Discriminating Instance Generation for Automated Constraint Model Selection

Ian P. Gent, **Bilal Syed Hussain**, Christopher Jefferson, Lars Kotthoff, Ian Miguel, Glenna F. Nightingale, and Peter Nightingale

CP 2014

Motivation

- One approach to automated constraint modelling is to generate, then select from a set of candidate models.
- To choose a subset of these models we need instances which are *discriminating*.
  - Allowing us to pick models with superior runtime performance.
- We will show methods which can generate these instances automatically.

Overview

- Automated Modelling
- Model Selection
- Generating *Discriminating* Instances
- Experimental Results

Automated modelling?

- The user writes a specification in Essence which supports:
  - Abstract types including relations, (multi-)sets and partitions.
  - These types can be arbitrarily nested.
- This abstracts over the modelling decisions.
- The specification is then refined into a set of constraint models that a solver can solve.
  - We want to pick 1 (or a subset) of the models with good performance.

Social Golfer Problem (SGP)

- In a golf club there are a number of golfers who wish to play together in g groups of size s.
- Find a schedule of play for w weeks such that no pair of golfers play together more than once.

Social Golfer Problem (SGP)

```
given w, g, s : int(1..100)
letting Golfers be new type of size g * s
find schedule : set (size w)
                    of partition (regular, size g) from Golfers
such that
    forAll {week1, week2} subsetEq schedule .
        forAll group1 in parts(week1) .
            forAll group2 in parts(week2) .
             |group1 intersect group2| < 2
```

Social Golfer Problem (SGP)

```
given w, g, s : int(1..100)
letting Golfers be new type of size g * s
find schedule : set (size w)
                of partition (regular, size g) from Golfers
such that
    forAll {week1, week2} subsetEq schedule .
        forAll group1 in parts(week1) .
            forAll group2 in parts(week2) .
             |group1 intersect group2| < 2
```

Social Golfer Problem (SGP)

```
given w, g, s : int(1..100)
letting Golfers be new type of size g * s
find schedule : set (size w)
                of partition (regular, size g) from Golfers
such that
    forAll {week1, week2} subsetEq schedule .
        forAll group1 in parts(week1) .
            forAll group2 in parts(week2) .
             |group1 intersect group2| < 2
```

Model Selection - Racing

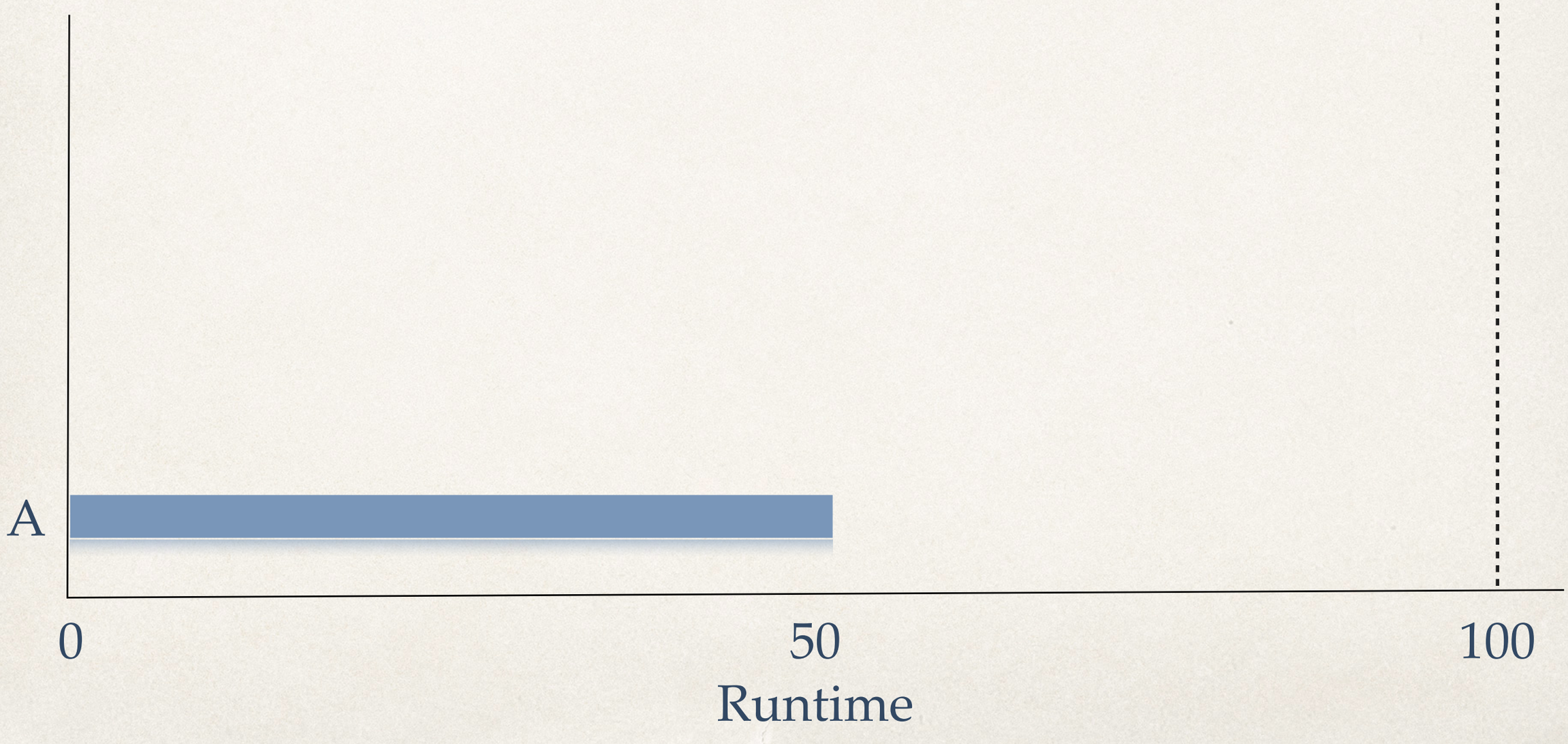
- To select a subset of the models, we perform a series of *races* with each instance as they are generated.

- A model is *ρ-dominated* on an instance by another model if the runtime for the second model is at least ρ times better than the first.

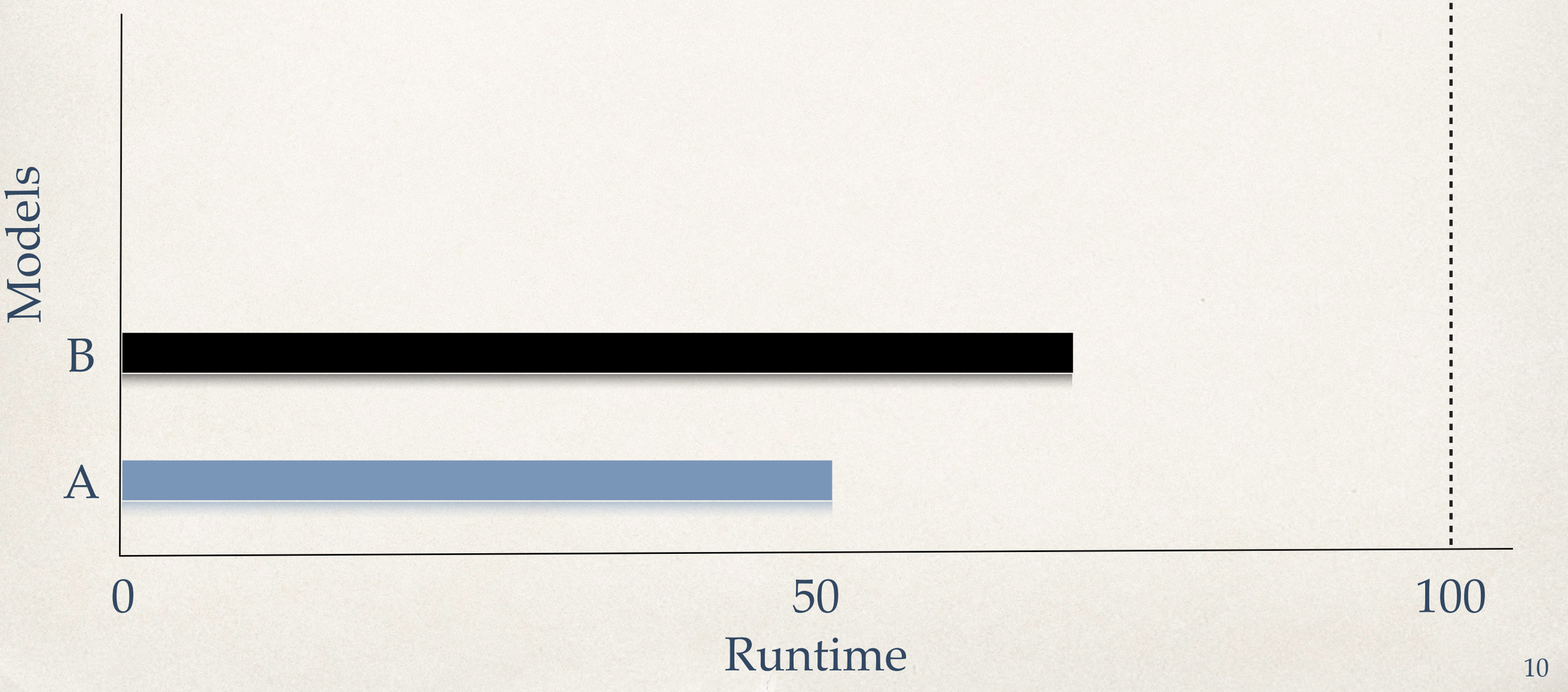
- The *winners* of an instance race are the models not *ρ-dominated*.

- The discriminatory quality of an instance is the fraction of models that are ρ-dominated.

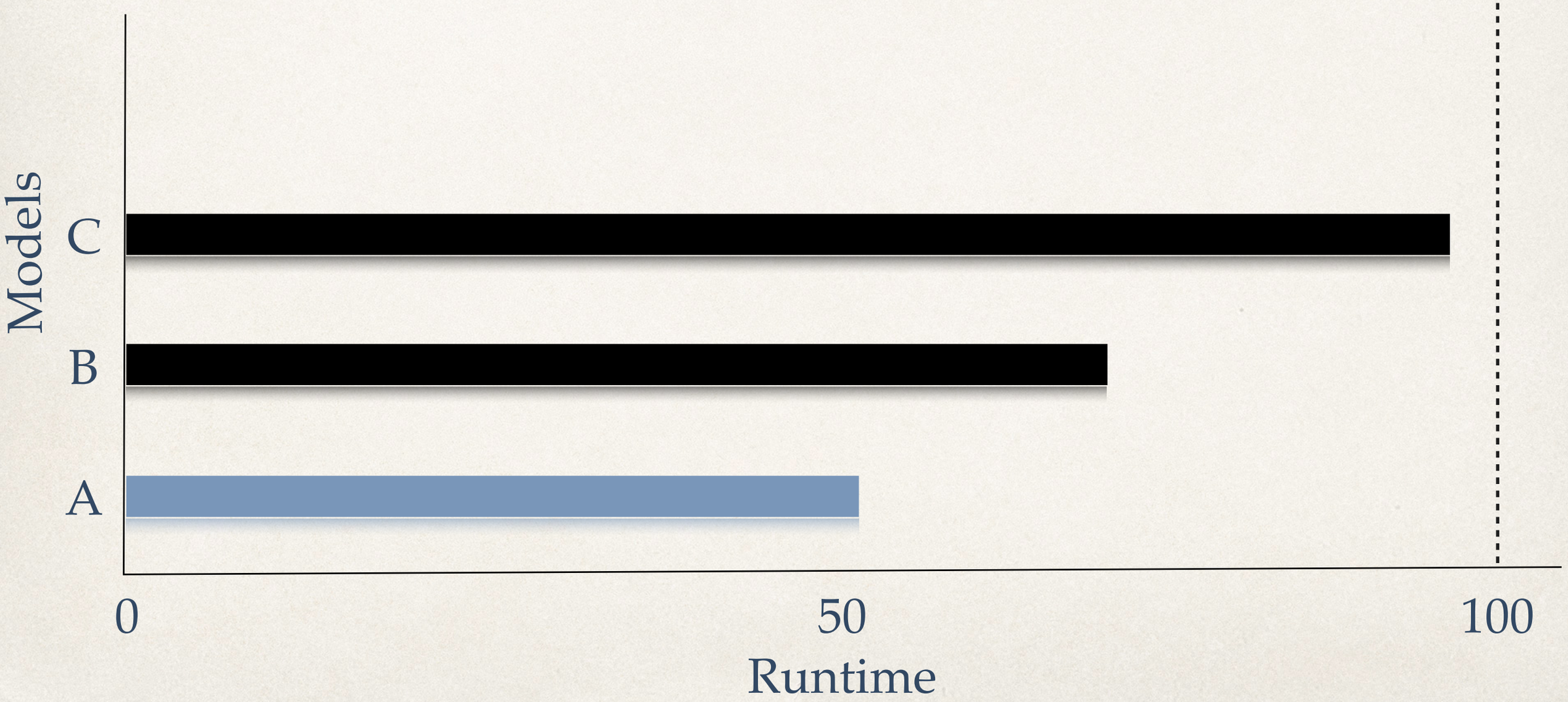
Model Selection - Racing



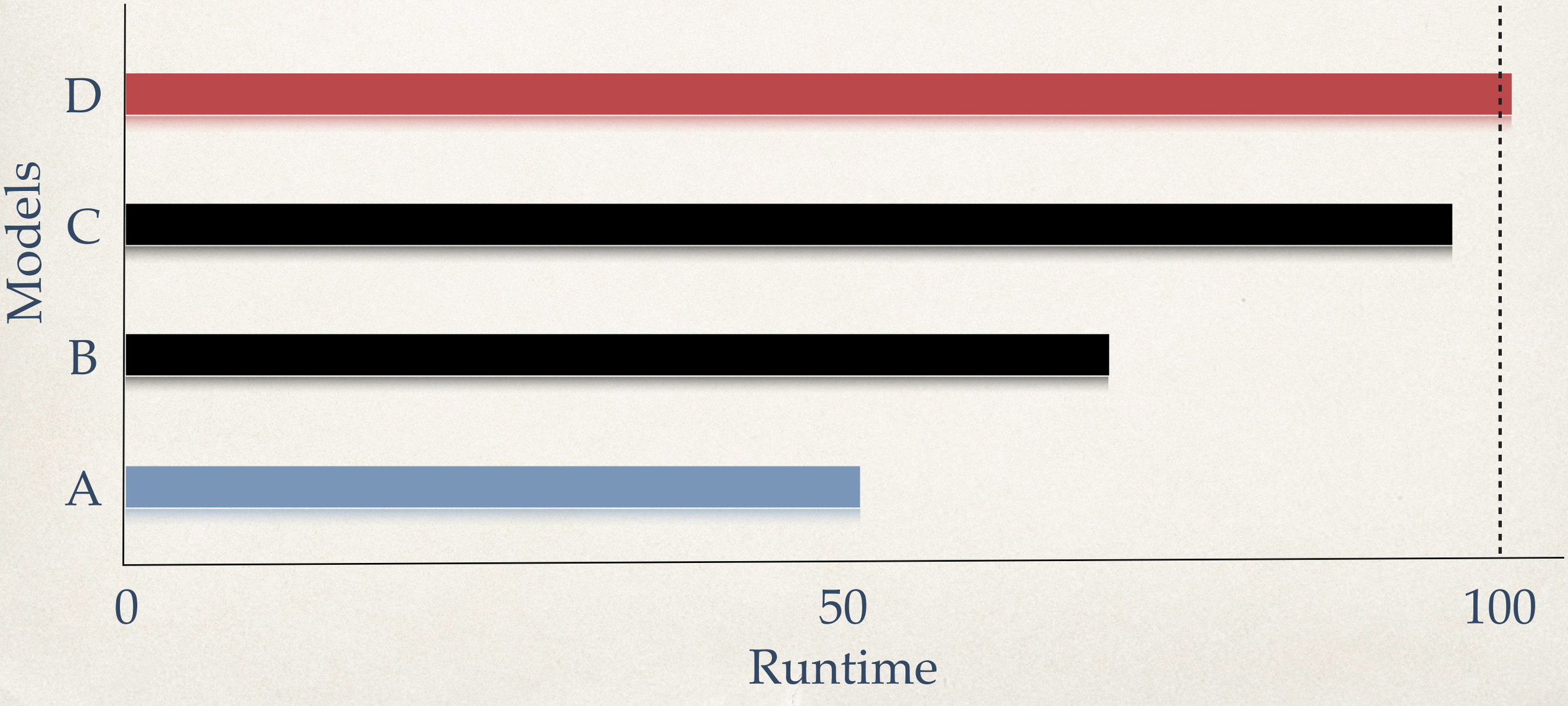
Model Selection - Racing



Model Selection - Racing



Model Selection - Racing



Instance Space

- The instance space is defined by the parameters of a problem class.
- Social Golfer Problem has 3 independent integer parameters.
  - given w, g, s : int(1..100)
- The Progressive Party (CSPLib 13) is timetabling problem:
  - Certain boats are to be designated *hosts*.
  - Remaining boats in turn visit the *host* boats for several successive half-hour periods.
  - Each boat has limited capacity.

The Progressive Party Problem(PPP)

- The Progressive Party has complex set of dependent parameters; 3 integers and 2 functions.
  - Two of the integers determine the domain of the two given functions.

given n_upper, n_boats, n_periods : int(1..)
letting Boat be domain int(1..n_boats)
given capacity, crew : function (total) Boat --> int(1..n_upper)
where forAll i : Boat . crew(i) <= capacity(i),

find … such that …

Generating Discriminating Instances

- Undirected — For each race in a sequence, undirected simply draws a sample from the instance space and runs a race.
- SMAC — Utilises a existing algorithm configuration system.
- Markov — This method is loosely based on the Markov chains Monte Carlo methods.

SMAC

- SMAC (Hutter et al) is an existing algorithm configuration system:
  - Given an algorithm, a description of its parameters, and a set of instances,
  - It finds the set of parameters for the algorithm that delivers the best performance on the set of instances.
- Finding discriminatory instances is a similar setting:
  - The algorithm is the problem class specification, its parameters instantiate particular instances of the class and the set of problem "instances" is the set of models.
- We encode the problem class parameters into SMAC's input format, which uses integer and categorical variables.
  - For more structured givens such as `functions` we use multiple SMAC parameters.
  - For a total function, we use *n* SMAC parameters, where *n* is the maximum number of mappings

Markov

- Markov uses an *acceptance* function to determine if an sampled instance should be run.
- It does this by utilising data about the quality of previous instances to infer the quality of the candidate instance.
- Attracted towards known discriminating instances.
- Repelled from known non-discriminating instances.

Markov — Distance calculation

- An instance's quality is used to effect other instance in its range of influence.

- Our measure of proximity between two instances is the distance between each pair of parameters combined using the Euclidean distance.

- The calculation for the distance between each pair of parameters is specified per type.
  - Ints : The absolute difference between the two values.
  - Sets: Given sets S and T , the distance is $\sqrt{|S \setminus T| + |T \setminus S|}$

Markov — Acceptance function

Previous instance

Estimated quality of proposed instance

$$A(x_{i-1}, x'_i) = \frac{G'(x'_i)}{G(x_{i-1})}$$

Proposed instance

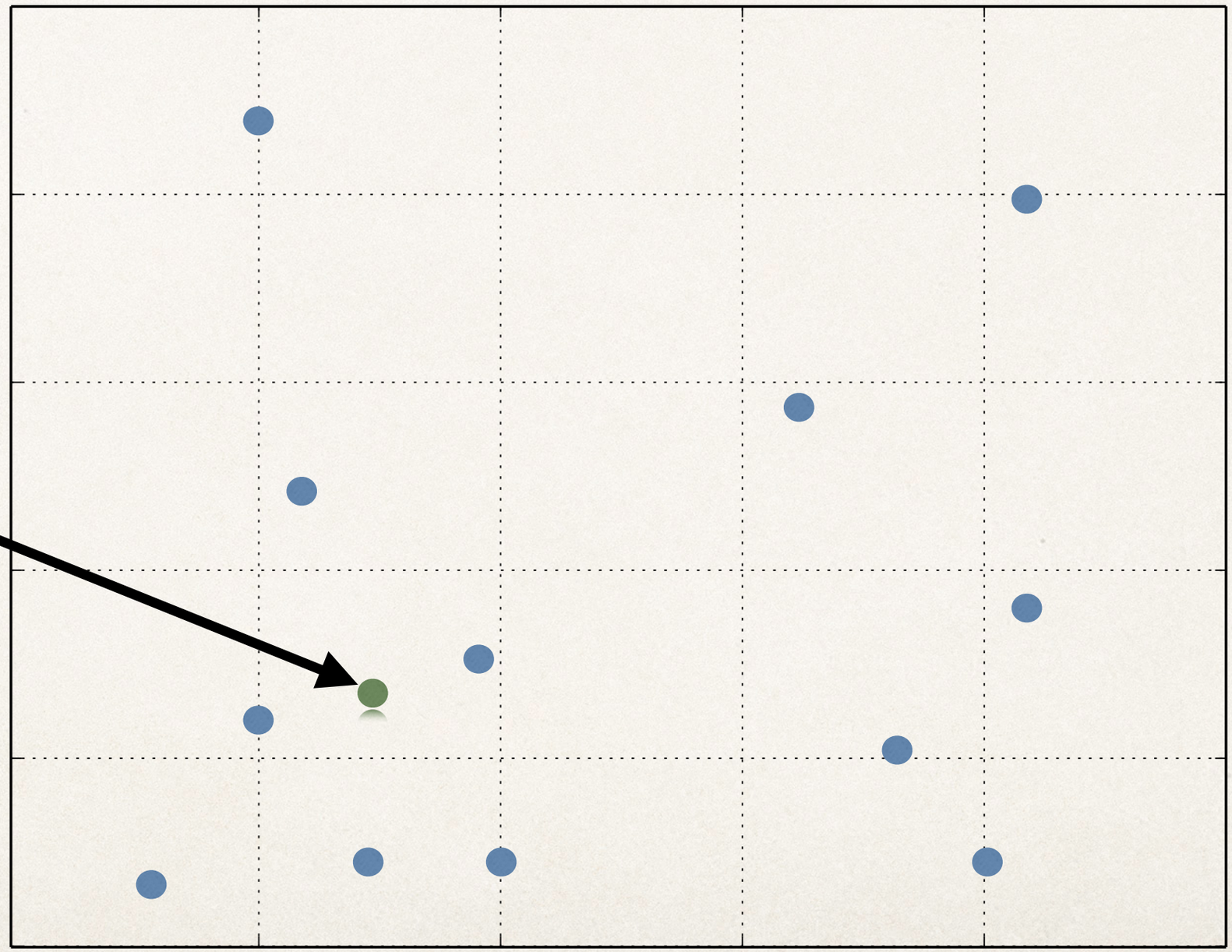
Quality of previous instance

Markov — Acceptance function

- G' uses the set of all previous accepted instances within distance r of x_i'.
  - If non-empty the mean of the set is returned otherwise 0.5
- A pseudorandom number u is generated within [0, 1], and x_i' is accepted if $A(x_{i-1}, x_i') > u$.
- Note that the proposed instance is always accepted if its estimated quality is greater then the previous instance's true quality.

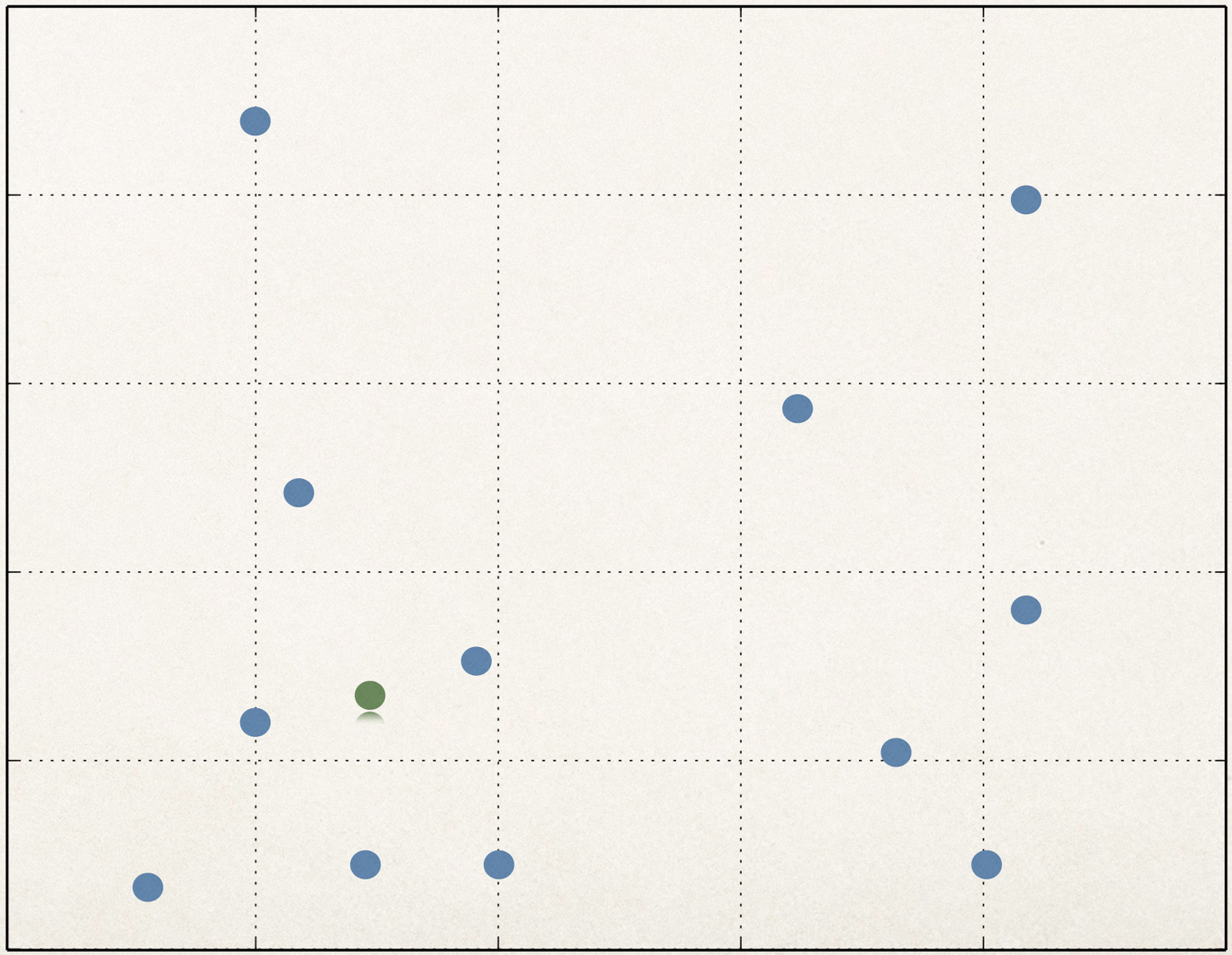
Markov

- accepted points



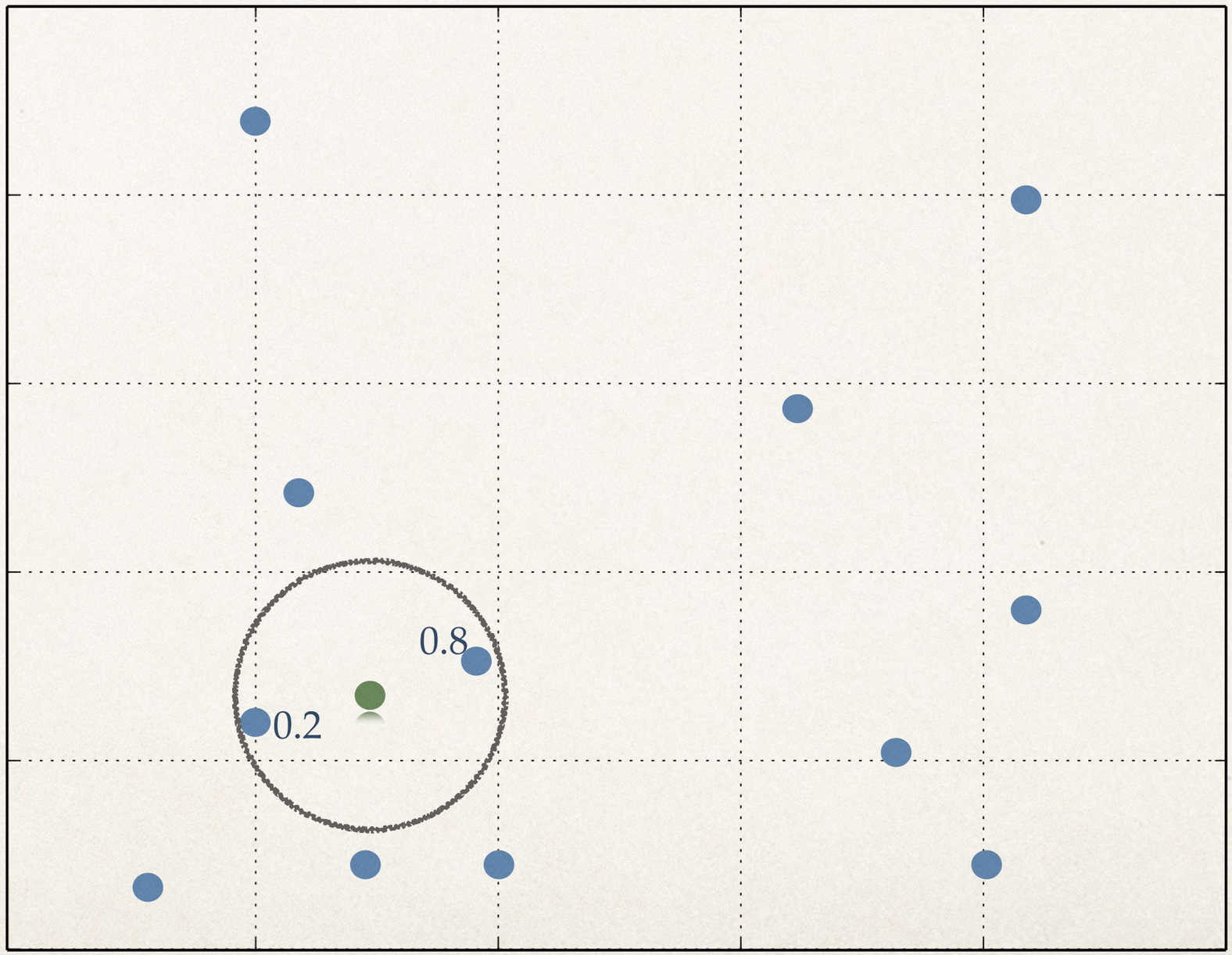
Markov

- accepted points



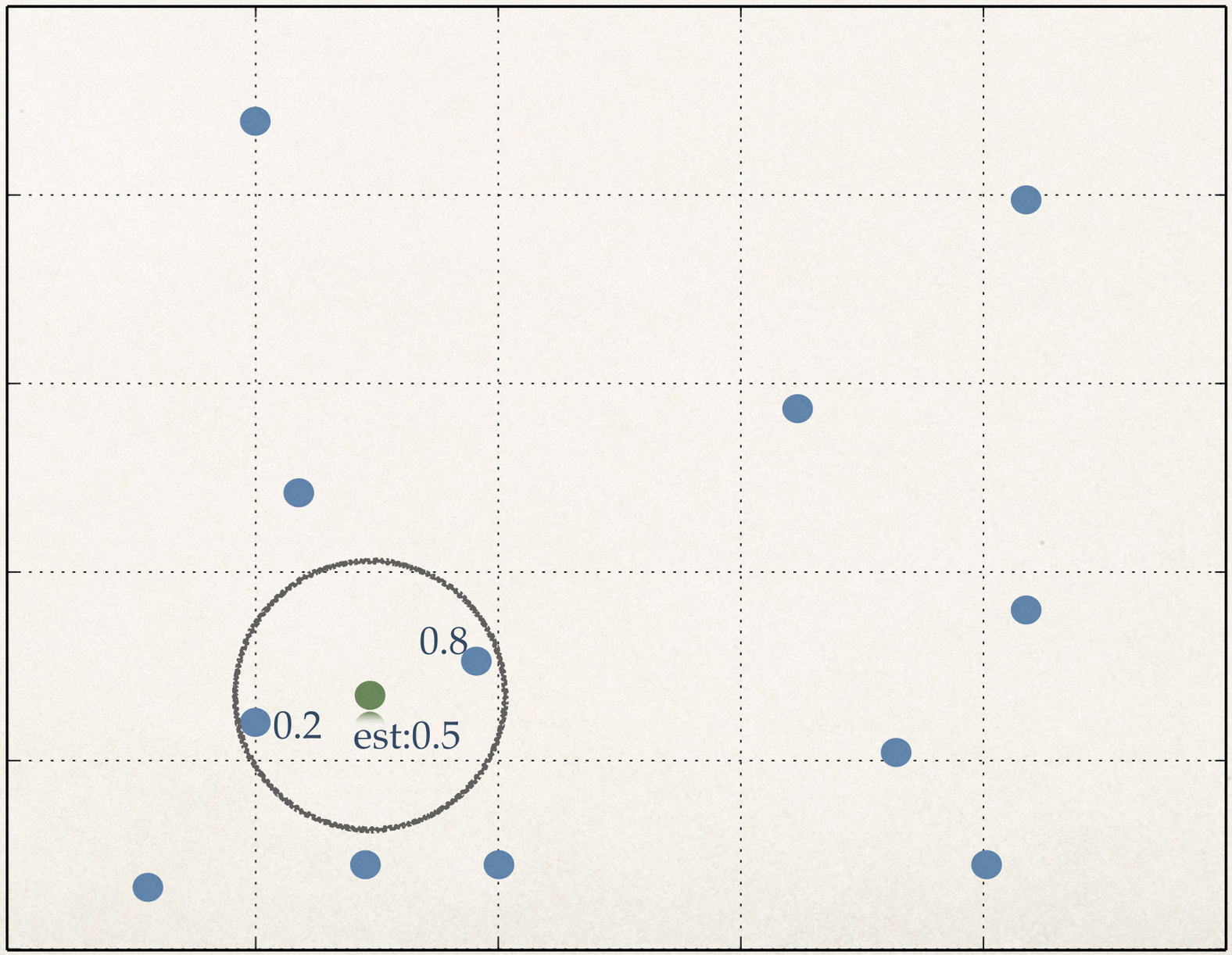
Markov

- accepted points



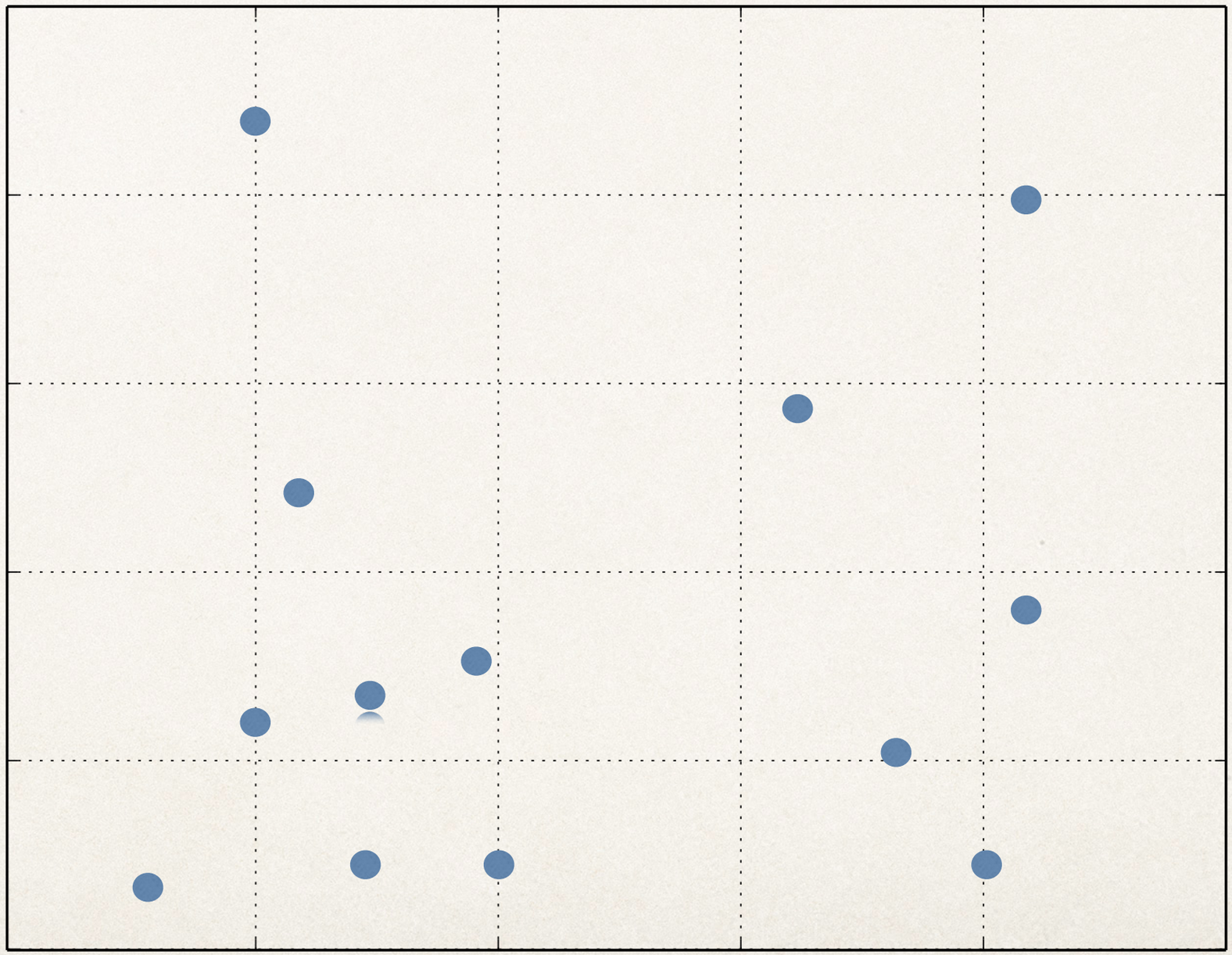
Markov

- accepted points



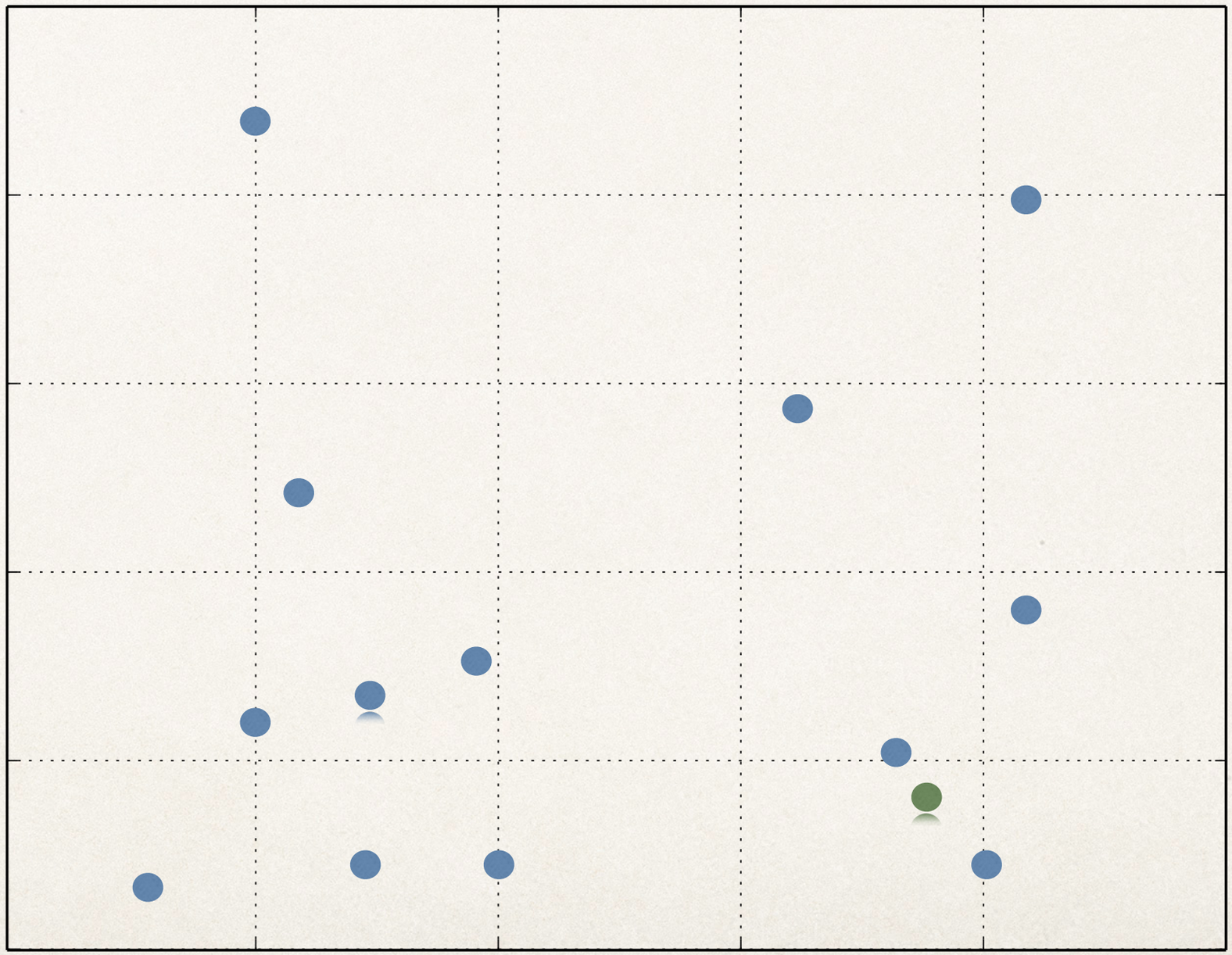
Markov

- accepted points



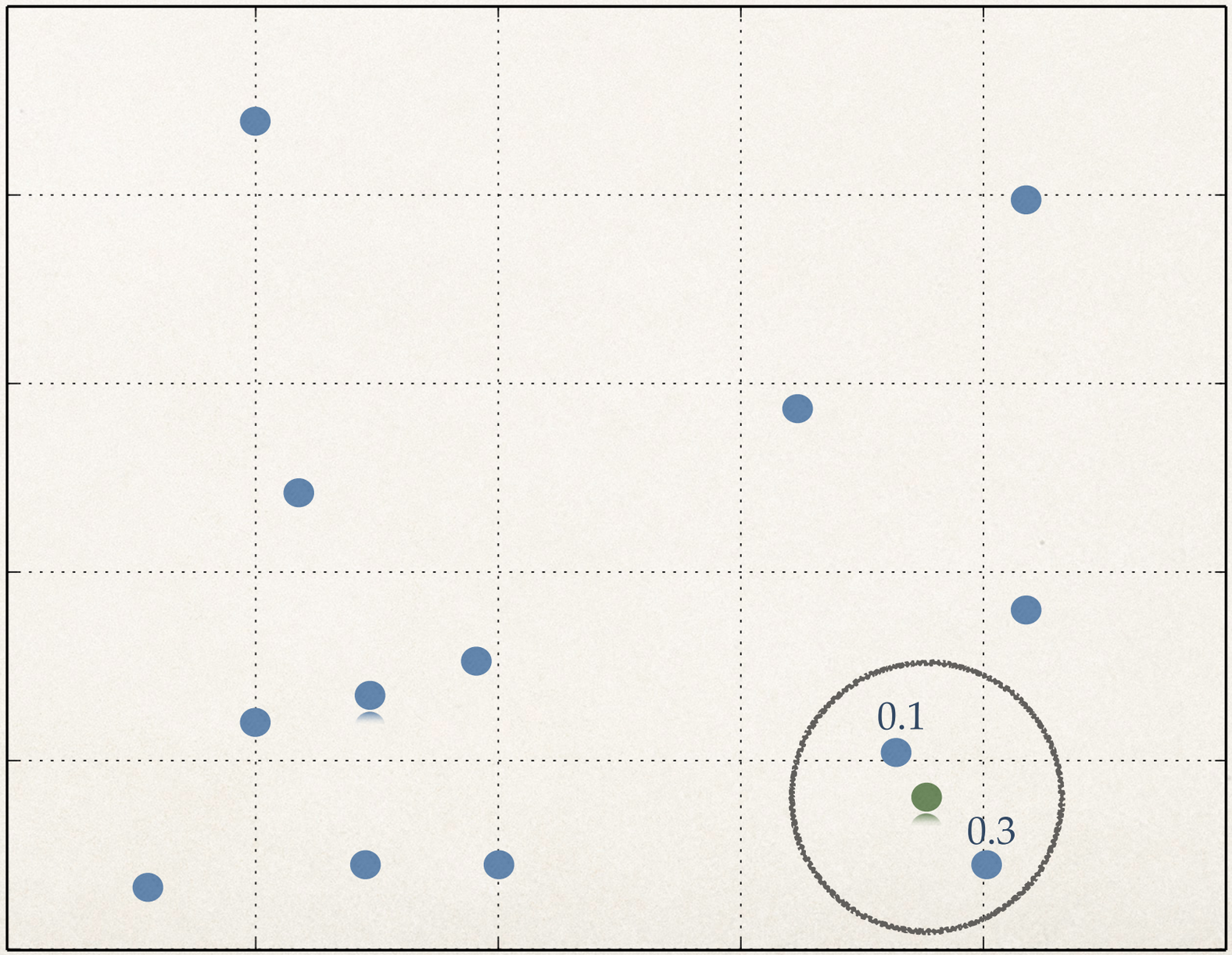
Markov

- accepted points



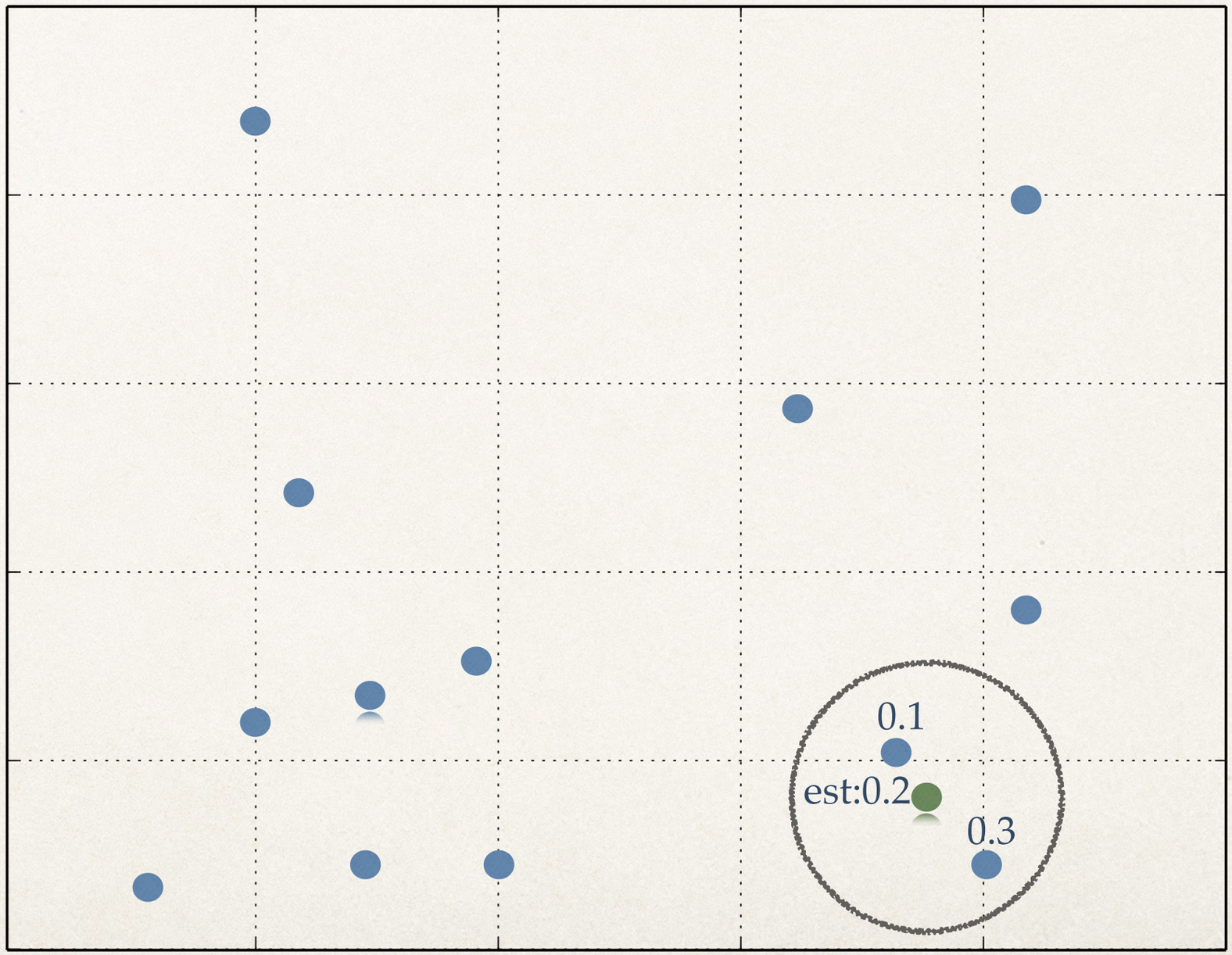
Markov

- accepted points



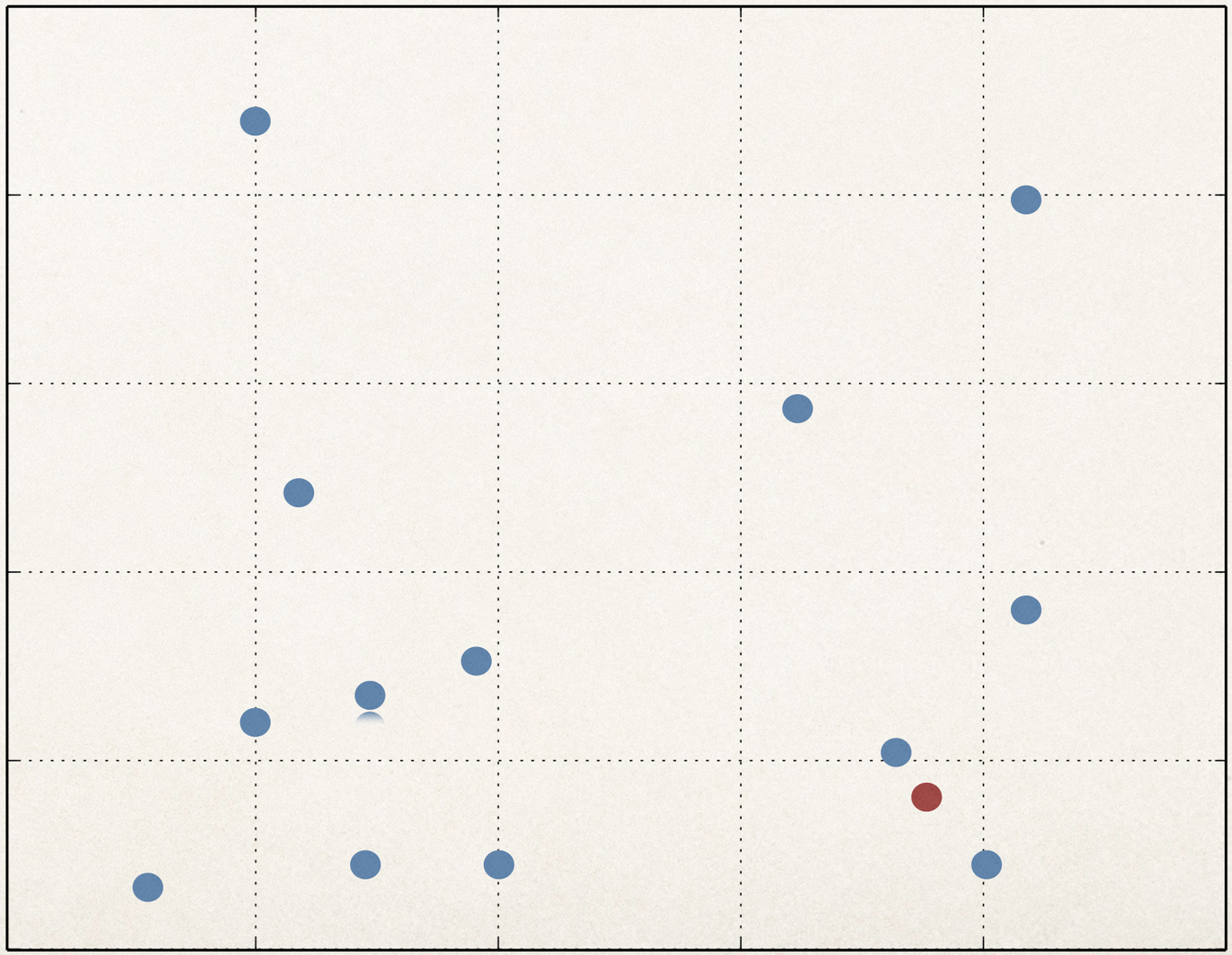
Markov

- accepted points



Markov

- accepted points
- rejected



Sampling Instances

- For independent parameters, uniform sampling is straightforward:
  - generate a value uniformly and independently for each parameter of the instance.
- For dependent parameters this is more difficult, consider where *n_upper* and *n_boats* are also parameters.
  - given capacity: function (total) int(1..n_boats) --> int(1..n_upper)
  - If generated independently, the sampling would be biased, since there are many more possible functions for large values of *n_boats* — meaning a particular function with less mappings is more likely to be picked.

Uniform Sampling

- One way of sampling uniformly from a specification that has dependent parameters is to enumerate all possible instances then select an instance from these.
- This is done creating a new Essence specification E^* which convert the original into an enumeration problem.
- given w, g, s : int(1..100) → find w, g, s : int(1..100)

Uniform Sampling

- A constraint model of E^*, is produced automatically, using Conjure's compact heuristic.
  - The compact heuristic greedily picks the transformations which produce smaller expressions.
- This is adequate since enumeration problems are usually easy, but time and space consuming.

Uniform Sampling

- Uniform sampling by this method is infeasible for large domains or large number of parameters.
- An alternative *solver-random* which uses Minion to produce a single solution by employing a random variable and value order. While scaleable this has an inherent bias, because of the distribution of the solutions.
- Experimentally over problems classes with instance spaces restricted to ~1 Billion instances, *solver-random* produced as discriminating instances as uniform sampling, while being more scalable.
- Therefore we use *solver-random* in the rest of the experiments.

Output

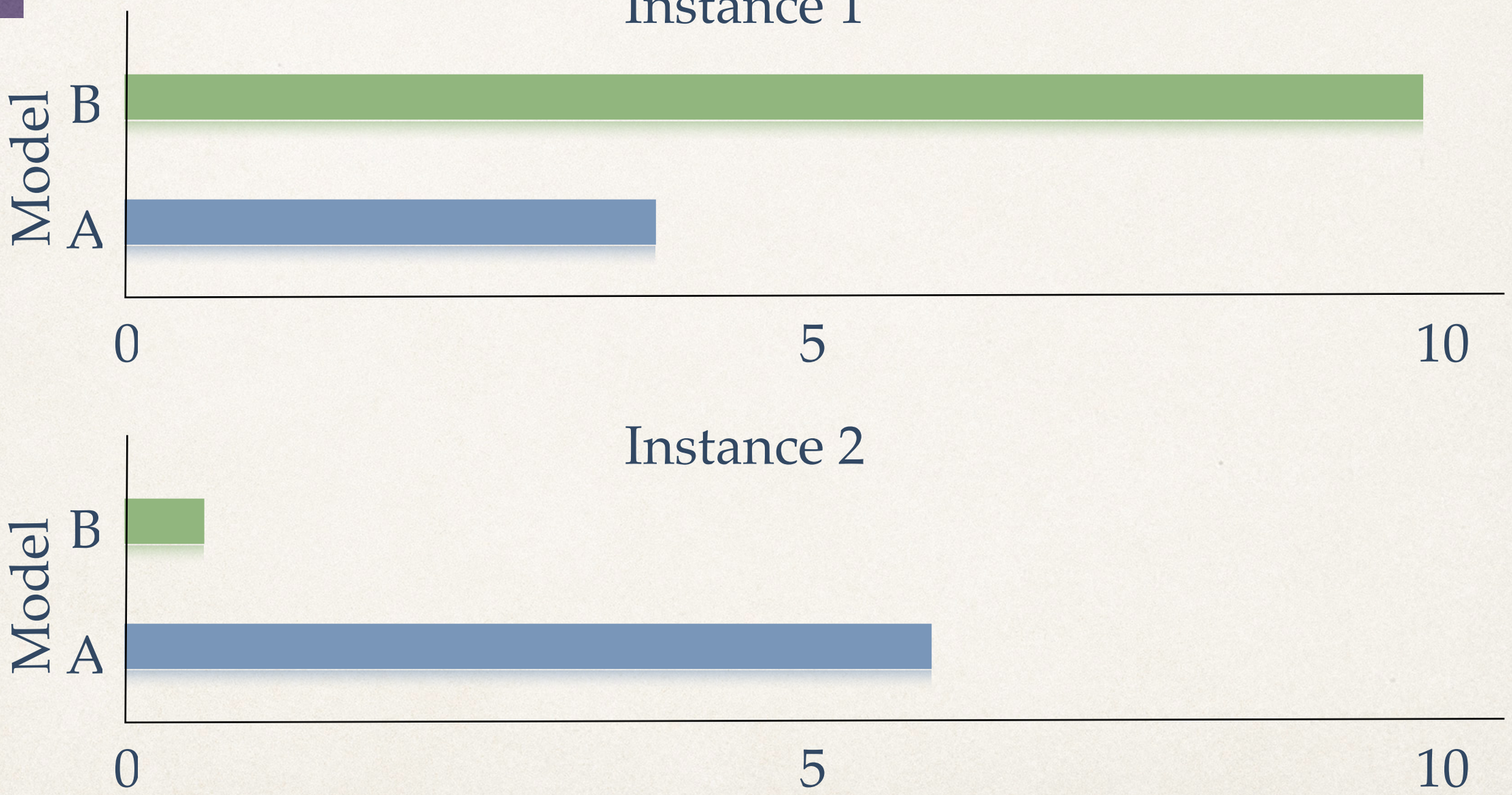
- We wish to pick a small set of models which have good performance.
- Once all the races on the generated instances are completed we have a set of models then has have the best runtime performance on each instance.
- Simple case: A subset of the models wins every races
  - We return that subset.

Fracturing

- The simple case does not always happen.
  - Sometimes no model wins every race.
- This means that means that instance space is fractured.

Fracturing

ρ=2



Fracturing

- A set of instances is *ρ-fractured* if every model is *ρ-dominated* on at least one instance.
- We find a minimum hitting sets of all *winners* of the races $\{a_1, a_2, a_3, \ldots\}$, which gives us coverage of all the instance while reducing the number of models
  - The choice of the hitting set is arbitrary, as with the non-fractured case we include models which we could not prove are worse (i.e non-dominated).
- We define the set A_i as the set of models that won every race that a_i won.
- The set of sets $\{A_1, A_2, A_3, \ldots\}$ then gives a summary of the winning models over all fractured parts of the instance space.

Output - Example

- Say we have 4 models *a*..*d* and 3 instances I_1..I_3
  - I_1 — *a* & *d* were winners.
  - I_2 — *a*, *b* & *d* were winners.
  - I_3 — *b* & *c* were winners.
- Minimum hitting set: {*a*, *b*}
  - *a* & *b* cover all instances, hence *c* is not needed.
- Set of sets : { {*a*, *d*}, {*b*} }
  - *d* won every race *a* did.

Experimental Setup

- 6 Problem classes
  - Knapsack, Langford's Problem, Social Golfers, Progressive Party,Warehouse Location Problem and Balanced Academic Curriculum.
- For each problem class, 30 races, each with a time budget of 6 hours:
  - The time budget is divided by the number of models to obtain the maximum time allowed for a particular model to solve an instance.
- We report the output set sizes, the steps to convergence and number of models the problem can started with, over three independent runs.

Experimental results - PPP

		Markov			SMAC		Undirected		
Problem	#Models	Output Sizes	Frac.	Steps to Conv.	Output Sizes	Frac.	Output Sizes	Frac.	Steps to Conv.
PPP-1	256	32,8	2	2	8	1	53,8	2	19
PPP-2	256	32,8	2	12	16	1	27,8	2	11
PPP-3	256	8	1	9	15	1	17	1	7

- PPP started of with most models.
- Instance space is fractured.
- When fracturing is detected, Markov is more consistent than Undirected in the size of the returned set.

Experimental results - PPP

		Markov			SMAC			Undirected		
Problem	#Models	Output Sizes	Frac.	Steps to Conv.	Output Sizes	Frac.		Output Sizes	Frac.	Steps to Conv.
PPP-1	256	32,8	2	2	8	1		53,8	2	19
PPP-2	256	32,8	2	12	16	1		27,8	2	11
PPP-3	256	8	1	9	15	1		17	1	7

- PPP started of with most models.
- Instance space is fractured.
- When fracturing is detected, Markov is more consistent than Undirected in the size of the returned set.

Experimental results

		Markov			SMAC		Undirected		
Problem	#Models	Output Sizes	Frac.	Steps to Conv.	Output Sizes	Frac.	Output Sizes	Frac.	Steps to Conv.
Knapsack-1	64	1,1	2	20	1,1	2	1,1	2	7
Knapsack-2	64	1,1	2	13	1,1	2	1,1	2	30
Knapsack-3	64	1,1	2	2	1,1	2	1,1	2	3
Langford-1	154	4	1	28	4	1	6	1	1
Langford-2	154	1	1	14	3	1	4	1	14
Langford-3	154	4	1	2	4	1	3	1	29

- Knapsack: All methods identified that the instance space is fractured into two parts.
- Langford: Markov shows a slight edge in performance returning a single model in one case.

Experimental results - BACP

		Markov			SMAC		Undirected		
Problem	#Models	Output Sizes	Frac.	Steps to Conv.	Output Sizes	Frac.	Output Sizes	Frac.	Steps to Conv.
BACP-1	48	1,1	2	24	26	1	1	1	8
BACP-2	48	1	1	1	8	1	1	1	25
BACP-3	48	1	1	2	25	1	1	1	7

- SMAC performed poorly, because of the difficulty of encoding:
  - BACP has a complex instance space defined by 7 integers, a function and a relation.
  - When one *given* depends on another, such as the size of the domain of the function: we must be conservative: sufficient SMAC parameters are used to accommodate the maximum size of the structured given.
  - Although SMAC has demonstrated that it is able to handle large parameter spaces, this conservative encoding may hinder its ability to cover the space effectively, since many of the values it is producing may be ignored.

Experimental results - Summary

- Our experimental evaluation shows that all of these methods are capable of reducing a large number of possible models to a much smaller set.
- Overall Markov was able to detect the most fractures and reduce the number of models the most.
- For SMAC, the necessary conservative encodings may hinder its ability to cover the space effectively.

Conclusion

- During Automated Modelling we generate many alternate models.
- To select between them we generate discriminating instances drawn from the problem class.
- The methods are able to detect fracturing if it occurs and successfully determine the best models for each fraction.