and Smith [2, 4], we also consider the special case where each payoff distribution is a generalized extreme value (GEV) distribution. The motivation for studying this special case is the Extremal Types Theorem [6], which singles out the GEV as the limiting distribution of the maximum of a large number of independent identically distributed (i.i.d.) random variables. Roughly speaking, one can think of the Extremal Types Theorem as an analogue of the Central Limit Theorem. Just as the Central Limit Theorem states that the sum of a large number of i.i.d. random variables converges in distribution to a Gaussian, the Extremal Types Theorem states that the maximum of a large number of i.i.d. random variables converges in distribution to a GEV. We provide a no-regret strategy for this special case, improving upon earlier theoretical work by Cicirello & Smith [2, 4].

References

[1] J. L. Bresina. Heuristic-biased stochastic sampling. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 271–278, 1996.

[2] Vincent A. Cicirello and Stephen F. Smith. Heuristic selection for stochastic search optimization: Modeling solution quality by extreme value theory. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming*, pages 197–211, 2004.

[3] Vincent A. Cicirello and Stephen F. Smith. Enhancing stochastic search performance by value-biased randomization of heuristics. *Journal of Heuristics*, 11(1):5–34, 2005.

[4] Vincent A. Cicirello and Stephen F. Smith. The max k-armed bandit: A new model of exploration applied to search heuristic selection. In *Proceedings of the Twentieth National Conference on Artificial Intelligence*, pages 1355–1361, 2005.

[5] Edith Cohen, Amos Fiat, and Haim Kaplan. Efficient sequences of trials. In *SODA '03: Proceedings of the* 14^{th} *annual ACM-SIAM Symposium on Discrete Algorithms*, pages 737–746, 2003.

[6] Stuart Coles. *An Introduction to Statistical Modeling of Extreme Values*. Springer-Verlag, London, 2001.

[7] Uriel Feige. A threshold of ln n for approximating set cover. *Journal of the ACM*, 45(4):634–652, 1998.

[8] Uriel Feige, László Lovász, and Prasad Tetali. Approximating min sum set cover. *Algorithmica*, 40(4):219–234, 2004.

[9] Thomas A. Feo and Mauricio G. C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6(2):109–133, 1995.

[10] Matteo Gagliolo and Jürgen Schmidhuber. Learning restart strategies. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07)*, pages 792–797, 2007.

[11] Daniel Golovin and Matthew Streeter. Online algorithms for maximizing submodular set functions. Manuscript, 2007.

[12] Carla Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, pages 431–437, 1998.

[13] Carla P. Gomes and Bart Selman. Algorithm portfolios. *Artificial Intelligence*, 126:43–62, 2001.

[14] Bernardo A. Huberman, Rajan M. Lukose, and Tad Hogg. An economics approach to hard computational problems. *Science*, 275:51–54, 1997.

[15] Haim Kaplan, Eyal Kushilevitz, and Yishay Mansour. Learning with attribute costs. In *STOC '05: Proceedings of the* 37^{th} *annual ACM symposium on Theory of Computing*, pages 356–365, 2005.

[16] Samir Khuller, Anna Moss, and Joseph (Seffi) Naor. The budgeted maximum coverage problem. *Information Processing Letters*, 70(1):39–45, 1999.

[17] Andreas Krause and Carlos Guestrin. A note on the budgeted maximization of submodular functions. Technical Report CMU-CALD-05-103, Carnegie Mellon University, 2005.

[18] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, 47:173–180, 1993.

[19] Kamesh Munagala, Shivnath Babu, Rajeev Motwani, Jennifer Widom, and Eiter Thomas. The pipelined set cover problem. In *Proceedings of the International Conference on Database Theory*, pages 83–98, 2005.

[20] G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher. An analysis of approximations for maximizing submodular set functions. *Mathematical Programming*, 14(1):265–294, 1978.

[21] Marek Petrik and Shlomo Zilberstein. Learning parallel portfolios of algorithms. *Annals of Mathematics and Artificial Intelligence*, 48(1-2):85–106, 2006.

[22] Tzur Sayag, Shai Fine, and Yishay Mansour. Combining multiple heuristics. In *Proceedings of the* 23^{rd} *International Symposium on Theoretical Aspects of Computer Science*, pages 242–253, 2006.

[23] Matthew Streeter, Daniel Golovin, and Stephen F. Smith. Combining multiple heuristics online. In *Proceedings of the Twenty-Second Conference on Artificial Intelligence (AAAI-07)*, pages 1197–1203, 2007.

[24] Matthew Streeter, Daniel Golovin, and Stephen F. Smith. Restart schedules for ensembles of problem instances. In *Proceedings of the Twenty-Second Conference on Artificial Intelligence (AAAI-07)*, pages 1204–1210, 2007.

[25] Matthew Streeter and Stephen F. Smith. An asymptotically optimal algorithm for the max k-armed bandit problem. In *Proceedings of the Twenty-First National Conference on Artificial lntelligence*, pages 135–142, 2006.

[26] Matthew Streeter and Stephen F. Smith. A simple distribution-free approach to the max k-armed bandit problem. In *Proceedings of the Twelfth International Conference on Principles and Practice of Constraint Programming*, 2006.

[27] Matthew Streeter and Stephen F. Smith. Using decision procedures efficiently for optimization. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling*, 2007.

[28] Maxim Sviridenko. A note on maximizing a submodular set function subject to a knapsack constraint. *Operations Research Letters*, 32:41–43, 2004.

[29] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla07: The design and analysis of an algorithm portfolio for SAT. In *Proceedings of CP 2007*, 2007.