Using Online Algorithms to Solve NP-Hard Problems More Efficiently in Practice

Matthew Streeter

Committee

Avrim Blum

Tuomas Sandholm

Stephen Smith, Chair

Carla Gomes, Cornell

John Hooker, Tepper School

Background

Background

- NP-hard problems are worst-case intractable under standard assumptions
- But, they arise frequently in practice

Background

- NP-hard problems are worst-case intractable under standard assumptions
- But, they arise frequently in practice
- Different techniques for dealing with this
  - Problem-specific theory (approximation algorithms, improved exponential-time algorithms, ...)
  - Benchmark-driven engineering (SAT solvers, job shop scheduling heuristics, ...)
  - Black box optimization (simulated annealing, genetic algorithms, ...)

Background

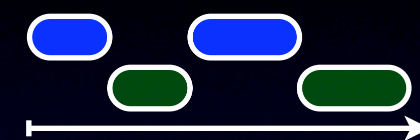
- NP-hard problems are worst-case intractable under standard assumptions
- But, they arise frequently in practice
- Different techniques for dealing with this
  - Problem-specific theory (approximation algorithms, improved exponential-time algorithms, ...)
  - Benchmark-driven engineering (SAT solvers, job shop scheduling heuristics, ...)
  - Black box optimization (simulated annealing, genetic algorithms, ...)

This thesis

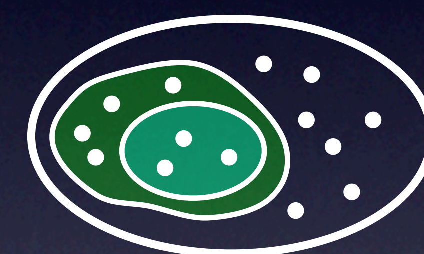
- **Goal:** boost performance of existing algorithms by adapting them to actual problem instance(s) encountered
  - use *black-box* techniques that can be applied to many problem domains
  - adaptation can be performed *online*, while solving a sequence of problem instances

Outline

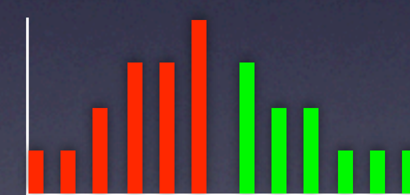
- Combining multiple heuristics online



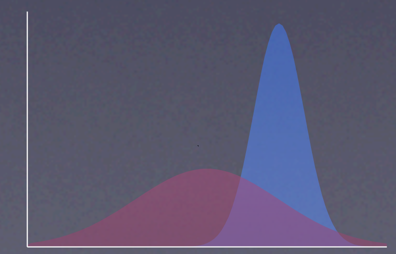
- Online algorithms for maximizing submodular functions



- Using decision procedures efficiently for optimization



- The max *k*-armed bandit problem



Combining Multiple Heuristics Online

Heuristics can have complementary strengths

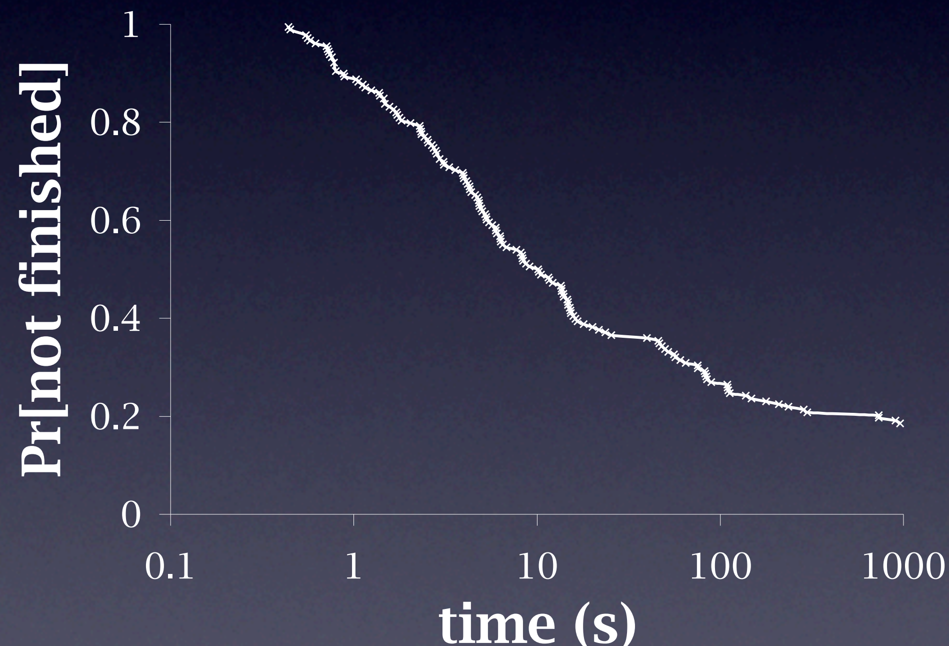
- Running time of heuristics varies widely across instances

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	≥ 12000
vliw-sat-2-0/9dlx_vliw_at_b_iq6_bug9	≥ 12000	131

- Can often reduce average-case running time by interleaving execution of multiple heuristics

The power of restarts

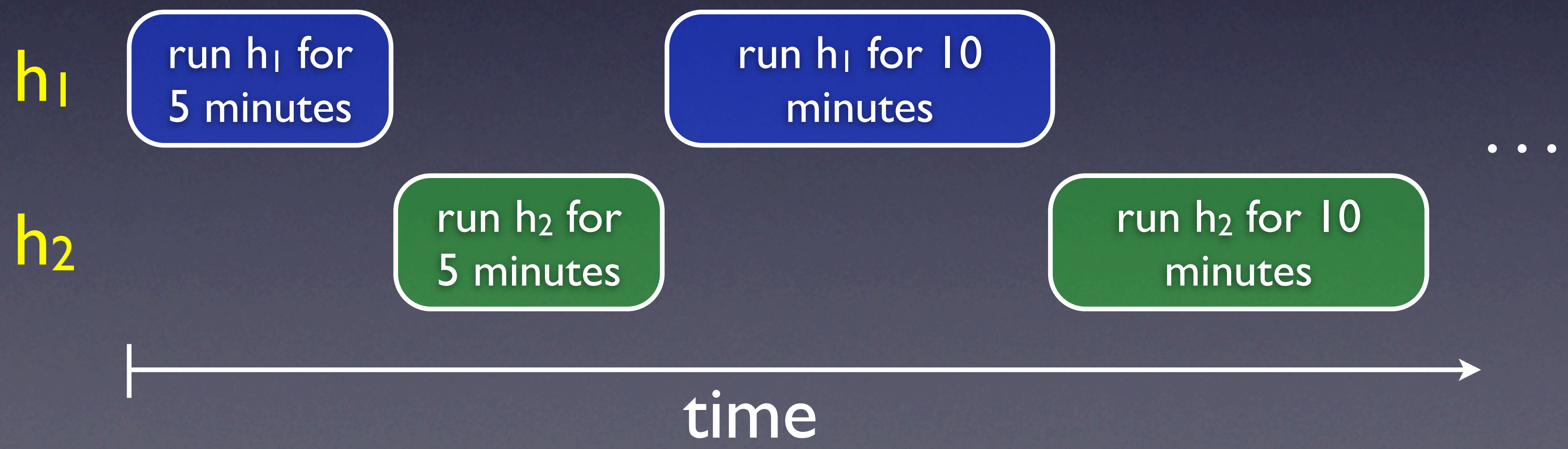
- Running time of randomized heuristics can vary widely across different random seeds



- Periodically restarting with fresh random seed can dramatically improve performance

Schedules

- Schedule = sequence of pairs (h,t) (a pair (h,t) represents running heuristic h for time t)
- Execute in suspend-and-resume model or restart model

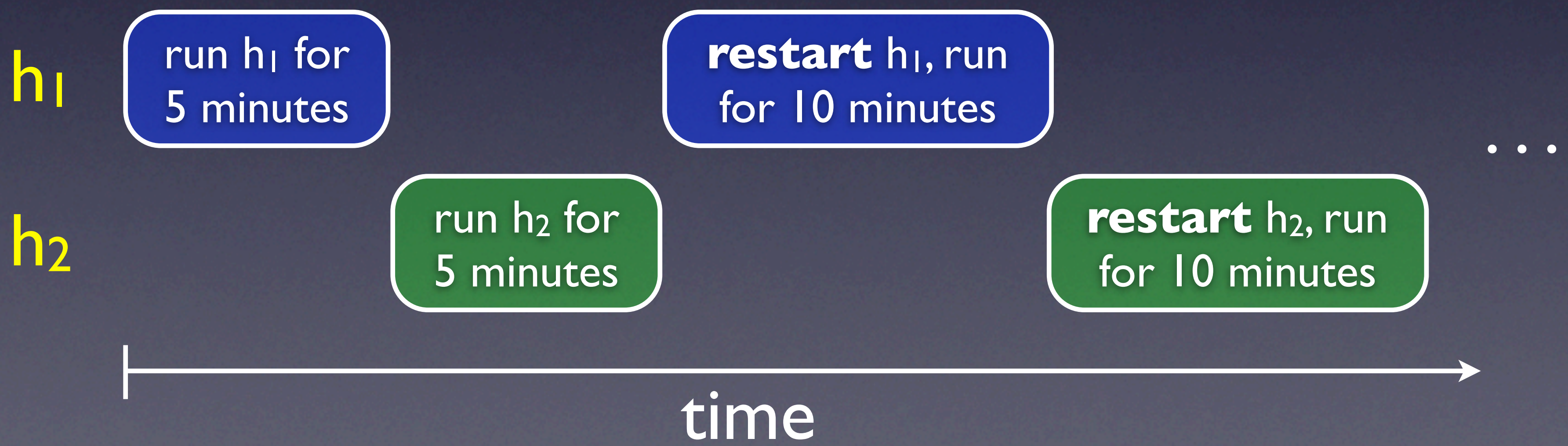


Schedules

- Schedule = sequence of pairs (h,t) (a pair (h,t) represents running heuristic h for time t)
- Execute in **suspend-and-resume model** or restart model

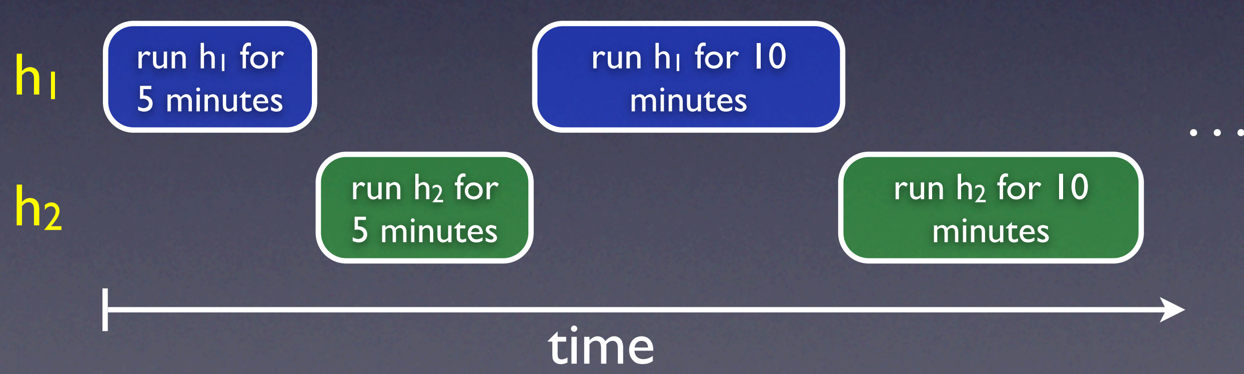
Schedules

- Schedule = sequence of pairs (h,t) (a pair (h,t) represents running heuristic h for time t)
- Execute in suspend-and-resume model or **restart model**



Schedules

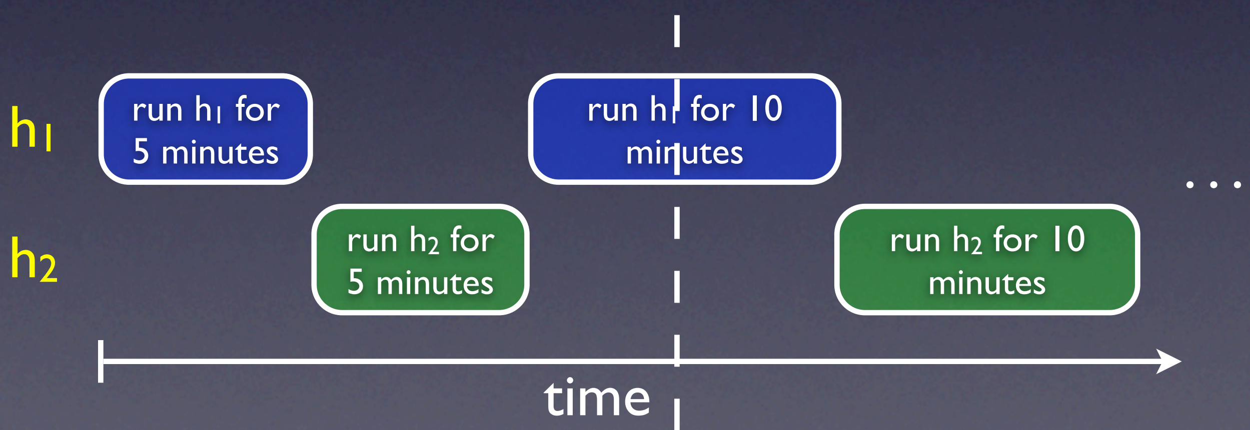
- Schedule = sequence of pairs (h,t) (a pair (h,t) represents running heuristic h for time t)
- Execute in suspend-and-resume model or restart model



Heuristic	Time to solve instance
h_1	10 min
h_2	20 min

Schedules

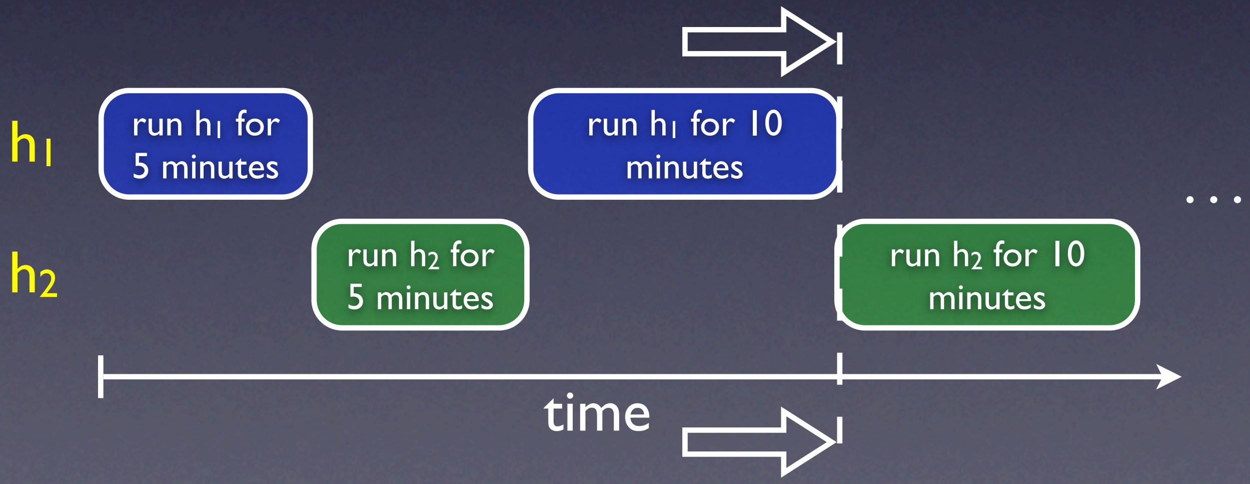
- Schedule = sequence of pairs (h,t) (a pair (h,t) represents running heuristic h for time t)
- Execute in **suspend-and-resume model** or restart model



Heuristic	Time to solve instance
h_1	10 min
h_2	20 min

Schedules

- Schedule = sequence of pairs (h,t) (a pair (h,t) represents running heuristic h for time t)
- Execute in suspend-and-resume model or **restart model**



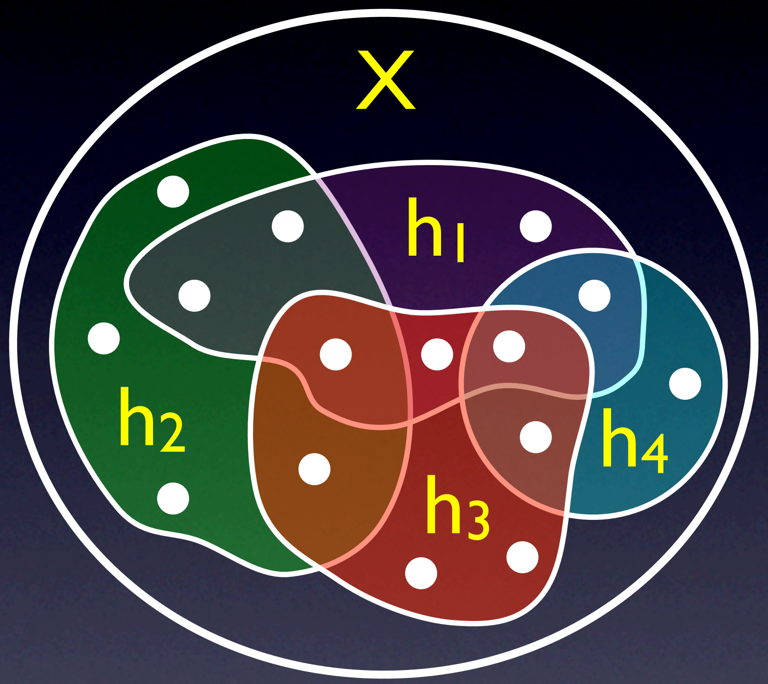
Heuristic	Time to solve instance
h_1	10 min
h_2	20 min

The offline problem

- Given: set H of deterministic heuristics, set X of instances of some decision problem. We know how long each heuristic takes to solve each instance (think of X as training data)
- Goal: construct schedule S that achieves one of two objectives:
  - maximize #(instances solved in time ≤ T), for some fixed T > 0
  - minimize average time to solve each instance

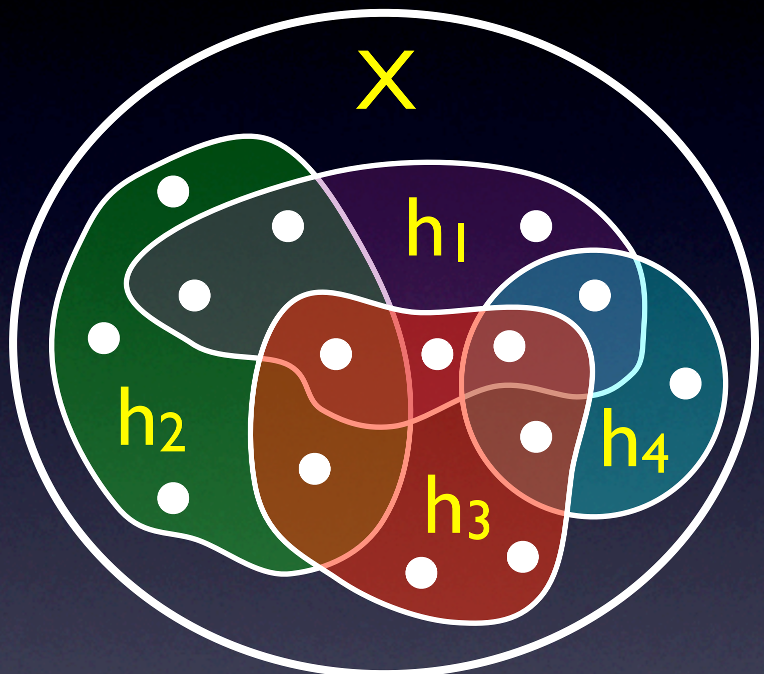
Computational complexity

- Let $H=\{h_1,h_2,...\}$ be a collection of subsets of a finite set X
- Think of each subset $h \in H$ as a heuristic, and each element $x \in X$ as an instance
- h solves x in unit time if $x \in h$, otherwise h never solves x



Computational complexity

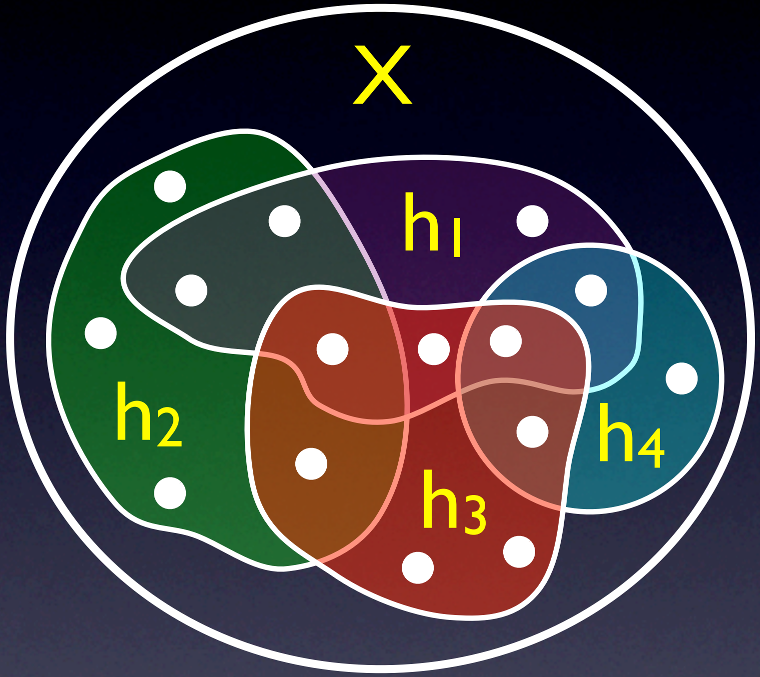
- Let $H=\{h_1,h_2,\ldots\}$ be a collection of subsets of a finite set X
- Think of each subset $h \in H$ as a heuristic, and each element $x \in X$ as an instance
- h solves x in unit time if $x \in h$, otherwise h never solves x



- Maximizing #instances solved in time $\leq T$ is Max k-Coverage ($k=T$). NP-hard to get $1-1/e+\epsilon$ approximation, for any $\epsilon > 0$ (Feige 1997)

Computational complexity

- Let $H=\{h_1,h_2,...\}$ be a collection of subsets of a finite set X
- Think of each subset $h \in H$ as a heuristic, and each element $x \in X$ as an instance
- h solves x in unit time if $x \in h$, otherwise h never solves x



- Maximizing #instances solved in time $\leq T$ is Max *k*-Coverage ($k=T$). NP-hard to get $1-1/e+\epsilon$ approximation, for any $\epsilon > 0$ (Feige 1997)
- Minimizing avg. time to solve each instance is Min-Sum Set Cover. NP-hard to get $4-\epsilon$ approximation, for any $\epsilon > 0$ (Feige *et al.*, 2004)

Greedy algorithm

- Let f(S) = #(instances solved by schedule S) (in restart model or suspend-and-resume, whichever we care about)
- Let G = empty schedule
- While f(G) < |X|:
    - Find the pair a = (h,t) maximizing [f(G + a) - f(G)] / t, and append it to G

Greedy algorithm

- Let f(S) = #(instances solved by schedule S) (in restart model or suspend-and-resume, whichever we care about)
- Let G = empty schedule
- While f(G) < |X|:
  - Find the pair a = (h,t) maximizing [f(G + a) - f(G)] / t, and append it to G

- Average CPU time for G at most 4 times optimal. Proof generalizes analysis of greedy algorithm for Min-Sum Set Cover by Feige *et al.* (2004)
- #(instances solved in time T) at least 1-1/e times optimal, for certain values of T. Follows from Khuller *et al.* (1999)

The online problem

- Given: set H of heuristics, fed sequence $x_1, x_2, ..., x_n$ of n instances
- Solve each x_i (via some schedule) before moving on to x_{i+1}. Only learn outcomes of runs we actually perform.
- Goal is to achieve one of two objectives:
  - maximize #(instances solved in time $\leq T$), for some fixed $T > 0$
  - minimize average time to solve each instance
- Assume for each x_i, some heuristic can solve in time $\leq B$. Also, time each heuristic takes is integer.

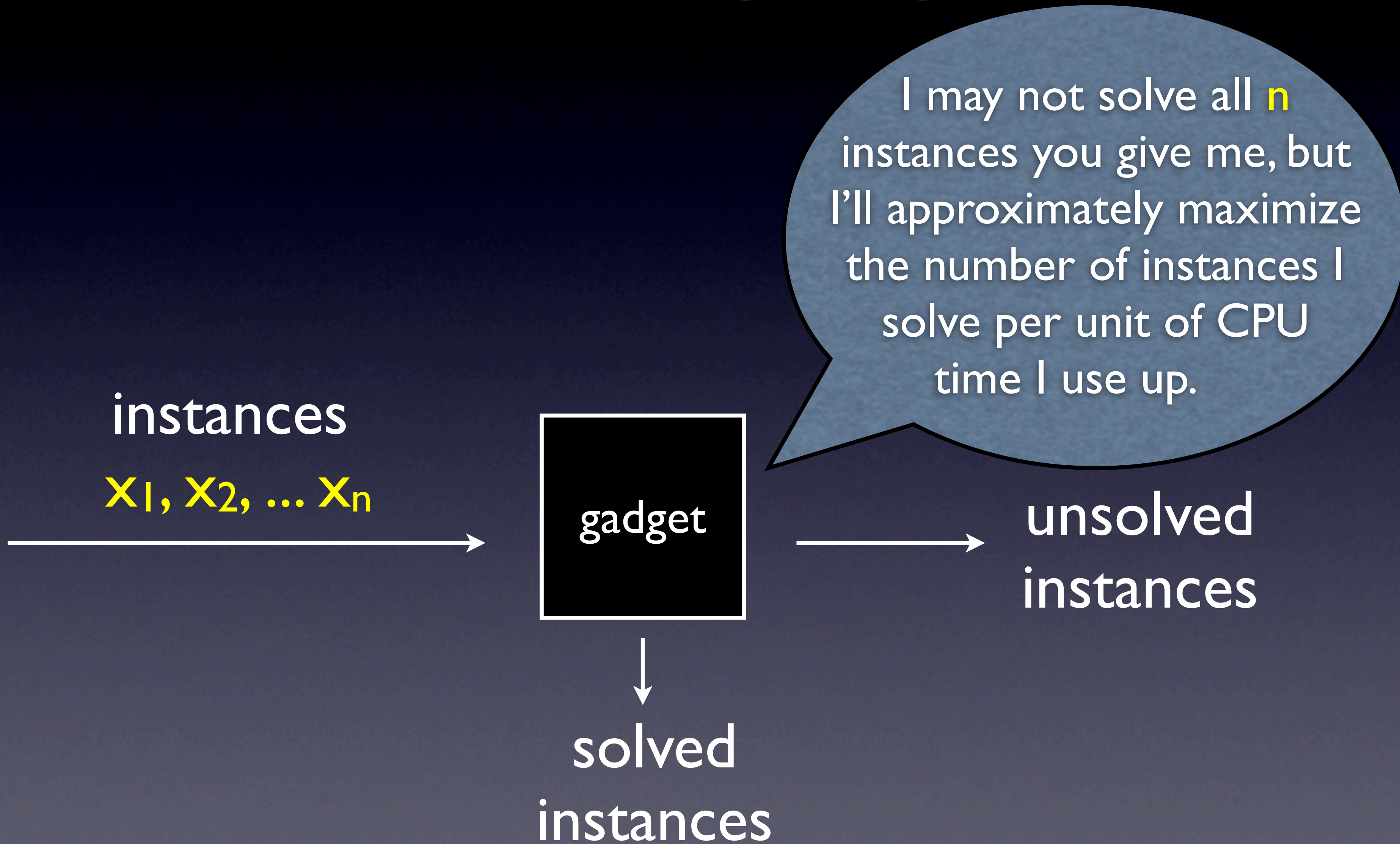
A solved problem

- Suppose instead of picking a schedule, you get to pick *one* heuristic and run it for unit time. Want to maximize #(instances solved)
- Define regret = $\max_{h \in H}$ #(instances h can solve in unit time) - #(instances you solve)
- Any online schedule-selection algorithm has worst-case regret $\geq n(1-1/k)$, where $k=|H|$
- But, **Exp3** algorithm (Auer *et al.*, 2002) has worst-case *expected* regret $O((n\ k \log k)^{1/2})$

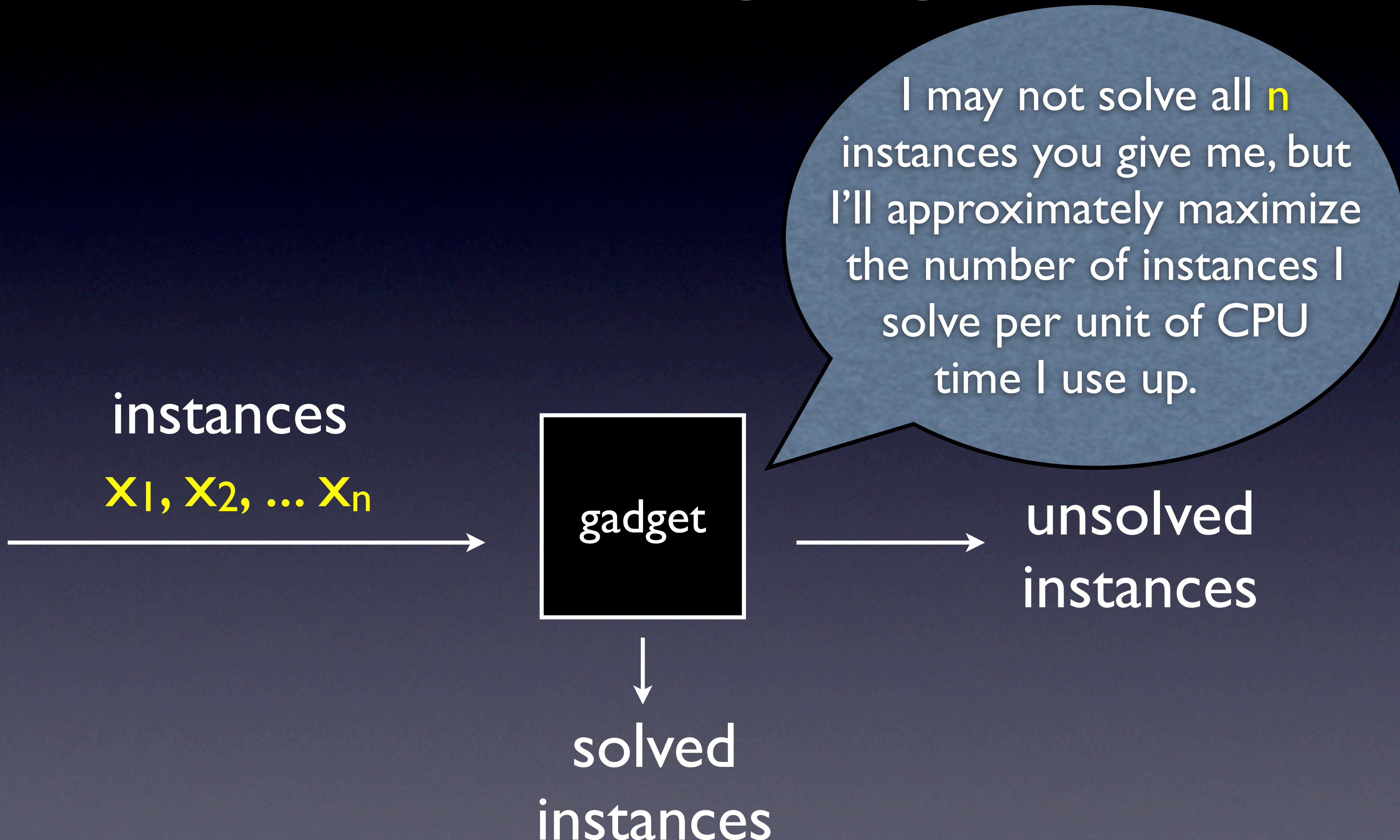
A useful gadget

- Suppose you still have to pick one heuristic, but now can run for unit time *in expectation*
- For example, could flip coin of bias 1/t, if heads run h for time t. Call this "action (h,t)"
- Using **Exp3** to pick actions, worst-case expected regret is $O((n A \log A)^{1/2})$, where regret now defined in terms of actions and A = #actions.
- Some algebra shows E[#(instances we solve)] is $\geq \max_{h,t}$ { #(instances solved by h in time t) / t } - E[regret] So we're maximizing # instances solved per unit time...

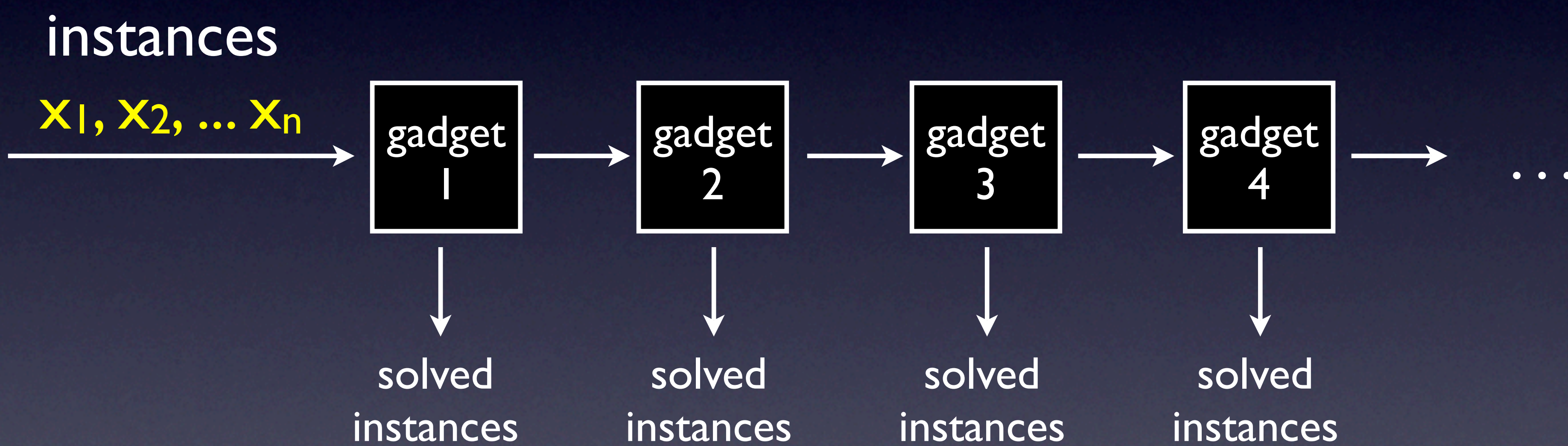
A useful gadget



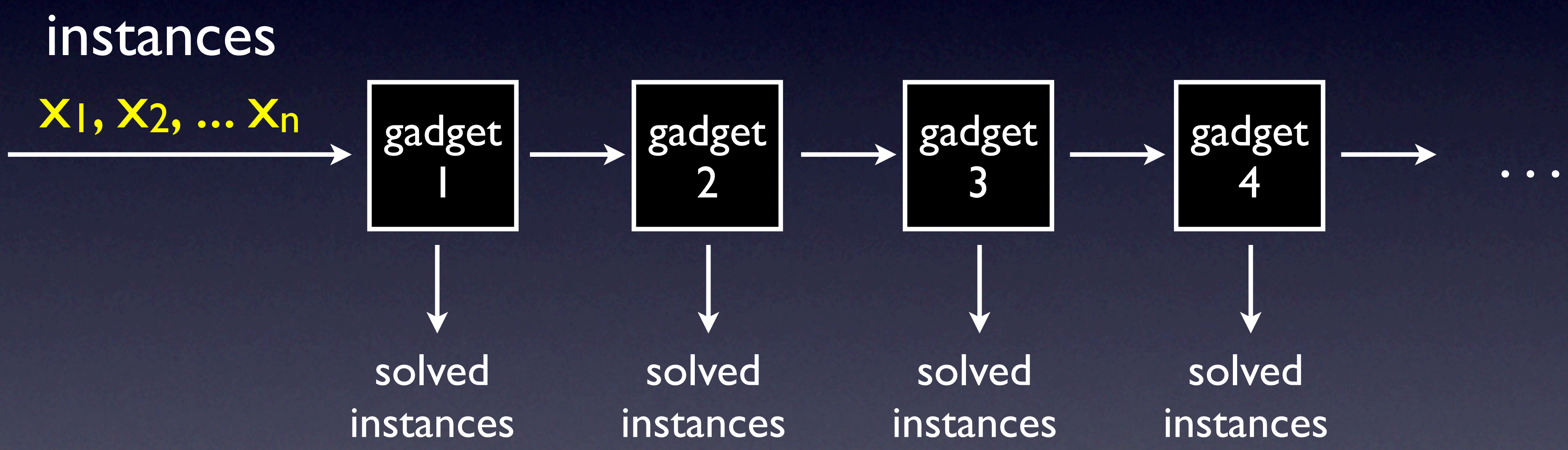
A useful gadget



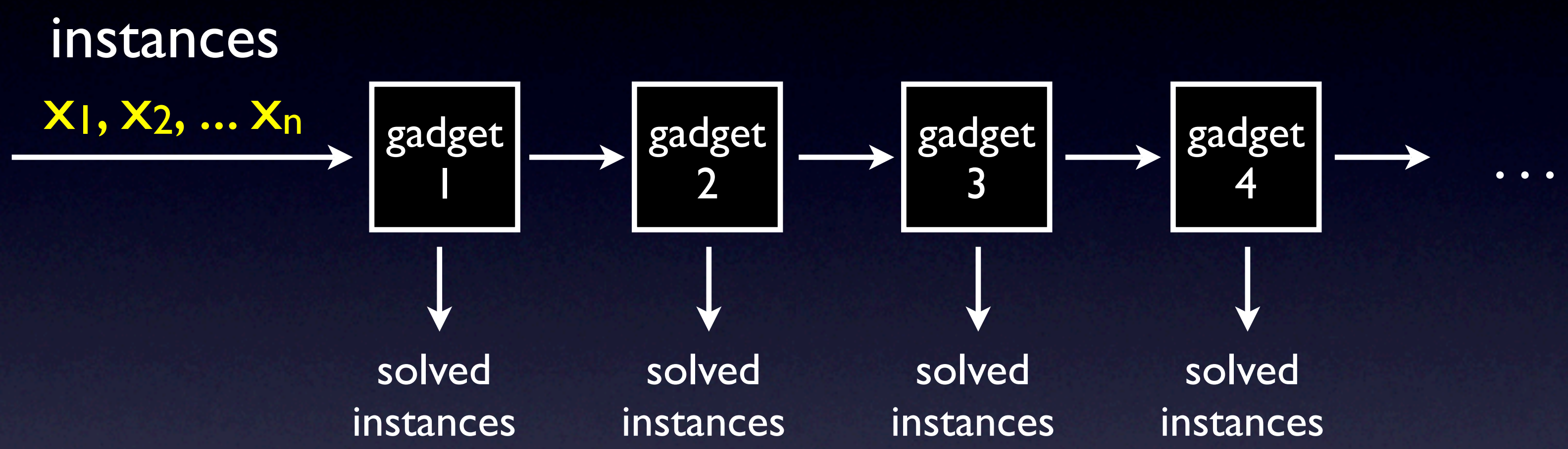
Online greedy algorithm



Online greedy algorithm



Online greedy algorithm



- As $n \to \infty$, online algorithm's performance guarantees converge to those of offline greedy algorithm
- Analysis views online algorithm as variant of offline greedy algorithm

Exploiting features

- Suppose each instance is labeled with the values of one or more Boolean features

Instance	industrial/ academic	small/ large
x_1	industrial	large
x_2	industrial	small
x_3	academic	large

Exploiting features

- Suppose each instance is labeled with the values of one or more Boolean features
- Let X_F = subsequence of instances with feature F
- Can get the following guarantee: simultaneously for each feature F, performance on X_F converges to that of offline greedy schedule for instances in X_F
  - Get this guarantee using known technique: use algorithms for *sleeping experts problem* (Freund *et al.*, 1997; Blum & Mansour 2007) as wrapper around multiple copies of online greedy algorithm

Randomized heuristics

- All results extend to randomized heuristics
- Can have some heuristics execute in restart model, others in suspend-and-resume

Other theoretical results

- Offline and online algorithms based on shortest paths
- Generalization bounds for learning a schedule from training data
- Lower bounds on regret for online schedule-selection problem

Previous work

- Algorithm portfolios
  - Idea of using schedules to improve average-case, offline algorithms for special cases (Huberman *et al.*, 1997; Gomes & Selman 2001, ...)
  - Using features to pick out a single heuristic (Leyton-Brown *et al.*, 2003; Xu *et al.*, 2007, ...)
- Restart schedules for single randomized algorithm (Luby *et al.,* 1993; Gomes *et al.*,1998, ...)
- Exponential-time offline algorithms for computing task-switching schedules (Petrik 2005; Sayag *et al.*, 2006)

Contributions

- New techniques for combining heuristics
  - consider a class of schedules that **generalizes** schedules considered in previous work
  - first **polynomial-time** approximation algorithms for constructing these schedules
  - **online algorithms** for selecting schedules on-the-fly while solving a sequence of problems
  - can exploit **features** in a principled way

Solver competitions

- Each year, various conferences hold solver competitions
  - Each submitted solver is run on a set of benchmark instances, subject to per-instance time limit
  - Solvers judged on how many instances they solve and how fast
- How would schedules created by our algorithms have fared in the competitions?
  - determine running time of each heuristic on each instance using data from competition web sites
  - removed instances that no solver could solve

Solver competitions

Competition	Problem domain
SAT 2007	Boolean satisfiability
SMT-COMP'07	satisfiability modulo theories
CASC-J3	theorem proving
MaxSAT-2007	maximum satisfiability
PB'07	zero-one integer programming
QBFEVAL'07	quantified Boolean formulae
CPAI'06	constraint satisfaction
IPC-5	A.I. planning

Results for SAT 2007, *random* category

Solver	**Avg. CPU** [lower,upper]	**Num. solved**
adaptg2wsat+	[2157,∞]	252
adaptg2wsat0	[2204,∞]	248
SATzilla	[2275,∞]	248
ranov	[2288,∞]	242
March KS	[2305,∞]	257
adaptnovelty	[2331,∞]	240
gnovelty+	[2359,∞]	242
KCNFS	[2554,∞]	237
sapsrt	[2804,∞]	188
MXC	[3642,∞]	135
minisat	[3676,∞]	140
SAT7	[3761,∞]	122
DEWSATZ 1A	[3797,∞]	121
MiraXTv3	[3940,∞]	106

Offline algorithms

Results for SAT 2007, *random* category

Solver	**Avg. CPU** [lower,upper]	**Num. solved**
Fastest individual solver	[2157,∞]	252

Offline algorithms

Results for SAT 2007, *random* category

Solver	**Avg. CPU** [lower,upper]	**Num. solved**
Parallel schedule	[1775,7571]	302
Fastest individual solver	[2157,∞]	252

Offline algorithms

Results for SAT 2007, *random* category

Solver	**Avg. CPU** [lower,upper]	**Num. solved**
Greedy schedule (restart)	[1320,3657]	342
Parallel schedule	[1775,7571]	302
Fastest individual solver	[2157,∞]	252

Offline algorithms

Results for SAT 2007, *random* category

Solver	**Avg. CPU** [lower,upper]	**Num. solved**
Greedy schedule (suspend)	[1223,2372]	350
Greedy schedule (restart)	[1320,3657]	342
Parallel schedule	[1775,7571]	302
Fastest individual solver	[2157,∞]	252

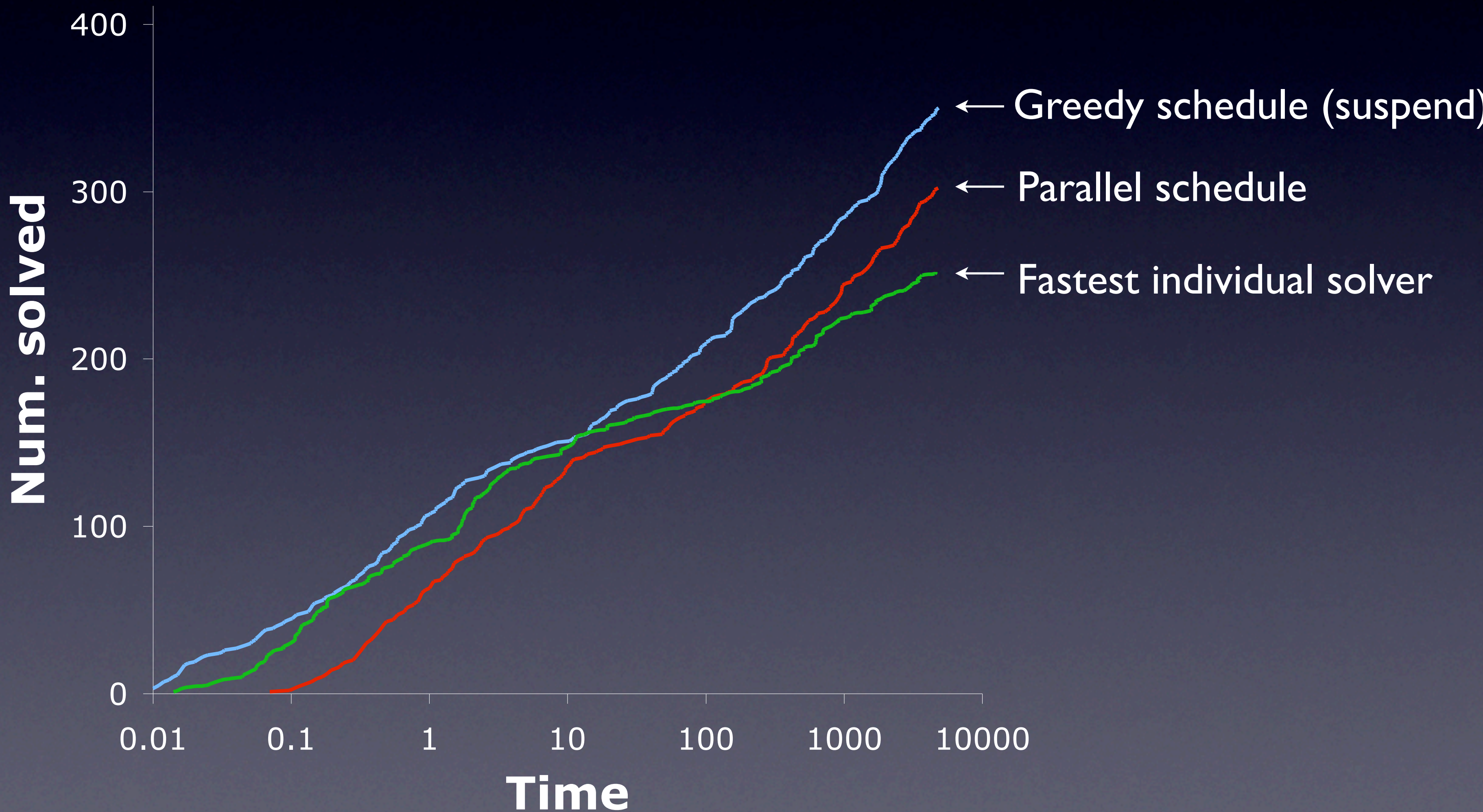
Offline algorithms

Results for SAT 2007, *random* category

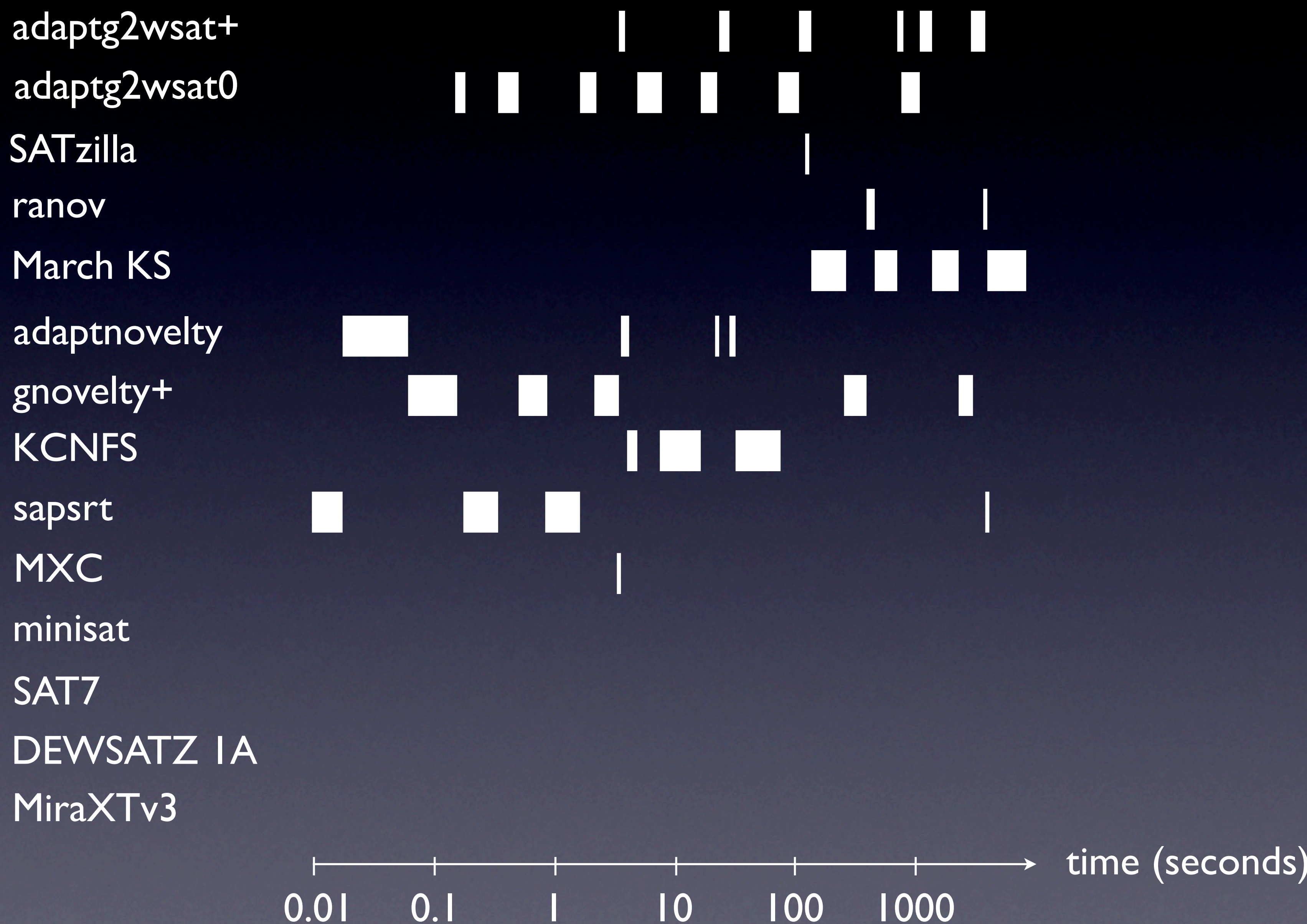
Solver	**Avg. CPU** [lower,upper]	**Num. solved**
Greedy schedule (suspend)	[1223,2372]	350
Greedy schedule (suspend) crossval	*[1337,3252]*	*344*
Greedy schedule (restart)	[1320,3657]	342
Greedy schedule (restart) crossval	*[1342,4804]*	*340*
Parallel schedule	[1775,7571]	302
Fastest individual solver	[2157,∞]	252

Offline algorithms

Results for SAT 2007, *random* category



Greedy schedule (restart model) for SAT 2007, *random* category



Online algorithms

- We consider two feedback models
  - *Full information:* after solving x_i, we learn how long each heuristic would have taken to solve x_i
  - *Partial information:* only learn outcome of runs we actually perform

Online algorithms

- We consider two feedback models
  - *Full information:* after solving x_i, we learn how long each heuristic would have taken to solve x_i
  - *Partial information:* only learn outcome of runs we actually perform
- Evaluate online greedy algorithm in both models
  - In *full info* model, gadget uses self-tuning version of **WMR** (Auer & Gentille, 2000)
  - In *partial info* model, gadget uses self-tuning version of **Exp3** (Auer *et al.*, 2002)

Online algorithms

- We consider two feedback models
  - *Full information:* after solving x_i, we learn how long each heuristic would have taken to solve x_i
  - *Partial information:* only learn outcome of runs we actually perform
- Also evaluate online algorithms that solve each instance by choosing a *single* heuristic to run
  - In *full info* model, use self-tuning version of **WMR**
  - In *partial info* model, use self-tuning version of **Exp3**

Online algorithms

Results for SAT 2007, *random* category

Solver	**Avg. CPU** [lower,upper]	**Num. solved**
Greedy schedule (suspend)	[1223,2372]	350
Greedy schedule (suspend) cross-val	*[1337,3252]*	*344*
Parallel schedule	[1775,7571]	302
Fastest individual solver	[2157,∞]	252

Online algorithms

Results for SAT 2007, *random* category

Solver	**Avg. CPU** [lower,upper]	**Num. solved**
Greedy schedule (suspend)	[1223,2372]	350
Greedy schedule (suspend) cross-val	*[1337,3252]*	*344*
Parallel schedule	[1775,7571]	302
Fastest individual solver	[2157,∞]	252
Online single-heur (full info)	[2184,∞]	255
Online single-heur (partial info)	[2835,∞]	191

Online algorithms

Results for SAT 2007, *random* category

Solver	**Avg. CPU** [lower,upper]	**Num. solved**
Greedy schedule (suspend)	[1223,2372]	350
Online greedy (full info)	[1304,4261]	347
Greedy schedule (suspend) cross-val	*[1337,3252]*	*344*
Parallel schedule	[1775,7571]	302
Fastest individual solver	[2157,∞]	252
Online single-heur (full info)	[2184,∞]	255
Online single-heur (partial info)	[2835,∞]	191

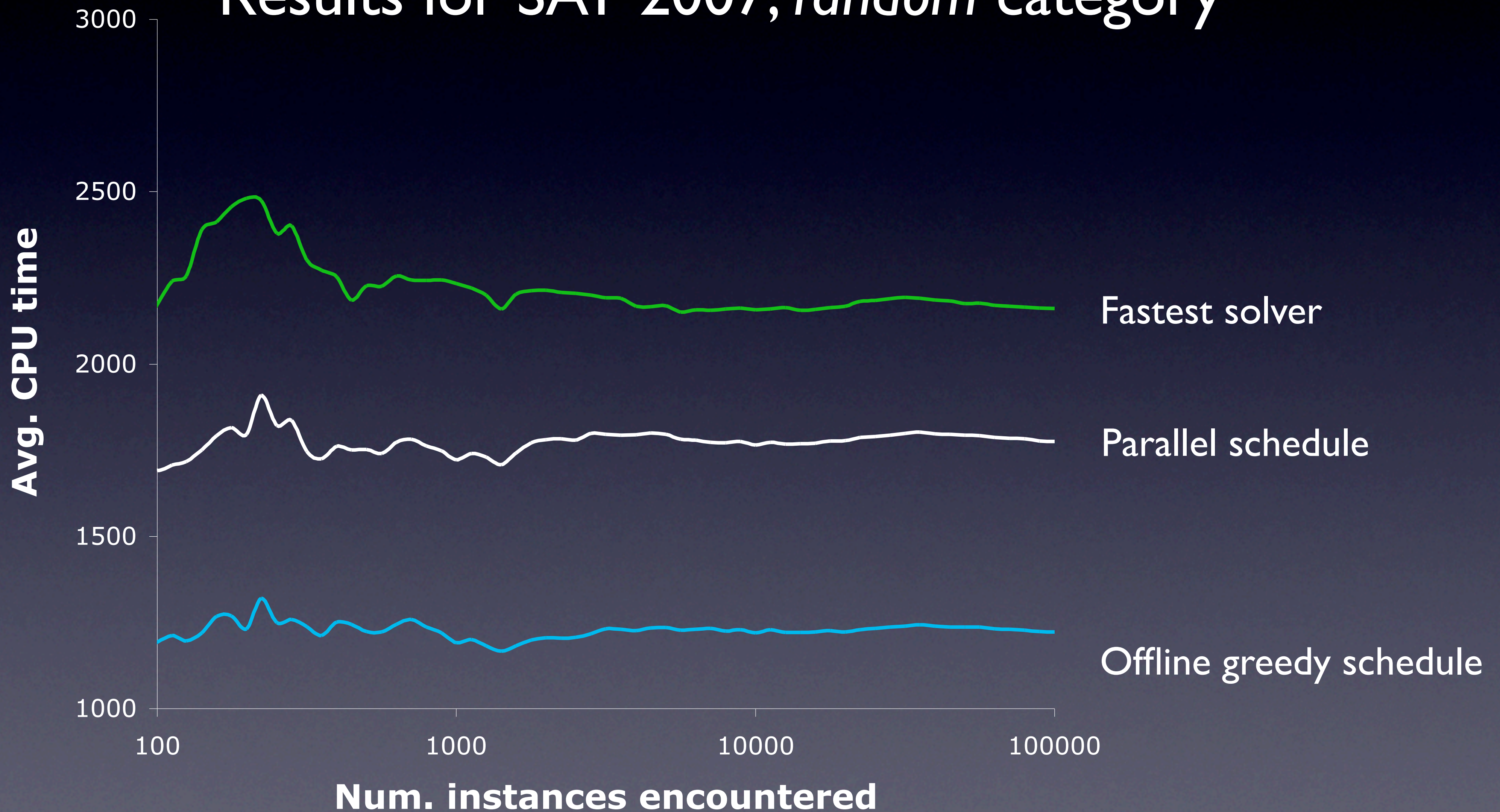
Online algorithms

Results for SAT 2007, *random* category

Solver	**Avg. CPU** [lower,upper]	**Num. solved**
Greedy schedule (suspend)	[1223,2372]	350
Online greedy (full info)	[1304,4261]	347
Greedy schedule (suspend) cross-val	*[1337,3252]*	*344*
Parallel schedule	[1775,7571]	302
Online greedy (partial info)	[2050,8127]	294
Fastest individual solver	[2157,∞]	252
Online single-heur (full info)	[2184,∞]	255
Online single-heur (partial info)	[2835,∞]	191

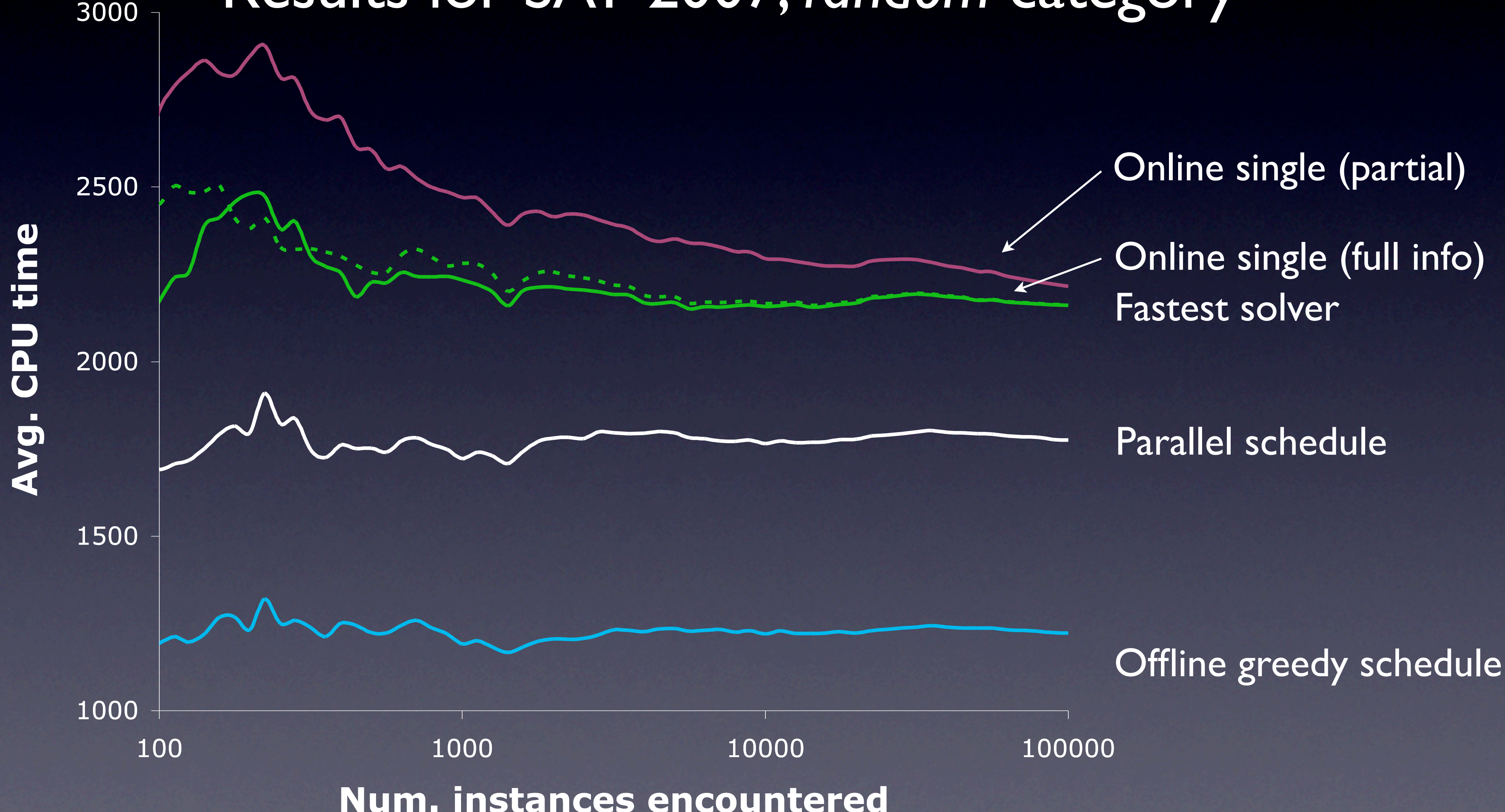
Online algorithms

Results for SAT 2007, *random* category



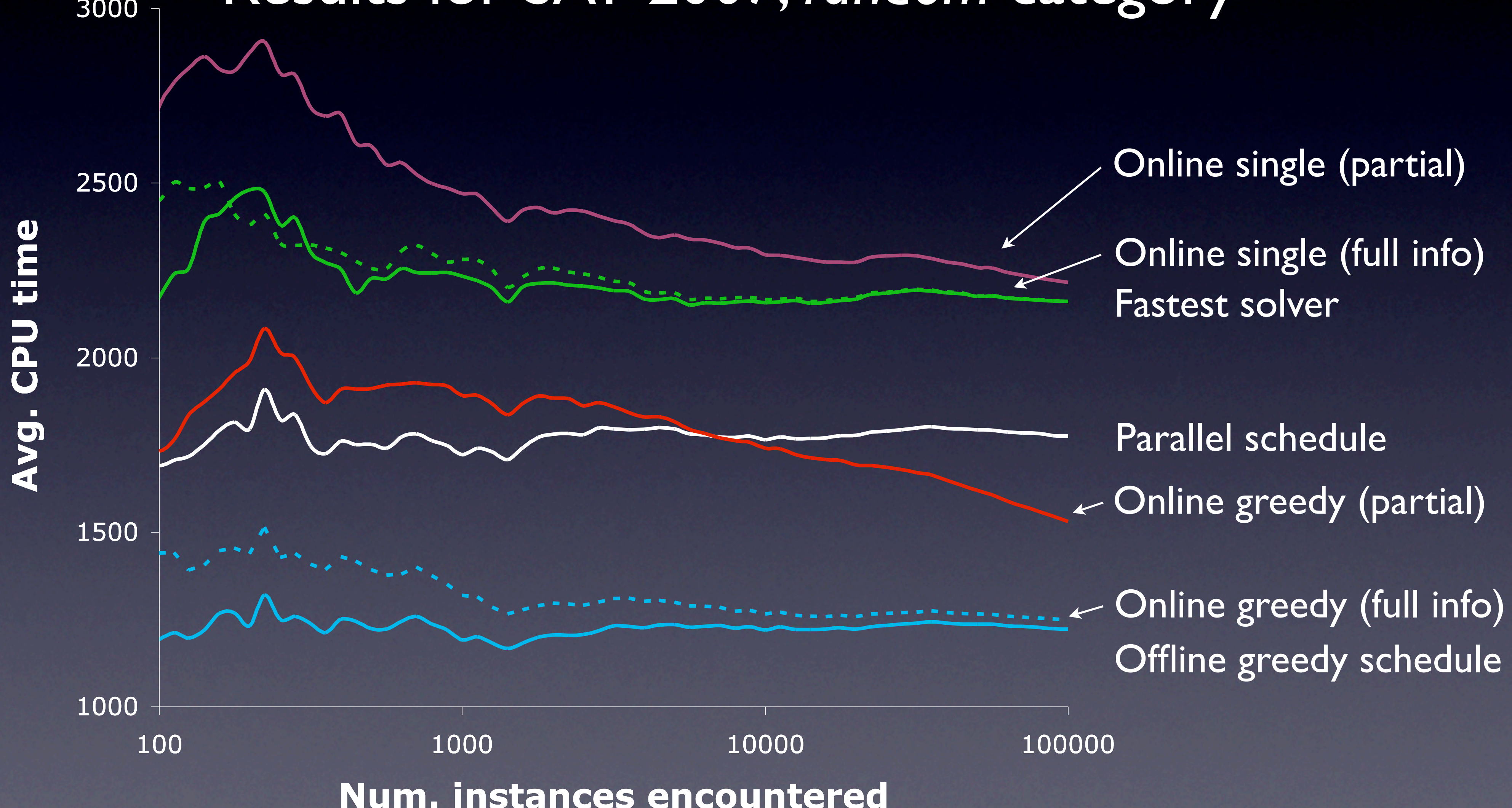
Online algorithms

Results for SAT 2007, *random* category



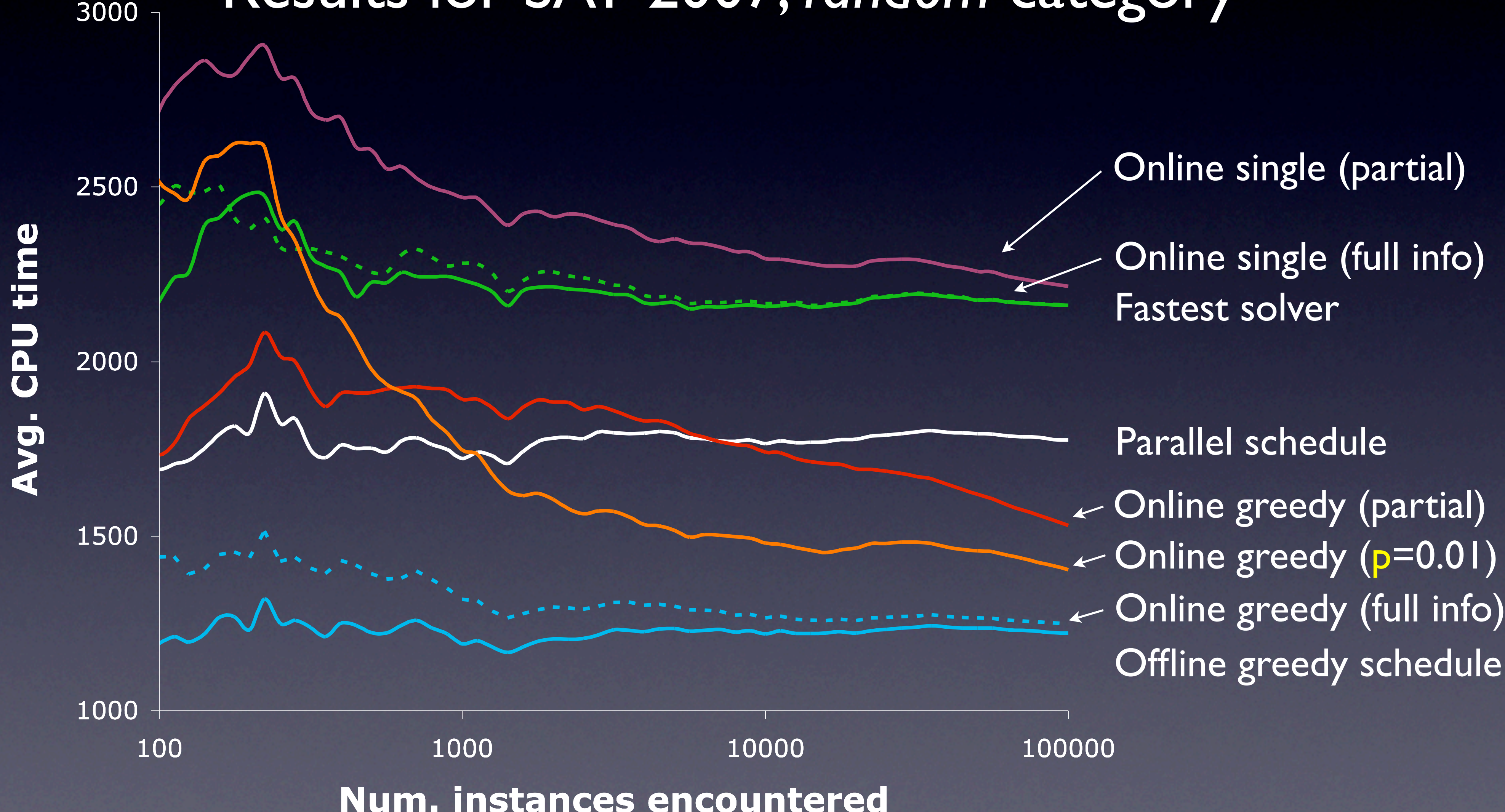
Online algorithms

Results for SAT 2007, *random* category



Online algorithms

Results for SAT 2007, *random* category



Exploiting features

- Created features based on competition benchmark directory structure
- For each subdirectory, have feature that is true if instance resides under that directory

```
SAT 2007, random
  ├─ Large
  │    ├─ 3SAT
  │    │    └── [40 instances]
  │    └─ 5SAT
  │         └── [60 instances]
  └─  ...
```

Exploiting features

Results for SAT 2007, *random* category

Solver	**Avg. CPU** [lower,upper]	**Num. solved**
Greedy schedule	[1223,2372]	350
Online greedy (full info)	[1304,4261]	347
Greedy schedule (cross-val)	[1337,3252]	344
Parallel schedule	[1775,7571]	302
Online greedy (partial info)	[2050,8127]	294
Fastest individual solver	[2157,∞]	252
Online single-heur (full info)	[2184,∞]	255
Online single-heur (partial info)	[2835,∞]	191

Exploiting features

Results for SAT 2007, *random* category

Solver	**Avg. CPU** [lower,upper]	**Num. solved**
Online greedy (full info) + features	[1044,3262]	365
Greedy schedule	[1223,2372]	350
Online greedy (full info)	[1304,4261]	347
Greedy schedule (cross-val)	[1337,3252]	344
Parallel schedule	[1775,7571]	302
Online greedy (partial info)	[2050,8127]	294
Fastest individual solver	[2157,∞]	252
Online single-heur (full info)	[2184,∞]	255
Online single-heur (partial info)	[2835,∞]	191

Speedup factors

- Speedup factor = ratio of (lower bound on) best solver's avg. CPU time to that of greedy schedule (suspend-and-resume, crossval)

Results for SAT 2007

Category	**Speedup factor**	**Speedup factor w/features**
random	1.61	2.24
hand-crafted	1.37	1.49
industrial	0.99	1.20

Speedup factors

Competition	**Speedup factor** (range across categories)	**Speedup factor w/features** (range across categories)
Boolean satisfiability	0.99 - 1.61	1.3 - 2.24
Satisfiability modulo theories	0.25 - 15.1	0.25 - 15.1
A.I. planning	1.61	1.78
Constraint satisfaction	0.28 - 2.10	0.28 - 3.03
Maximum satisfiability	0.82 - 1.31	0.99 - 1.68
0/1 integer programming	0.98 - 2.71	1.1 - 3.09
Quantified Boolean formulae	0.81 - 2.19	0.81 - 2.19
Theorem proving	0.56 - 5.49	0.58 - 4.83

Other experimental results

- Optimization heuristics
  - suppose heuristics are *anytime* algorithms that return solutions of decreasing cost over time
  - can modify objective function to get schedules with good anytime behavior
  - good results for 0/1 int. programming competition
- Randomized heuristics
  - we develop an improved restart schedule for the SAT solver satz-rand

Online Algorithms for Maximizing Submodular Functions

Generalizing the greedy algorithm

- Greedy algorithm for combining heuristics (offline + online) can be generalized to solve wider class of problems
- Instance x becomes function from schedules to [0,1], satisfying certain conditions. Sufficient conditions based on *submodularity*

Problems that fit into this framework

	Problem	References
cost-minimization	Min-Sum Set Cover	Feige *et al.* (2004)
cost-minimization	Pipelined Set Cover	Munagala *et al.* (2005), Kaplan *et al.* (2005)
cost-minimization	Efficient sequences of trials	Cohen *et al.* (2003)
coverage-maximization	Maximizing a monotone, submodular set function subject to knapsack constraint	Sviridenko (2004), Krause & Guestrin (2005)
coverage-maximization	Budgeted Maximum Coverage	Khuller *et al.* (1999)
coverage-maximization	Max *k*-Coverage	Nemhauser *et al.* (1978)

Problems that fit into this framework

	Problem	References
	Min-Sum Set Cover	Feige *et al.* (2004)
cost-minimization	Pipelined Set Cover	Munagala *et al.* (2005), Kaplan *et al.* (2005)
	Efficient sequences of trials	Cohen *et al.* (2003)
	Maximizing a monotone, submodular set function subject to knapsack constraint	Sviridenko (2004), Krause & Guestrin (2005)
coverage-maximization	Budgeted Maximum Coverage	Khuller *et al.* (1999)
	Max k-Coverage	Nemhauser *et al.* (1978)

- Applications to database query processing, sensor placement, and market-sharing games

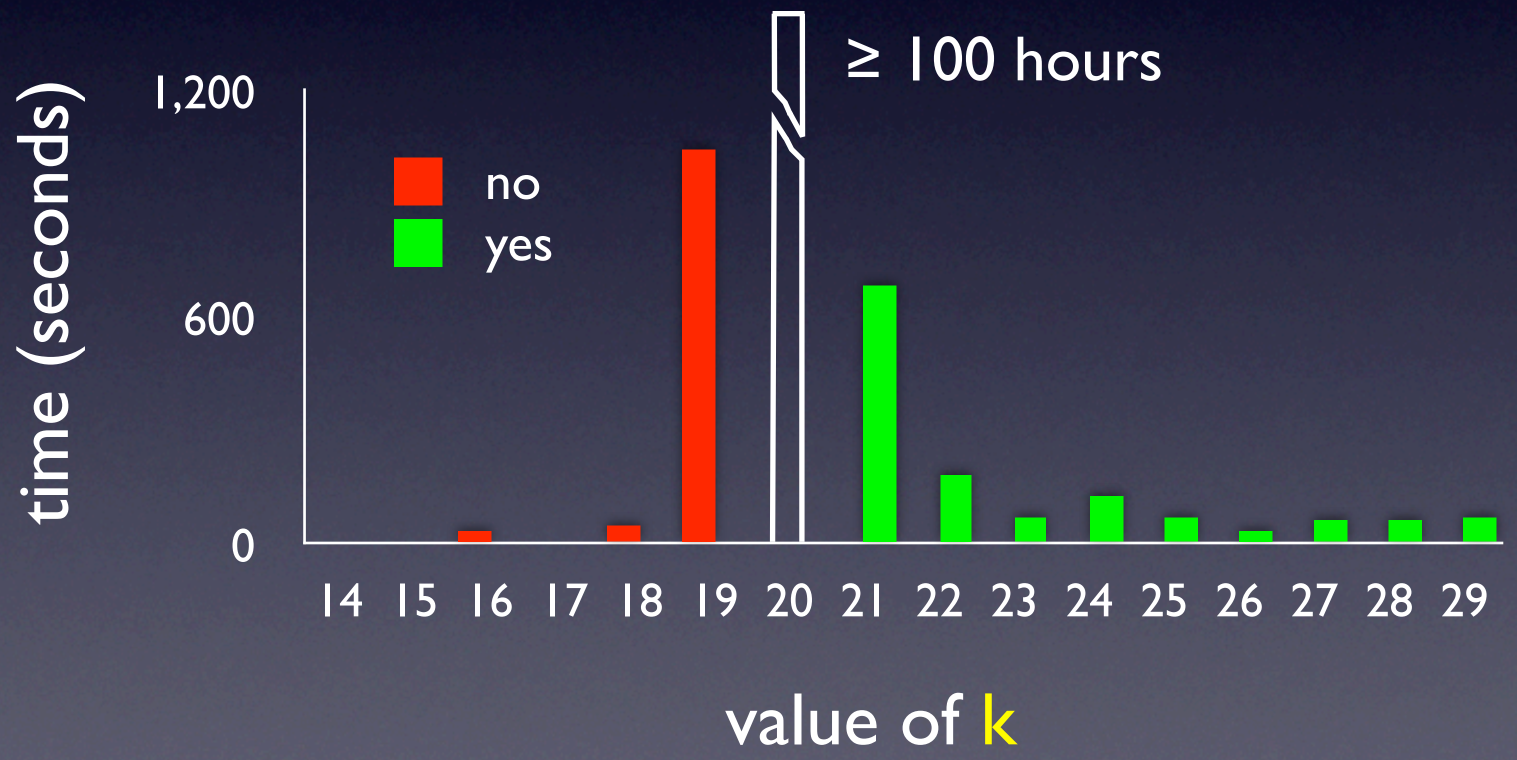
Using Decision Procedures Efficiently for Optimization

Introduction

- Optimization problems can be solved by asking a decision procedure questions of the form "is there a solution of cost ≤ k?"
- E.g., state-of-the art algorithms for A.I. planning use SAT solver to determine if plan of length ≤ k exists
- How to decide which questions to ask?
  - SATPLAN starts from k=1 and works upward
  - Maxplan starts from upper bound and works downward
  - Is there a better way?

Motivations

- Query strategy can dramatically affect time needed to find (provably) approximately optimal solution



Query Strategies

- A *query* (k,t) runs the decision procedure with time limit t, and asks it "is there a solution of cost ≤ k?" Result can be *yes*, *no*, or *timeout.*
- A *query strategy* determines the next query to execute, as a function of the results of previous queries

Query Strategies

- A *query* (k,t) runs the decision procedure with time limit t, and asks it "is there a solution of cost ≤ k?" Result can be *yes*, *no*, or *timeout.*
- A *query strategy* determines the next query to execute, as a function of the results of previous queries
- Notation:
  - $\tau(k)$ = time required by decision proc. on input k
  - OPT = minimum solution cost

Metrics & Assumptions

Metrics & Assumptions

- Performance metric: worst-case competitive ratio. Equals max, over all k, of

$$\frac{\text{time required to prove } k \leq \text{OPT or } k \geq \text{OPT}}{\tau(k)}$$

Metrics & Assumptions

- Performance metric: worst-case competitive ratio. Equals max, over all k, of

$$\frac{\text{time required to prove } k \le OPT \text{ or } k \ge OPT}{\tau(k)}$$

- Without any assumptions about τ(k), can't do better than trying all k-values in parallel. Competitive ratio = #(possible k-values)

Metrics & Assumptions

- Performance metric: worst-case competitive ratio. Equals max, over all k, of

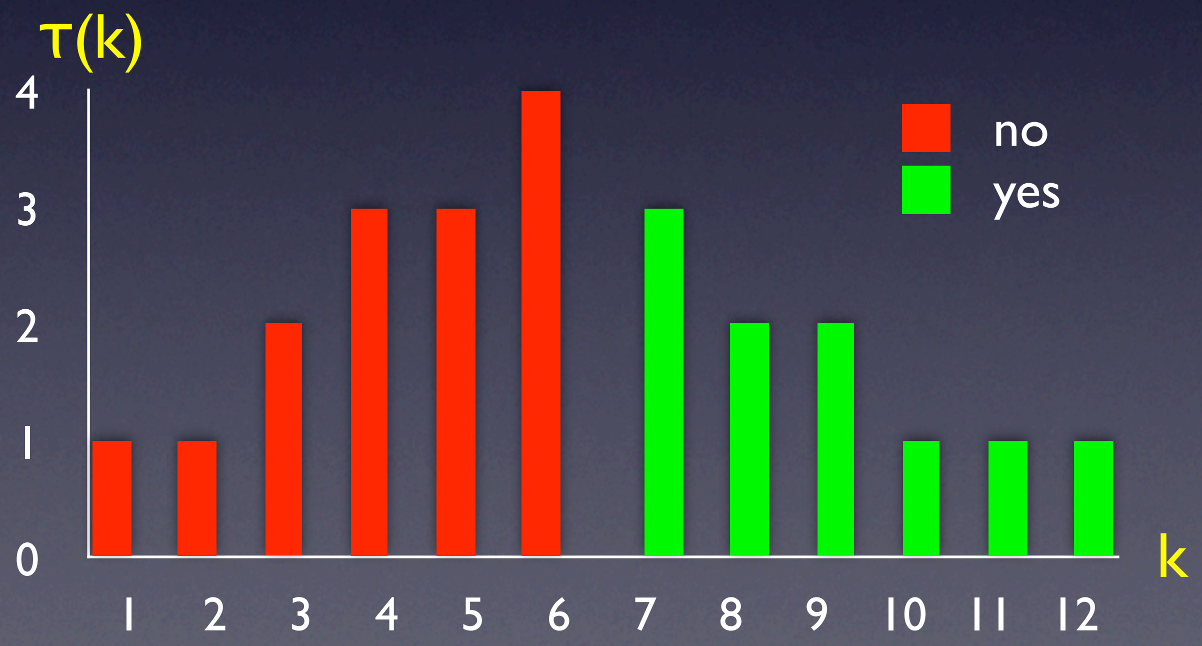
$$\frac{\text{time required to prove } k \le \text{OPT or } k \ge \text{OPT}}{\tau(k)}$$

- Without any assumptions about $\tau(k)$, can't do better than trying all k-values in parallel. Competitive ratio = #(possible k-values)

- We'll assume $\tau(k)$ is (approximately) increasing-then-decreasing

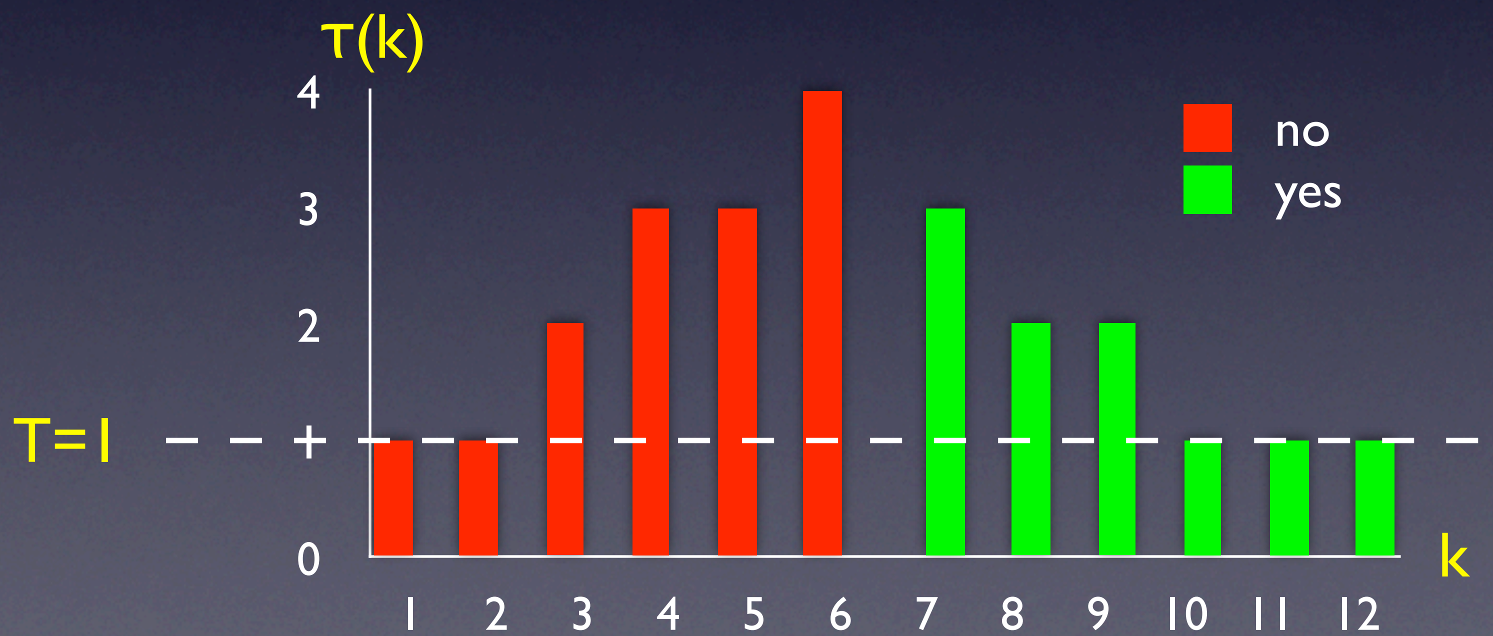
Query strategy S_2

- Initialize $T \leftarrow 1$
- Use two-sided binary search to find range of k-values such that $\tau(k) > T$
- Double T and repeat



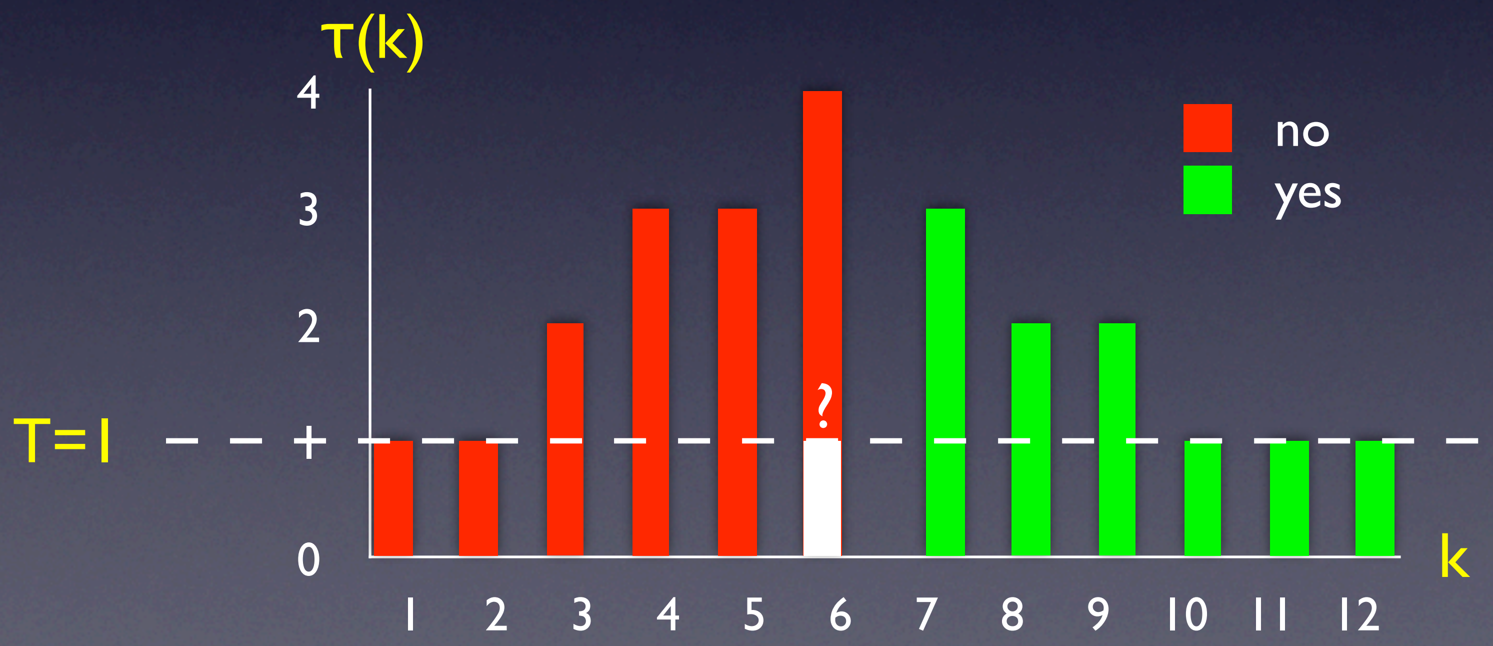
Query strategy S_2

- Initialize $T \leftarrow 1$
- Use two-sided binary search to find range of k-values such that $\tau(k) > T$
- Double T and repeat



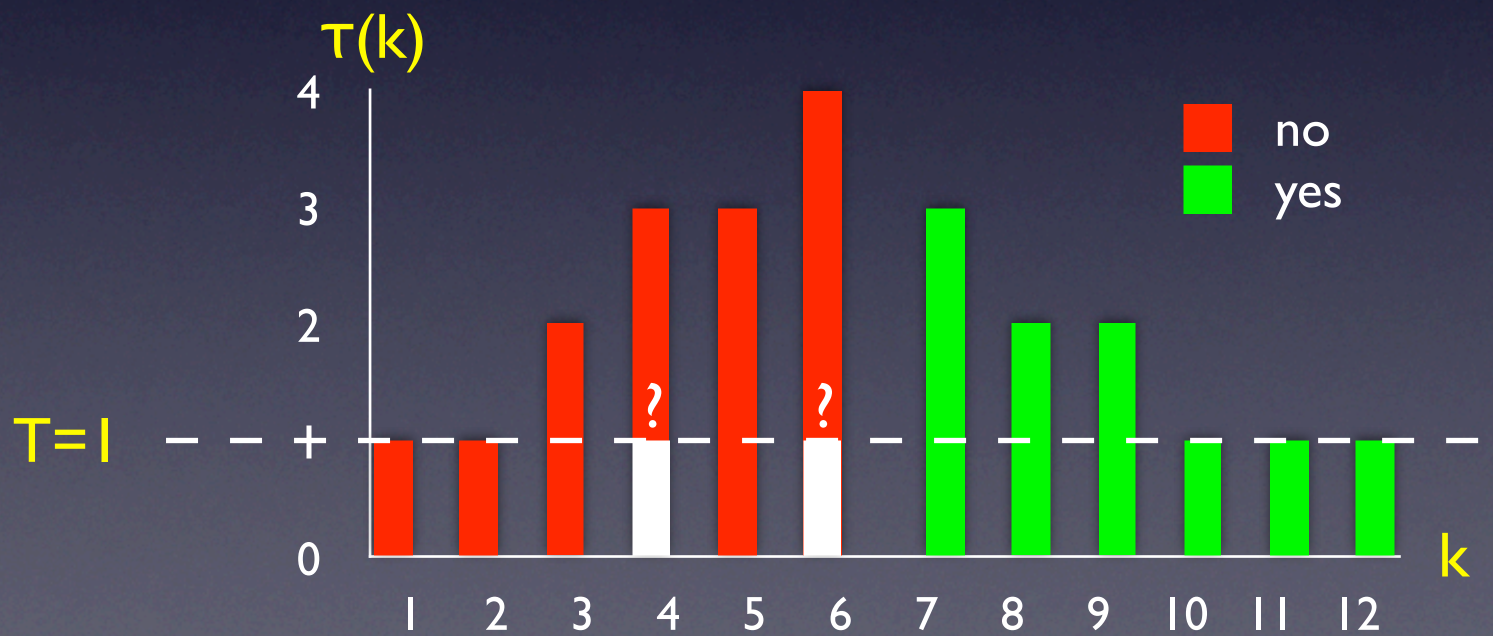
Query strategy S_2

- Initialize $T \leftarrow 1$
- Use two-sided binary search to find range of k-values such that $\tau(k) > T$
- Double T and repeat



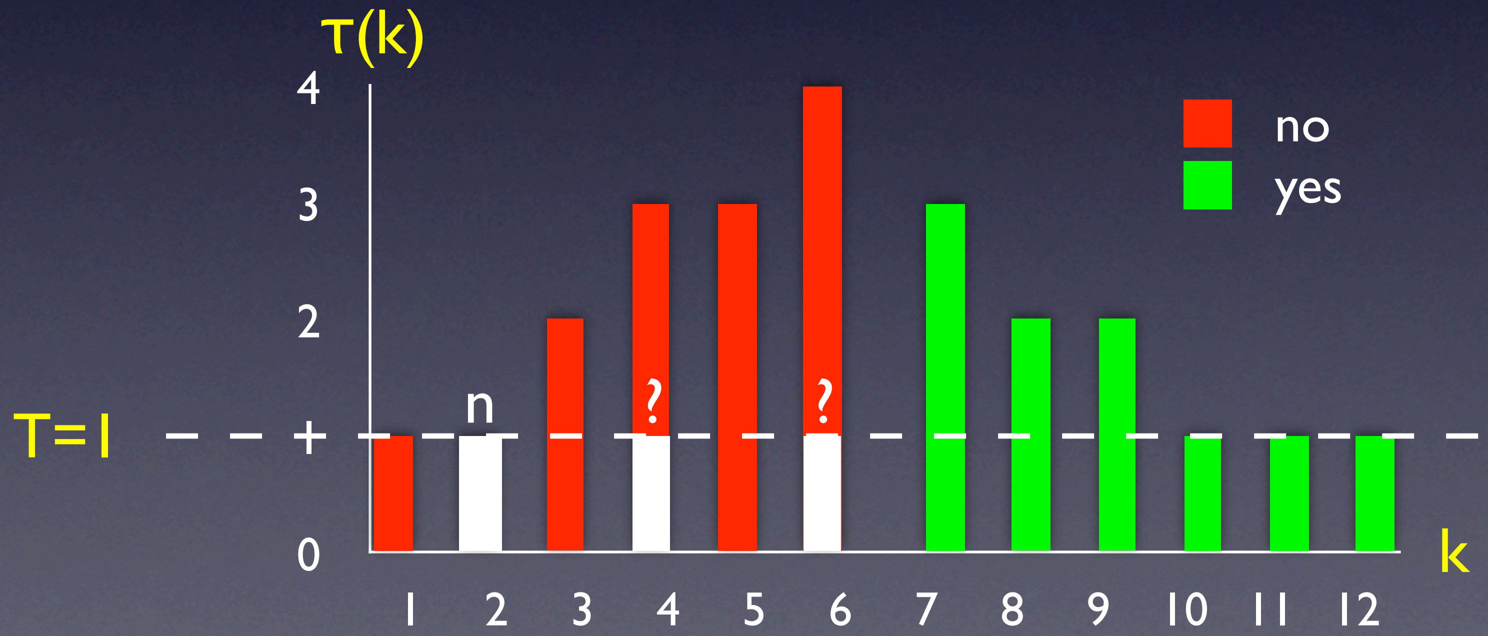
Query strategy S_2

- Initialize $T \leftarrow 1$
- Use two-sided binary search to find range of k-values such that $\tau(k) > T$
- Double T and repeat



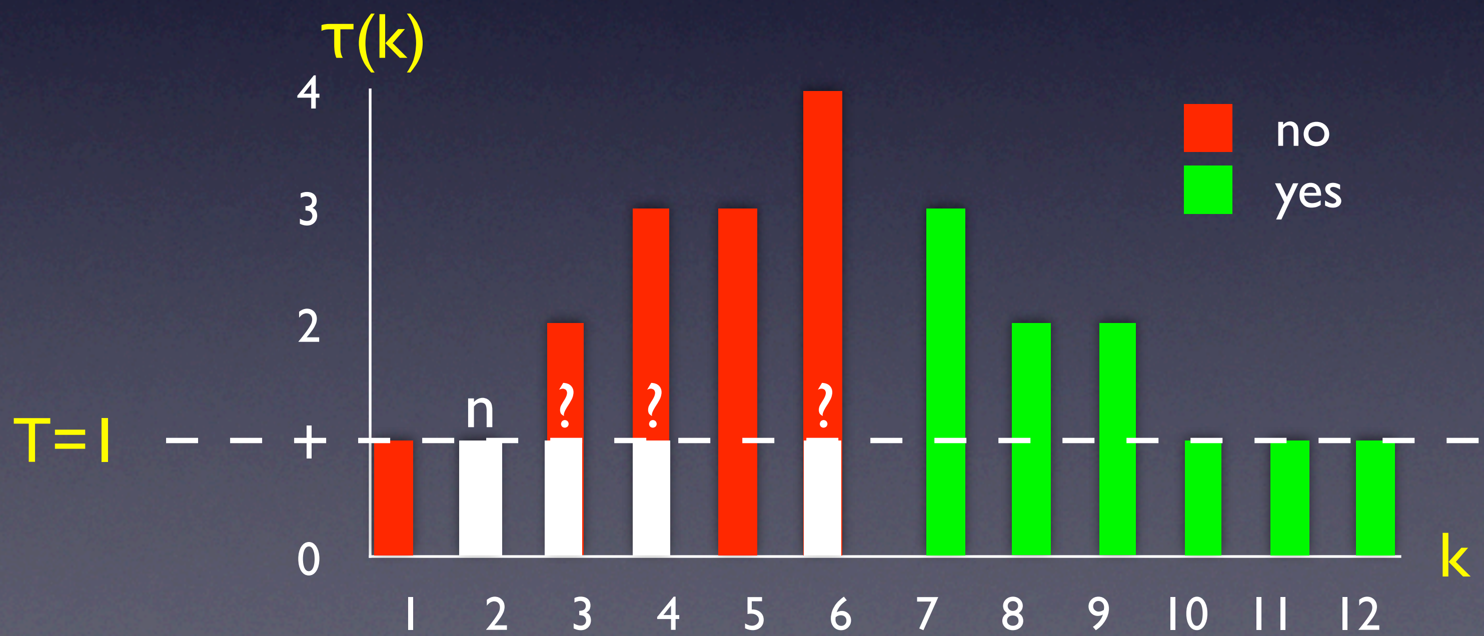
Query strategy S_2

- Initialize $T \leftarrow 1$
- Use two-sided binary search to find range of k-values such that $\tau(k) > T$
- Double T and repeat



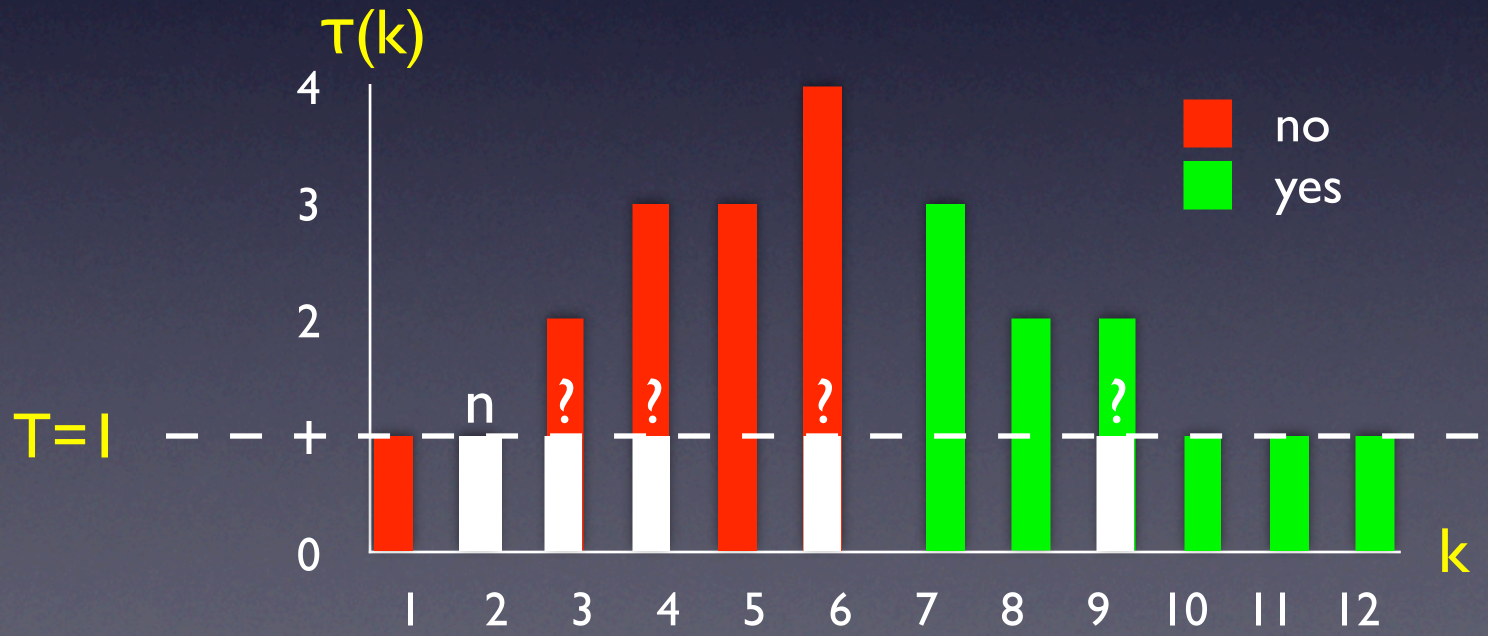
Query strategy S_2

- Initialize $T \leftarrow 1$
- Use two-sided binary search to find range of k-values such that $\tau(k) > T$
- Double T and repeat



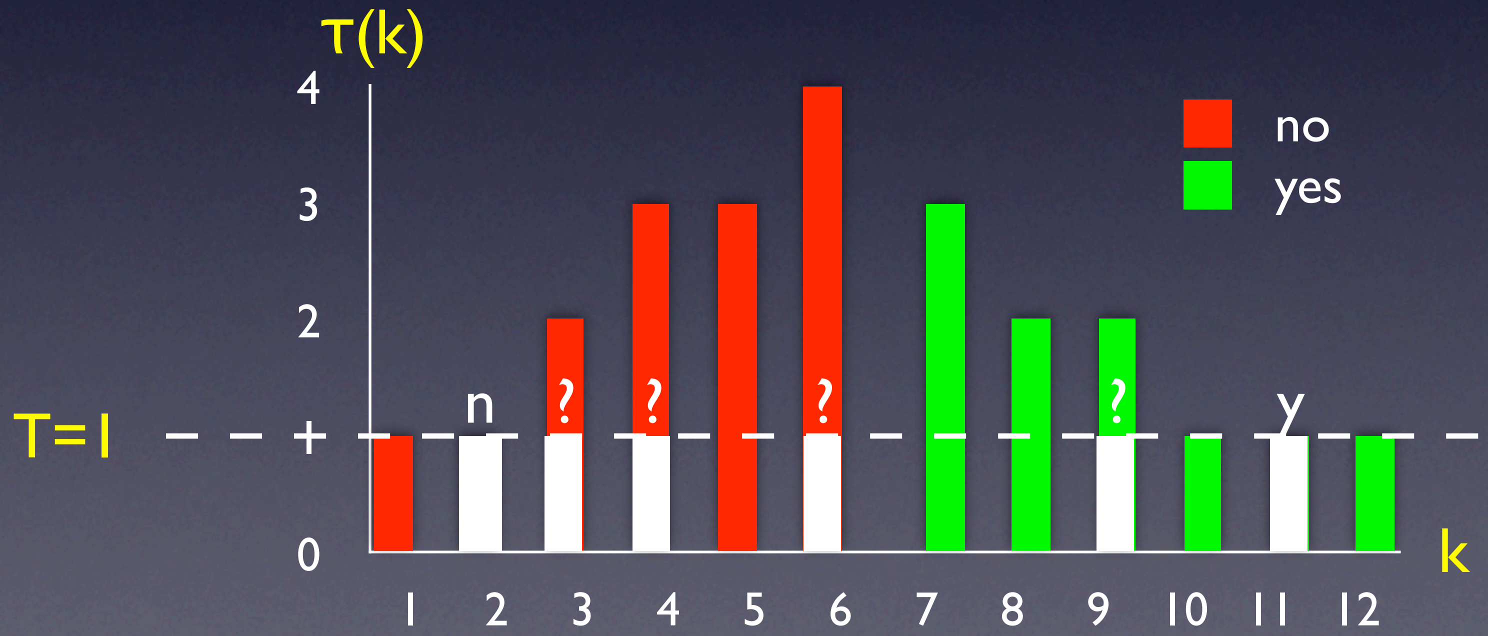
Query strategy S_2

- Initialize $T \leftarrow 1$
- Use two-sided binary search to find range of k-values such that $\tau(k) > T$
- Double T and repeat



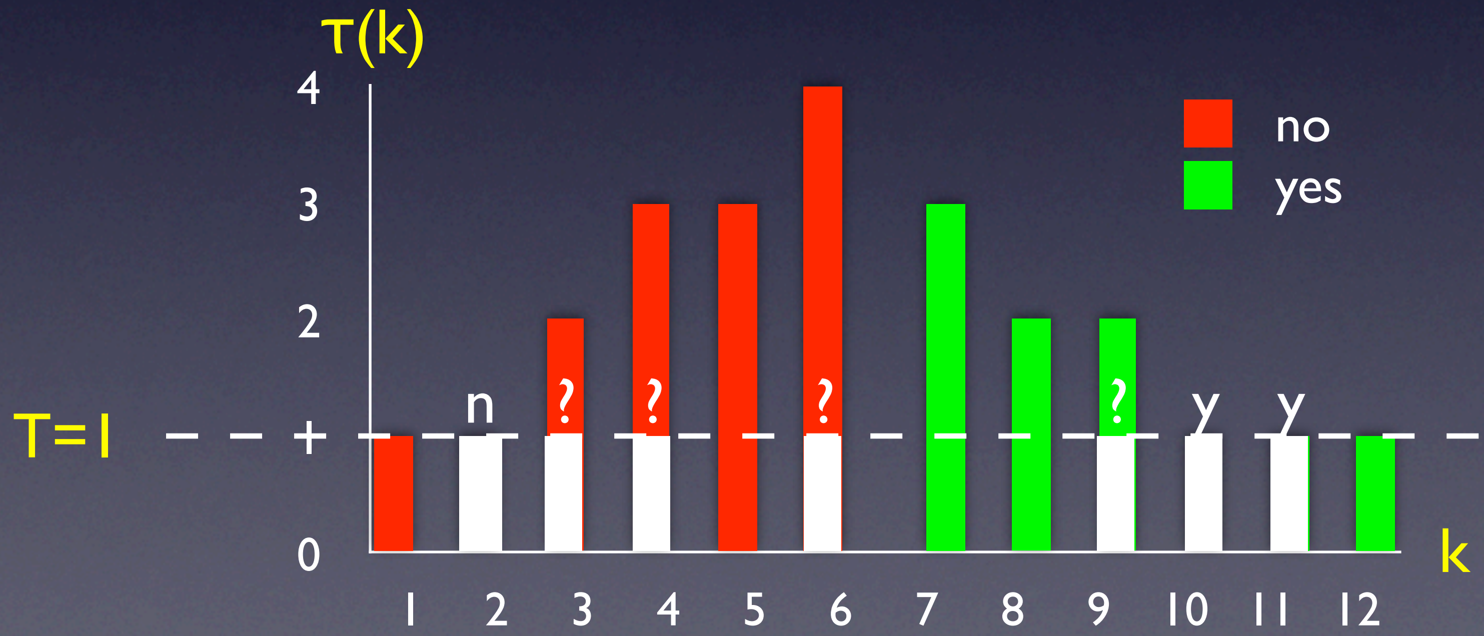
Query strategy S_2

- Initialize $T \leftarrow 1$
- Use two-sided binary search to find range of k-values such that $\tau(k) > T$
- Double T and repeat



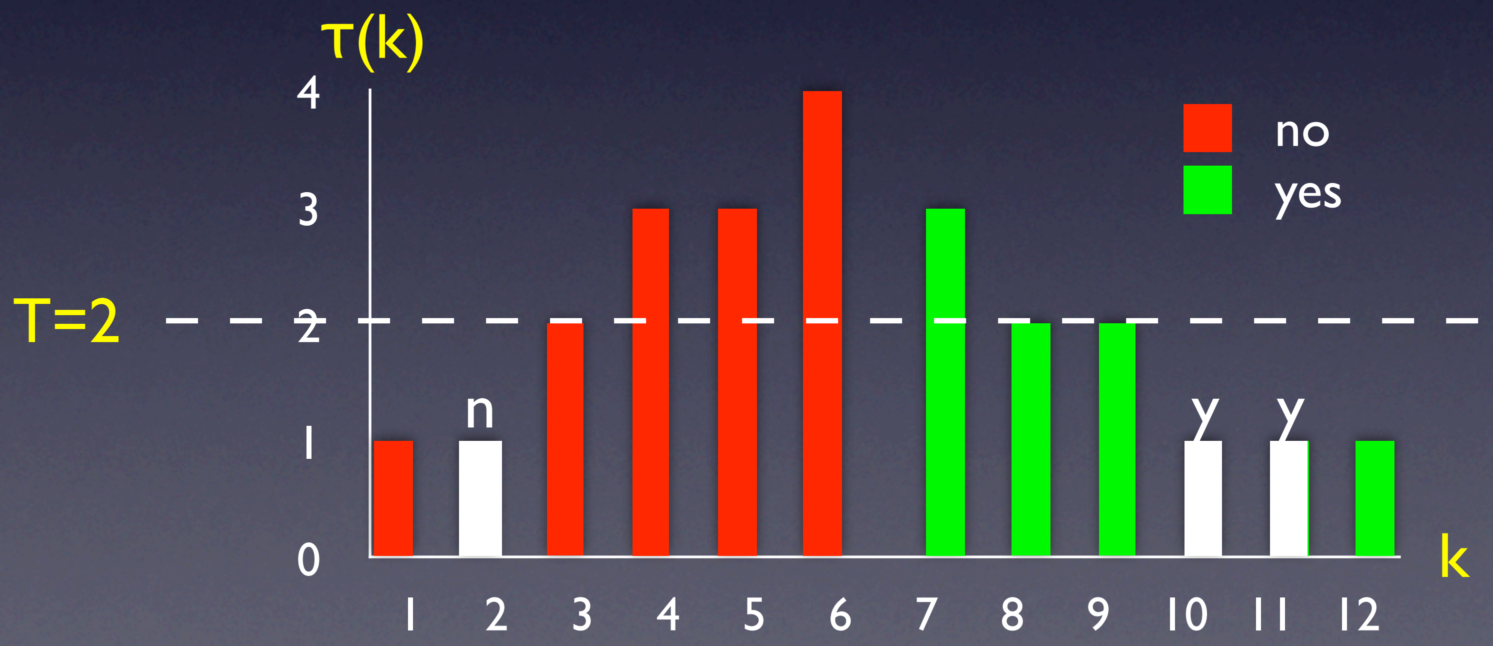
Query strategy S_2

- Initialize $T \leftarrow 1$
- Use two-sided binary search to find range of k-values such that $\tau(k) > T$
- Double T and repeat



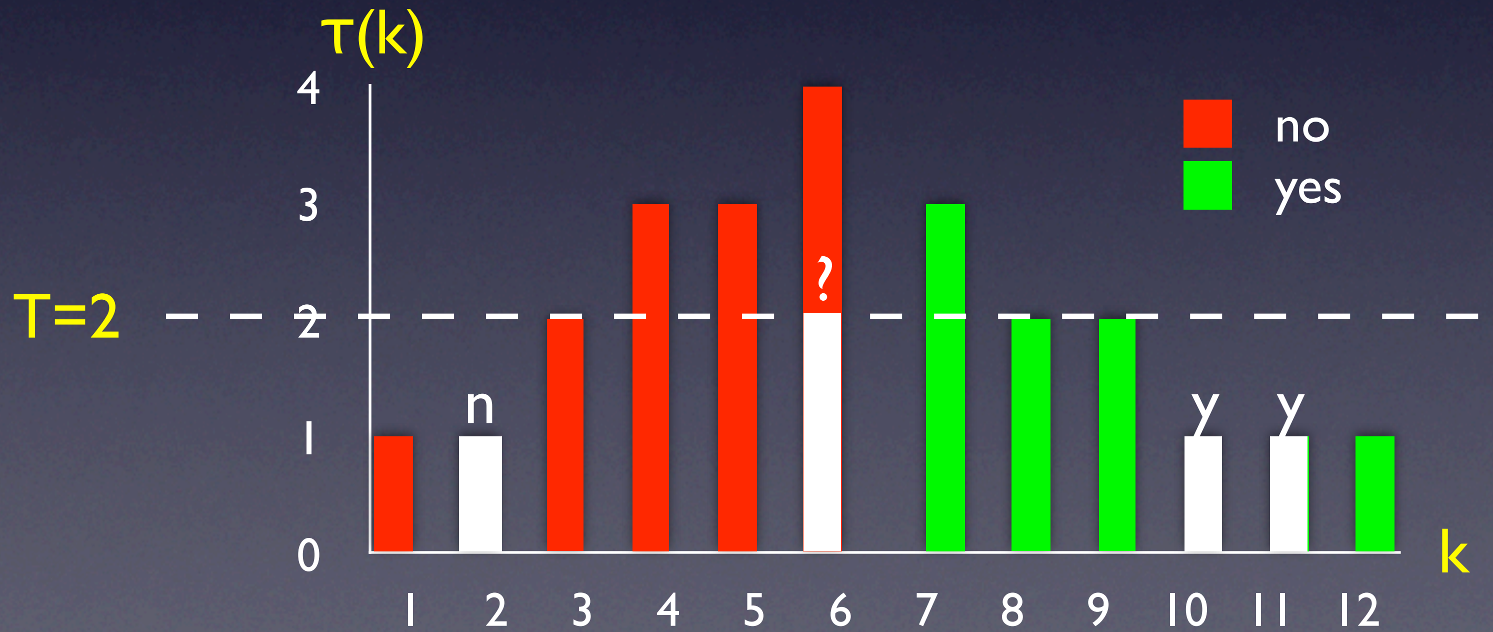
Query strategy S_2

- Initialize $T \leftarrow 1$
- Use two-sided binary search to find range of k-values such that $\tau(k) > T$
- Double T and repeat



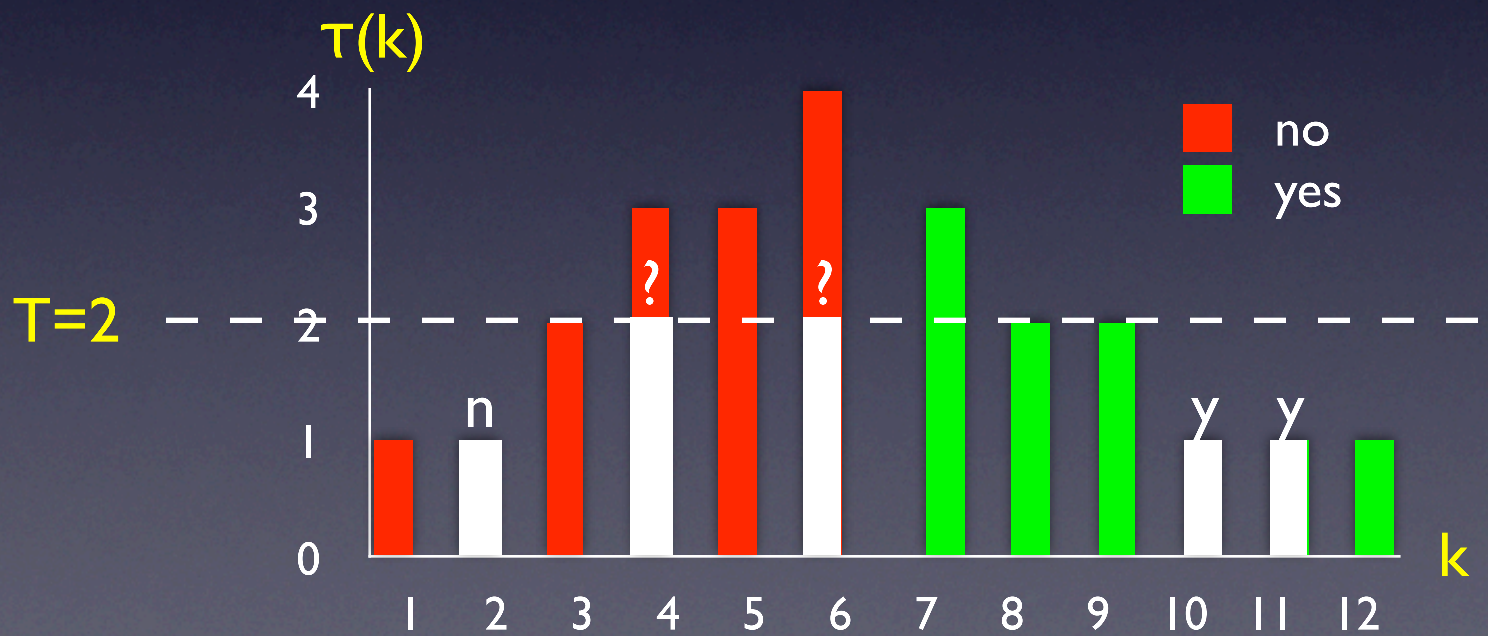
Query strategy S_2

- Initialize $T \leftarrow 1$
- Use two-sided binary search to find range of k-values such that $\tau(k) > T$
- Double T and repeat



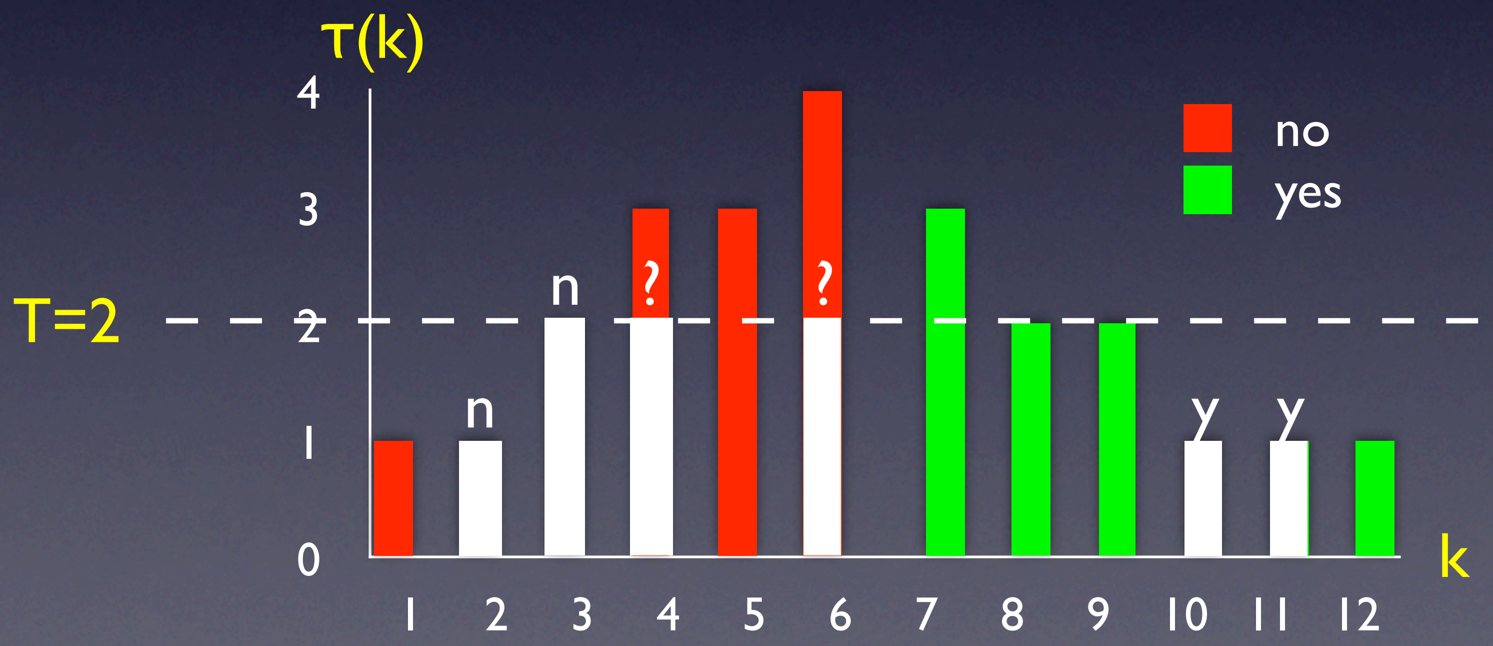
Query strategy S_2

- Initialize $T \leftarrow 1$
- Use two-sided binary search to find range of k-values such that $\tau(k) > T$
- Double T and repeat



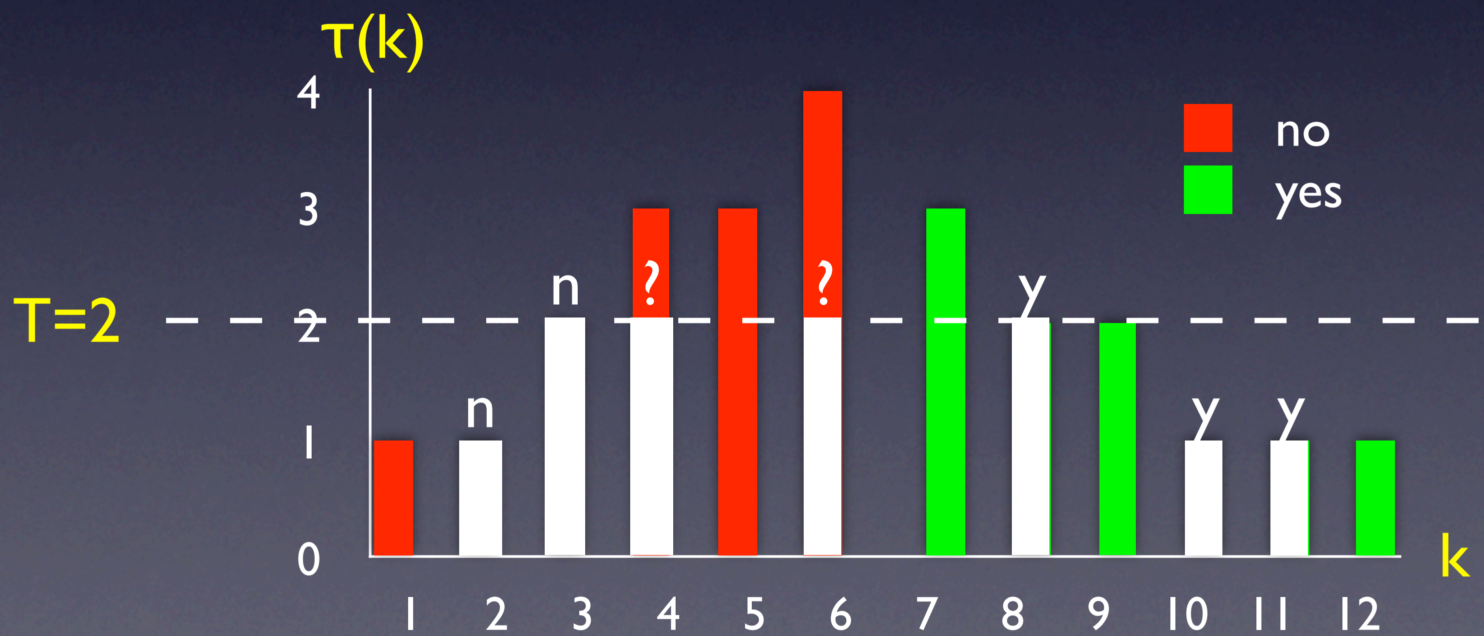
Query strategy S_2

- Initialize $T \leftarrow 1$
- Use two-sided binary search to find range of k-values such that $\tau(k) > T$
- Double T and repeat



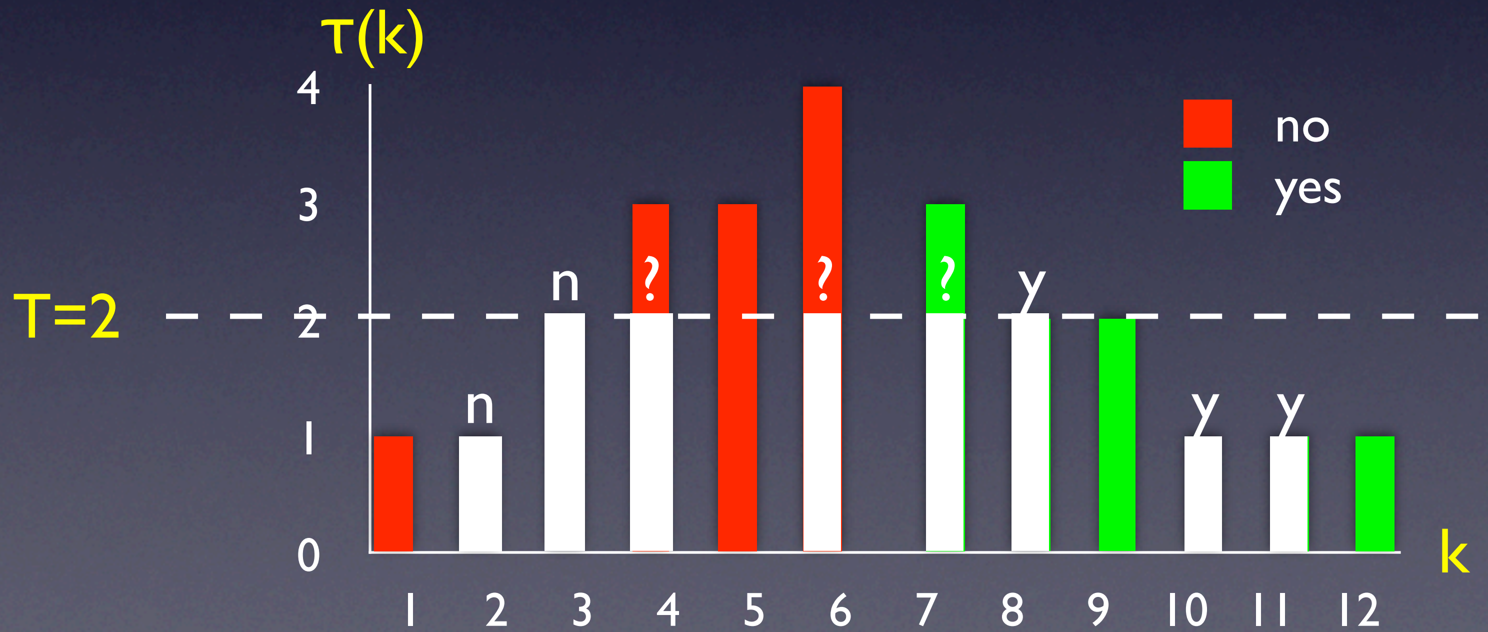
Query strategy S_2

- Initialize $T \leftarrow 1$
- Use two-sided binary search to find range of k-values such that $\tau(k) > T$
- Double T and repeat



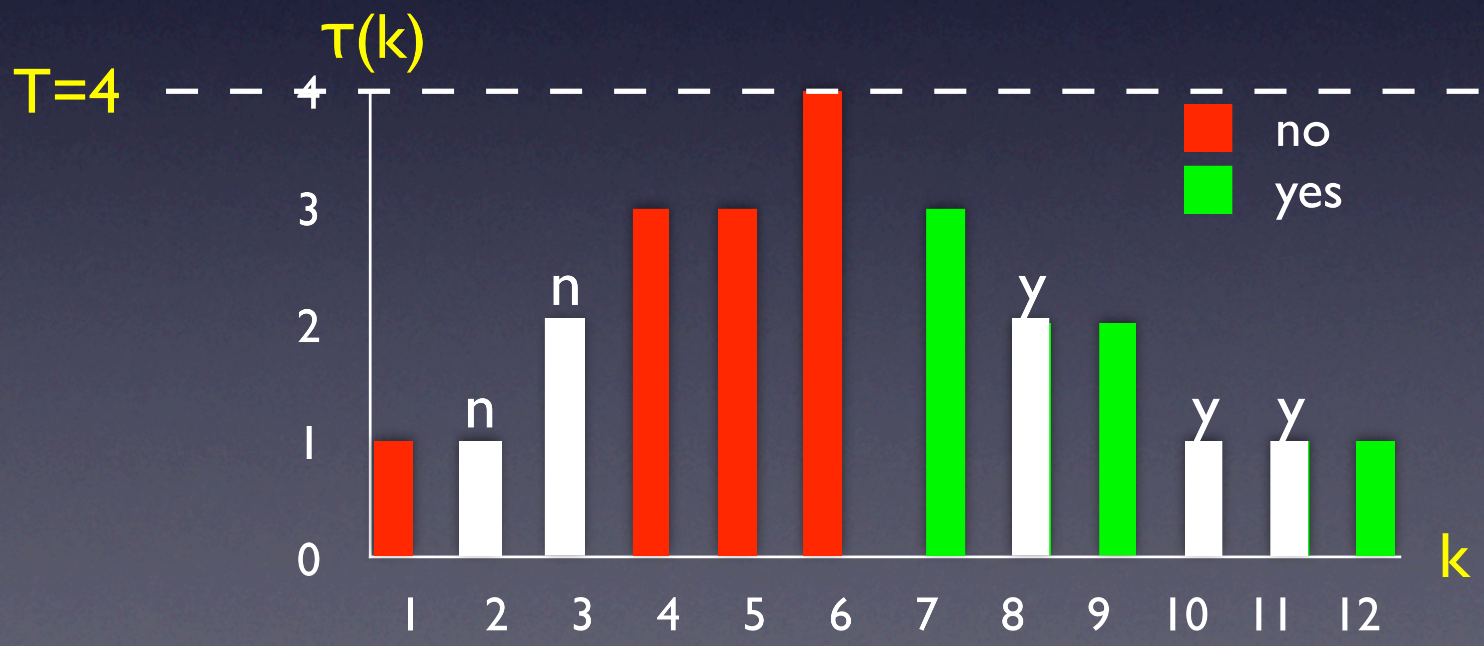
Query strategy S_2

- Initialize T ← 1
- Use two-sided binary search to find range of k-values such that τ(k) > T
- Double T and repeat



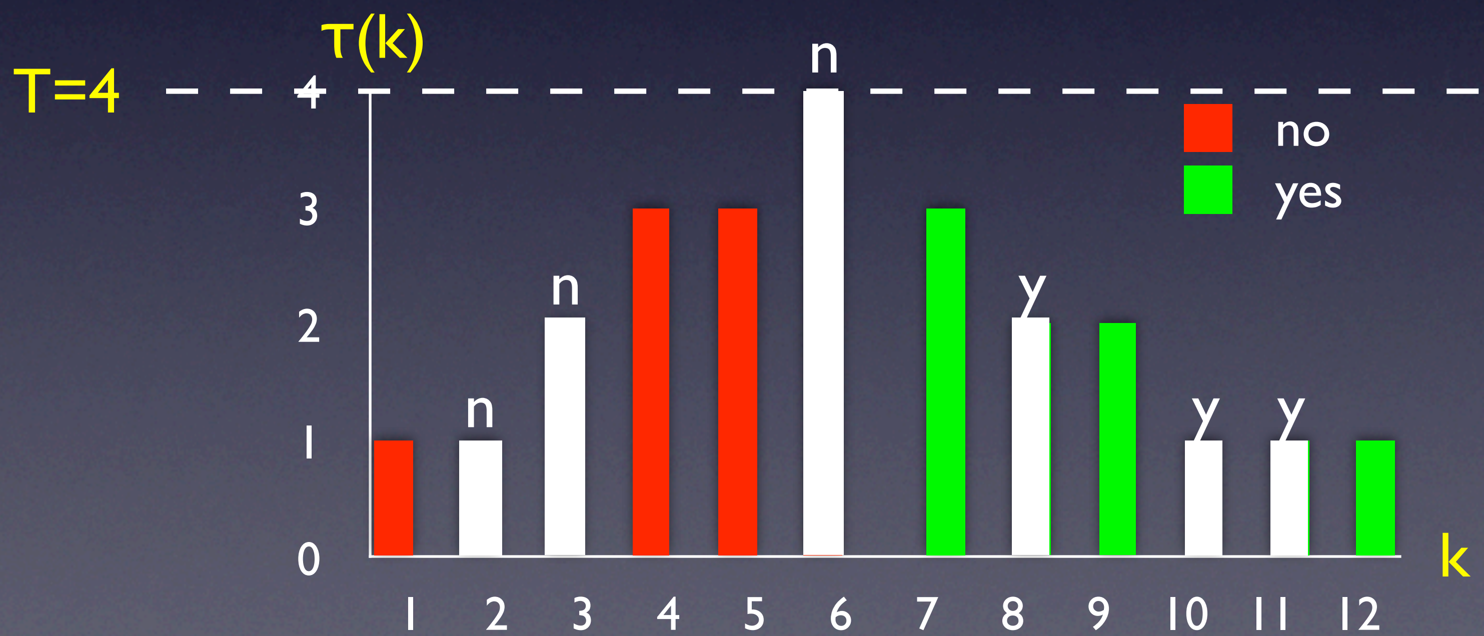
Query strategy S_2

- Initialize $T \leftarrow 1$
- Use two-sided binary search to find range of k-values such that $\tau(k) > T$
- Double T and repeat



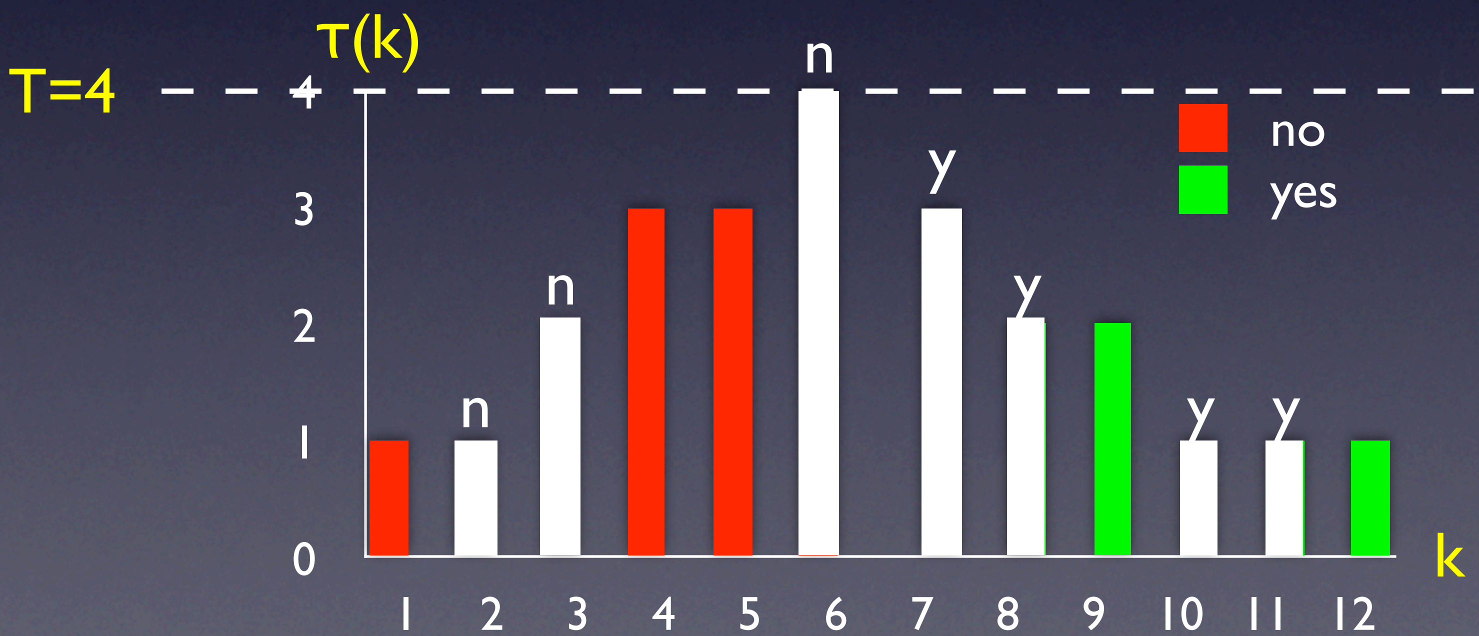
Query strategy S_2

- Initialize $T \leftarrow 1$
- Use two-sided binary search to find range of k-values such that $\tau(k) > T$
- Double T and repeat



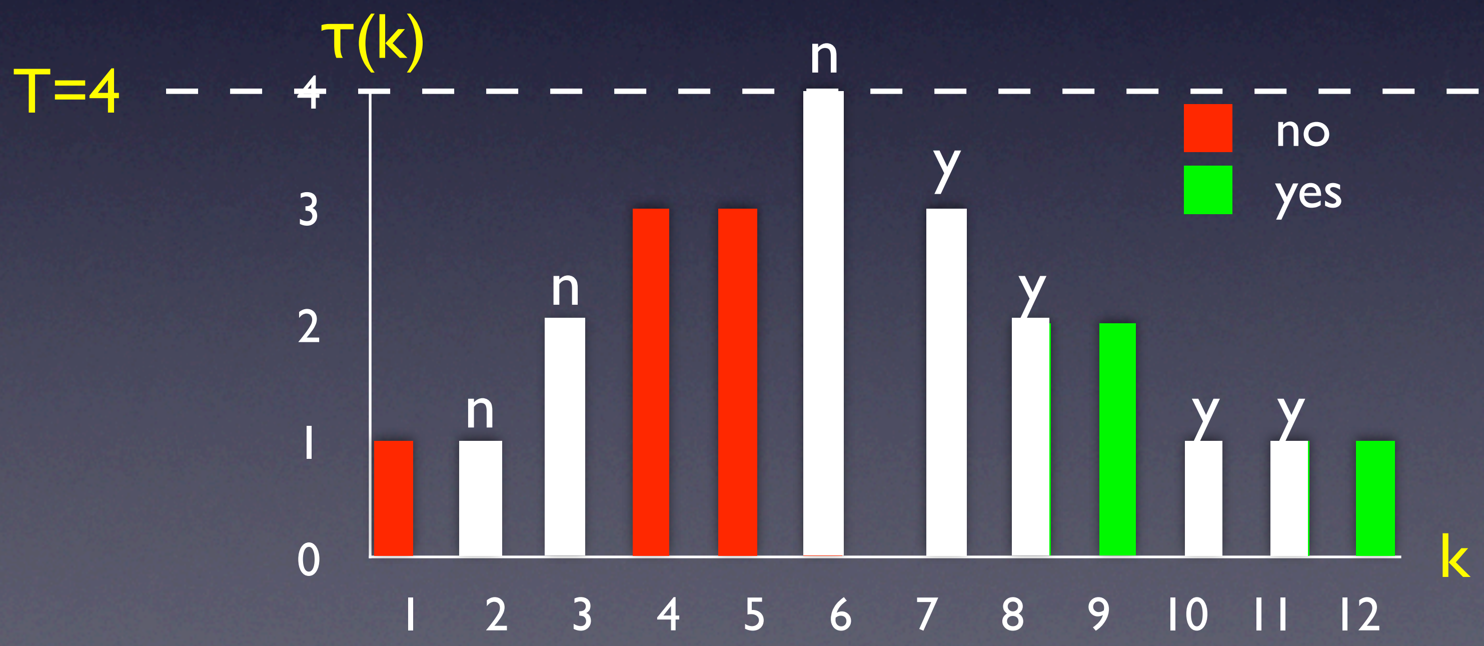
Query strategy S_2

- Initialize $T \leftarrow 1$
- Use two-sided binary search to find range of k-values such that $\tau(k) > T$
- Double T and repeat



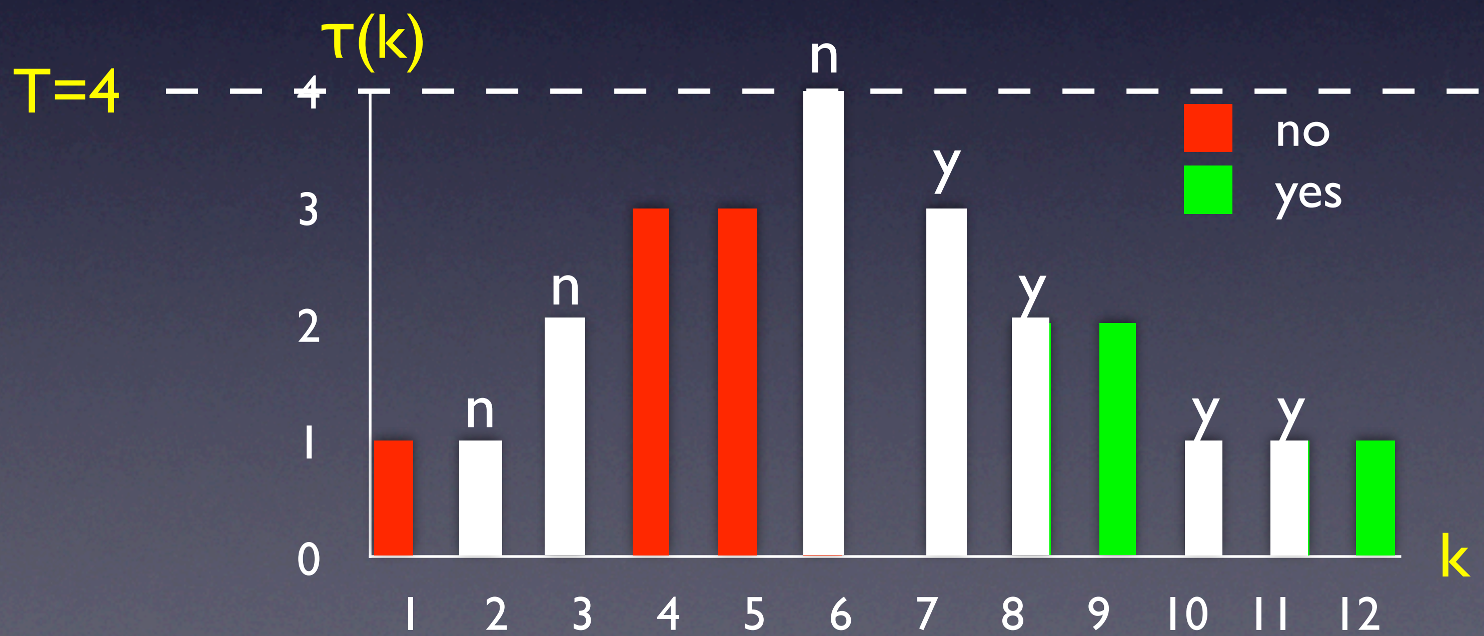
Query strategy S_2

- **Theorem:** if $\tau(k)$ is increasing-then-decreasing, then S_2 has competitive ratio $O(\log \#(\text{possible k-values}))$



Query strategy S_2

- **Theorem:** if $\tau(k)$ is increasing-then-decreasing, then S_2 has competitive ratio $O(\log \#(\text{possible k-values}))$
- If $\tau(k)$ becomes increasing-then-decreasing after multiplying each $\tau(k)$ by a factor $\alpha_k \leq \Delta$, ratio goes up by factor $\leq \Delta$



Experiments

- **A.I. Planning:** we use S_2 to create a variant of SATPLAN that finds approximately optimal plans quickly
- **Job shop scheduling:** we use S_2 to create a variant of a branch and bound algorithm for job shop scheduling that finds improved upper & lower bounds

Job shop scheduling

- Created variant of branch and bound algorithm of Brucker *et al.* (1994) that uses query strategy S_2
  - To execute query (k,t), set upper bound to $k+1$ and see if problem is feasible
- Ran on each instance in OR library, one hour time limit per instance

Job shop scheduling

Upper and lower bounds on OPT

Job shop scheduling

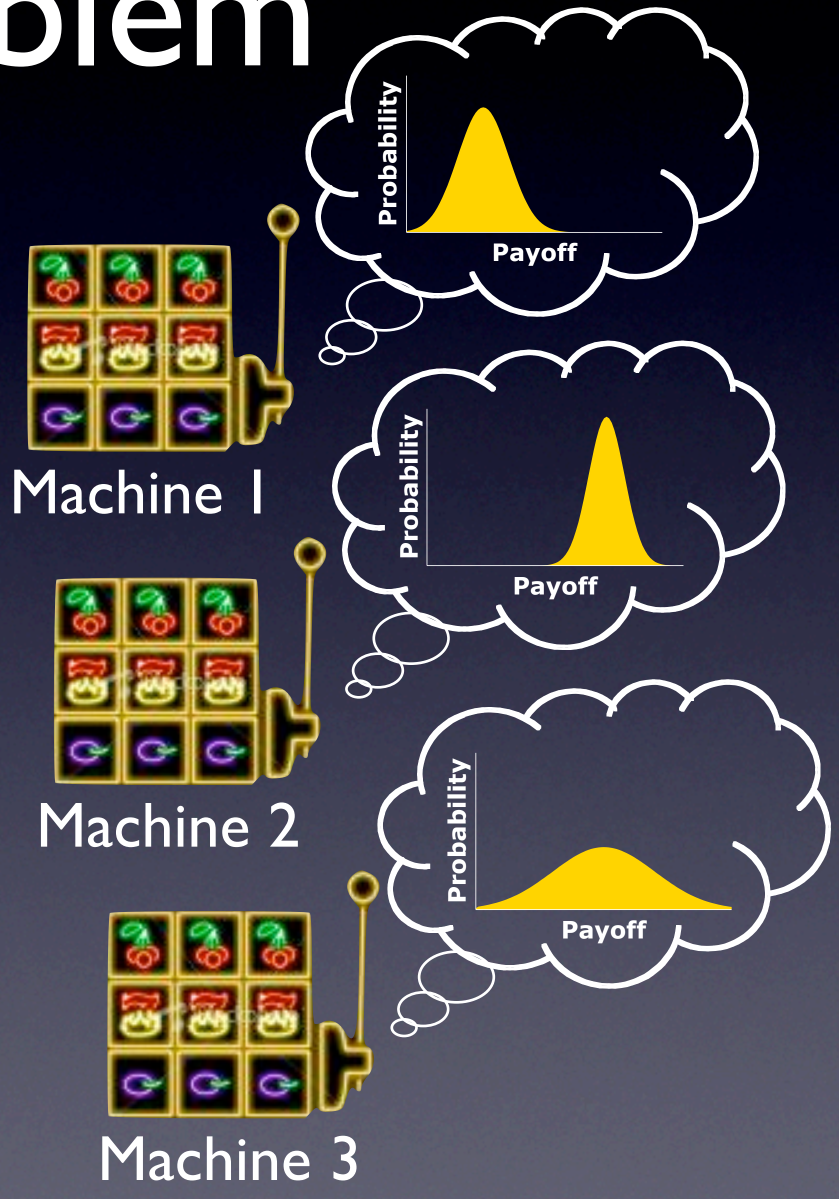
Upper and lower bounds on OPT

Instance	Brucker (S_2) [lower,upper]	Brucker (orig.) [lower,upper]
abz7	[650,712]	[650,726]
abz8	[622,725]	[597,767]
abz9	[644,728]	[616,820]
...	...	...
yn1	[813,987]	[763,992]
yn2	[835,1004]	[795,1037]
yn3	[812,982]	[793,1013]
yn4	[899,1158]	[871,1178]

The Max *k*-Armed Bandit Problem

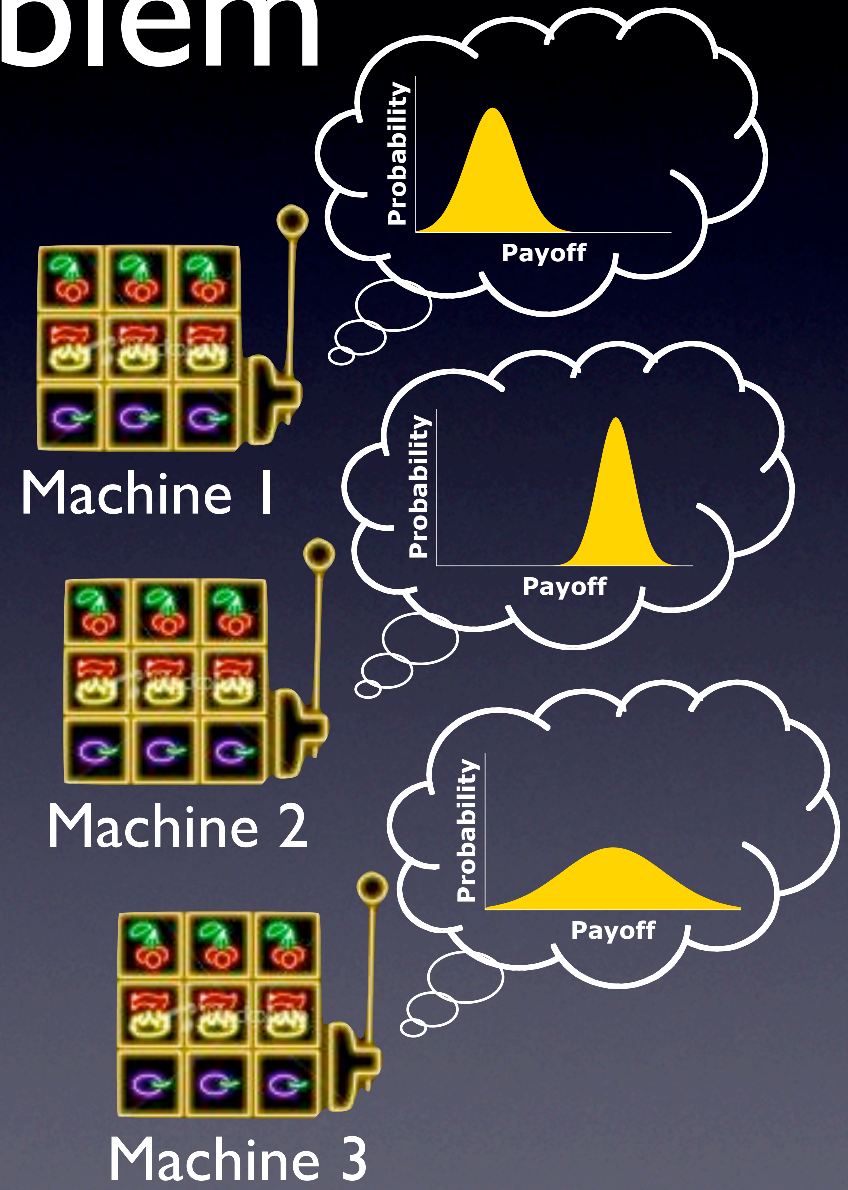
The *k*-armed bandit problem

- You are in a room with k slot machines
- Pulling arm of i^{th} machine returns payoff drawn from unknown distibution D_i
- Given budget of n pulls, want to maximize total payoff received
- Researched for 50+ years



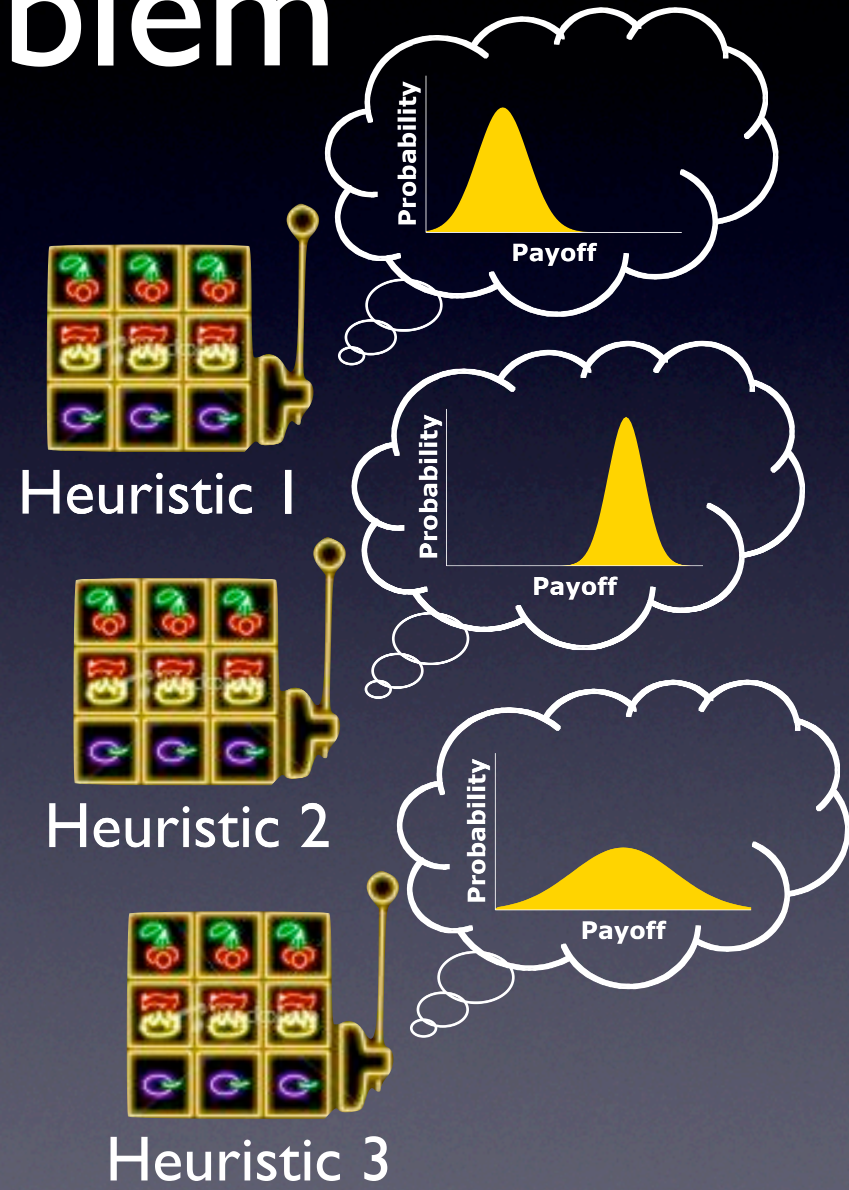
The max *k*-armed bandit problem

- You are in a room with k slot machines
- Pulling arm of i^{th} machine returns payoff drawn from unknown distribution D_i
- Given budget of n pulls, want to maximize **highest** payoff received
- Introduced by Cicirello & Smith (2003)



The max *k*-armed bandit problem

- Given: a *single* optimization problem, k randomized heuristics
- Each time you run a heuristic, get a solution with certain quality
- Given budget of n runs, want to maximize quality of best solution

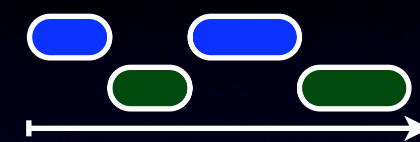


Our results

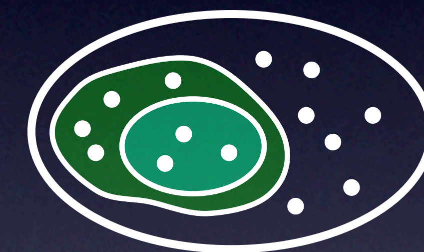
- Theoretical guarantees when each arm draws payoff from a *generalized extreme value* distribution
- Simple distribution-free approach that works well in practice
- Experiments allocating time among randomized greedy heuristics for resource-constrained project scheduling

Summary & contributions

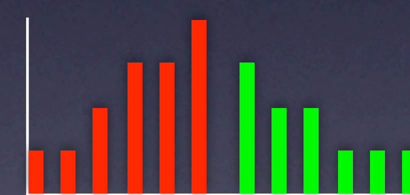
- New techniques for combining multiple heuristics



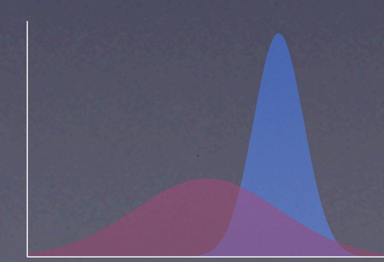
- An online algorithm for maximizing submodular functions



- Query strategy for solving optimization problems using decision algorithms



- Max *k*-armed bandit strategies



Thank You