Combining Multiple Heuristics in an Adversarial Online Setting

CMU theory lunch 2/14/07

Daniel Golovin Stephen F. Smith Matthew Streeter

Why heuristics?

- Many interesting problems are NP-hard, sometimes even to approximate
- Heuristics can be very effective in practice
  - SAT solvers handle formulae with 10^6 variables, used for hardware and software verification
  - CPLEX used widely in industry to solve integer programs
- Much interest in improving performance of heuristics (e.g., SAT conference holds annual competitions)

Pitfalls

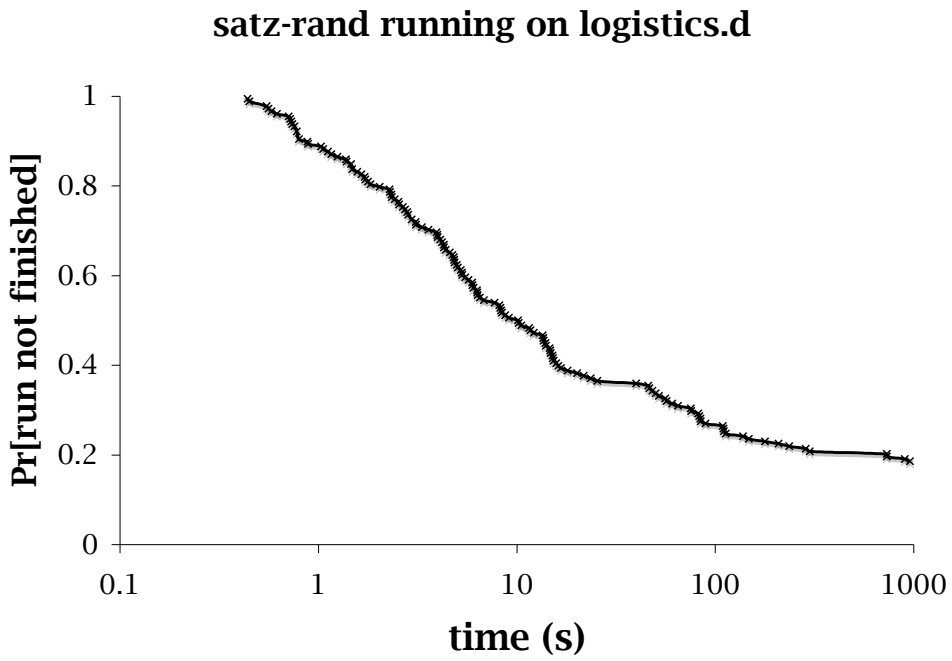
- Behavior of a heuristic on a particular instance is hard to predict

Instance	`SatELiteGTI` CPU (s)	`MiniSat` CPU (s)
liveness-unsat-2-01dlx_c_bp_u_f_liveness	33	15
vliw-sat-2-0/9dlx_vliw_at_b_iq6_bug4	376	$\geq$ 120000
vliw-sat-2-0/9dlx_vliw_at_b_iq6_bug9	$\geq$ 120000	131

- Might do better on average by running several heuristics in parallel

Pitfalls

- Running time of a randomized heuristic can vary widely across different random seeds



- Randomized SAT solvers can exhibit heavy-tailed run length distributions (Gomes et al. 1998)

Previous work

- *Algorithm portfolios* (Huberman et al. 1997, Gomes et al. 2001, ...)
- Assign each heuristic a fixed proportion of CPU time, plus a fixed restart threshold
- Assumed each heuristic has a known run length distribution that does not vary across instances

ties perpendicular to the magnetic field. The frequency of the lower hybrid waves is between the gyrofrequencies of the electrons (ω_{ce}) and the ions (ω_{ci}) which means that these waves can be in simultaneous Cherenkov resonance with the relatively slow but unmagnetized ions perpendicular to the magnetic field and fast magnetized (hence magnetic field aligned) electrons. Cherenkov resonance occurs when the phase velocity of the wave and the particle velocity are equal; under these conditions strong interaction between the waves and particles is possible and results in energy transfer from the wave to the particle or vice versa. The lower hybrid waves provide the intermediary step in transferring energy between the ions and electrons.

5. M. J. Mumma *et al.*, *Science* **272**, 1310 (1996).
6. D. Krankowsky *et al.*, *Nature* **321**, 326 (1986).
7. H. S. Hudson, W.-H. Ip, D. A. Mendis, *Planet. Space Sci.* **29**, 1373 (1981).
8. J. B. McBride, E. Ott, P. B. Jay, J. H. Orens, *Phys. Fluids* **157**, 2367 (1972). A two stream instability results when two charged particle populations traveling in opposite directions interact.
9. M. K. Wallis and R. S. B. Ong, *Planet. Space Sci.* **23**, 713 (1975). A more accurate calculation based on the analysis of the solar wind dynamics, mass-loaded by the picked-up cometary ions lead to the same formula for the ion density.
10. D. A. Mendis, H. L. F. Houpis, M. L. Marconi, *Physics of Comets Fundamentals of Cosmic Physics* (1985), vol. 10.
11. L. D. Landau, *J. Phys. USSR* **10**, 25 (1946); F. F. Chen, *Introduction to Plasma Physics and Controlled Fusion* (Plenum, New York, 1984), vol. 1, p. 240.
12. V. D. Shapiro and V. I. Shevchenko, *Sov. Sci. Rev. E, Astrophys. Space Phys.* **6**, 425 (1988).
13. D. F. Post, R. V. Jensen, C. B. Tarter, W. H. Grasberger, W. A. Lokke, *Princeton Plasma Physics Laboratory Report PPPL-1352* (1977).
14. J. M. Dawson, in *Fusion*, E. Teller, Ed. (Academic Press, New York, 1981), p. 465.
15. J. W. Chamberlain, *Physics of the Aurora and Airglow* (Academic Press, New York, 1961).
16. This work was supported in part by NSF grant PH-9319198;003 and NASA NAGW-1502.

21 June 1996; accepted 17 October 1996

An Economics Approach to Hard Computational Problems

Bernardo A. Huberman, Rajan M. Lukose, Tad Hogg

A general method for combining existing algorithms into new programs that are unequivocally preferable to any of the component algorithms is presented. This method, based on notions of risk in economics, offers a computational portfolio design procedure that can be used for a wide range of problems. Tested by solving a canonical NP-complete problem, the method can be used for problems ranging from the combinatorics of DNA sequencing to the completion of tasks in environments with resource contention, such as the World Wide Web.

Extremely hard computational problems are pervasive in fields ranging from molecular biology to physics and operations research. Examples include determining the most probable arrangement of cloned fragments of a DNA sequence (*1*), the global minima of complicated energy functions in physical and chemical systems (*2*), and the shortest path visiting a given set of cities (*3*), to name a few. Because of the combinatorics involved, their solution times grow exponentially with the size of the problem (a basic trait of the so-called NP-complete problems), making it impossible to solve very large instances in reasonable times (*4*).

In response to this difficulty, a number of efficient heuristic algorithms have been developed. These algorithms, although not always guaranteed to produce a good solution or to finish in a reasonable time, often provide satisfactory answers fairly quickly. In practice, their performance varies greatly from one problem instance to another. In many cases, the heuristics involve randomized algorithms (*5*), giving rise to performance variability even across repeated trials on a single problem instance.

In addition to combinatorial search problems, there are many other computational situations where performance varies from one trial to another. For example, programs operating in large distributed systems or interacting with the physical world can have unpredictable performance because of changes in their environment. A familiar example is the action of retrieving a particular page on the World Wide Web. In this case, the usual network congestion leads to a variability in the time required to retrieve the page, raising the dilemma of whether to restart the process or wait.

In all of these cases, the unpredictable variation in performance can be characterized by a distribution describing the probability of obtaining each possible performance value. The mean or expected values of these distributions are usually used as an overall measure of quality (*6*–*9*). We point out, however, that expected performance is not the only relevant measure of the quality of an algorithm. The variance of a performance distribution also affects the quality of an algorithm because it determines how likely it is that a particular run's performance will deviate from the expected one. This variance implies that there is an inherent risk associated with the use of such an algorithm, a risk that, in analogy with the economic literature, we will identify with the standard deviation of its performance distribution (*10*).

Risk is an important additional characteristic of algorithms because one may be willing to settle for a lower average performance in exchange for increased certainty in obtaining a reasonable answer. This situation is often encountered in economics when trying to maximize a utility that has an associated risk. It is usually dealt with by constructing mixed strategies that have desired risk and performance (*11*). In analogy with this approach, we here present a widely applicable method for constructing "portfolios" that combine different programs in such a way that a whole range of performance and risk characteristics become available. Significantly, some of these portfolios are unequivocally preferable to any of the individual component algorithms running alone. We verify these results experimentally on graph-coloring, a canonical NP-complete problem, and by constructing a restart strategy for access to pages on the Web.

To illustrate this method, consider a simple portfolio of two Las Vegas algorithms, which, by definition, always produce a correct solution to a problem but with a distribution of solution times (*5*). Let t_1 and t_2 denote the random variables, which have distributions of solution times $p_1(t)$ and $p_2(t)$. For simplicity, we focus on the case of discrete distributions, although our method applies to continuous distributions as well. The portfolio is constructed simply by letting both algorithms run concurrently but independently on a serial computer. Let f_1 denote the fraction of clock cycles allocated to algorithm 1 and $f_2 = 1 - f_1$ be the fraction allocated to the other. As soon as one of the algorithms finds a solution, the run terminates. Thus, the solution time t is a random variable related to those of the individual algorithms by

$$t = \min\left(\frac{t_1}{f_1}, \frac{t_2}{f_2}\right) \qquad (1)$$

The resulting portfolio algorithm is characterized by the probability distribution $p(t)$ that it finishes in a particular time t. This probability is given by the probability that both constituent algorithms finish in time $\geq t$ minus the probability that both algorithms finish in time $> t$

$$p(t) = \left[\sum_{t' \geq f_1 t} p_1(t')\right]\left[\sum_{t' \geq f_2 t} p_2(t')\right] - \left[\sum_{t' > f_1 t} p_1(t')\right]\left[\sum_{t' > f_2 t} p_2(t')\right] \qquad (2)$$

Dynamics of Computation Group, Xerox Palo Alto Research Center, Palo Alto, CA 94304, USA.

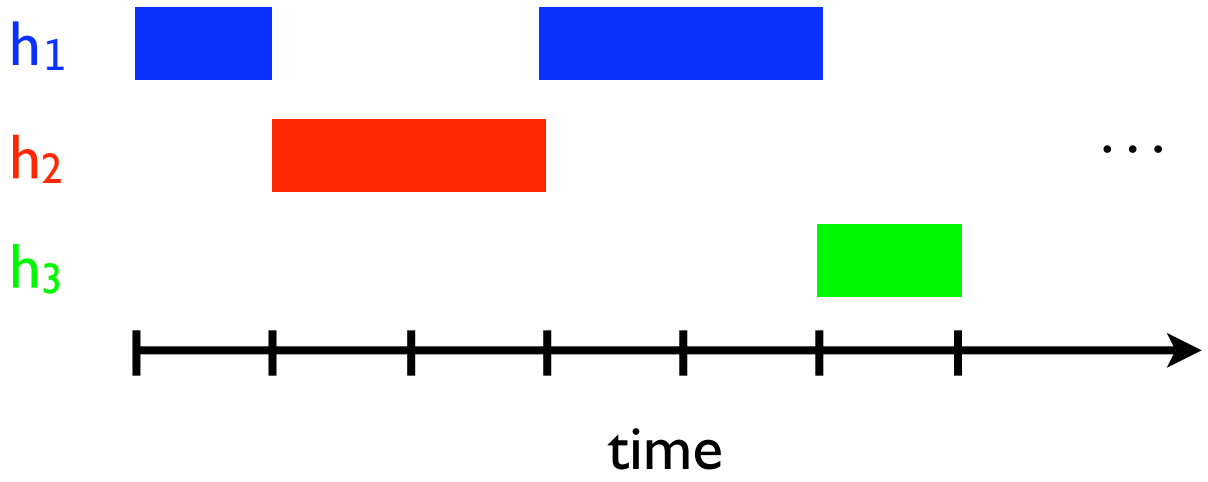
Previous work

- "Combining Multiple Heuristics" (Sayag, Fine & Mansour, STACS 2006)
  - considered *resource-sharing schedules* and *task-switching schedules*
  - gave offline algorithms + sample complexity bounds
  - algorithms are exponential in #heuristics

This talk: formal setup

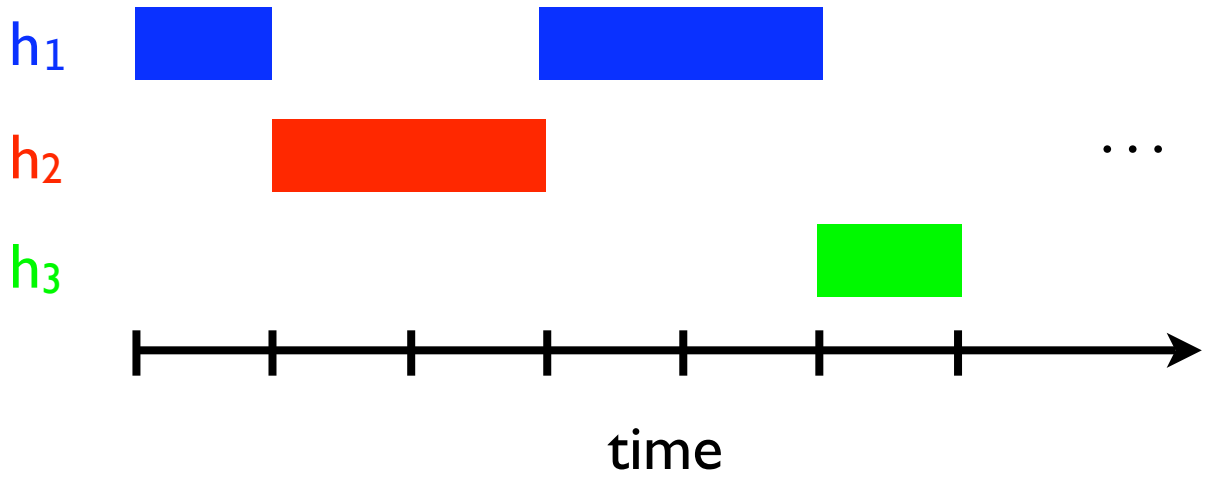
- Given set $H=\{h_1,h_2,..., h_k\}$ of heuristics (for now assume deterministic)
- Fed sequence of n decision problems to solve
- On i^{th} instance, h_j takes time $\tau_{ij} \in \{1, 2, ..., B\} \cup \{\infty\}$
- Assume for each i, $\min_j \tau_{ij} < \infty$
- Solve each problem by interleaving execution of heuristics, stopping as soon as one of them returns an answer

Task-switching schedules



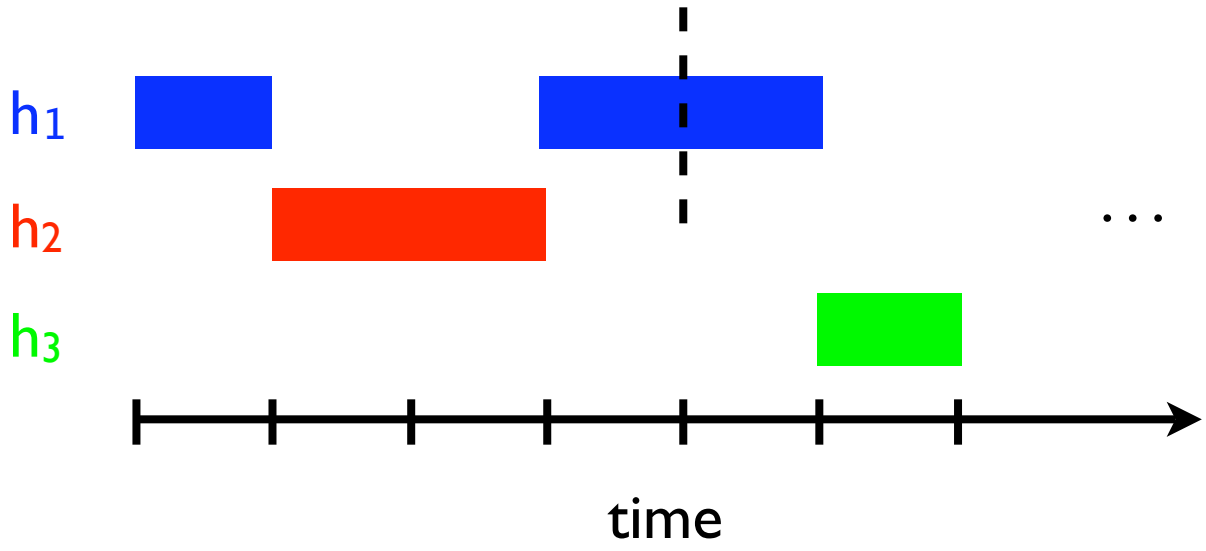
Task-switching schedules

- Mapping $S: \mathbb{Z} \mapsto H$ from time slices to heuristics; S(t) = heuristic to run from time t to time t+1
- Example:



Task-switching schedules

- Mapping S:$\mathbb{Z} \mapsto$H from time slices to heuristics; S(t) = heuristic to run from time t to time t+1
- Example:

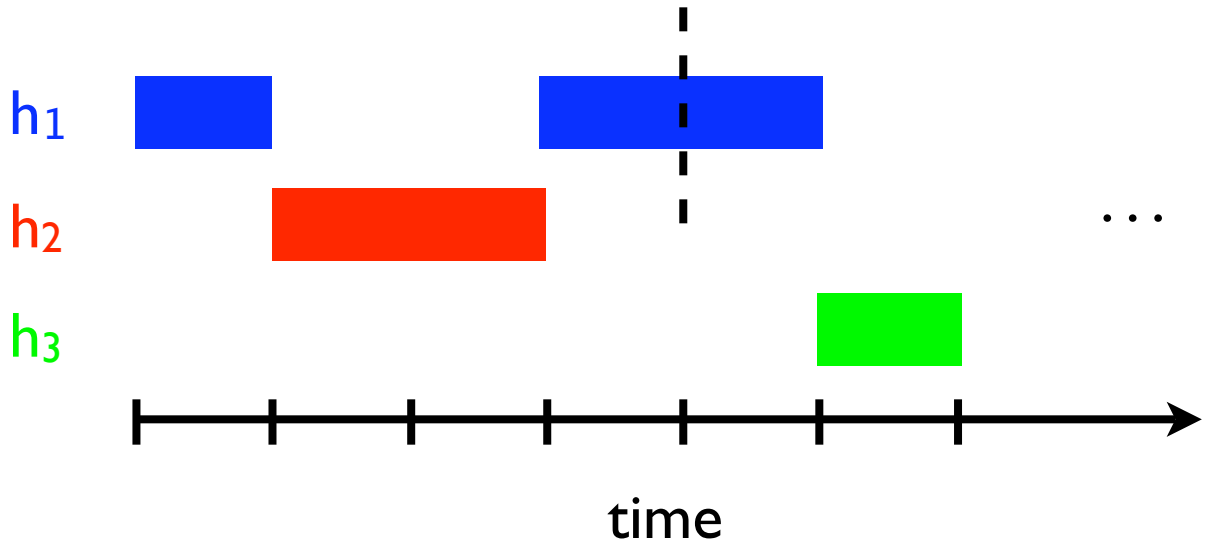


Completion times

	h_1	h_2	h_3
I_1	2	7	7

Task-switching schedules

- Mapping $S: \mathbb{Z} \mapsto H$ from time slices to heuristics; S(t) = heuristic to run from time t to time t+1
- Example:



Completion times

	h_1	h_2	h_3
I_1	2	7	7

- Note: this assumes we can keep multiple heuristics in memory and switch between them at zero cost (will come back to this later)

Outline

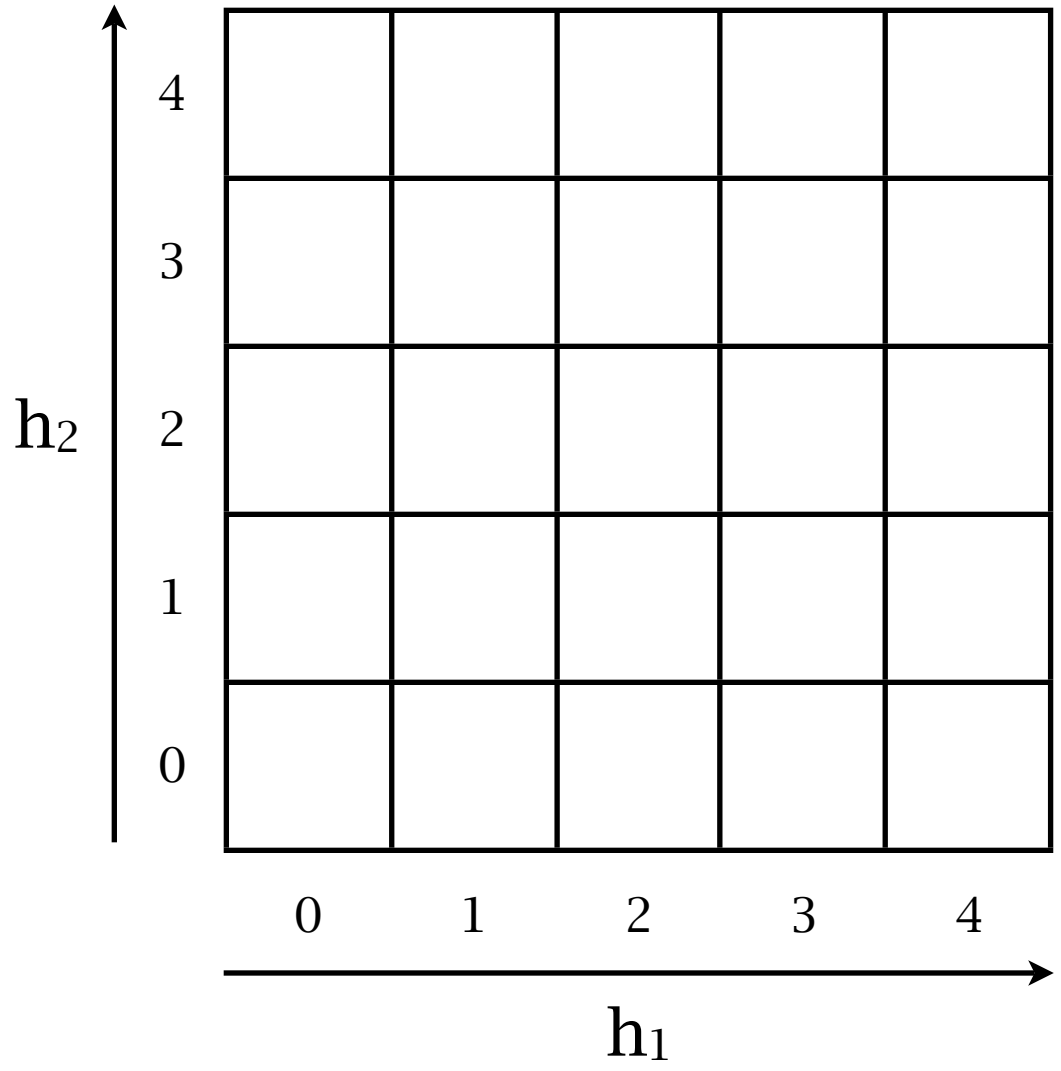
- Offline algorithms:
  - Exact algorithm based on shortest paths (Theorem 12 of Sayag et al. 2006)
  - Hardness of approximation
  - Greedy approximation algorithm
- Online algorithms
- Generalization to restart schedules
- Experiments

The offline problem

- Offline problem: given table τ of completion times, compute task-switching schedule that minimizes sum of CPU time over all instances

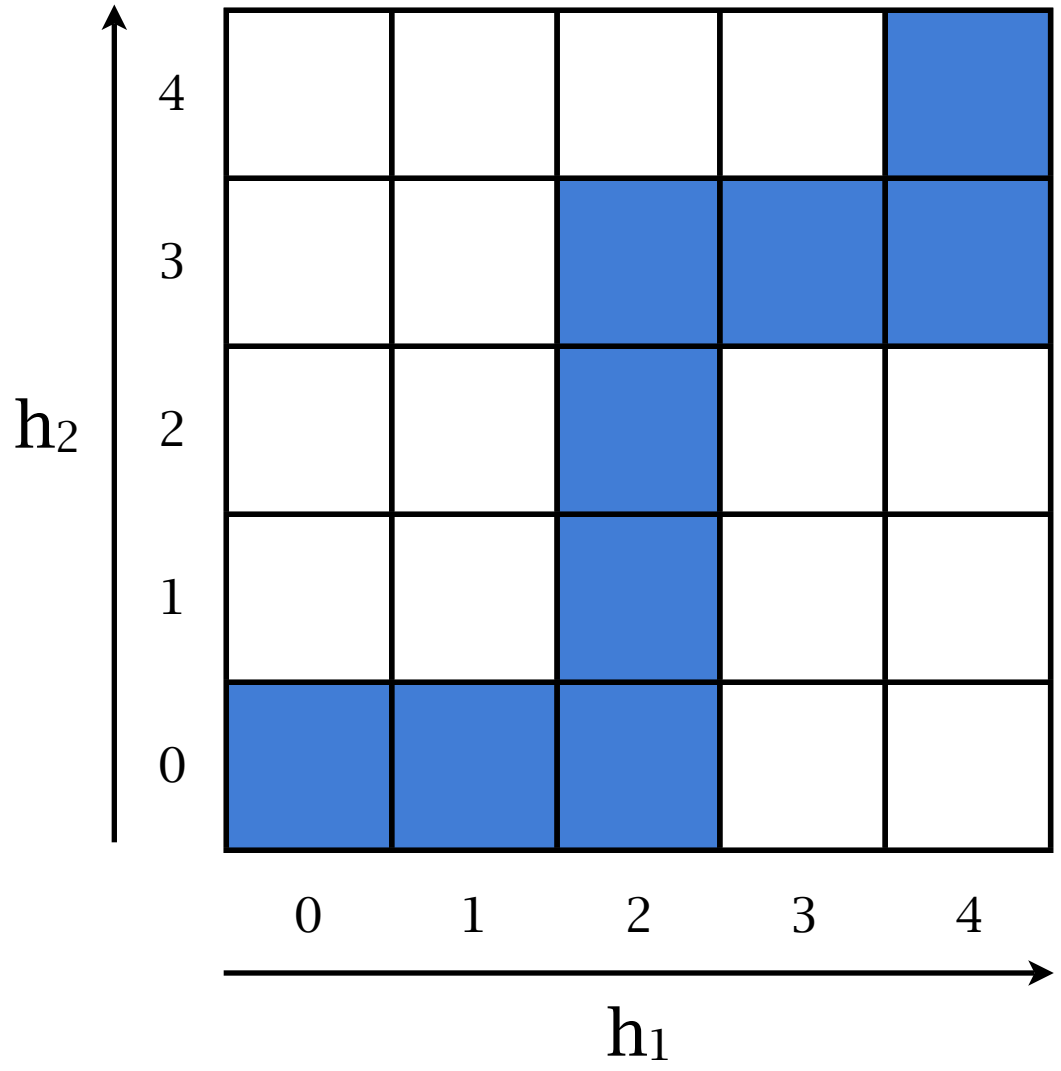
Solving the offline problem

- Can think of a task-switching schedule as a path in a k-dimensional grid with sides of length B+1 (here B=4)



Solving the offline problem

- Can think of a task-switching schedule as a path in a k-dimensional grid with sides of length B+1 (here B=4)
- E.g. "run h_1 for 2 seconds, then run h_2 for 3 seconds..."

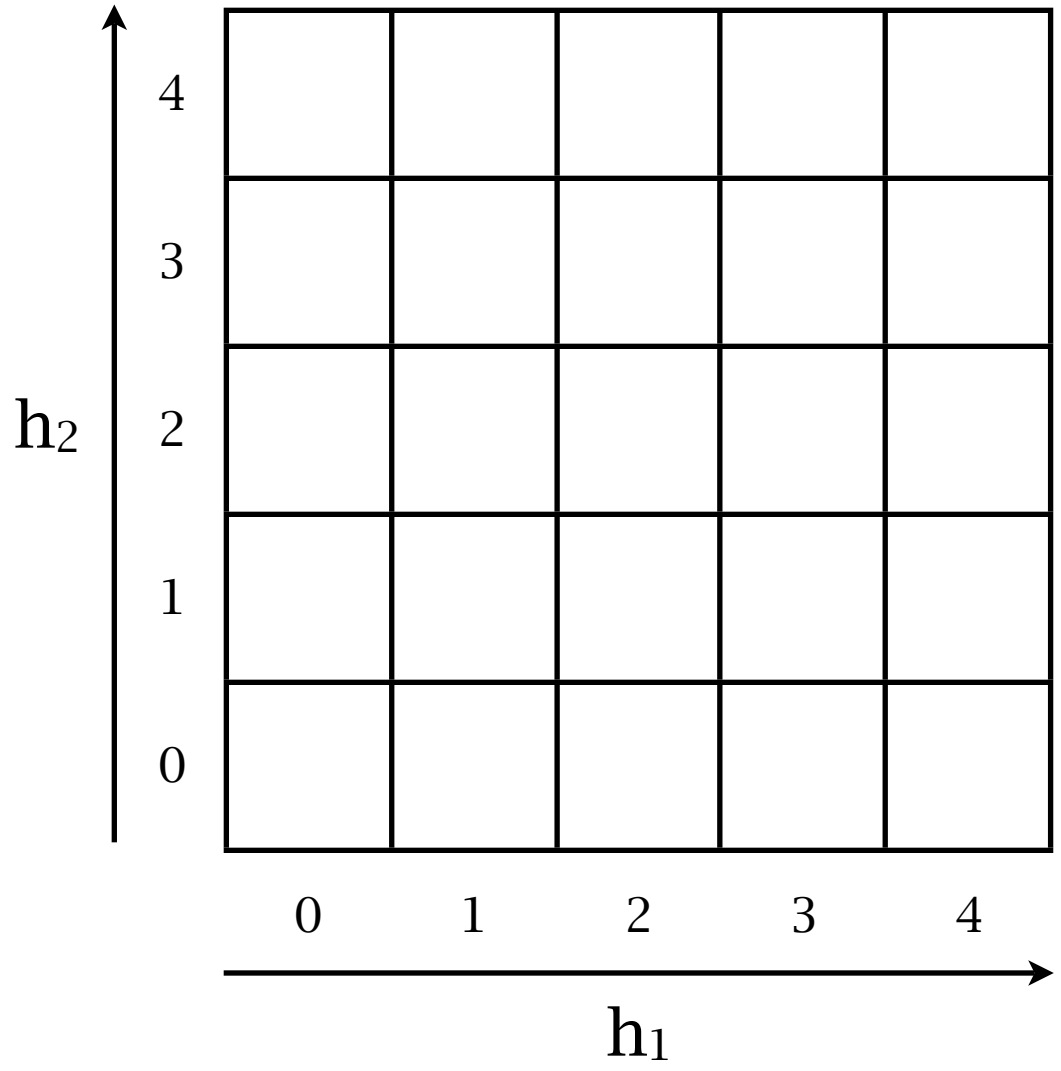


Solving the offline problem

Completion times

	h_1	h_2
I_1		
I_2		
I_3		
I_4		

Shortest path problem

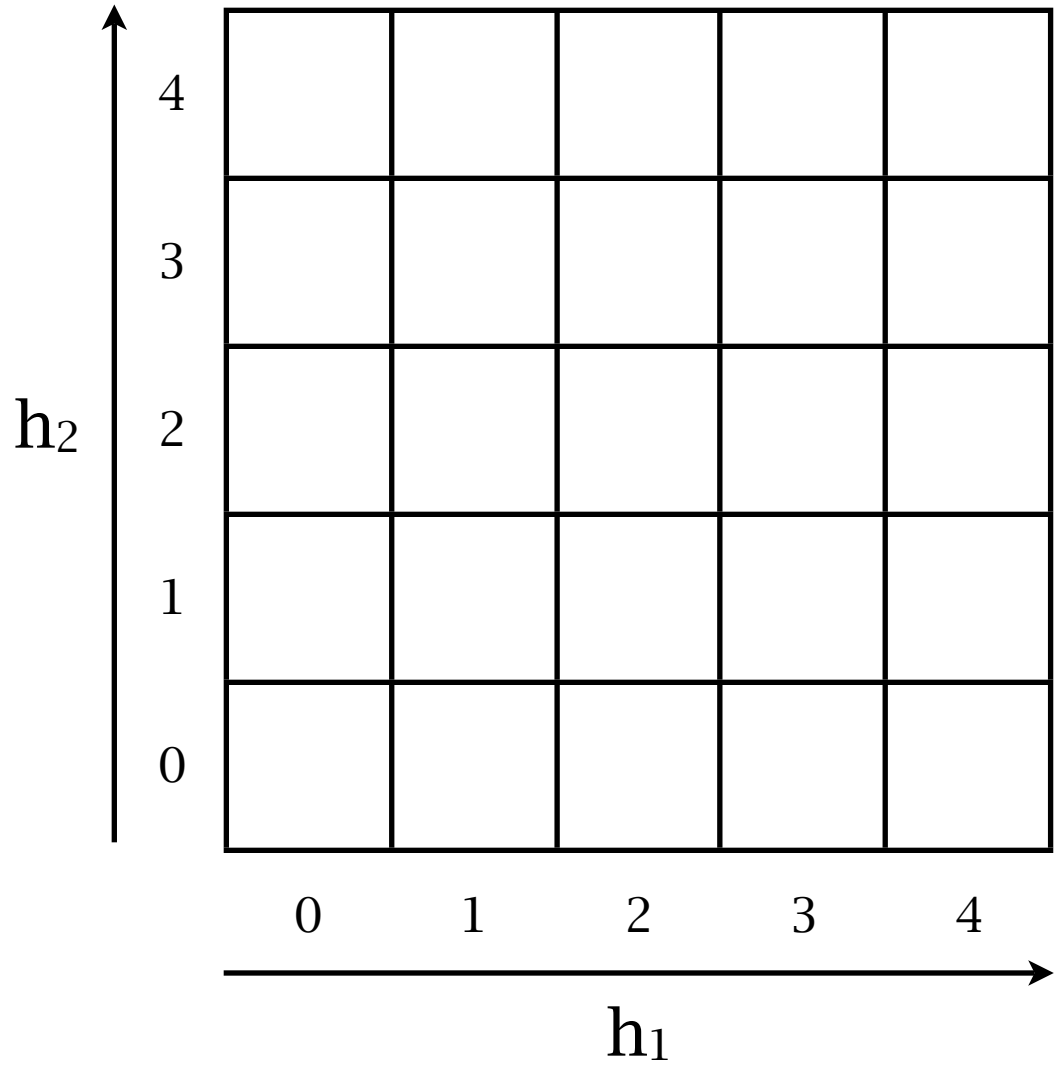


Solving the offline problem

Completion times

	h_1	h_2
I_1		
I_2		
I_3		
I_4		

Shortest path problem

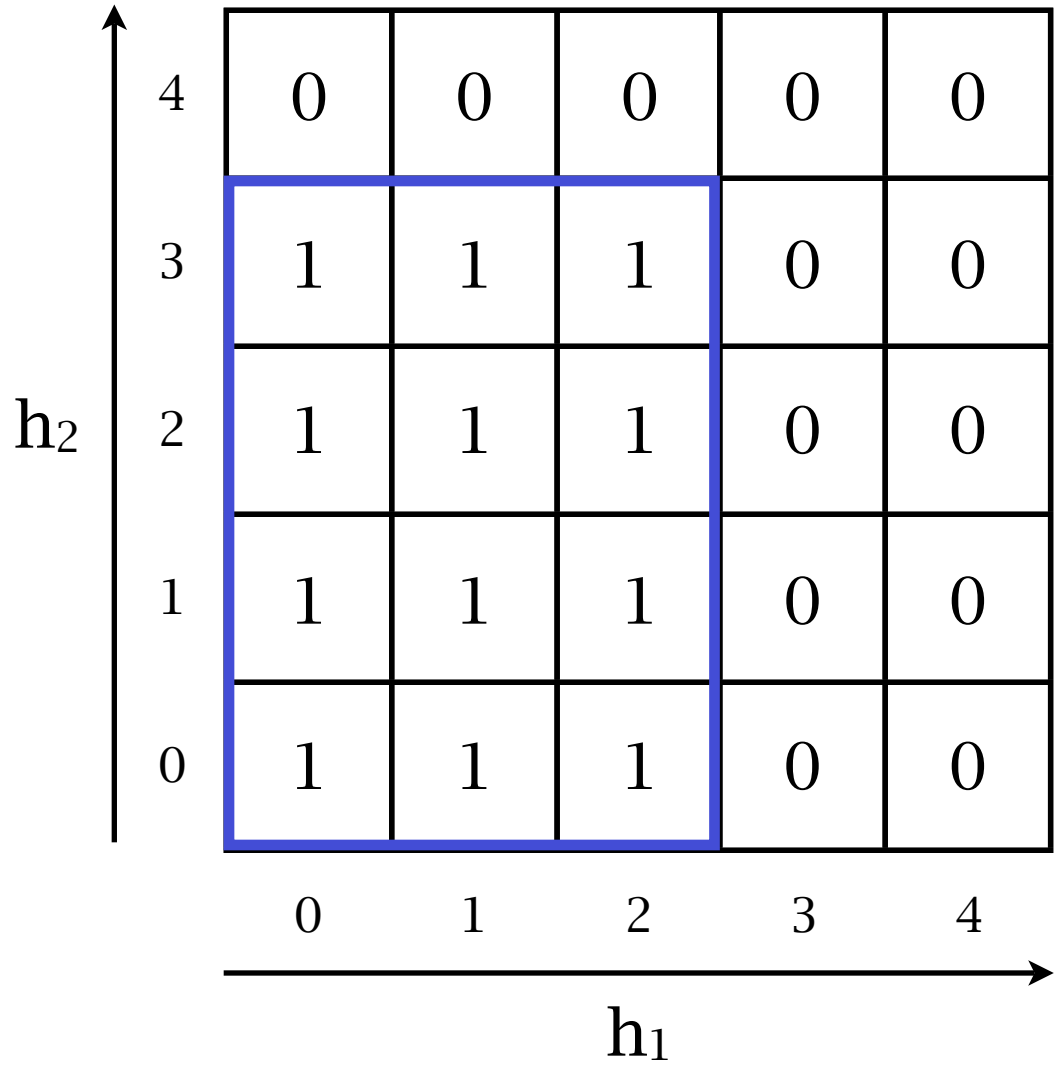


Solving the offline problem

Completion times

	h_1	h_2
I_1	3	4
I_2		
I_3		
I_4		

Shortest path problem

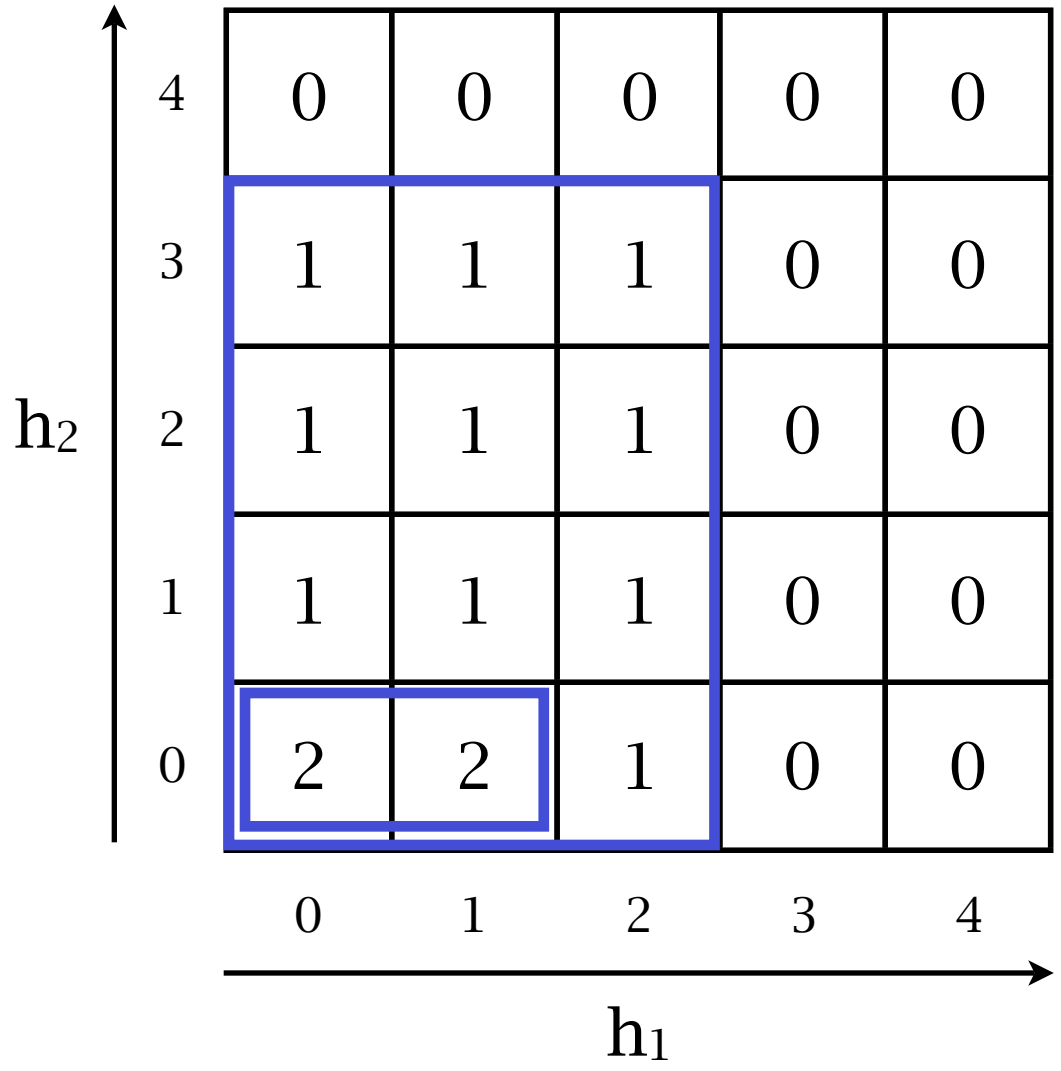


Solving the offline problem

Completion times

	h_1	h_2
I_1	3	4
I_2	2	1
I_3		
I_4		

Shortest path problem

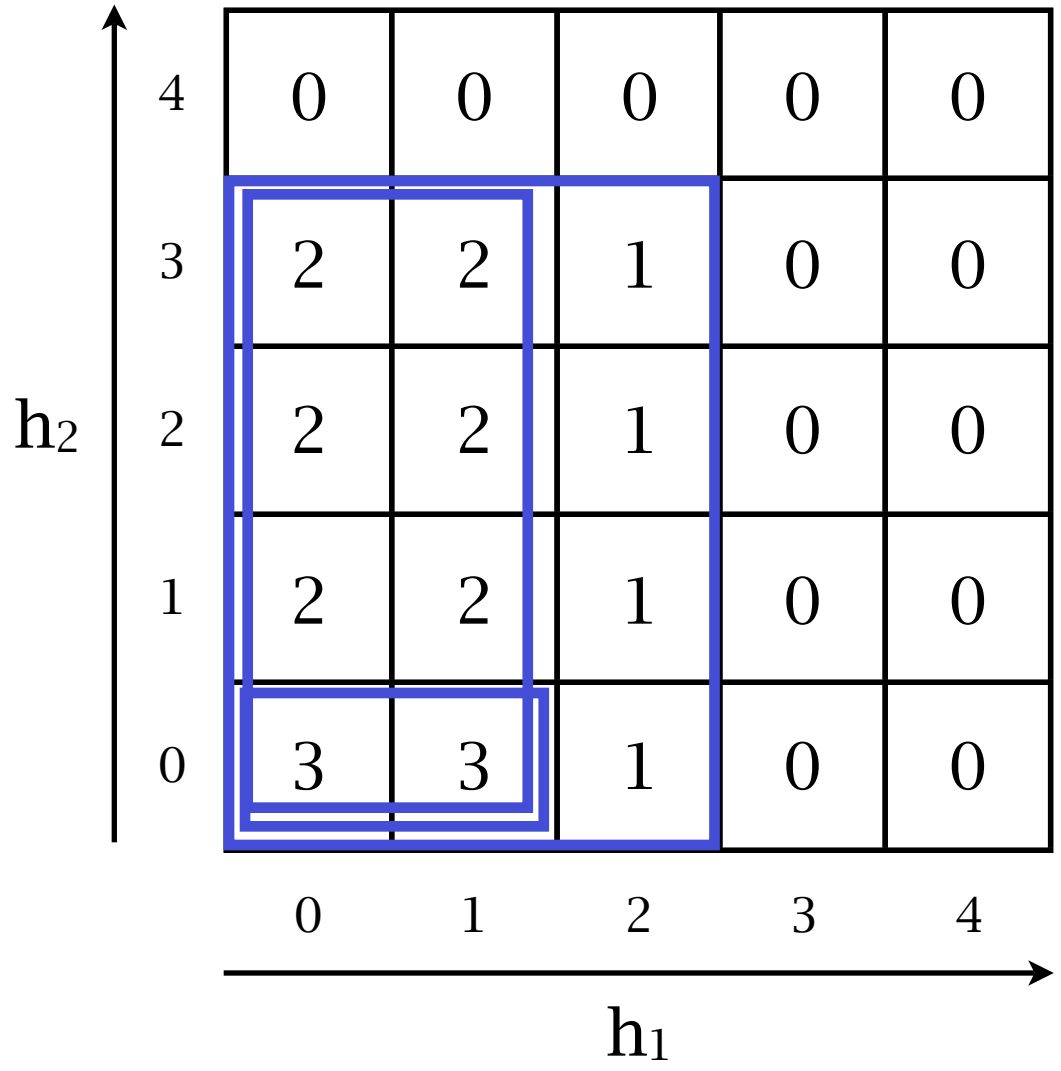


Solving the offline problem

Completion times

	h_1	h_2
I_1	3	4
I_2	2	1
I_3	2	4
I_4		

Shortest path problem

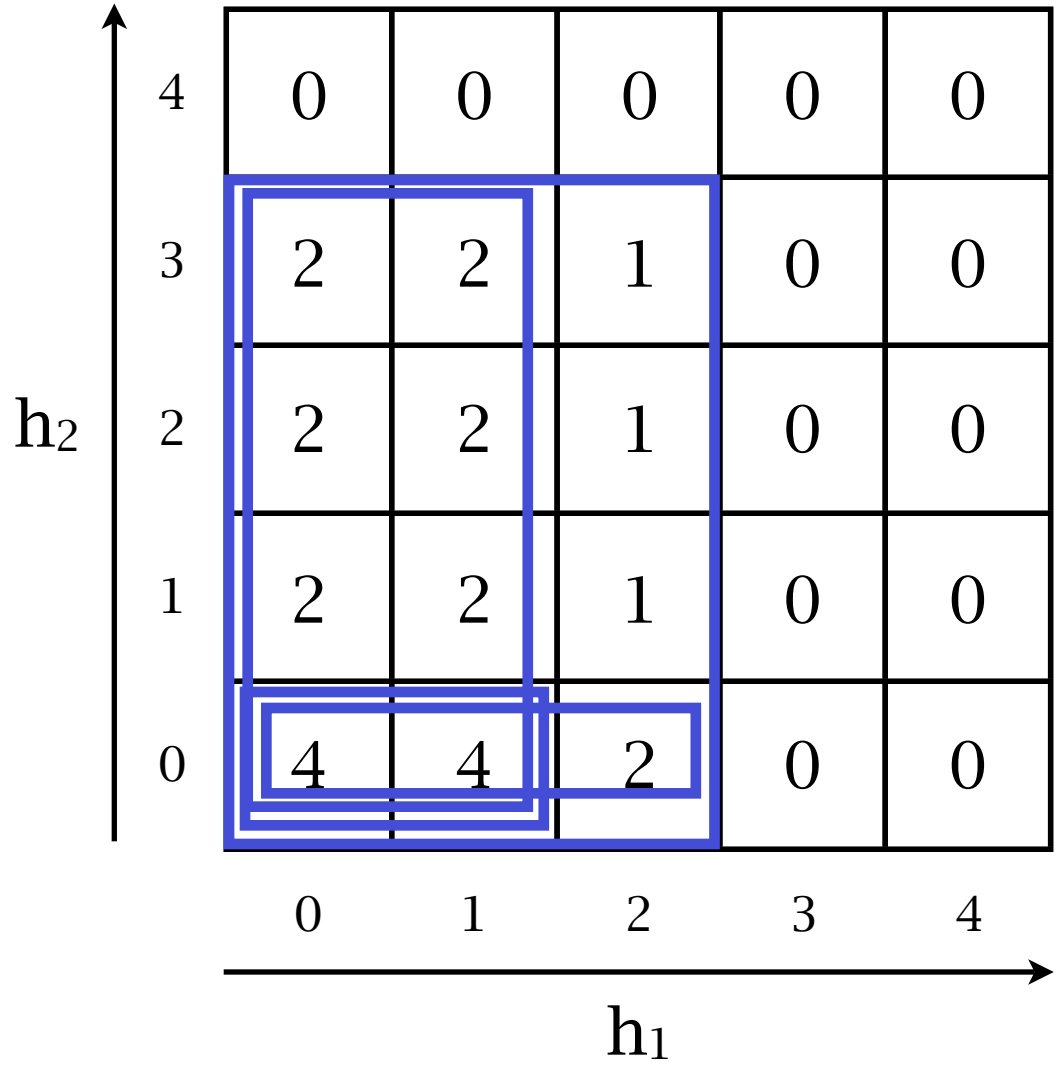


Solving the offline problem

Completion times

	h_1	h_2
I_1	3	4
I_2	2	1
I_3	2	4
I_4	3	1

Shortest path problem

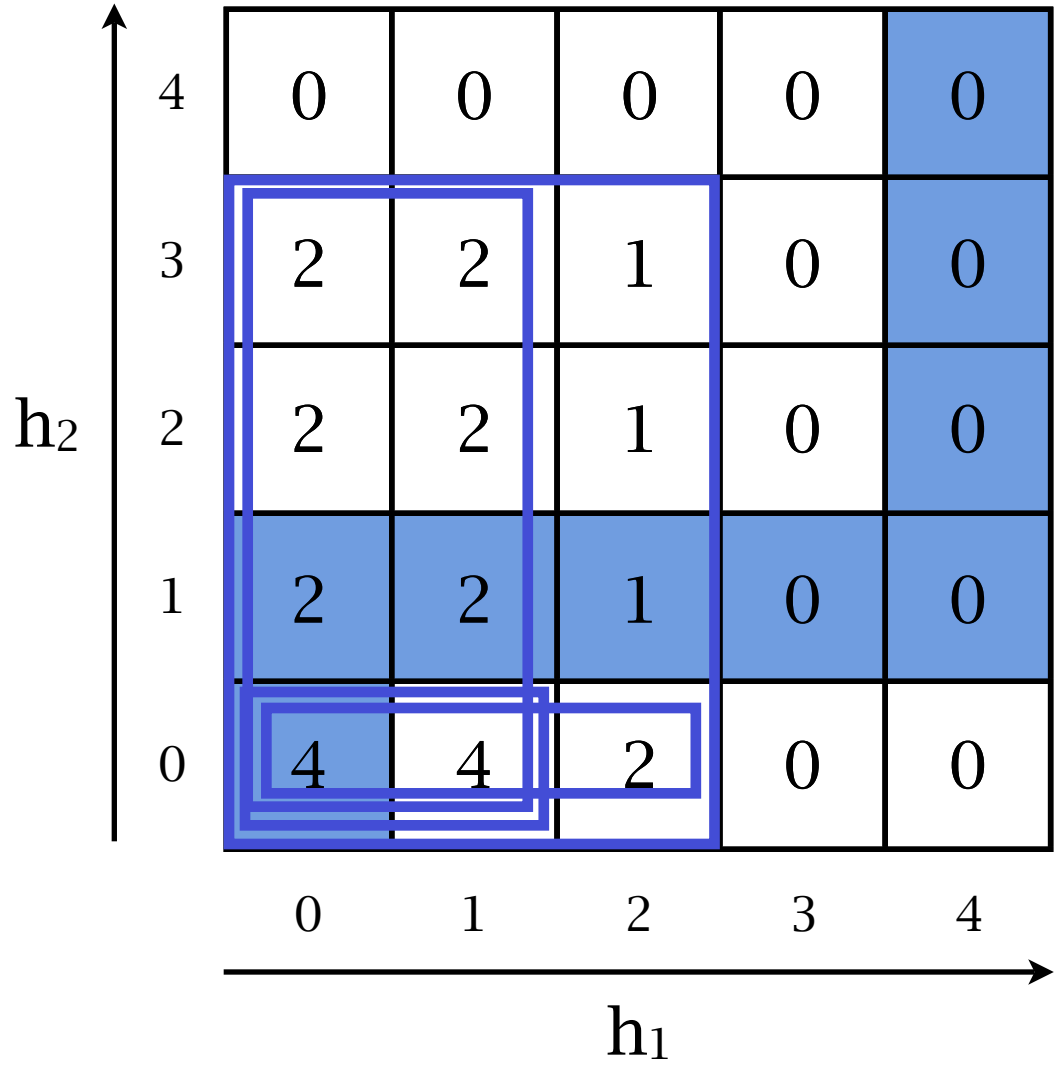


Solving the offline problem

Completion times

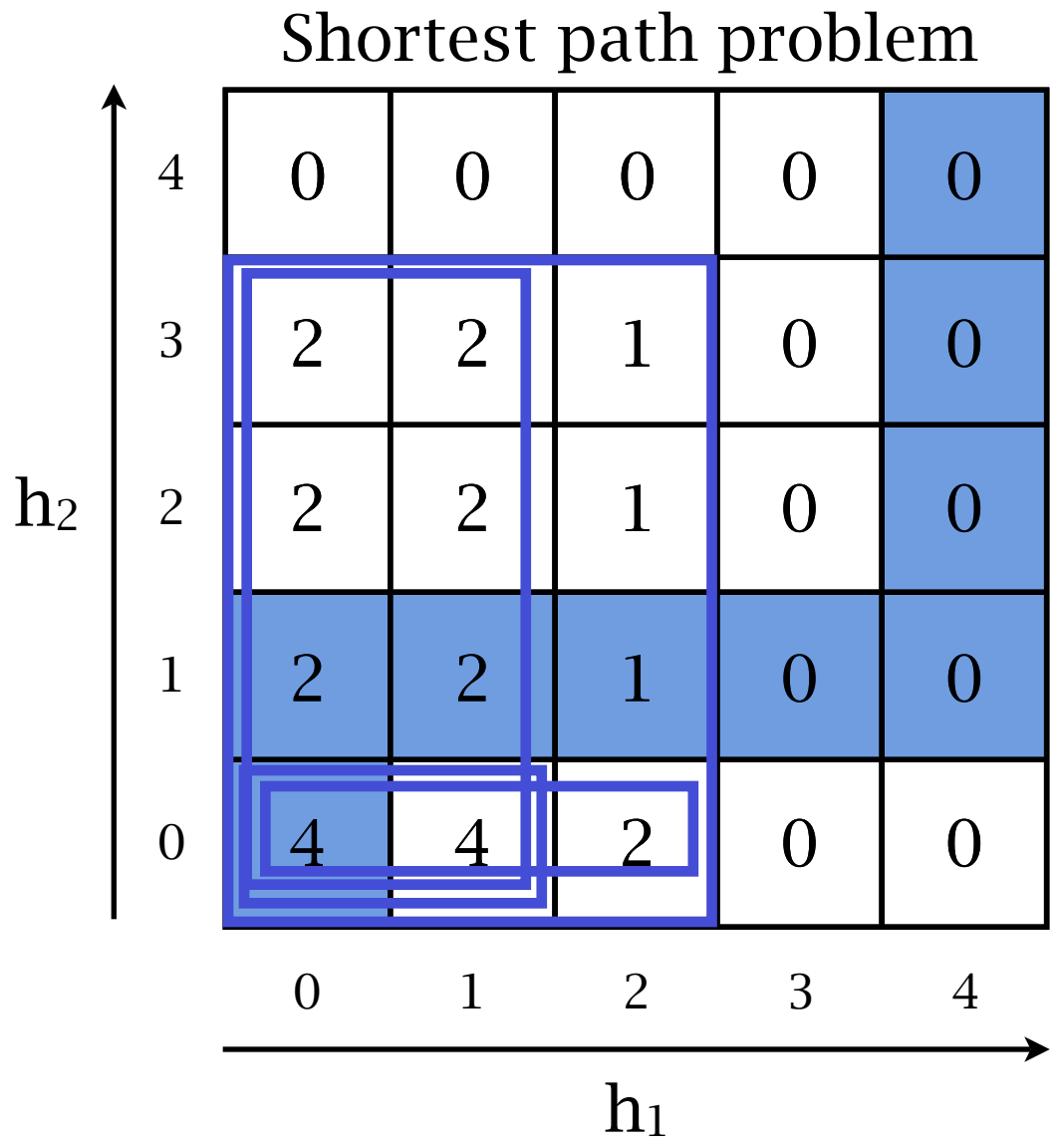
	h_1	h_2
I_1	3	4
I_2	2	1
I_3	2	4
I_4	3	1

Shortest path problem



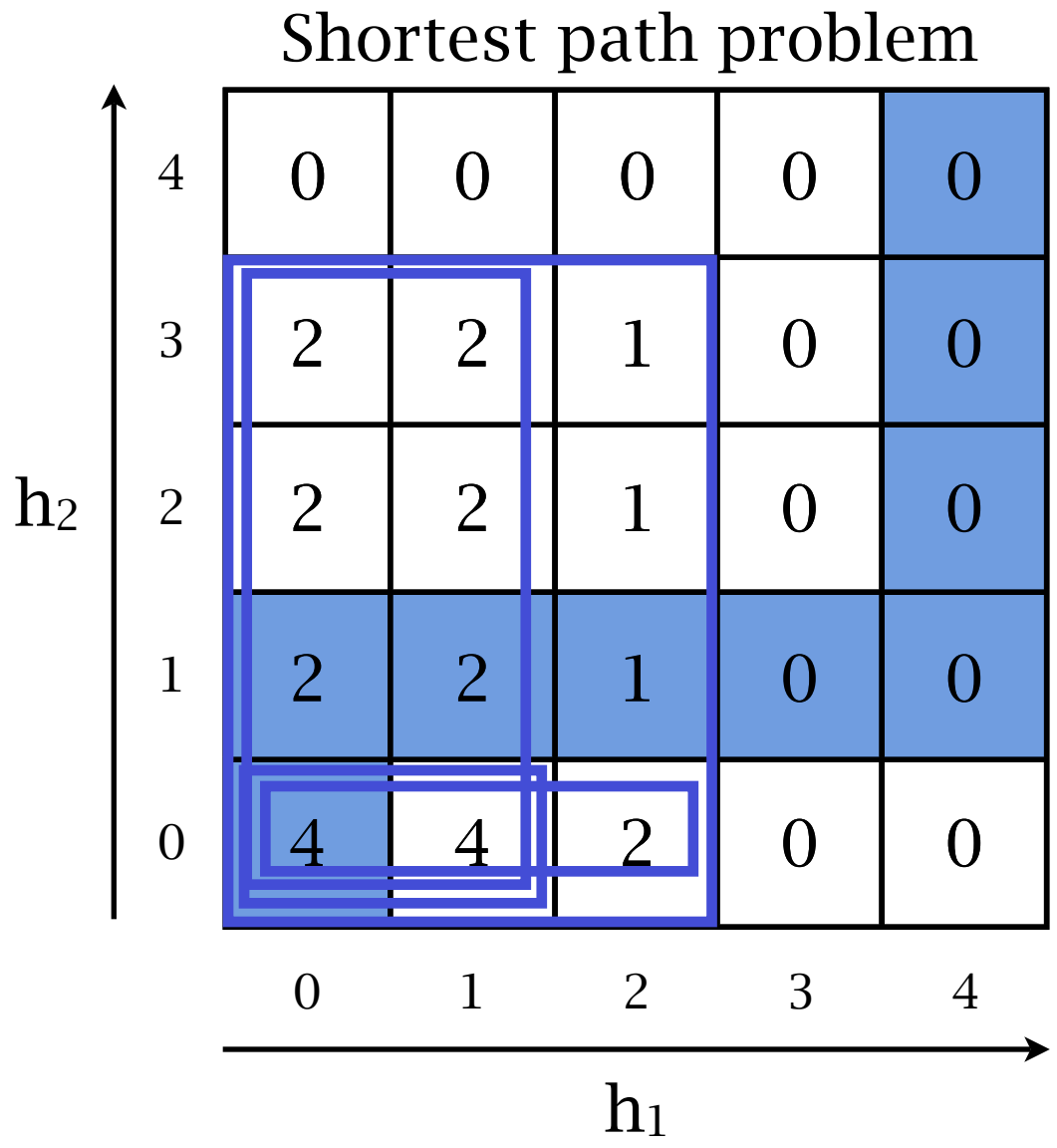
Solving the offline problem

- Time complexity is $O(nk(B+1)^k)$



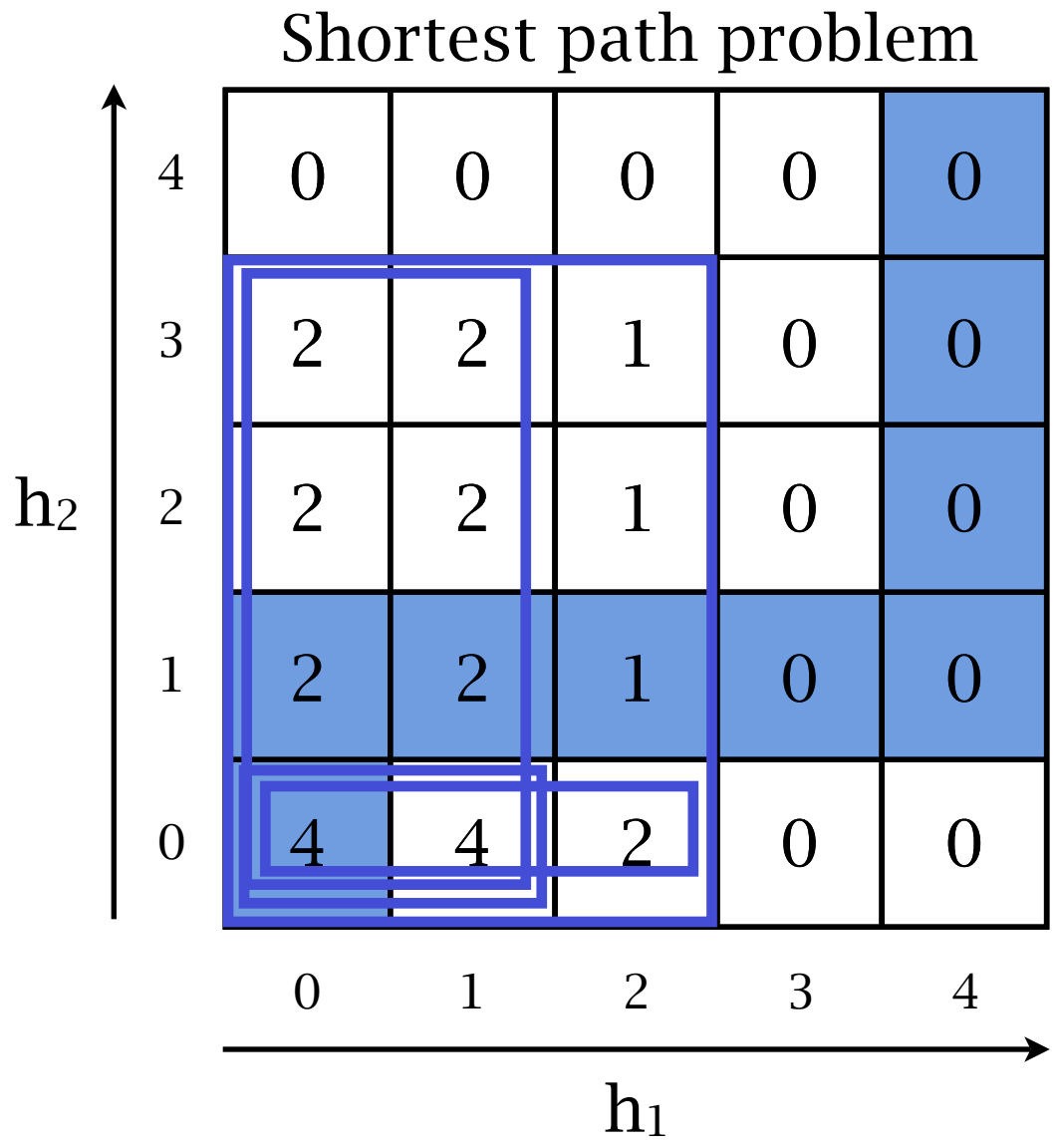
Solving the offline problem

- Time complexity is $O(nk(B+1)^k)$
- Can get α-approximation in time $O(nk(1+\log_\alpha B)^k)$



Solving the offline problem

- Time complexity is $O(nk(B+1)^k)$
- Can get α-approximation in time $O(nk(1+\log_\alpha B)^k)$
- Can also replace B with n (Theorem 12 of Sayag et al. 2006)



Hardness of approximation

- Offline problem of computing an optimal task-switching schedule generalizes *min-sum set cover*; obtaining an α-approximation is NP-hard for any $\alpha < 4$ (Feige, Lovász, & Tetali, APPROX 2002)

Min-sum set cover

(Feige, Lovász, & Tetali, 2002)

Min-sum set cover

(Feige, Lovász, & Tetali, 2002)

- Input: k sets, n elements
- Output: ordering of the sets that minimizes
  $\Sigma_{\text{elements } x}$ *coverage-time*(x)
  where *coverage-time*(x) = position of first set containing x
- Example: in ordering {a,b},{a,c},{d},
  coverage-time(a)=1 and *coverage-time*(c)=2

Min-sum set cover

(Feige, Lovász, & Tetali, 2002)

- Input: k sets, n elements
- Output: ordering of the sets that minimizes
  $\Sigma_{\text{elements } x}$ *coverage-time*(x)
  where *coverage-time*(x) = position of first set containing x
- Example: in ordering {a,b},{a,c},{d},
  coverage-time(a)=1 and *coverage-time*(c)=2
- Our problem is equivalent when B=1, so $\tau_{ij} \in \{1,\infty\}$

  (sets = heuristics, elements = instances)

Min-sum set cover

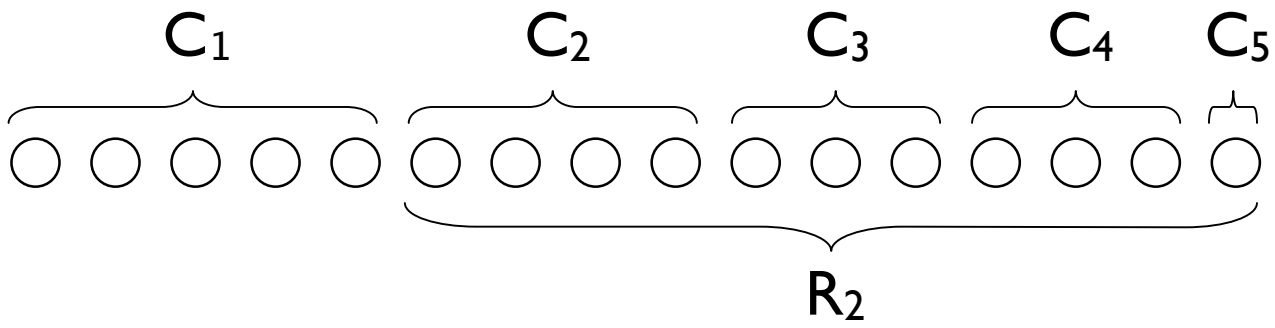
(Feige, Lovász, & Tetali, 2002)

- Can get a 4-approximation by greedily choosing the set that covers the max #elements to go next in the ordering
- Will generalize to get 4-approximation for task-switching schedules

Greedy min-sum set cover

(Feige, Lovász, & Tetali, 2002)

- Let C_i = #(elements with coverage time i under greedy ordering); let $R_i = C_i + C_{i+1} + ... + C_k$



- Key fact: under *any* ordering, at least $R_i - t*C_i$ elements have coverage time > t (for all i,t)

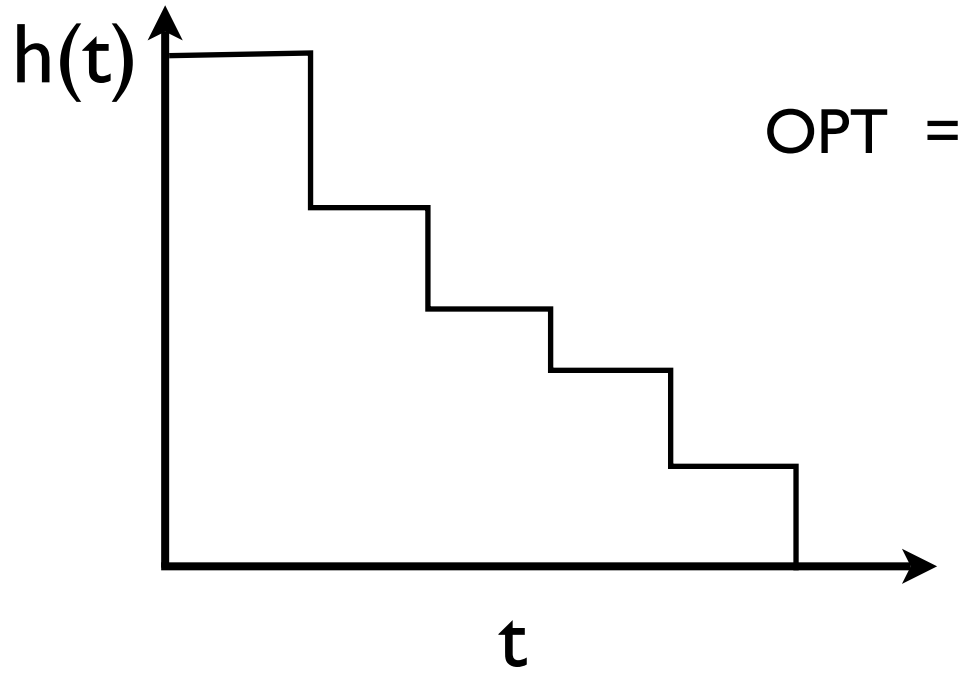
Greedy min-sum set cover

(Feige, Lovász, & Tetali, 2002)

Greedy min-sum set cover

(Feige, Lovász, & Tetali, 2002)

- Let h(t) = #(elements with coverage time > t under optimal ordering), so OPT = Σ_t h(t)

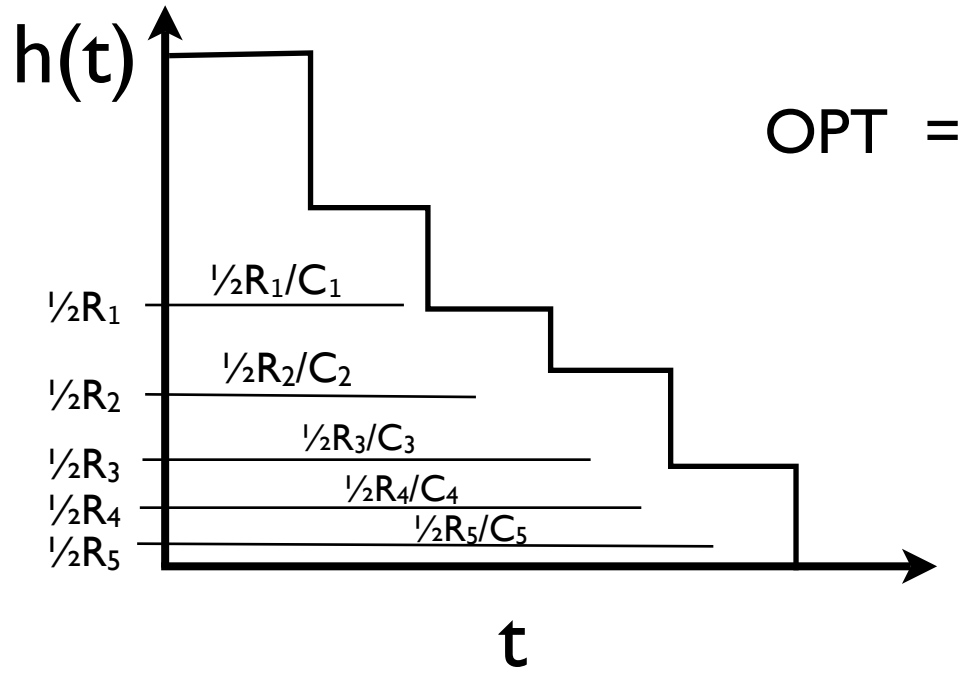


OPT = area under curve

Greedy min-sum set cover

(Feige, Lovász, & Tetali, 2002)

- Let h(t) = #(elements with coverage time > t under optimal ordering), so OPT = Σ_t h(t)
- Key fact: $h(t) \geq R_i - t*C_i$. In particular, $h(\frac{1}{2}R_i/C_i) \geq \frac{1}{2}R_i$

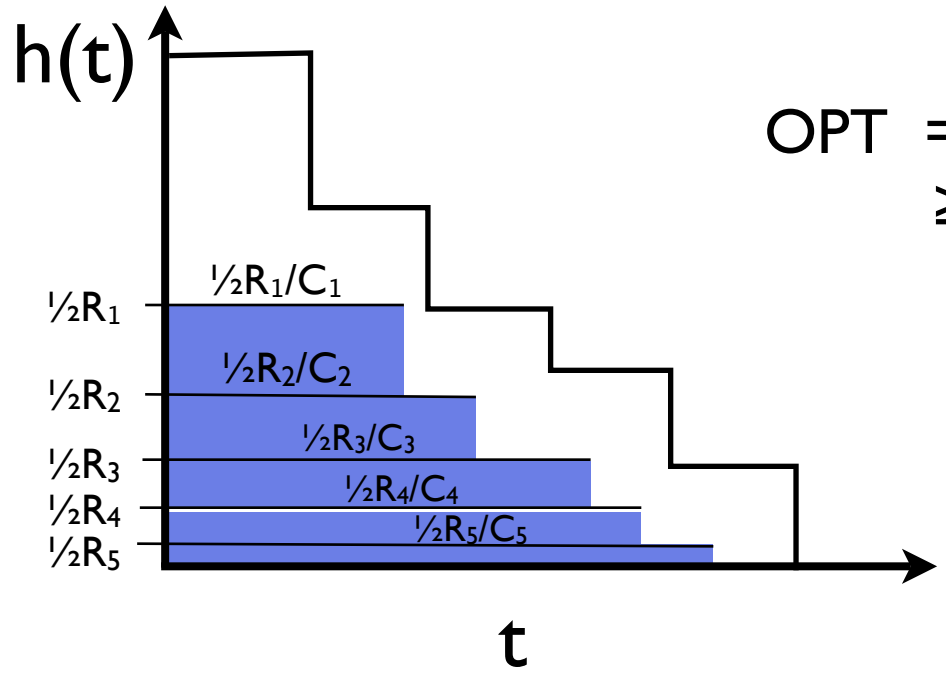


OPT = area under curve

Greedy min-sum set cover

(Feige, Lovász, & Tetali, 2002)

- Let h(t) = #(elements with coverage time > t under optimal ordering), so OPT = Σ_t h(t)
- Key fact: $h(t) \geq R_i - t*C_i$. In particular, $h(\frac{1}{2}R_i/C_i) \geq \frac{1}{2}R_i$

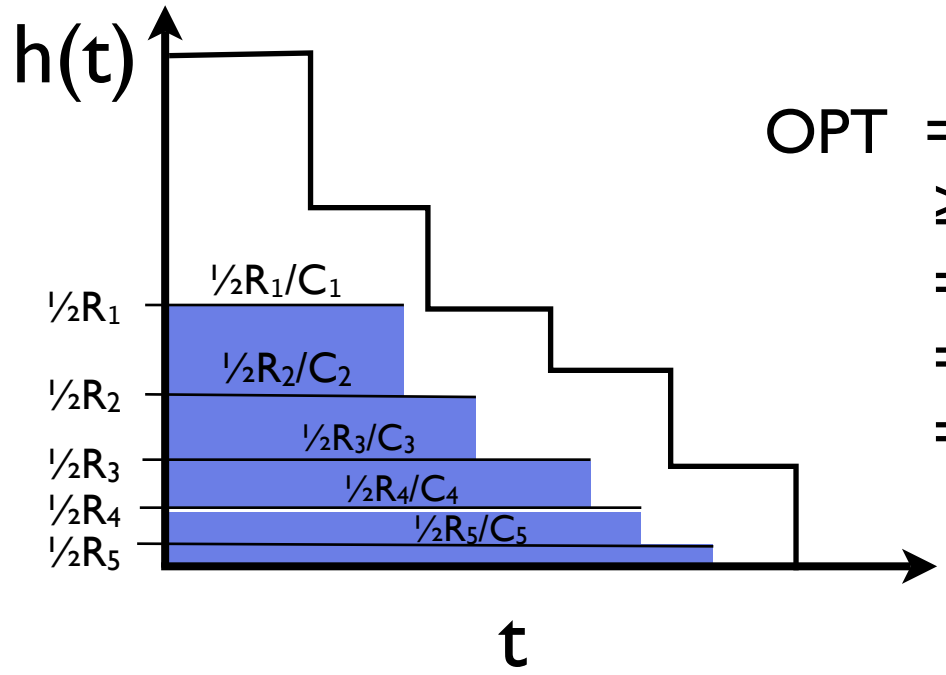


OPT = area under curve
≥ shaded area

Greedy min-sum set cover

(Feige, Lovász, & Tetali, 2002)

- Let h(t) = #(elements with coverage time > t under optimal ordering), so OPT = Σ_t h(t)
- Key fact: $h(t) \geq R_i - t*C_i$. In particular, $h(\frac{1}{2}R_i/C_i) \geq \frac{1}{2}R_i$



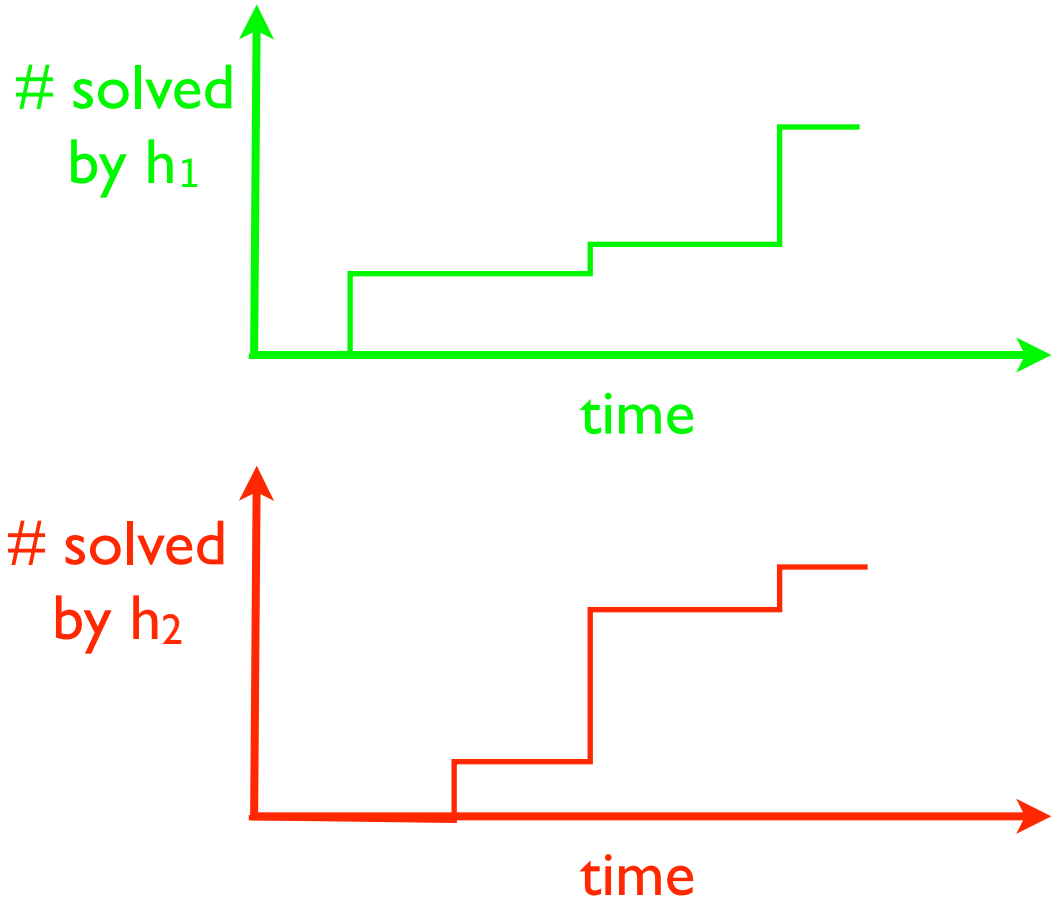
OPT = area under curve
≥ shaded area
= Σ_i ($\frac{1}{2}R_i/C_i$)*($\frac{1}{2}R_i$-$\frac{1}{2}R_{i+1}$)
= Σ_i $R_i/4$
= GREEDY/4

Greedy task-switching schedules

- Algorithm: greedily choose pair (h,t) such that running h for t (additional) time steps maximizes #(new instances solved)/t

Greedy task-switching schedules

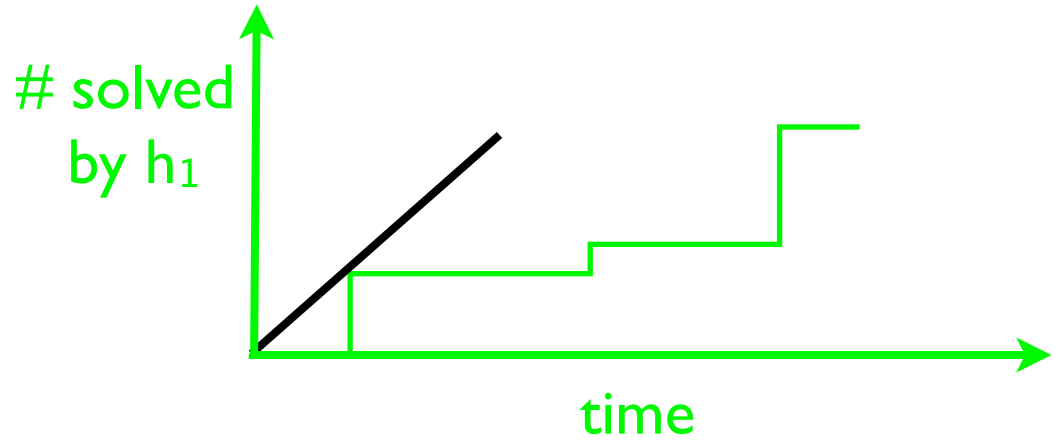
- Algorithm: greedily choose pair (h,t) such that running h for t (additional) time steps maximizes #(new instances solved)/t



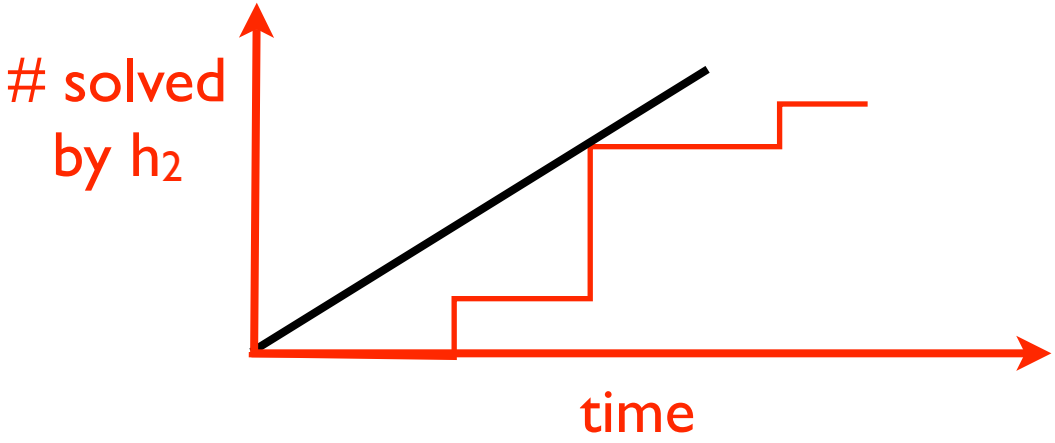
Schedule

Greedy task-switching schedules

- Algorithm: greedily choose pair (h,t) such that running h for t (additional) time steps maximizes #(new instances solved)/t

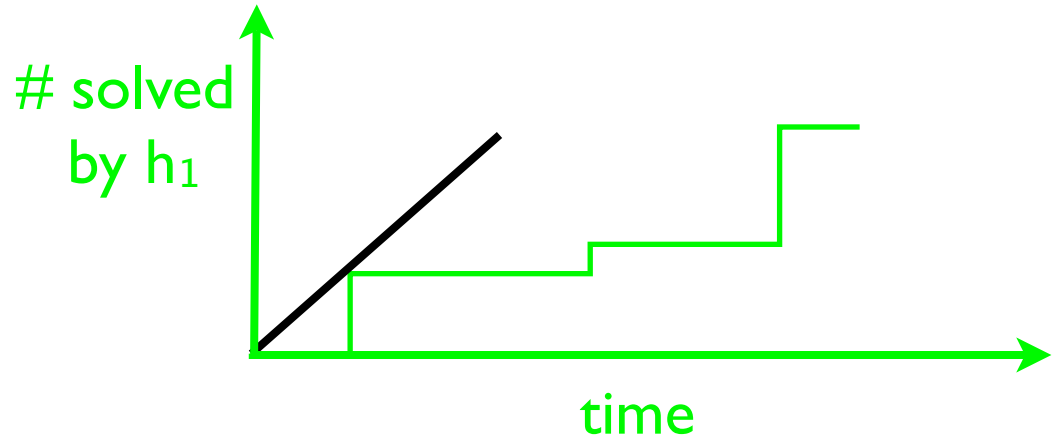


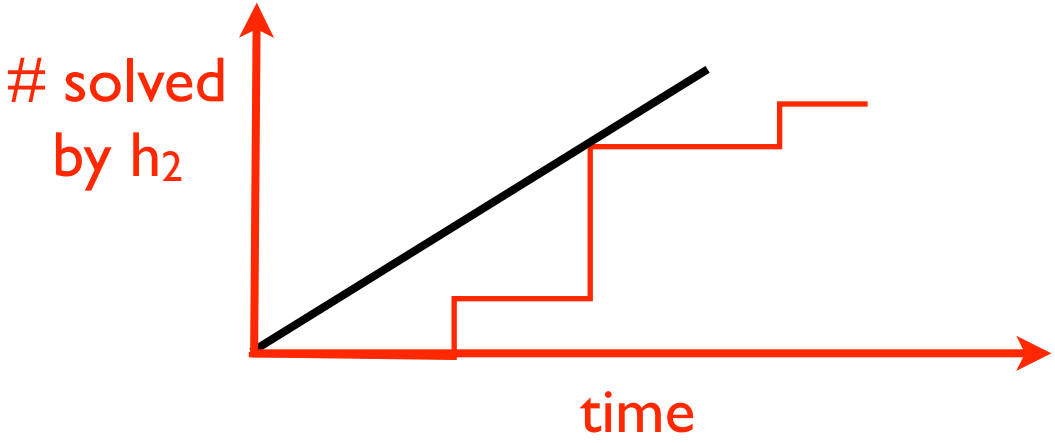
Schedule



Greedy task-switching schedules

- Algorithm: greedily choose pair (h,t) such that running h for t (additional) time steps maximizes #(new instances solved)/t

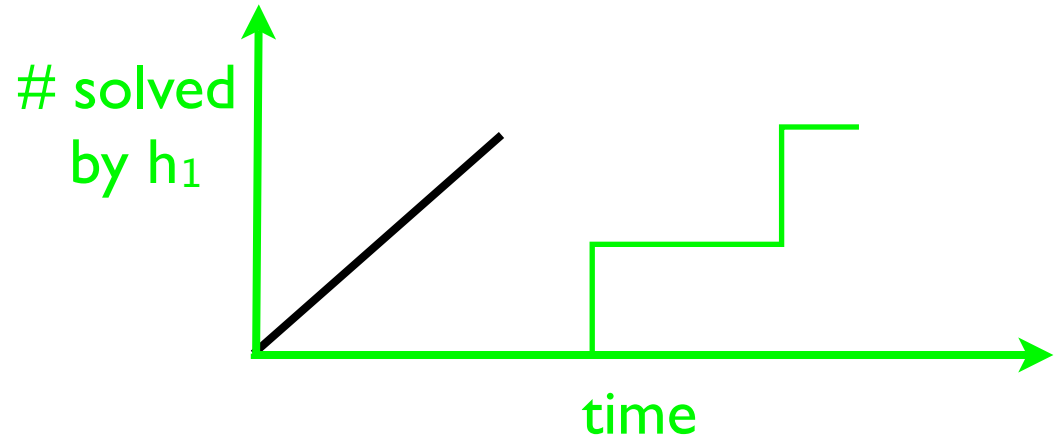


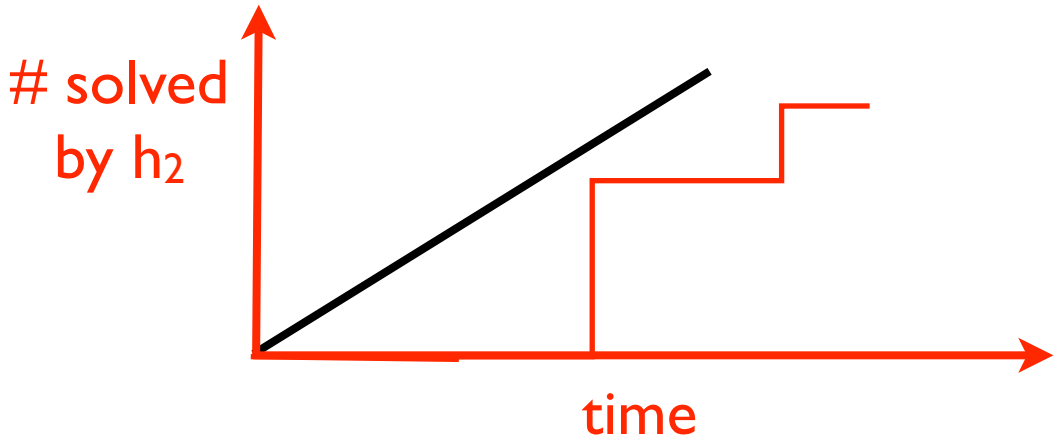


Schedule
run h_1 for 1 time step

Greedy task-switching schedules

- Algorithm: greedily choose pair (h,t) such that running h for t (additional) time steps maximizes #(new instances solved)/t

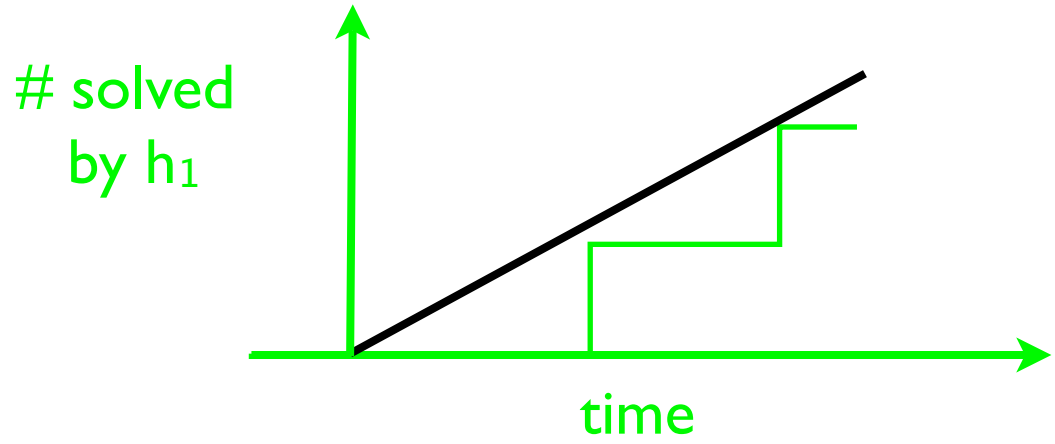


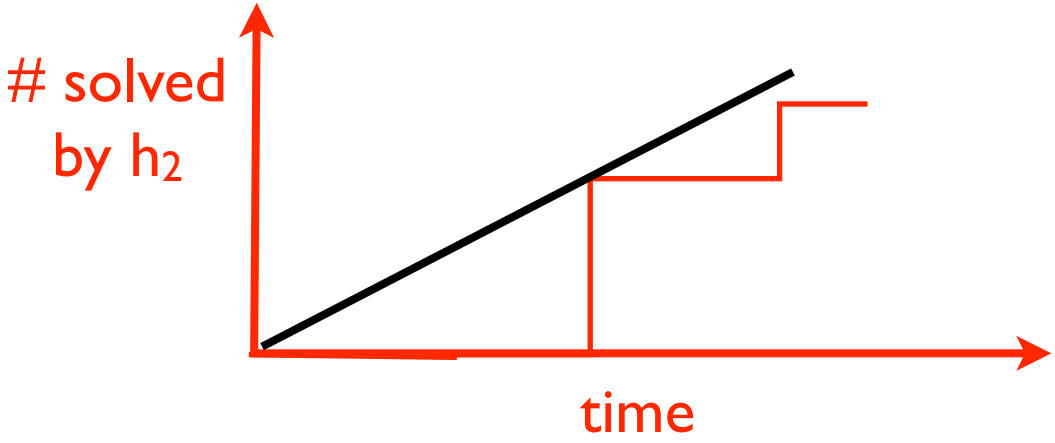


Schedule
run h_1 for 1 time step

Greedy task-switching schedules

- Algorithm: greedily choose pair (h,t) such that running h for t (additional) time steps maximizes #(new instances solved)/t

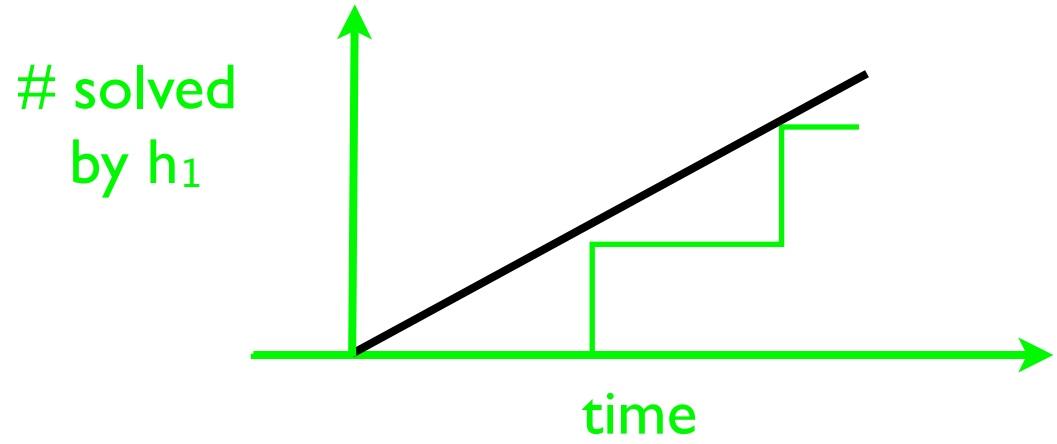


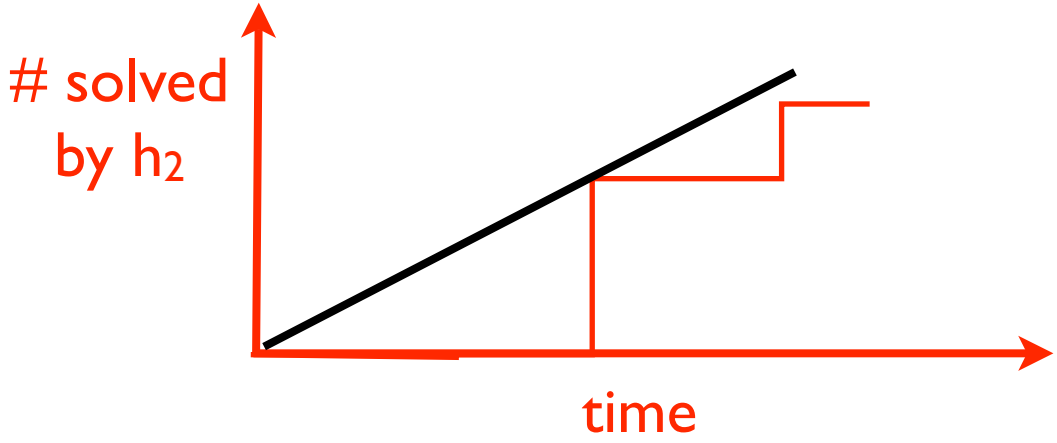


Schedule
run h_1 for 1 time step

Greedy task-switching schedules

- Algorithm: greedily choose pair (h,t) such that running h for t (additional) time steps maximizes #(new instances solved)/t





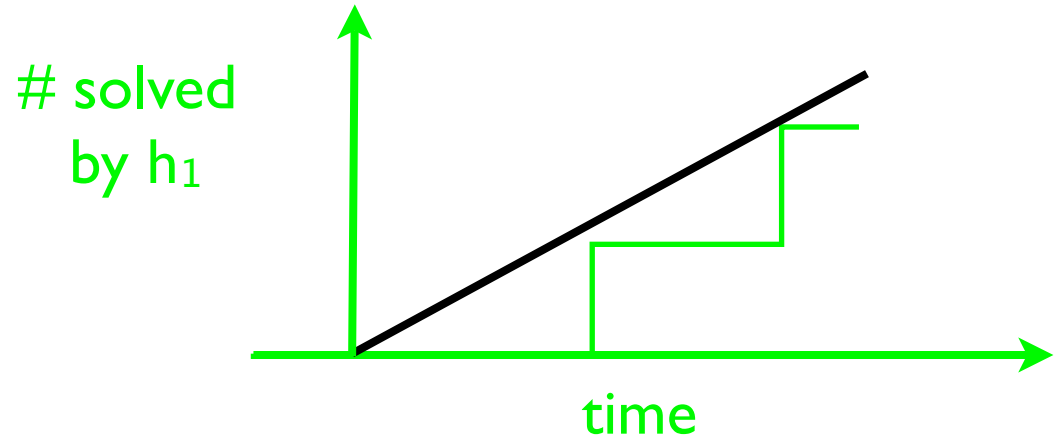
Schedule

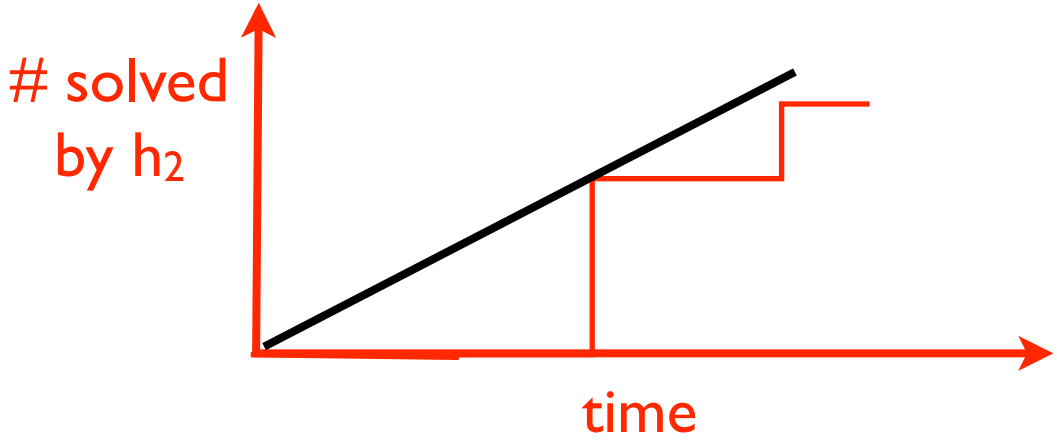
run h_1 for 1 time step

run h_2 for 4 time steps

Greedy task-switching schedules

- Algorithm: greedily choose pair (h,t) such that running h for t (additional) time steps maximizes #(new instances solved)/t





Schedule
run h_1 for 1 time step
run h_2 for 4 time steps

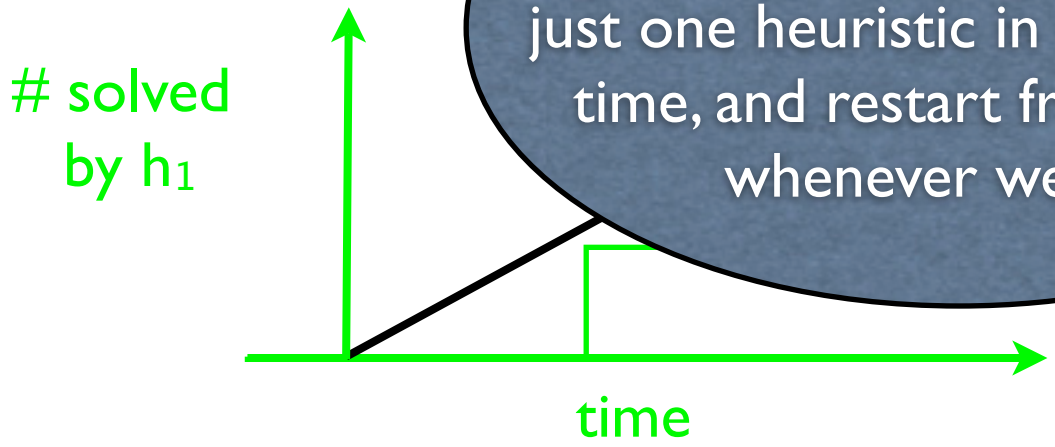
Can show any schedule has at least $R_i - t*C_i$ instances unsolved at time t, where C_i = i^{th} slope and R_i = #(instances unsolved before i^{th} phase)

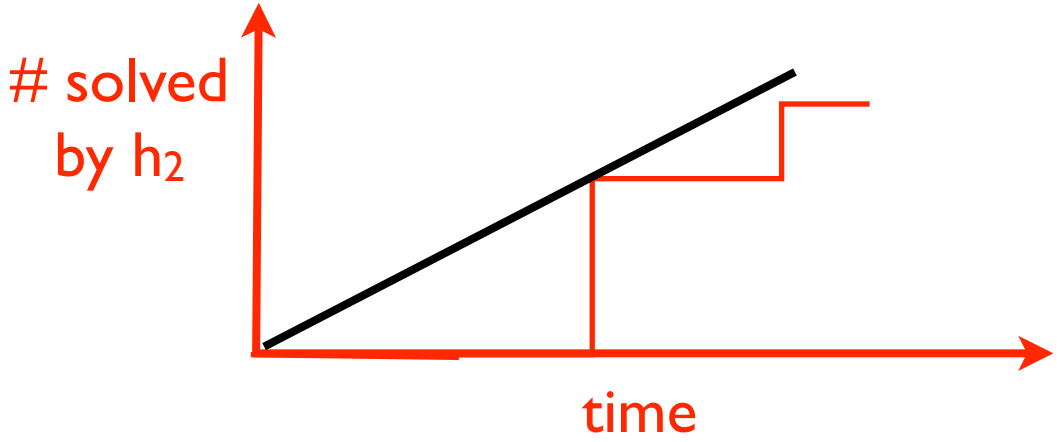
Then use similar proof by picture

Greedy task-switching schedules

- Algorithm: gree… …h that running h for t (addition… …stances solved)/t

In fact, we obtain a 4-approximation even if we keep just one heuristic in memory at a time, and restart from scratch whenever we switch





…chedule
…or 1 time step
…for 4 time steps

Can show any schedule has at least $R_i - t*C_i$ instances unsolved at time t, where C_i = i^{th} slope and R_i = #(instances unsolved before i^{th} phase)

Then use similar proof by picture

The online problem

- Nature (or an adversary) fills in table τ of completion times. Then:
- For j from 1 to n
  - You select task-switching schedule S_j
  - You incur cost $c_j(S_j)$ = time it takes to j^{th} instance using S_j
  - Your feedback is $c_j(S_j)$
- Regret = $\mathbf{E}[\sum_j c_j(S_j) - \min_S \sum_j c_j(S)]$
- Want worst-case regret that is o(n)

Background: experts algorithms

Background: experts algorithms

- General framework: have M experts that make predictions every day; following expert e's advice on day j costs $c_j(e)$
  - Every day you pick expert e_j and incur costs $c_j(e_j)$
  - You then learn $c_j(e)$ for all experts
  - regret = $\mathbf{E}[\sum_j c_j(e_j) - \min_e \sum_j c_j(e)]$
- Randomized weighted majority (RWM) gives worst-case regret $O((n \log M)^{1/2})$

Background: experts algorithms

- General framework: have M experts that make predictions every day; following expert e's advice on day j costs $c_j(e)$
  - Every day you pick expert e_j and incur costs $c_j(e_j)$
  - You then learn $c_j(e)$ for all experts
  - regret = $\mathbf{E}[\sum_j c_j(e_j) - \min_e \sum_j c_j(e)]$
- Randomized weighted majority (RWM) gives worst-case regret $O((n \log M)^{1/2})$
- Suppose that to learn c_j you must pay an "exploration cost" C that is added to regret. Running RWM using data from a random subset of the days gives regret $O(n^{2/3}(C \log M)^{1/3})$ (Cesa-Bianchi et al., 2005)

Online shortest path algorithm

- Using existing no-regret strategies for online shortest paths in "bandit" feedback setting would give regret poly(#edges)

- By paying Bk, can reveal weights of all edges. Using Cesa-Bianchi et al. (2005) gives regret $O(Bkn^{2/3}(Lk \log k)^{1/3})$, where L = length of sides of grid

- Using dynamic programming, can implement RWM so decision-making time is O(#edges) (György et al., 2006)

Online greedy algorithm

(ongoing work)

Online greedy algorithm

(ongoing work)

- Consider running RWM on a sequence of n instances, using the following pool of experts:
  - For each heuristic h and each time t, have an expert that behaves as follows: w/prob. 1/t it runs h for t time steps; and w/prob. 1-1/t it does nothing
  - Expert's payoff is 1 if it solves the problem, 0 otherwise

Online greedy algorithm

(ongoing work)

- Consider running RWM on a sequence of n instances, using the following pool of experts:
  - For each heuristic h and each time t, have an expert that behaves as follows: w/prob. 1/t it runs h for t time steps; and w/prob. 1-1/t it does nothing
  - Expert's payoff is 1 if it solves the problem, 0 otherwise
- Will consume n time steps in expectation

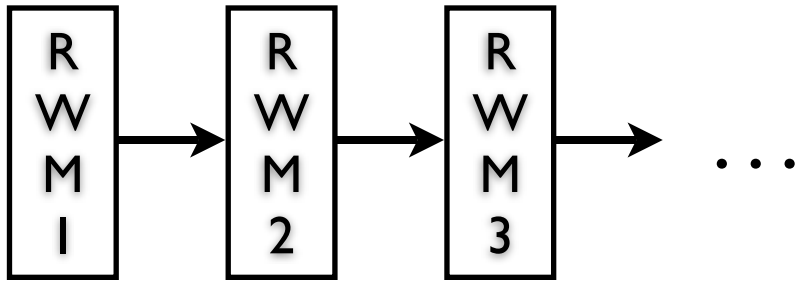
Online greedy algorithm

(ongoing work)

- Consider running RWM on a sequence of n instances, using the following pool of experts:
  - For each heuristic h and each time t, have an expert that behaves as follows: w/prob. 1/t it runs h for t time steps; and w/prob. 1-1/t it does nothing
  - Expert's payoff is 1 if it solves the problem, 0 otherwise
- Will consume n time steps in expectation
- To get regret/n $\rightarrow$ 0, must solve as many instances as possible per unit time (like offline greedy)

Online greedy algorithm

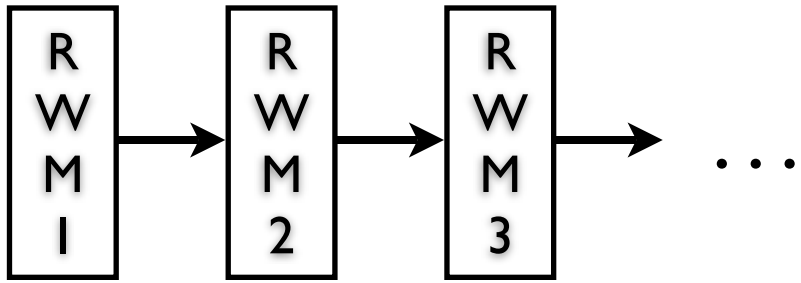
(ongoing work)



Online greedy algorithm

(ongoing work)

- Idea: define task-switching schedule using a series of such RWM algorithms, operating independently

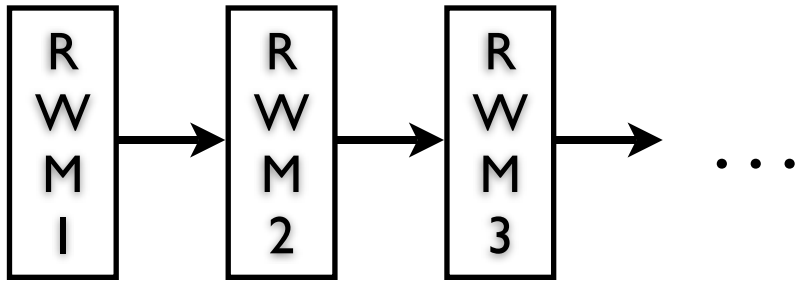


- Can show 4-regret is $O(\text{poly}(B,k)*n^{2/3})$

Online greedy algorithm

(ongoing work)

- Idea: define task-switching schedule using a series of such RWM algorithms, operating independently



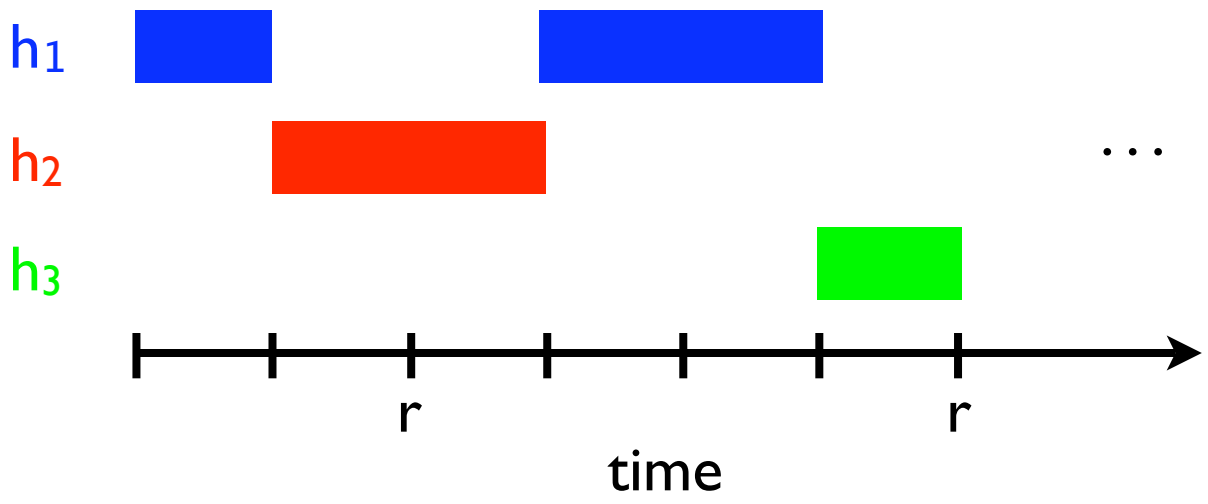
- Can show 4-regret is $O(poly(B,k)*n^{2/3})$
- Using result of Kakade, Kalai & Ligett also gives 4-regret that is o(1), but exponential in #heuristics

Previous work

- Special case: deterministic heuristics with fixed known running time
  - Munagala et al., "The pipelined set cover problem" (ICDT 2005) — asymptotic O(log n) competitive ratio in adversarial online setting
  - Kaplan et al. "Learning with attribute costs" (STOC 2005) — asymptotically 4-competitive with better bounds than ours, but only in distributional online setting

Generalization: restart schedules

- Restart schedule = task-switching schedule augmented with flag that says whether to restart at each time slice (i.e., mapping $S: \mathbb{Z} \mapsto H \times \{0,1\}$)
- If $|H|=1$, this is just a sequence of restart thresholds $t_1, t_2, \ldots$



Generalization: restart schedules

- Offline greedy algorithm maximizes *expected* number of instances solved per unit time
- For online version, need to interpret B as a bound on total time devoted to a single heuristic (across multiple runs)

Experiments

Solver competitions

- Each year, various conferences hold solver competitions with the following format:
  - each submitted heuristic is run on a sequence of instances (subject to time limit)
  - awards for heuristics that solve the most instances in various instance categories
- Downloaded tables of completion times, computed (approximately) optimal task-switching schedules, and compared them to best individual solver

Results for ICAPS 2006 Planning Competition

- A.I. planning involves finding a minimum-length sequence of actions that lead from a start state to a goal state
- Six "optimal" planners were submitted to 2006 A.I. planning competition
  - each run on 240 instances with 30 minute time limit per instance
  - 110 instances were solved by at least one of the six

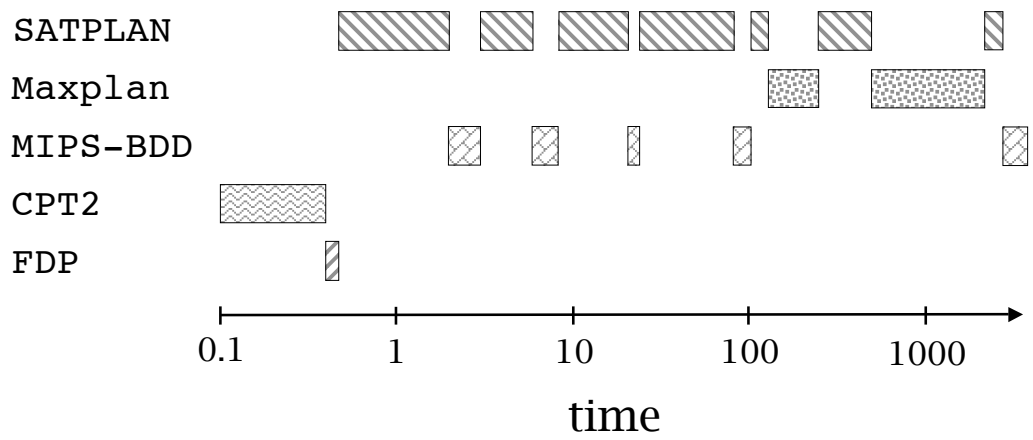
Results for 2006 A.I. Planning Competition

Solver	**Avg. CPU (s)**	**Num. solved**
Greedy schedule	358	98
Single-run greedy	476	96
`SATPLAN`	507	83
`Maxplan`	641	88
`MIPS-BDD`	946	54
`CPT2`	969	53
`FDP`	1079	46
Parallel schedule	1244	89
`IPPLAN-1SC`	1437	23

Results for 2006 A.I. Planning Competition

Solver	Avg. CPU (s)	Num. solved
Greedy schedule	358	98
Single-run greedy	476	96
SATPLAN	507	83
Maxplan	641	88
MIPS-BDD	946	54
CPT2	969	53
FDP	1079	46
Parallel schedule	1244	89
IPPLAN-1SC	1437	23

Greedy schedule:



Summary

Solver competition	**Domain**	**Speedup factor** (range across categories)
SAT 2005	satisfiability	1.2–2.0
ICAPS 2006	planning	1.4
CP 2006	constraint satisfaction	1.0–1.5
IJCAR 2006	theorem proving	1.0–7.7

Optimization heuristics

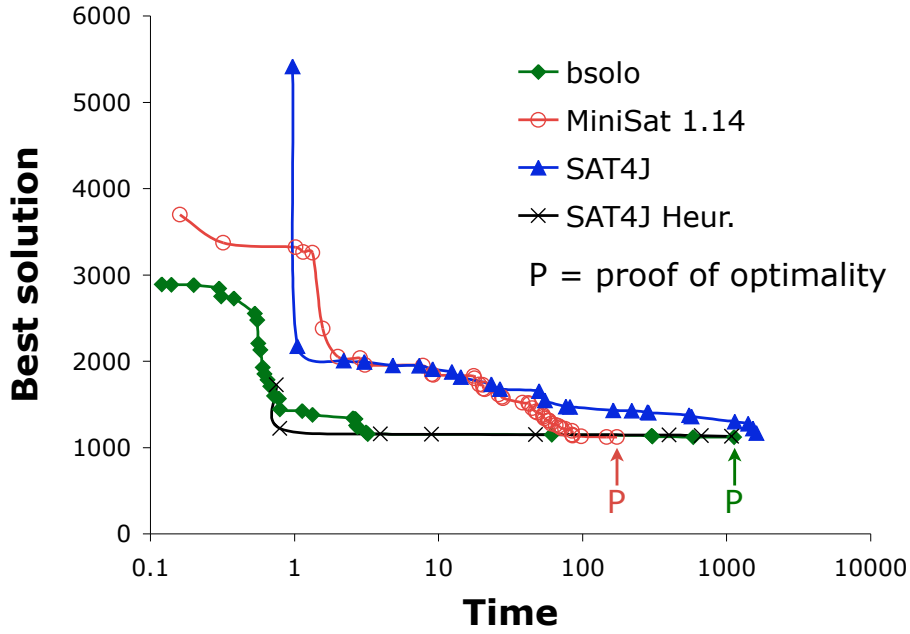
- For optimization heuristics, cost of a task-switching schedule should reflect how solution quality changes as a function of time
- Our results generalize to cost functions of the form $\sum_q w_q$*(time to get solution of quality at least q)

Results for PB 2006 evaluation

- "Pseudo-Boolean optimization" means using a SAT solver for 0/1 integer programming

- Several possible objectives. Used greedy algorithm to minimize

  (time to find feasible solution)
  + (time to find optimal solution)
  + (time to prove optimality)



Results for PB 2006 evaluation

Solver	**Avg. CPU to Prove Opt.**	**Avg. CPU to Find Opt.**	**Avg. CPU to Find Feas.**
Greedy schedule	116	85	29
`MiniSat 1.14`	277	257	86
`bsolo`	279	211	94
`SAT4J`	433	323	56
`SAT4J Heur.`	408	302	44

- Greedy schedule outperforms each individual solver with respect to all three criteria

Conclusions & Future Work

- We presented no-regret algorithms for selecting task-switching/restart schedules online
- Open problems:
  - matching upper & lower bounds on regret
  - better results for restart schedules when |H|=1?