CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Computing@CIT

Metaheuristic Optimization

LS for SAT/ Simulated Annealing

Dr. Diarmuid Grimes

Additional resources:

SAT Solvers: https://www.cs.ubc.ca/~davet/ubcsat/algorithms/

SAT benchmarks: https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html

Example random k-SAT instance generation method

Random k-CNF formulas with *N* variables and *L* clauses:

Sample uniformly from space of all possible *k*-clauses

nClauses = 0

while nClauses < L:

- Pick with uniform probability a set of *k* atoms over *N*
- Randomly negate each atom with probability 0.5
- Create a disjunction of the resulting literals
- If this clause not already in our set of clauses, add it and increment nClauses

Why random?

- Allows us to generate instances of varying difficulty
- Most difficult instances are ones of interest for assessing performance of solvers

  For given *k* fix *N*
  For increasing *L*, randomly generate and solve x instances for these k, N, and each L
  Solve all instances with exact method and store satisfiability percentages for each L/N

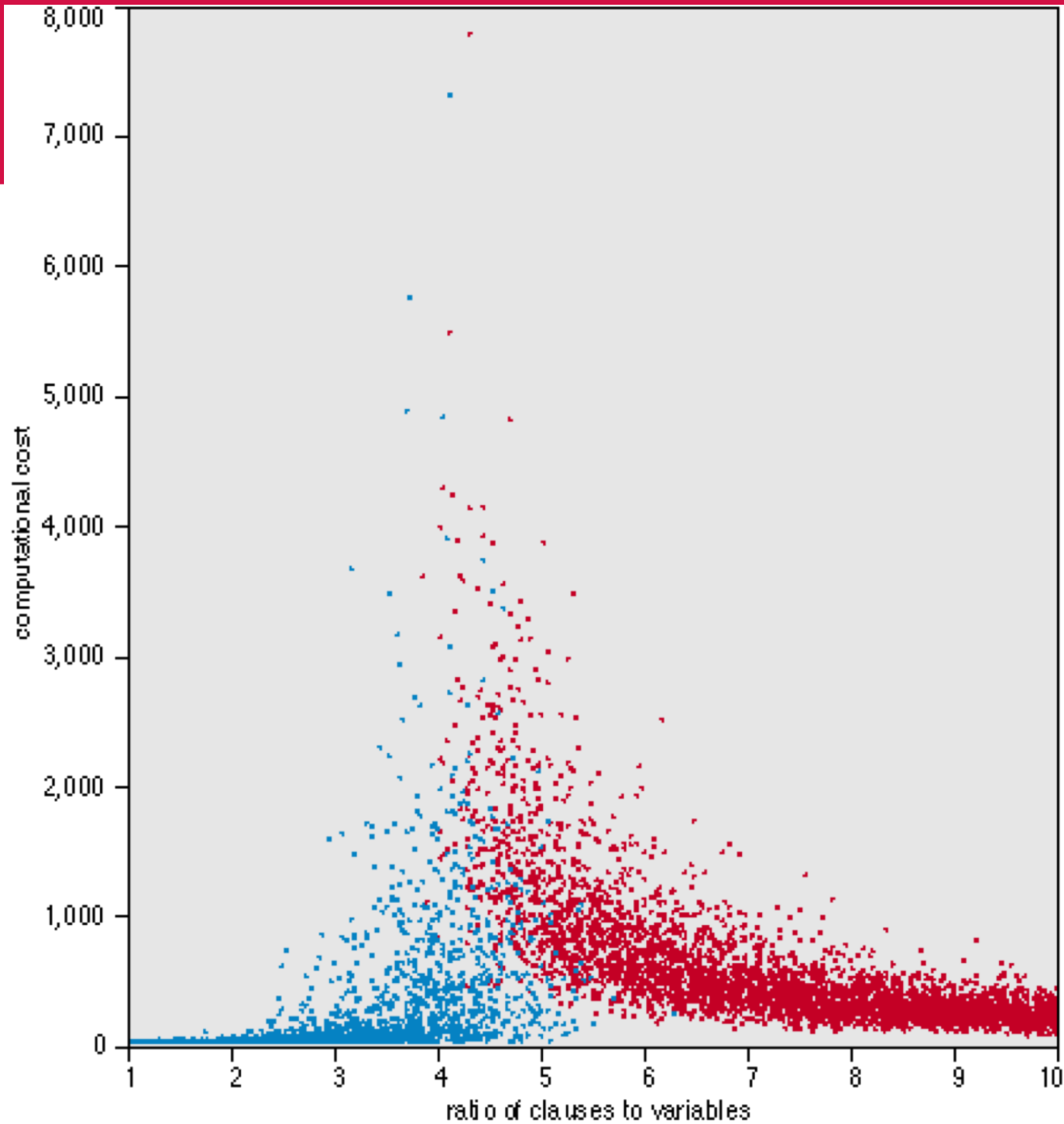
- Most difficult instances are (typically) at the *phase transition*, the point where the set of instances goes from mainly being satisfiable to mainly being unsatisfiable

Phase Transition for SAT

- 2-SAT occurs at $l/n=1$

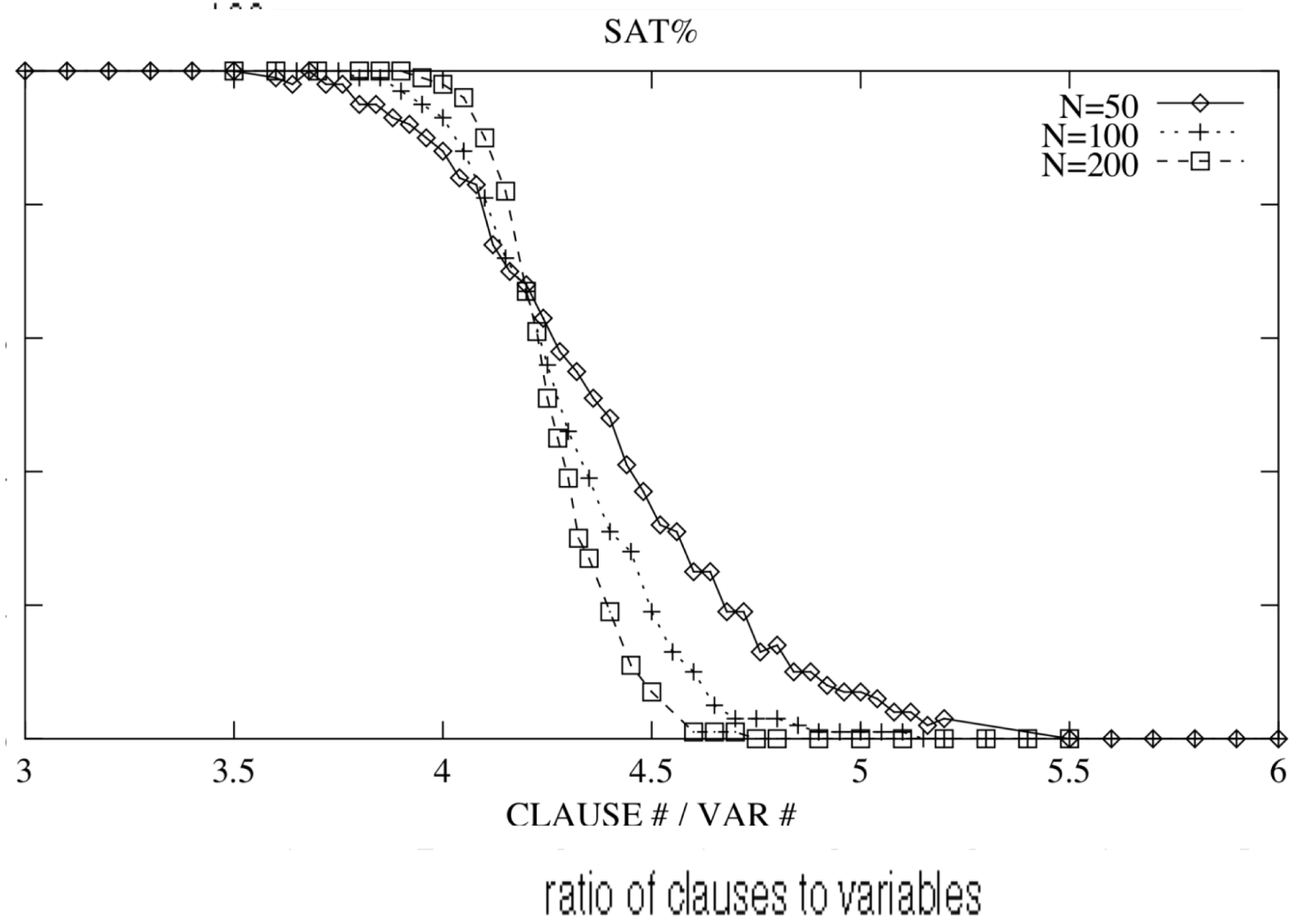
  [Chavatal & Reed 92, Goerdt 92]

- 3-SAT occurs at $3.26 < l/n < 4.598$
- Which are the hardest instances?
  - around $l/n = 4.3$



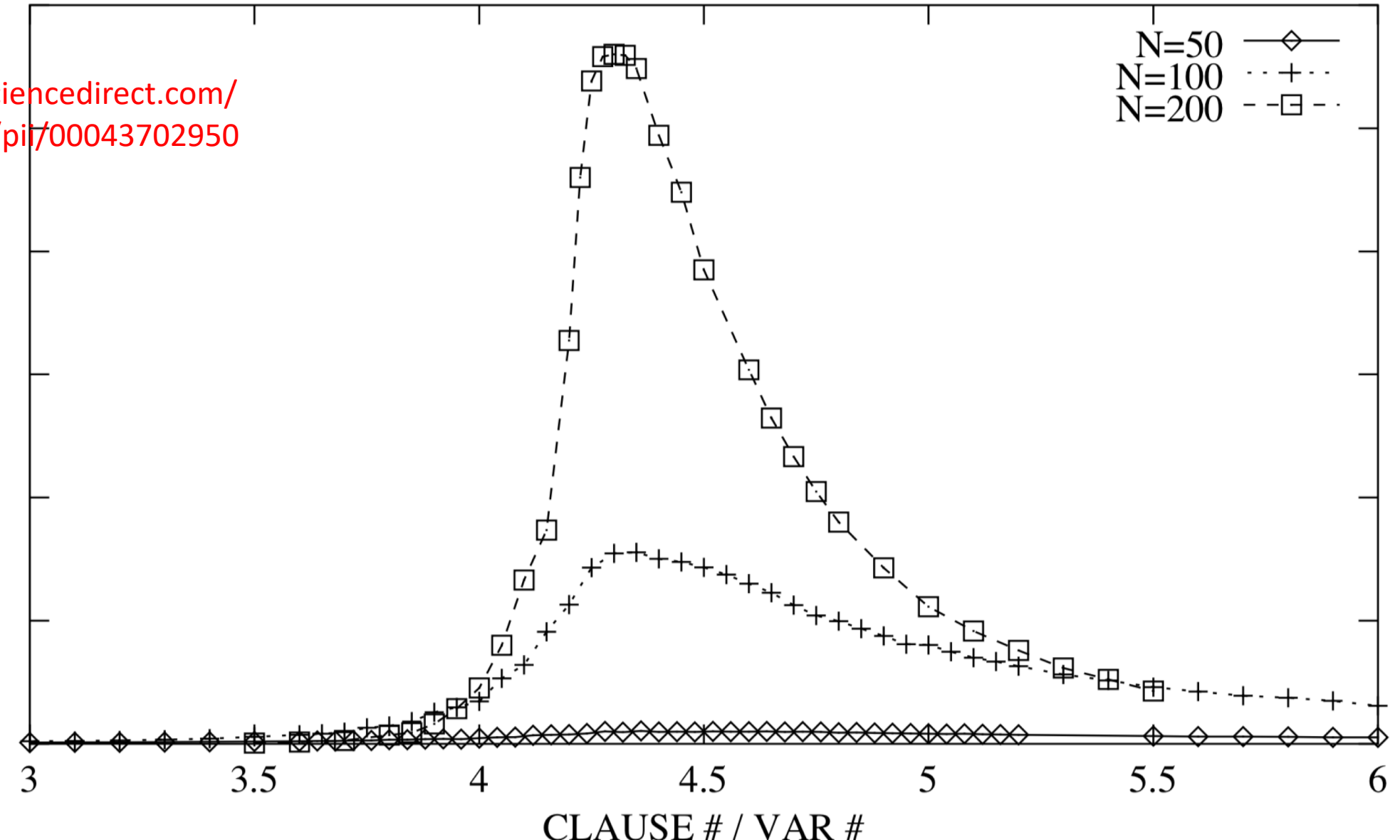
Phase Transition for 3-SAT

- l/n < 4.3 problems under-constrained and SAT
- l/n > 4.3 problems over-constrained and UNSAT
- l/n=4.3, problems on “knife-edge” between SAT and UNSAT



Phase Transition for 3-SAT: Solving cost for different N

Related work:
https://www.sciencedirect.com/science/article/pii/0004370295000453



Local Search for SAT

Local Search for SAT

- Choose a random initial state I
  - Random values for the variables (0 or 1 to all variables)
- If I is not a valid solution, repeatedly choose a variable p and change its value in I (flip the variable)
  - Flip(I,p) determines the successor state
- Flipped variables are chosen using heuristics or randomly or both
  - The evaluation function is usually the number of satisfied clauses in a state
- Most algorithms use random restarts if no solution has been found after a fixed number of search steps
- Several instantiations of this general framework have been proposed

Definitions

- Positive/Negative/Net ***Gain***
  - Given candidate variable assignment T for CNF formula F, let
    - B_0 be the total # of clauses that are currently unsatisfied in F
    - T' be the state of F if variable V is flipped
    - B_1 be the total # of clauses which would be unsatisfied in T'.
  - The net gain of V is B_0-B_1. The negative gain of V is the # of clauses which are satisfied in T but unsatisfied in T'. The positive gain of V is the # of clauses which are unsatisfied in T but satisfied in T'.

- Variable Age
  - The age of a variable is the # of flips when it was last flipped.

Example: 20 variables, 91 clauses instance

C1: 4 -18 19

C2: 3 18 -5

⋮

- Let's say our current state has

… X3 = F, X4 = F, X5 = T, …, X18 = T, X19 =F, …

And we are considering the flip of variables in unsat clauses such as C1

- If we flip X18 then C1 will now be satisfied, as will any other unsat clause that contained X18 (since these must have contained -X18, otherwise X18 would have been satisfied).
- However, C2 will now be unsat (since X3=F and X5=T)

GSAT - Greedy SAT

- The GSAT Algorithm (*Selman, Mitchell, Levesque, 1992*)
  - search initialisation: randomly chosen assignment
  - in each search step, flip the variable which gives the maximal increase in the number of satisfied clauses (*maximum net gain*)
    - ties are broken randomly
  - if no model found after *maxSteps* steps, restart from another randomly chosen assignment

- **HSAT** (*Gent and Walsh, 1993)* – Same as GSAT, but break ties in favor of maximum age variable.

GSAT

- Important point in efficiently implementing GSAT is to store a **score** for each variable which is the number of **unsatisfied** clauses the variable is in.
- The complete set of scores are computed only once at the **beginning** of each try (i.e. after initial assignment)
- Then, after each flip, update **only** the scores of those variables which were possibly affected by the flipped variable

GWSAT

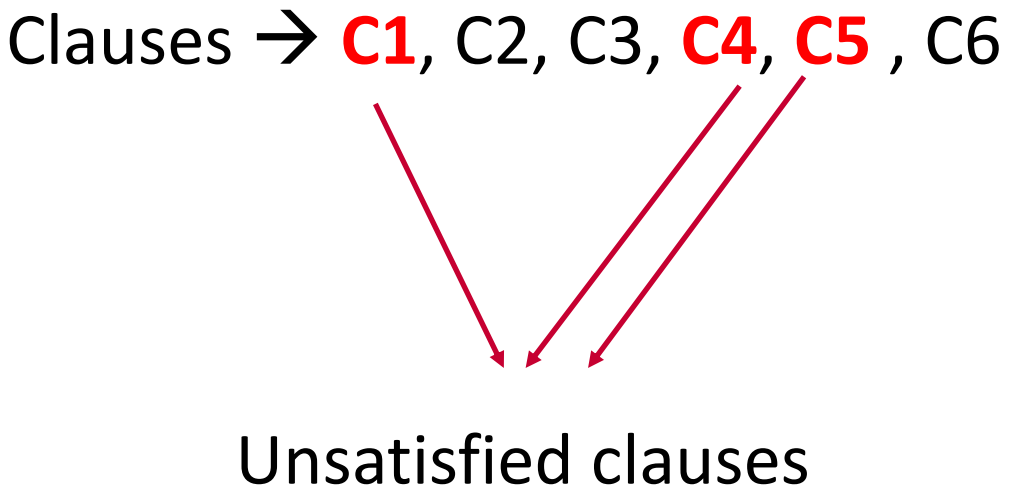
- The GWSAT Algorithm (Selman, Kautz, Cohen, 1994)
  - Search initialization: randomly chosen assignment
  - Random Walk Step: randomly choose a currently unsatisfied clause and a literal in that clause, flip the corresponding variable to satisfy the clause
  - GWSAT steps: choose probabilistically between a GSAT step and a Random WalkSAT step with *'walk probability'* (noise) ***wp***

Diversification, speed

WalkSAT

- The WalkSAT algorithm family
  - Search initialization: randomly chosen assignment
  - Search step
    1. Randomly select a currently unsatisfied clause
    2. Select a variable from this clause according to a heuristic h
  - If no valid solution found after *maxSteps* iterations, restart from randomly chosen assignment
- Original WalkSAT (Selman, Kautz, Cohen 1994)
  - Pick random unsatisfied clause BC
  - If at least one variable in BC has negative gain of 0, randomly select one of these variables.
  - Otherwise, with probability p, select random variable from BC to flip, and with probability (1-p), select variable in BC with minimal negative gain (break ties randomly)

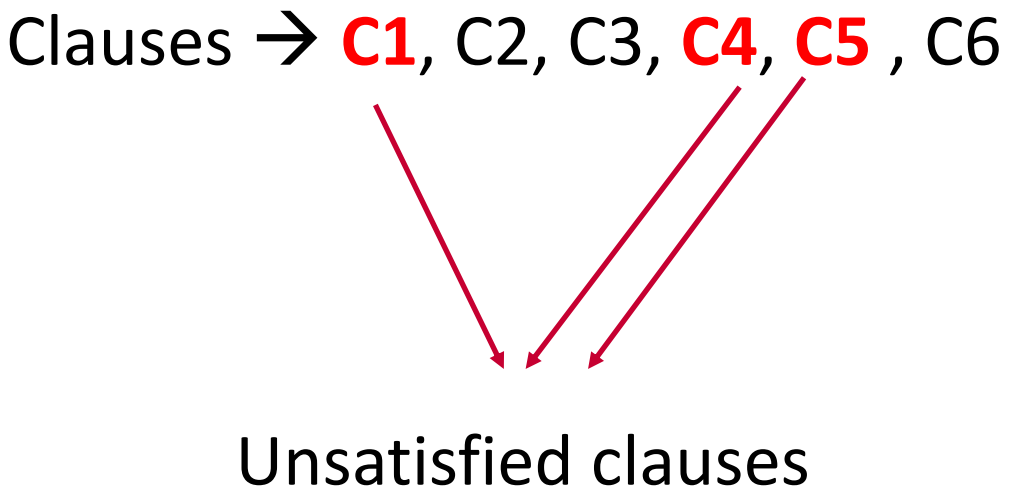
WalkSAT General



Select an unsatisfied clause (Random)

Select a variable in the selected clause

WalkSAT



Select an unsatisfied clause (Random)

With probability *wp*

Else (so with probability 1-*wp*)

Select a random variable in the selected clause

Select the “best” variable

WalkSAT

- Beating a correct, optimized implementation of Walksat is difficult!
- Many variants of “Walksat” appear in literature
  - Many of them are incorrect interpretations/ implementations of [Selman, Kautz, Cohen] Walksat.
  - Most of them perform significantly worse.
  - None of them perform better.

- Positive gain: #clauses that were unsatisfied and now become satisfied
- Negative gain: #clauses that were satisfied and now become unsatisfied

Variations of WalkSAT

Basic is often referred to as SKC:

- Select variable such that minimal number of currently satisfied clauses become unsatisfied by flipping
  - If ‘zero-damage’ possible, always go for it
  - Otherwise, with probability *wp* variable is randomly selected

Tabu:

- use constant length tabu-list for flipped variables and random tie-breaking.

Tabu search (SAT)

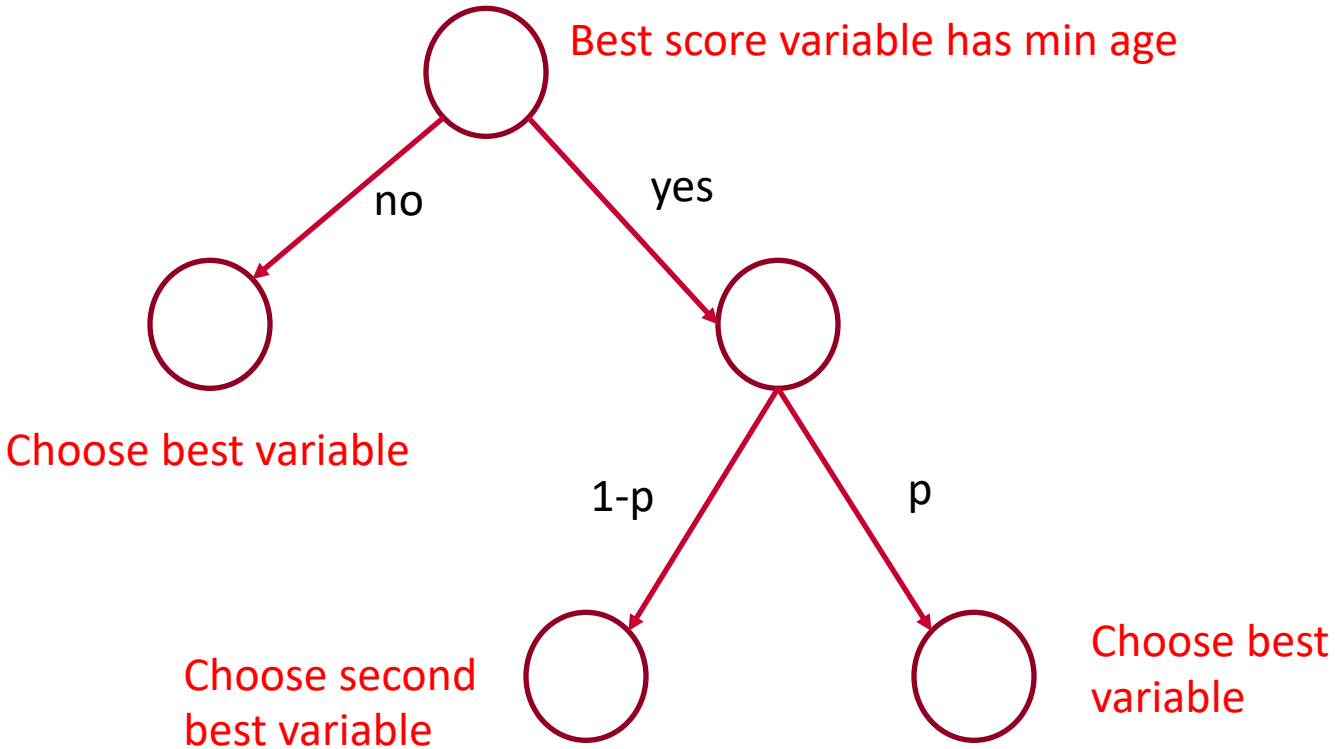
- Performance of Tabu search depends crucially on setting of tabu length tl:
  - tl too low → search stagnates due to inability to escape from local minima
  - tl too high → search becomes ineffective due to overly restricted search paths (adjust neighborhoods too small)

SAT Local Search: Novelty Family

- **Novelty** [McAllester, Selman, Kautz 1997]
  - Pick random unsatisfied clause *BC*.
  - Select variable *v* in *BC* with **maximal net gain**, unless *v* has the **minimal age** in *BC*.
  - In the latter case, select *v* with probability (1-*p*); otherwise, flip v_2 with 2nd highest net gain.
- **Novelty+** [Hoos and Stutzle 2000] – Same as Novelty, but after BC is selected, with probability p, select random variable in BC; otherwise continue with Novelty
- **R-Novelty** [McAllester, Selman, Kautz 1997] and **R-Novelty+** [Hoos and Stutzle 2000]– similar to Novelty/Novelty+, but more complex.

Novelty

- Choose a random unsatisfied clause c
- Pick a variable x from c according to the following heuristic:



Example of Novelty move

- Let's say we have an unsat clause with variables X1, X2 and X3
- X1 is currently in 3 unsat clauses, x2 is in 4 and x3 is in 2 unsat clauses
- BUT ... X1 is most recently flipped.

Best Variable

2nd Best Variable

	X1	X2	X3
# Cs currently unsat for V	3	4	2
# Cs unsat if V is flipped	4	8	4
Net gain	-1	-4	-2
Iteration last flipped	47	36	21

Best h-value

2nd Best h-value

GSAT/HSAT/GWSAT/WalkSAT move

	X1	X2	X3	X4
# Cs currently unsat for V	3	4	2	2
# Cs unsat if V is flipped	4	8	4	3
Net gain	-1	-4	-2	-1
Iteration last flipped	47	36	21	61
Negative gain	?	?	?	

- Wp = 0.3 for GWSAT and for WalkSAT
- What happens if random number generated is 0.1?

Adaptive Novelty+ (very effective)

- With probability p (initially set to say 1%) (the "+" part) **Diversification**
  - Choose randomly among all variables that appear in at least one **unsatisfied** clause
- Else be greedier (other 99%) using Novelty heuristic **Intensification**
  - Randomly choose an **unsatisfied** clause C
  - Choose the most-improving variable in C depending on its variable age and on a probability for choosing 2nd most-improving variable (the "Novelty" part)
- The "adaptive" part:
  - If we improved # of satisfied clauses, decrease p slightly **Increase Intensification**
  - If we haven't got an improvement for a while, increase p **Increase Diversification**
  - If we've been searching too long, restart the whole algorithm
  - Idea is to use the random walk escape mechanism only when it is really needed.

Structure of the standard SAT local search variable selection heuristics

- **Score variables with respect to gain metric**
  - Walksat uses negative gain
  - GSAT and Novelty use net gain
- **Restricted set of candidate variables**
  - GSAT – any variable in formula F
  - Walksat/Novelty – pick variable from a single random unsatisfied clause
- **Ranking of variables with respect to scoring metric**
  - Greedy choices considered by all heuristics.
  - Novelty variants also consider 2nd best variable.
- **Variable Age – prevent cycles and force exploration**
  - Novelty variants, HSAT, Walksat+tabu

Some observations on the variable selection heuristics

- Difficult to determine a priori how effective any given heuristic is.
- Many heuristics look similar to Walksat, but performance varies significantly
- Empirical evaluation necessary to evaluate complex heuristics. Previous efforts to discover new heuristics involved significant experimentation.
- Humans are skilled at identifying primitives, but find it difficult/time-consuming to combine the primitives into effective composite heuristics.

Simulated Annealing

Hill-climbing: stochastic variations

- Stochastic hill-climbing
  - Random selection among the uphill moves.
  - The selection probability can vary with the steepness of the uphill move.

- To escape local minima
  - Random-restart hill-climbing
  - Random-walk hill-climbing
  - Hill-climbing with both

Hill-climbing: stochastic variations

- When the state-space landscape has local minima, any search that moves only in the greedy direction cannot be complete
- Random walk, on the other hand, is asymptotically complete

- Idea: Put random walk into greedy hill-climbing

Hill-climbing with random restarts

- **If at first you don't succeed, try, try again!**
- Different variations
  - For each restart: run until termination vs. run for a fixed time
  - Run a fixed number of restarts or run indefinitely
- Analysis
  - Say each search has probability p of success
    - E.g., for 8-queens, p = 0.14 with no sideways moves
  - Expected number of restarts?
  - Expected number of steps taken?

Hill-climbing with random walk

- At each step do one of the two
  - Greedy: With prob p move to the neighbor with largest value
  - Random: With prob 1-p move to a random neighbor

Hill-climbing with both

- At each step do one of the three
  - Greedy: move to the neighbor with largest value
  - Random Walk: move to a random neighbor
  - Random Restart: Resample a new current state

Also known as Randomized Iterative Improvement

Metropolis(-Hastings) Heuristics

- Basic idea: rather than any random move choose a move relative to how good it is
  - Accept a move if it improves the objective value
  - Accept "bad moves" as well with some probability
  - The probability depends on how "bad" the move is
  - Inspired by statistical physics

Intensification

Diversification

- How to choose the probability?
  - t is scaling parameter (called temperature)
  - Δ is the difference between fitness of state with bad move versus current: f(s') – f(s)
  - A degrading move is accepted with probability $e^{-\Delta/t}$ **Will always be between 0 and 1**

Metropolis Heuristics

- What happens for a large t?
  - Probability of accepting a degrading move is large
- What happens for a small t?
  - Probability of accepting a degrading move is small
- Finding the "*correct*" temperature is hard
- Idea: gradually change the temperature

Simulated Annealing

- Simulated Annealing = physics inspired twist on random walk
- Basic ideas:
  - Like hill-climbing identify the quality of the local improvements
    - *But, instead of picking the best move, pick move randomly*
    - *Compute the change in objective function, δ.*
      - *If δ is positive, then move to that state*
      - *Otherwise: Move to this state with probability proportional to δ*
  - Thus worse moves (very large negative δ) are executed less often
  - However, there is always a chance of escaping from local maxima
  - Over time, make it less likely to accept locally bad moves

Physical Interpretation of Simulated Annealing

- A Physical Analogy:
  - imagine letting a ball roll downhill on the function surface
    - this is like hill-climbing (for minimization)
  - now imagine shaking the surface, while the ball rolls, gradually reducing the amount of shaking
    - this is like simulated annealing
- Annealing = physical process of heating/cooling a liquid or metal until particles achieve a certain frozen crystal state
  - Simulated annealing:
    - Free variables are like particles
    - Seek “low energy” (high quality) configuration
    - Slowly reducing temp. T with particles moving around randomly

Simulated Annealing

function SIMULATED-ANNEALING(*problem, schedule*) **return** a solution state

input: *problem*, a problem
schedule, a mapping from time to temperature

local variables: *current*, a node.
next, a node.
T, a "temperature" controlling the prob. of downward steps

current ← MAKE-NODE(INITIAL-STATE[*problem*])
for t ← 1 to ∞ do
T ← *schedule*[*t*]
if *T = 0* **then return** *current*
next ← a randomly selected successor of *current*
ΔE ← VALUE[*next*] - VALUE[*current*]
if *ΔE* > 0 **then** *current* ← *next*
else *current* ← *next* only with probability $e^{\Delta E / T}$

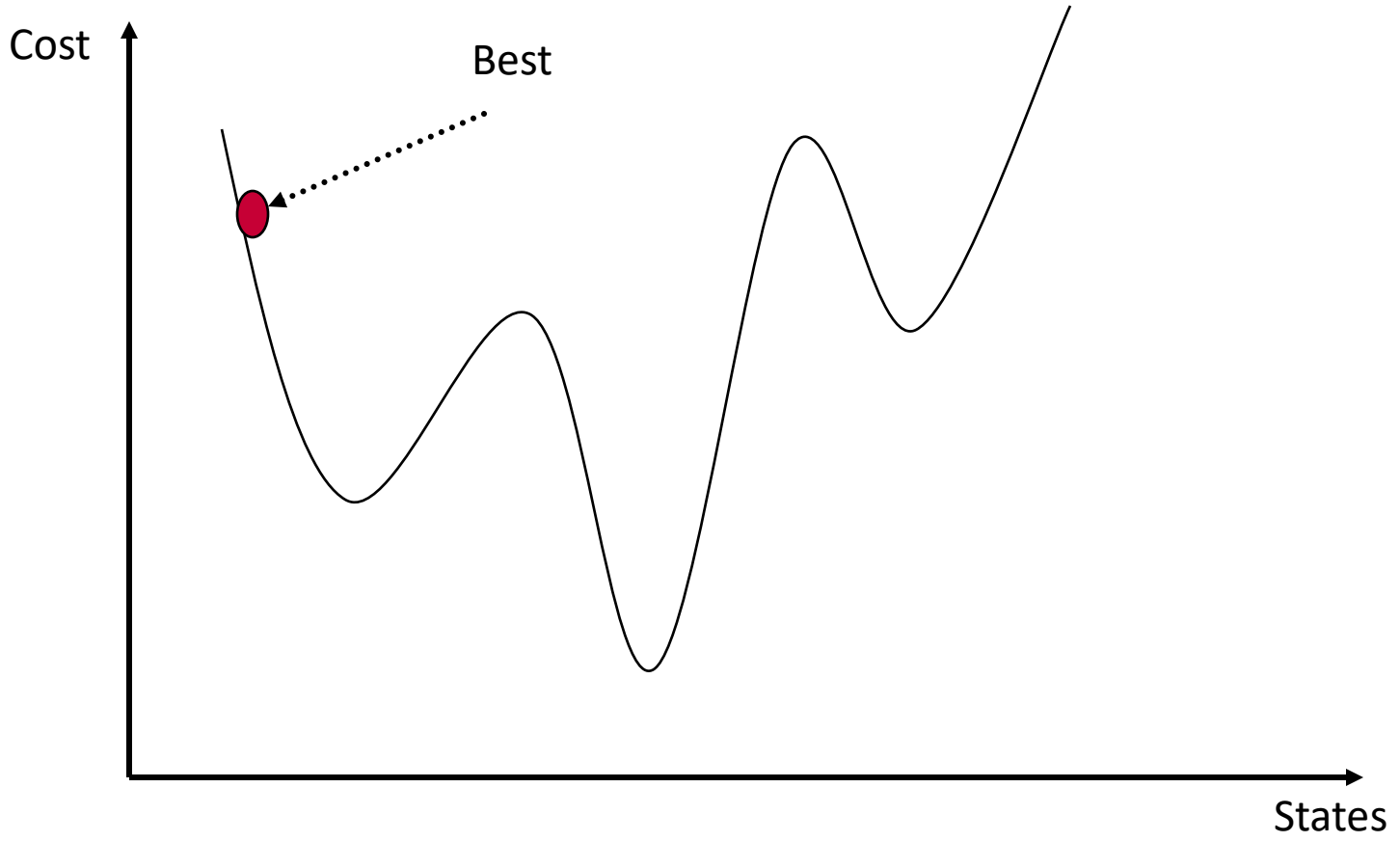
- say the change in objective function is δ
- **if** δ is **positive**, then move to that state
- **otherwise**:
  - move to this state with probability proportional to δ
  - thus: worse moves (very large negative δ) are executed less often

Similar to hill-climbing, but a random move instead of best move

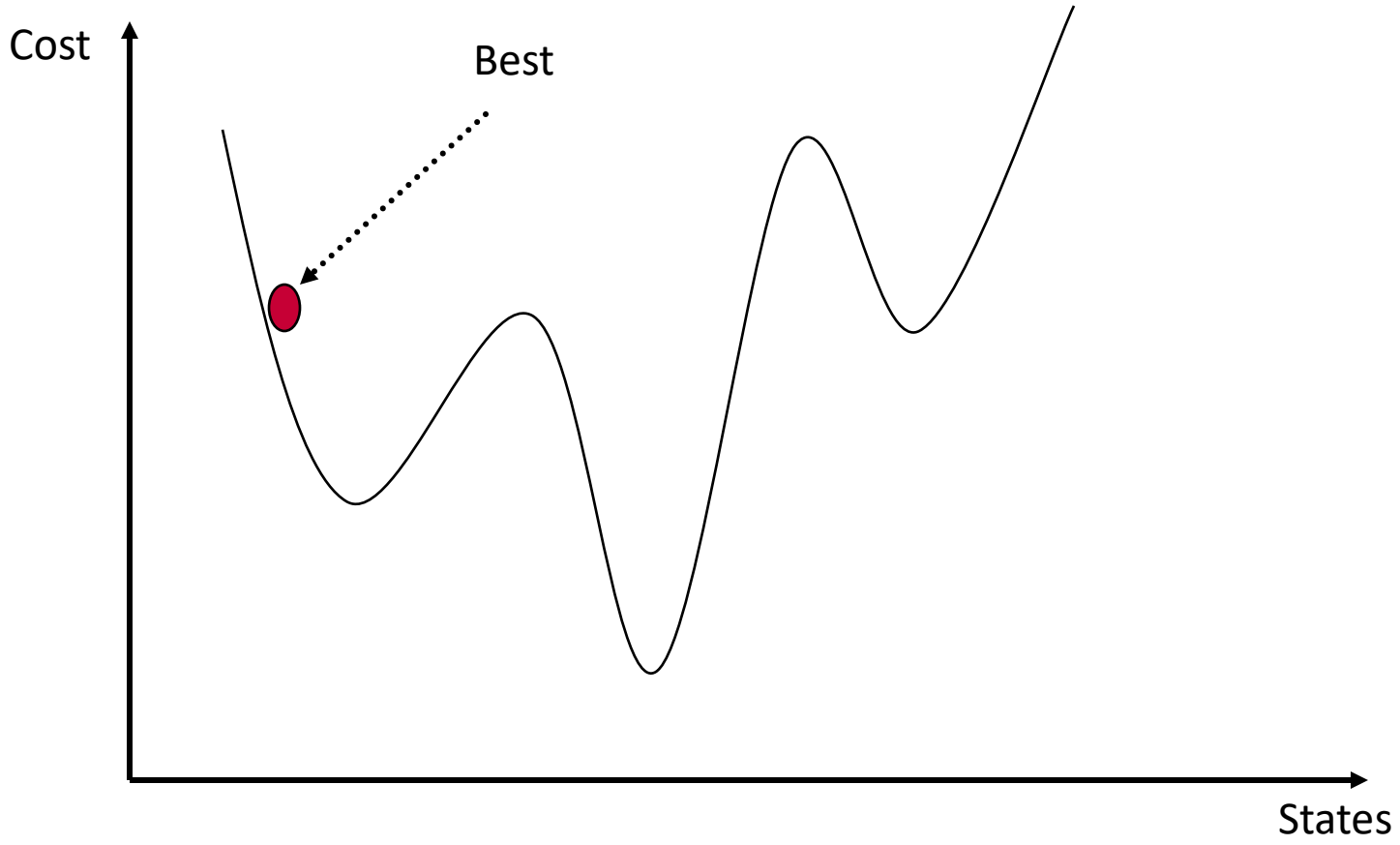
If improvement, make the move

Otherwise, choose this move with probability that decreases exponentially with the "badness" of the move

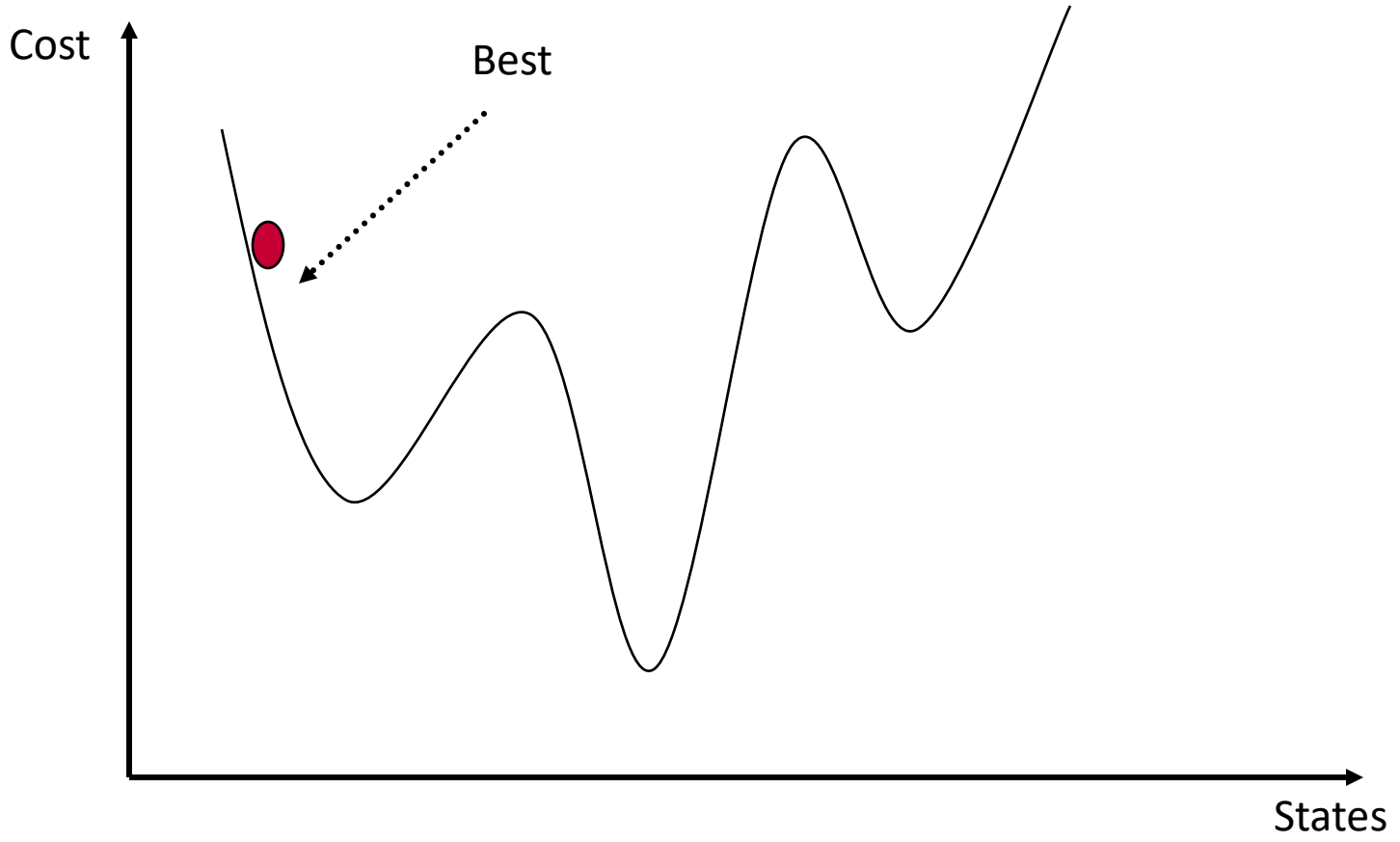
Simulated annealing



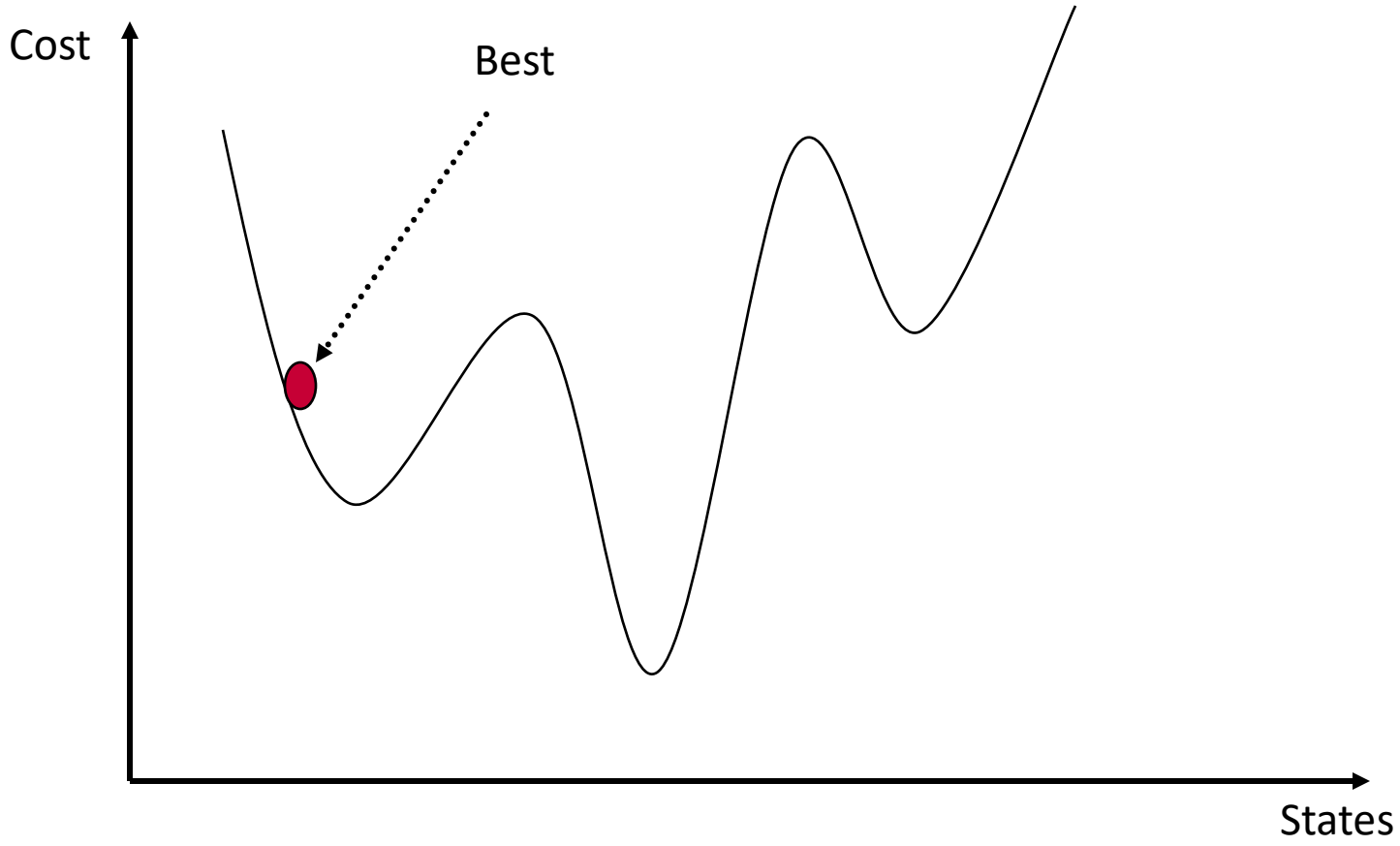
Simulated annealing



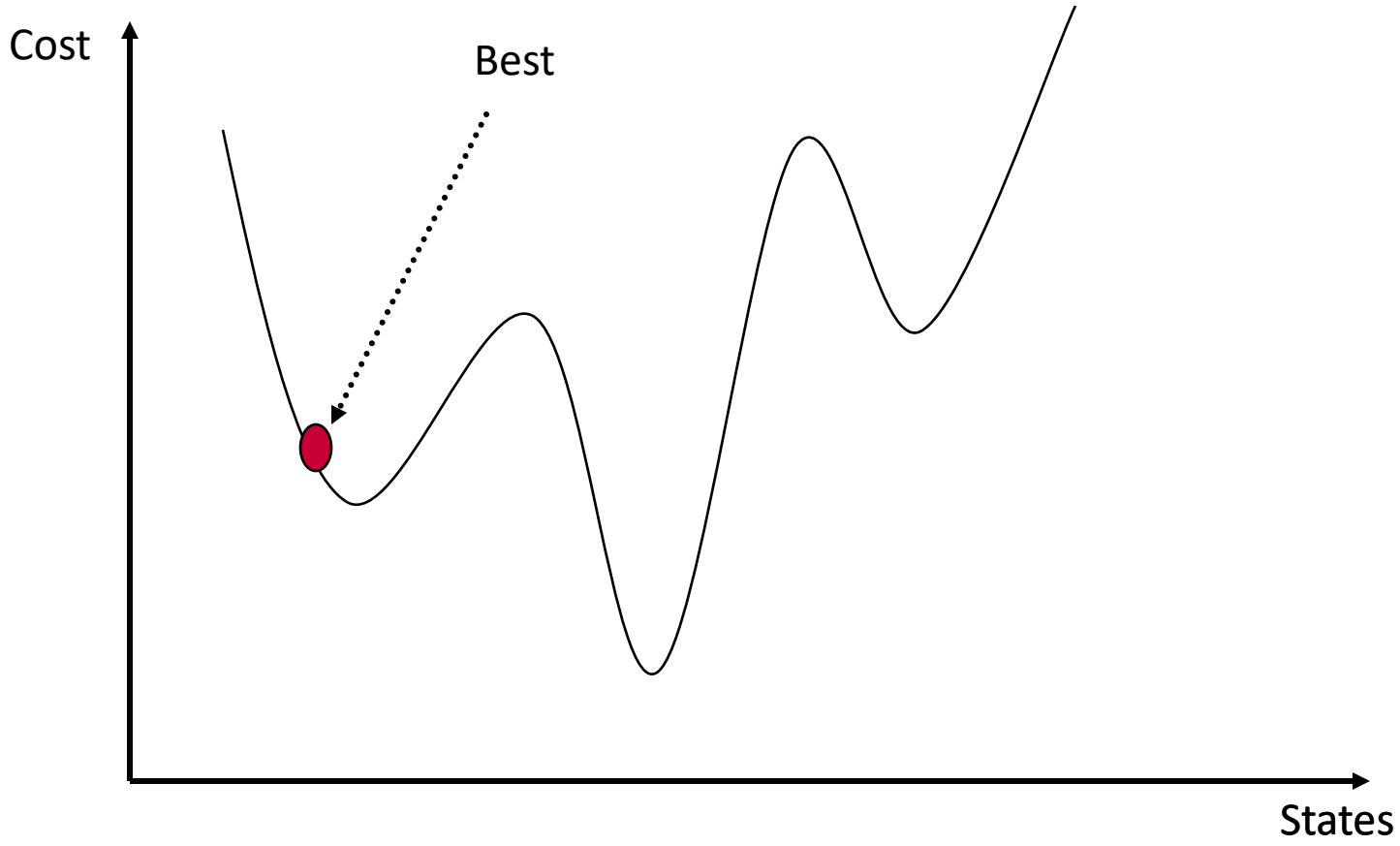
Simulated annealing



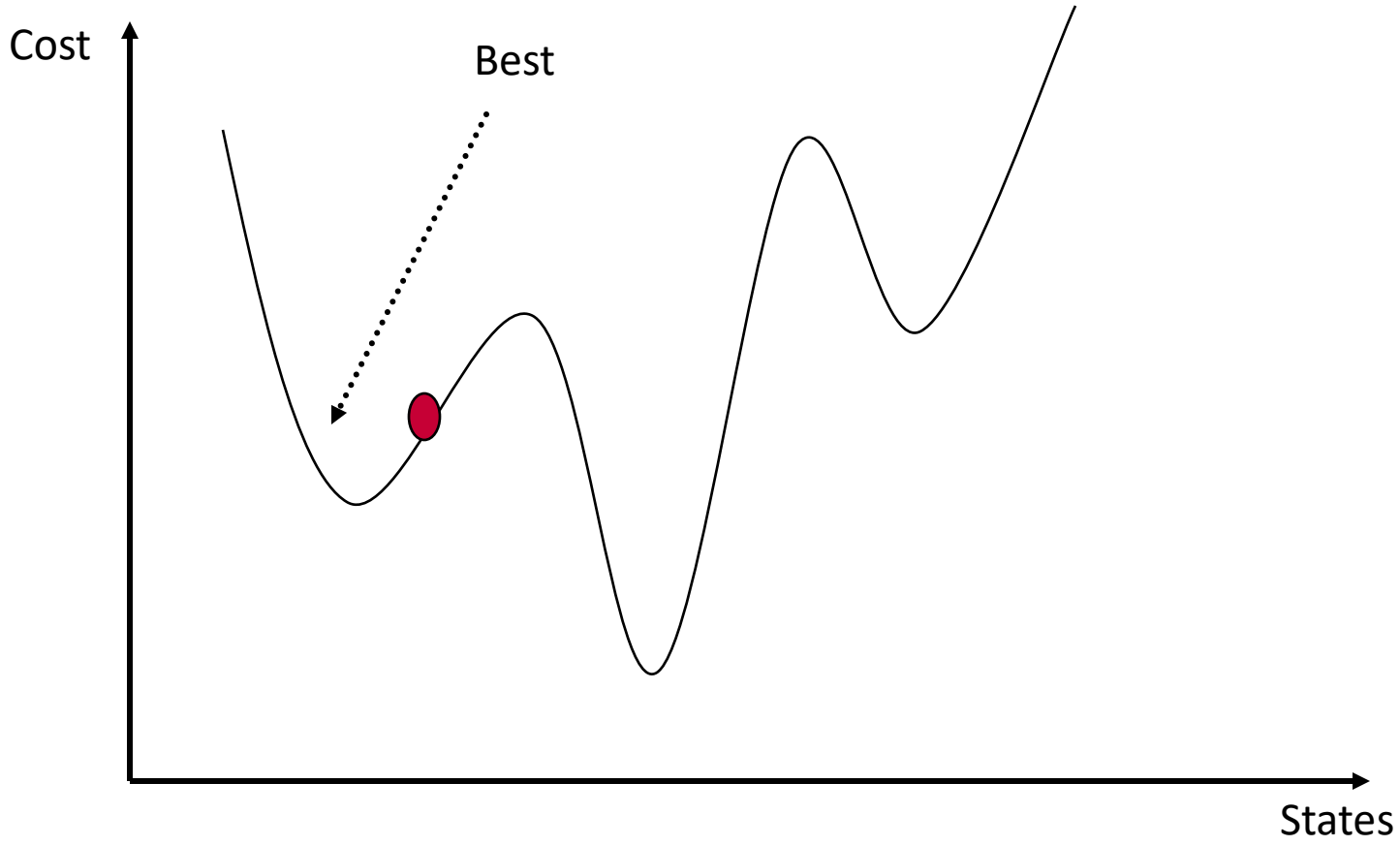
Simulated annealing



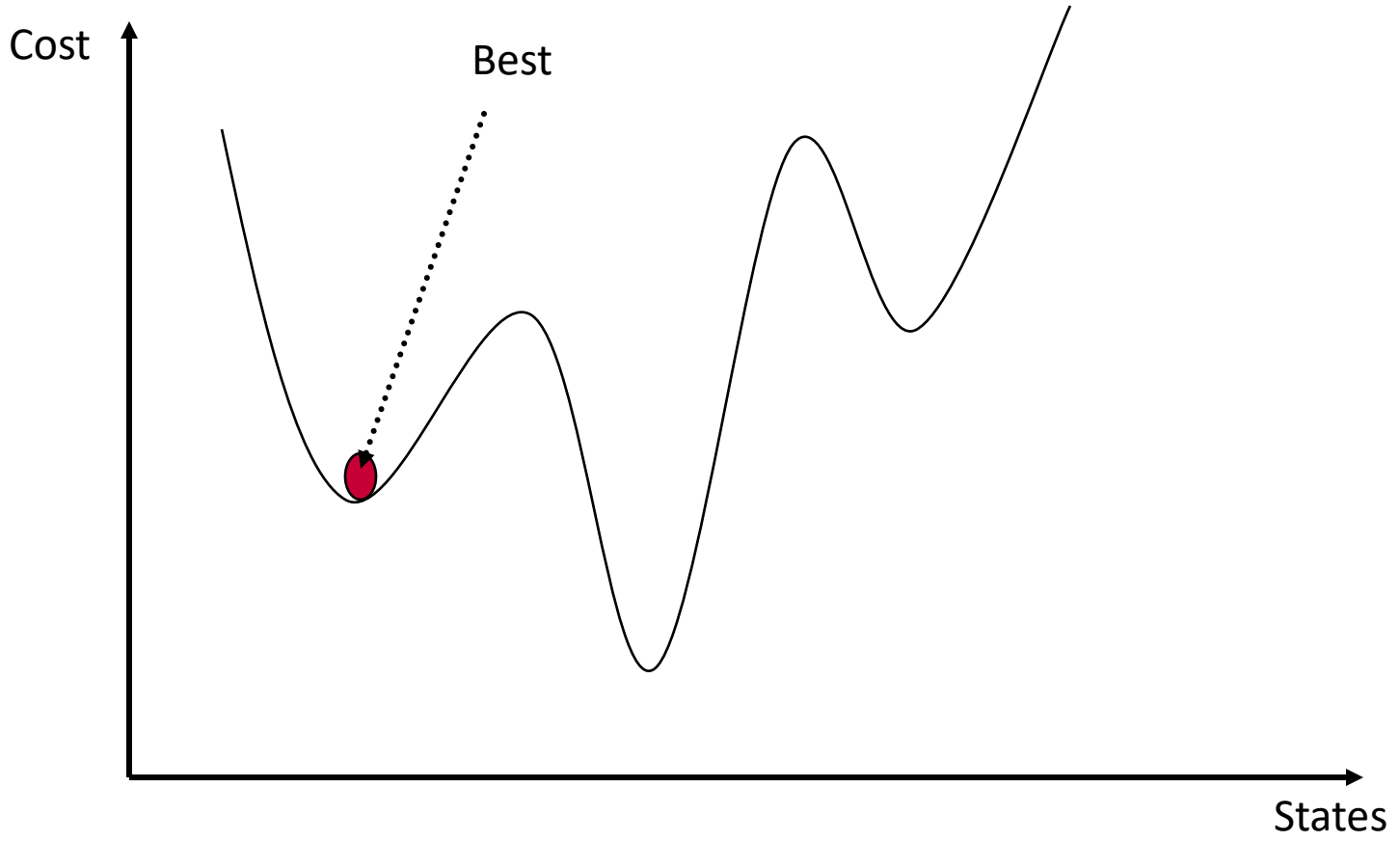
Simulated annealing



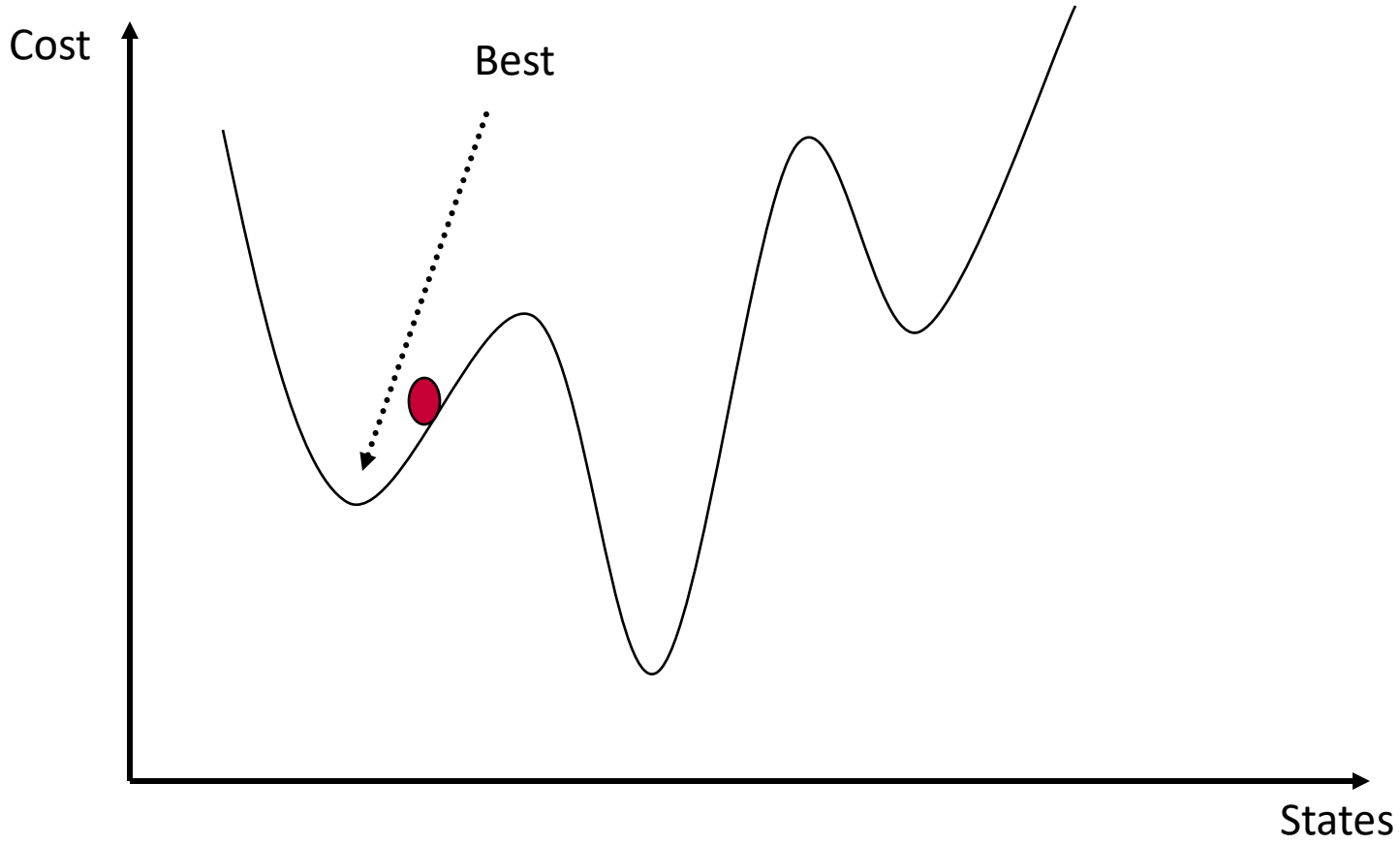
Simulated annealing



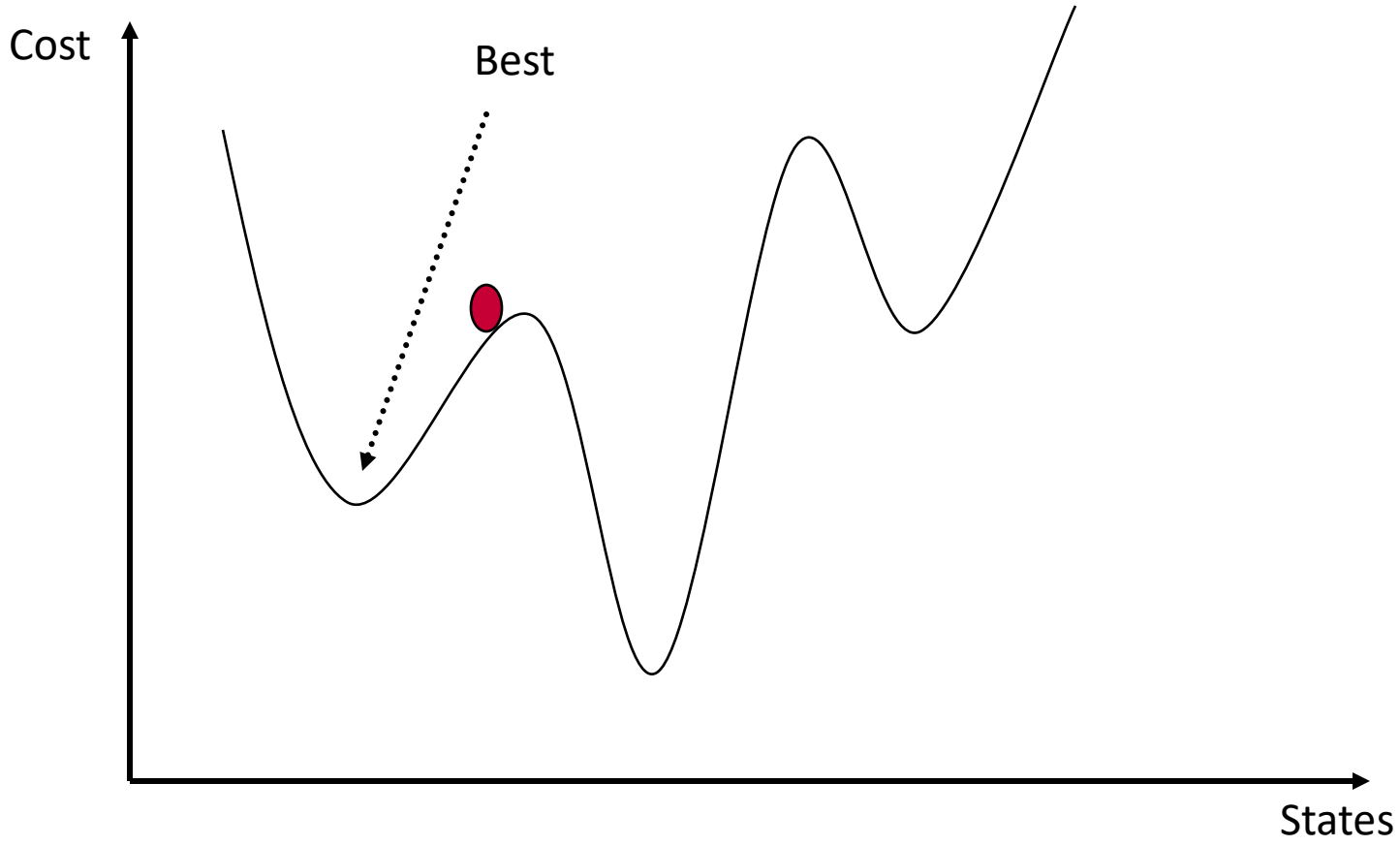
Simulated annealing



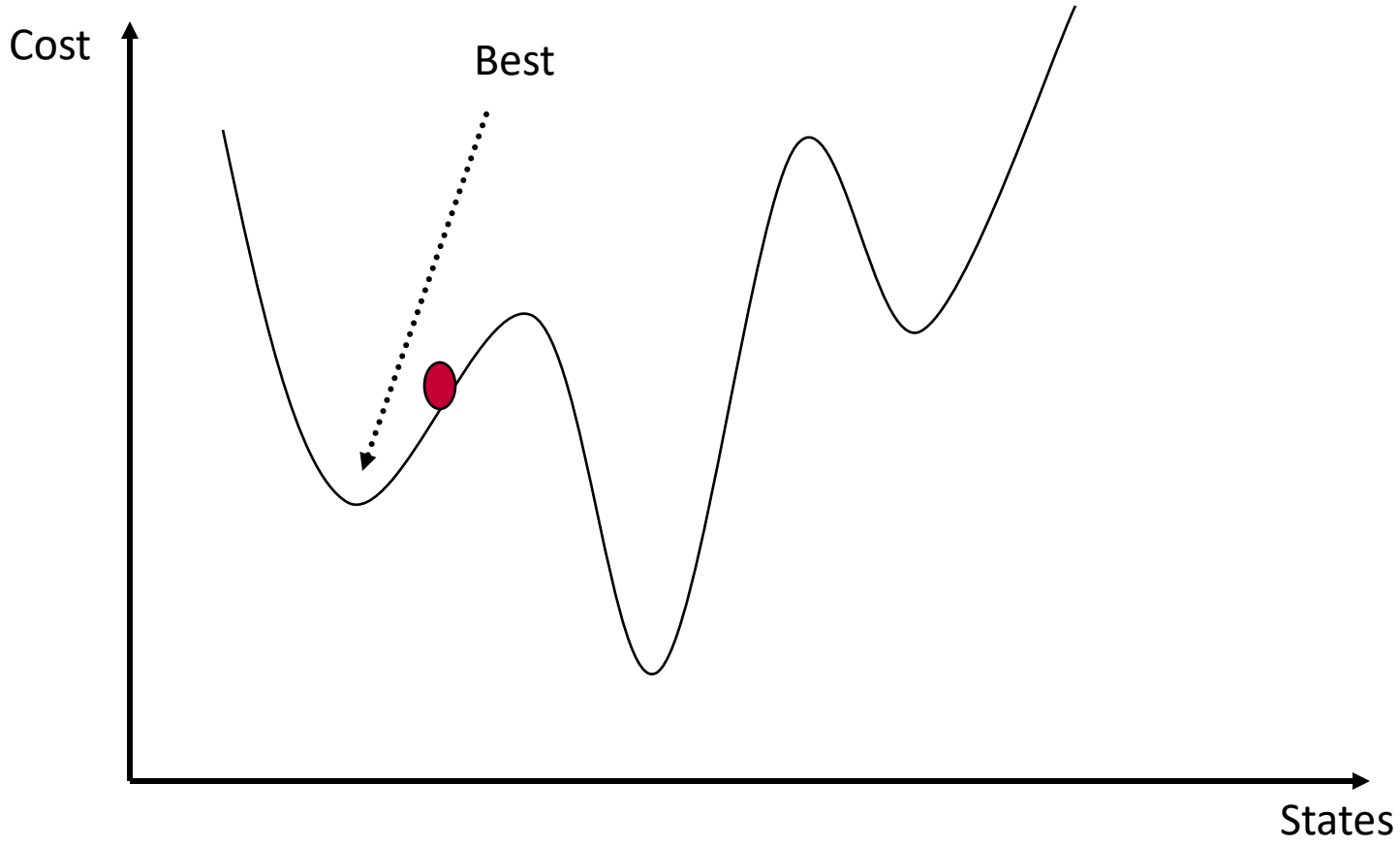
Simulated annealing



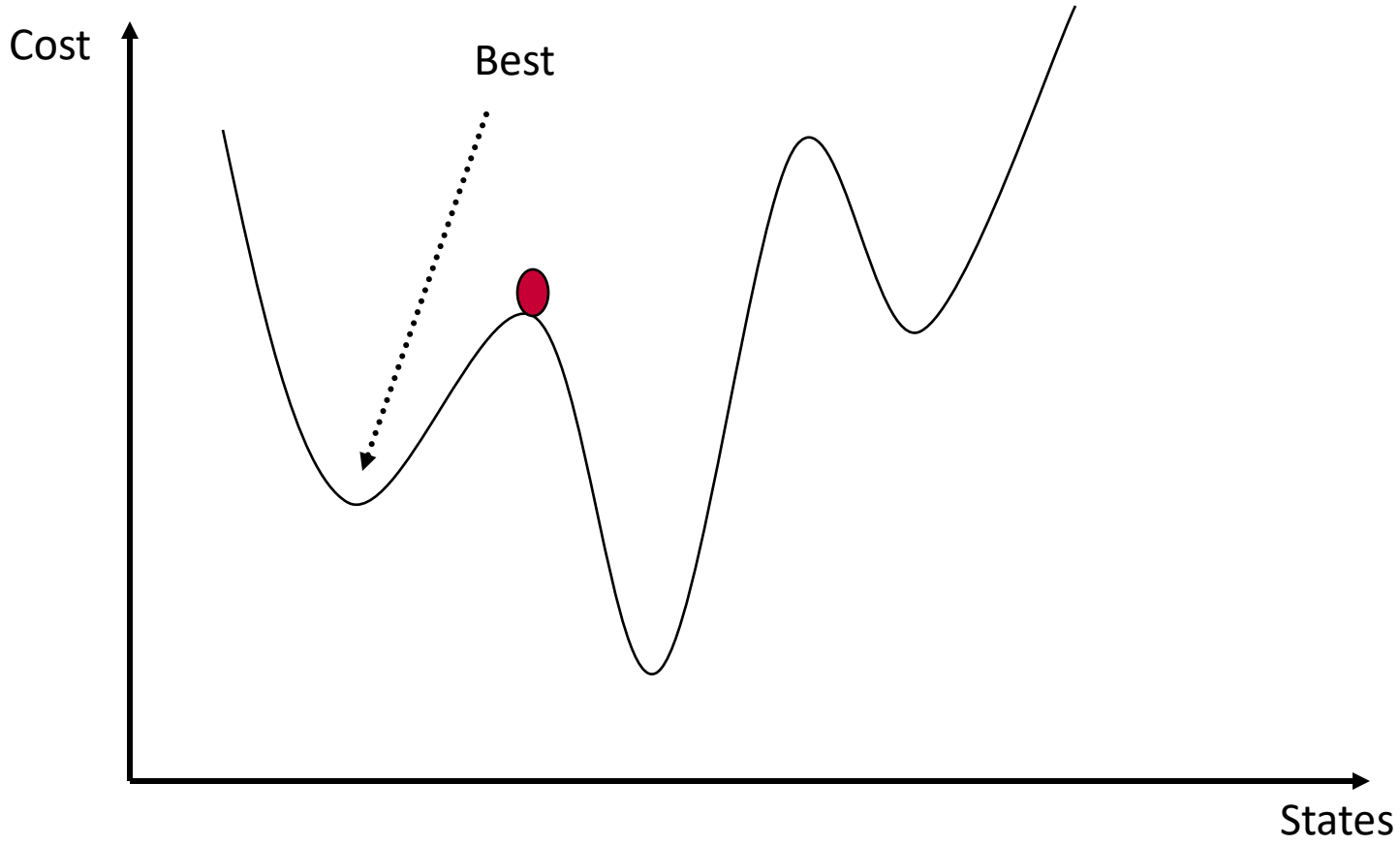
Simulated annealing



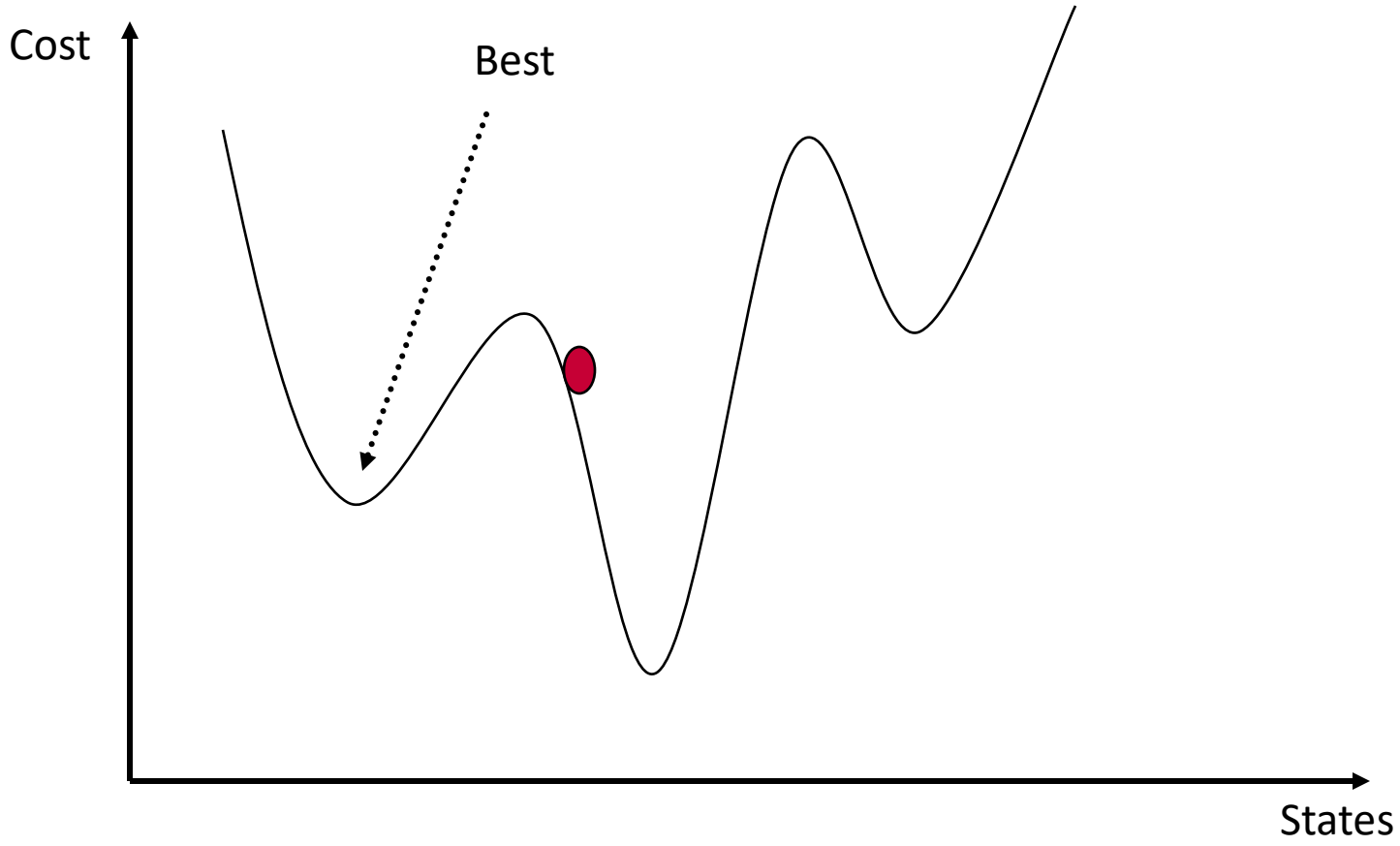
Simulated annealing



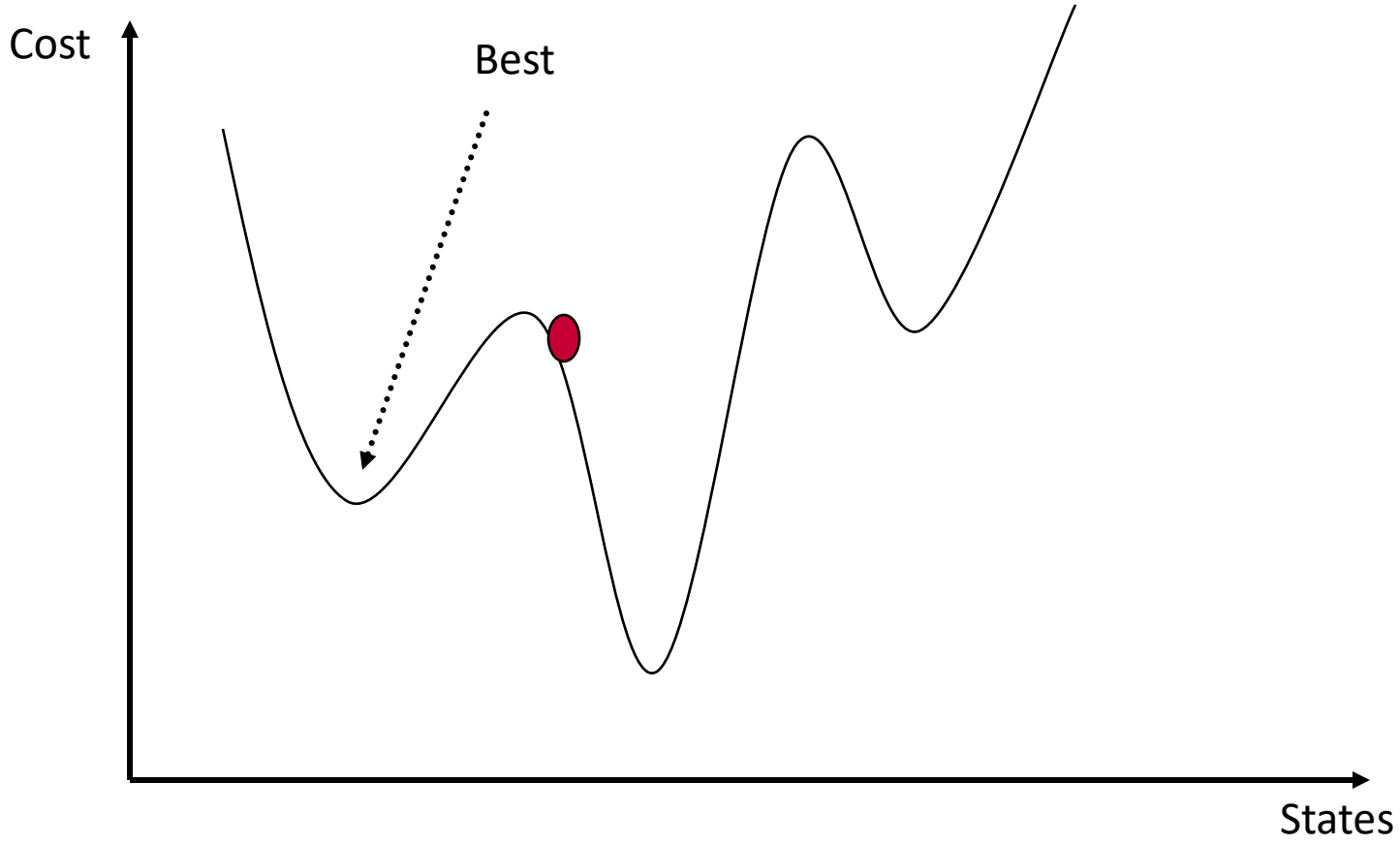
Simulated annealing



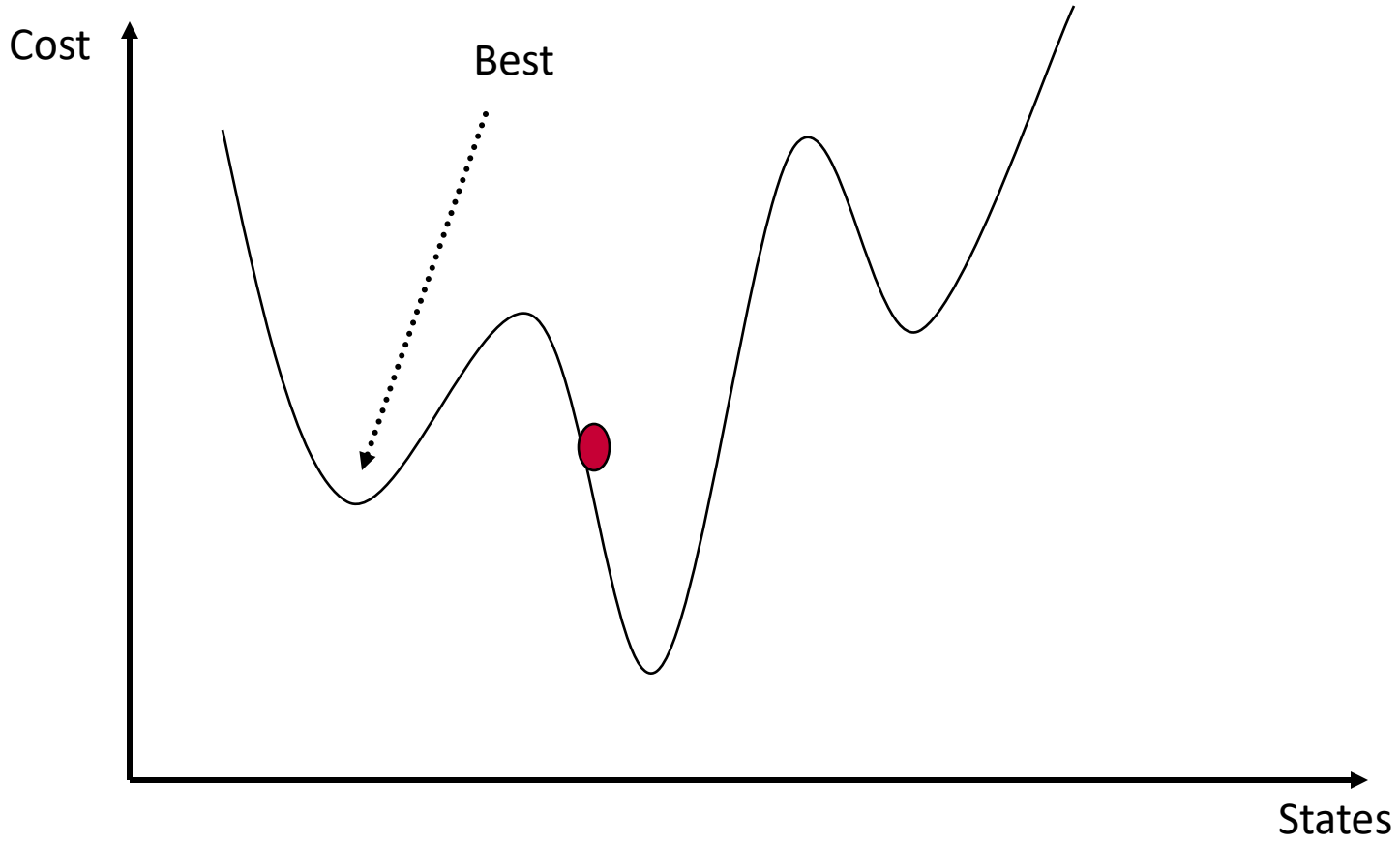
Simulated annealing



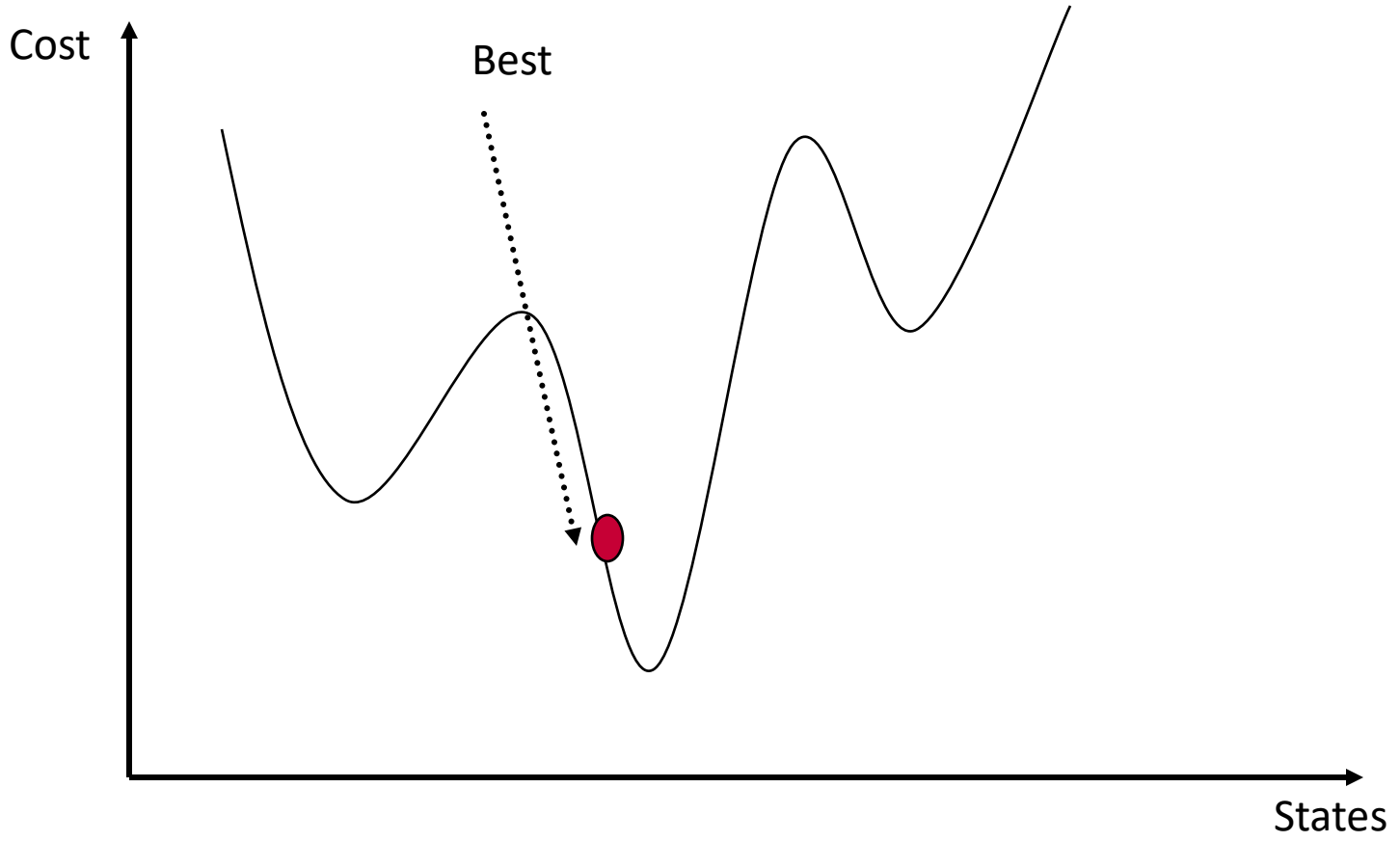
Simulated annealing



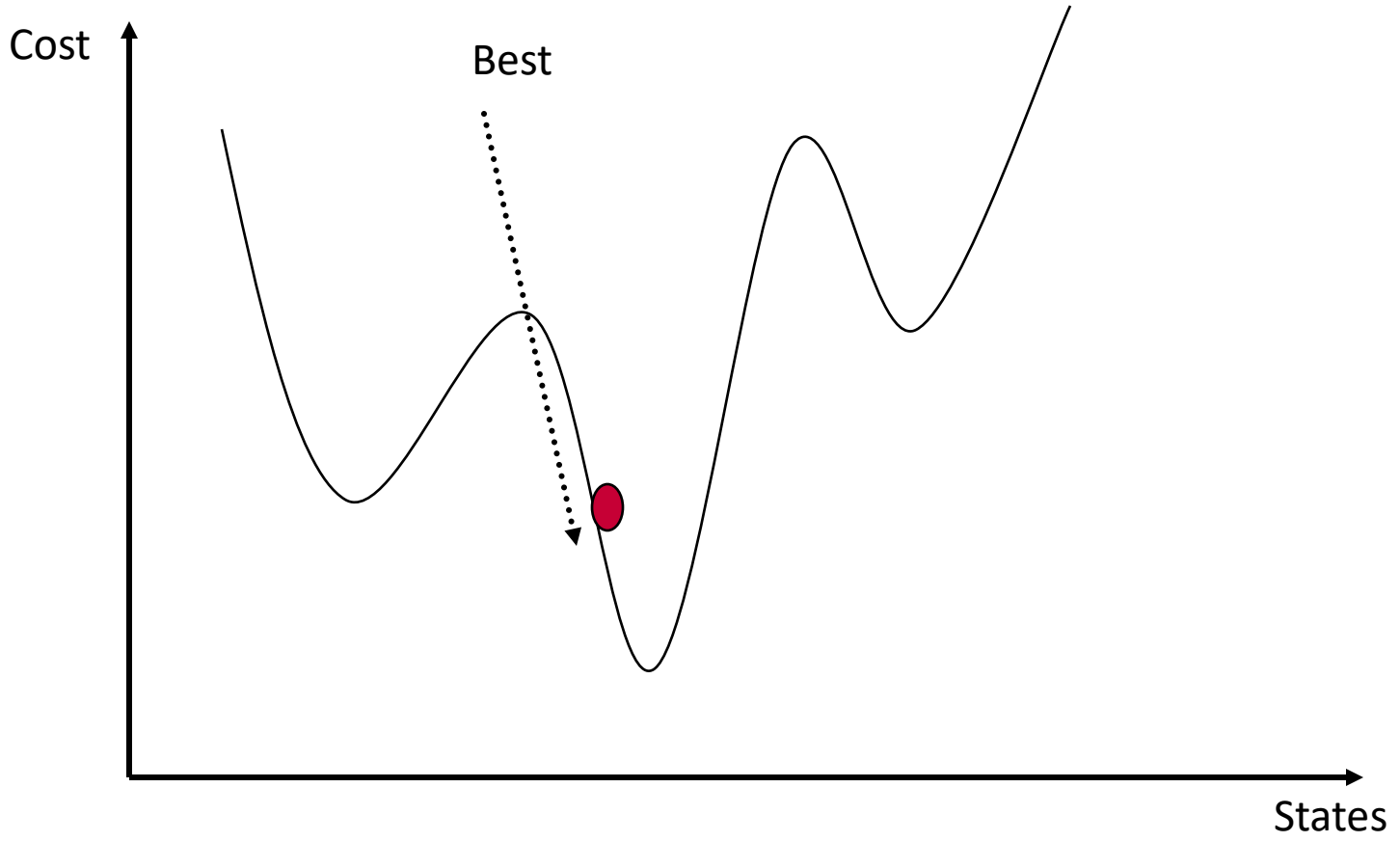
Simulated annealing



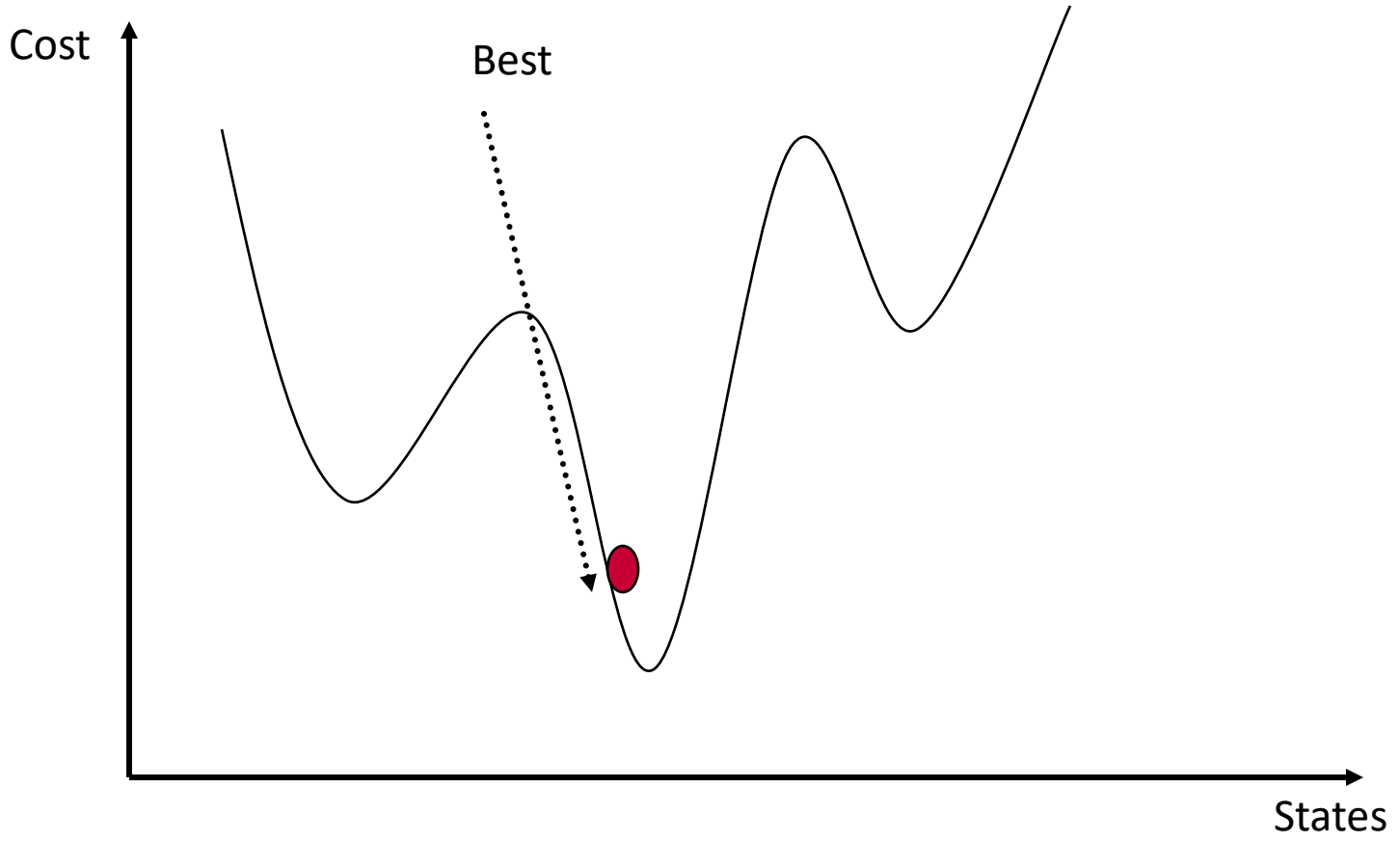
Simulated annealing



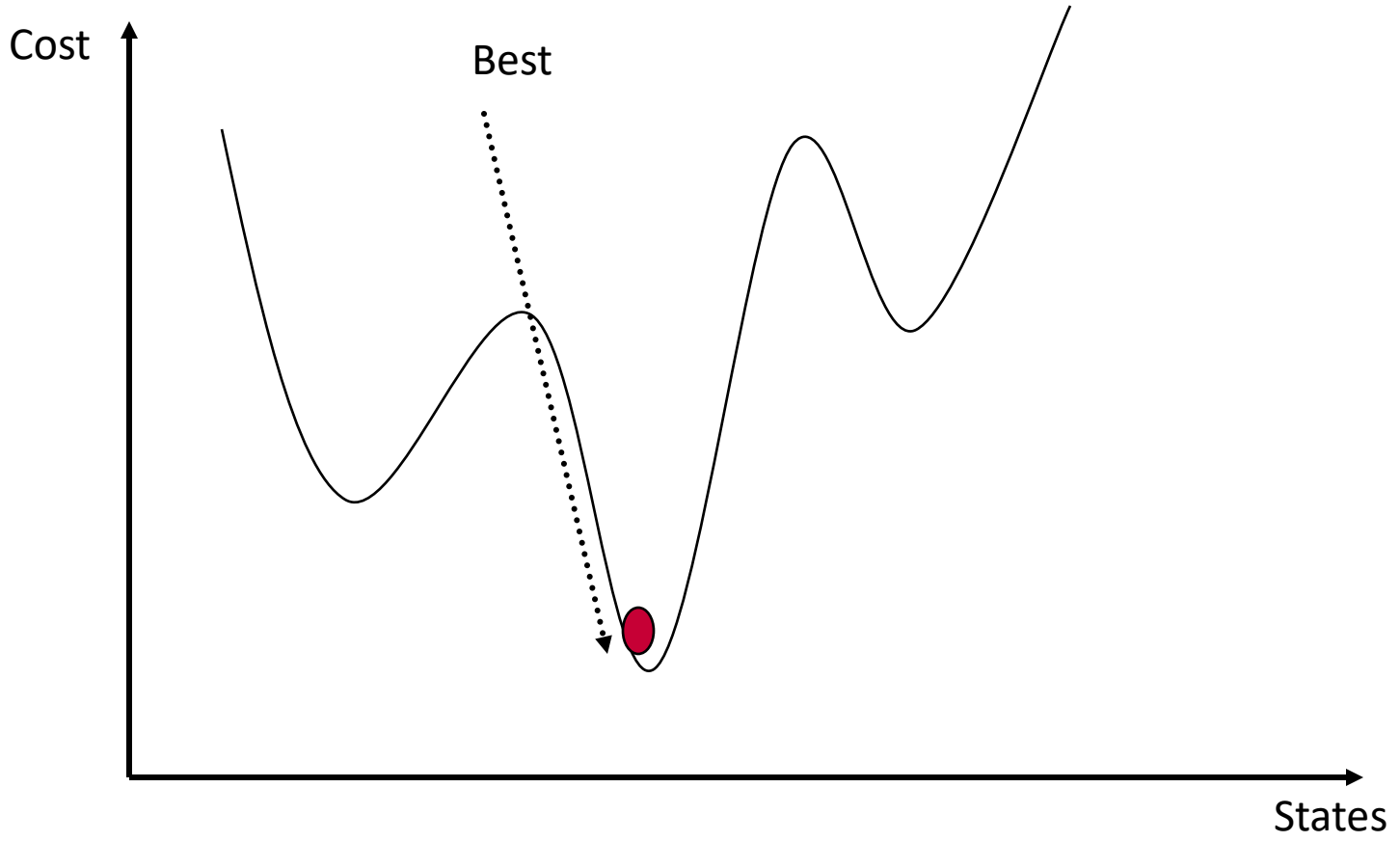
Simulated annealing



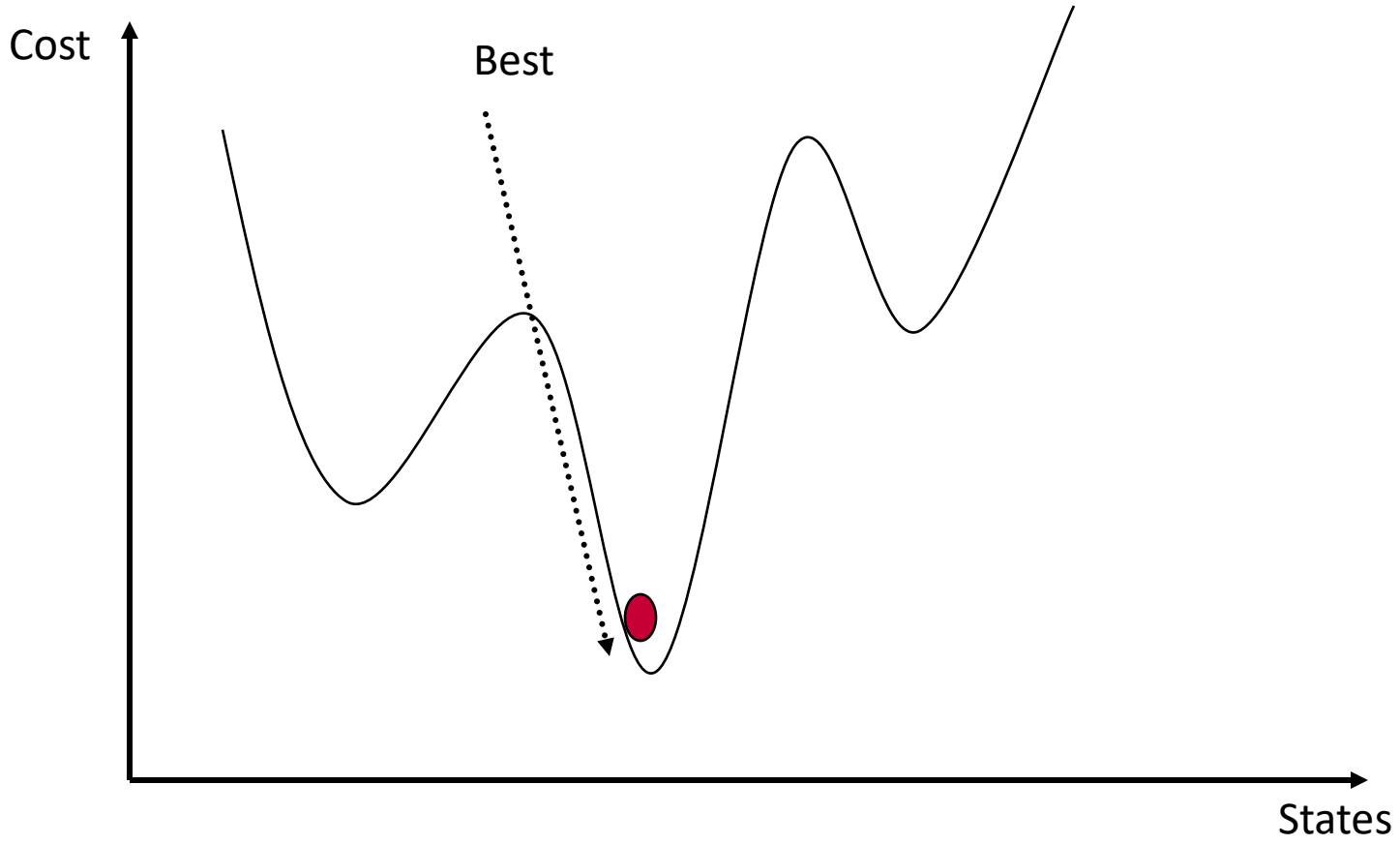
Simulated annealing



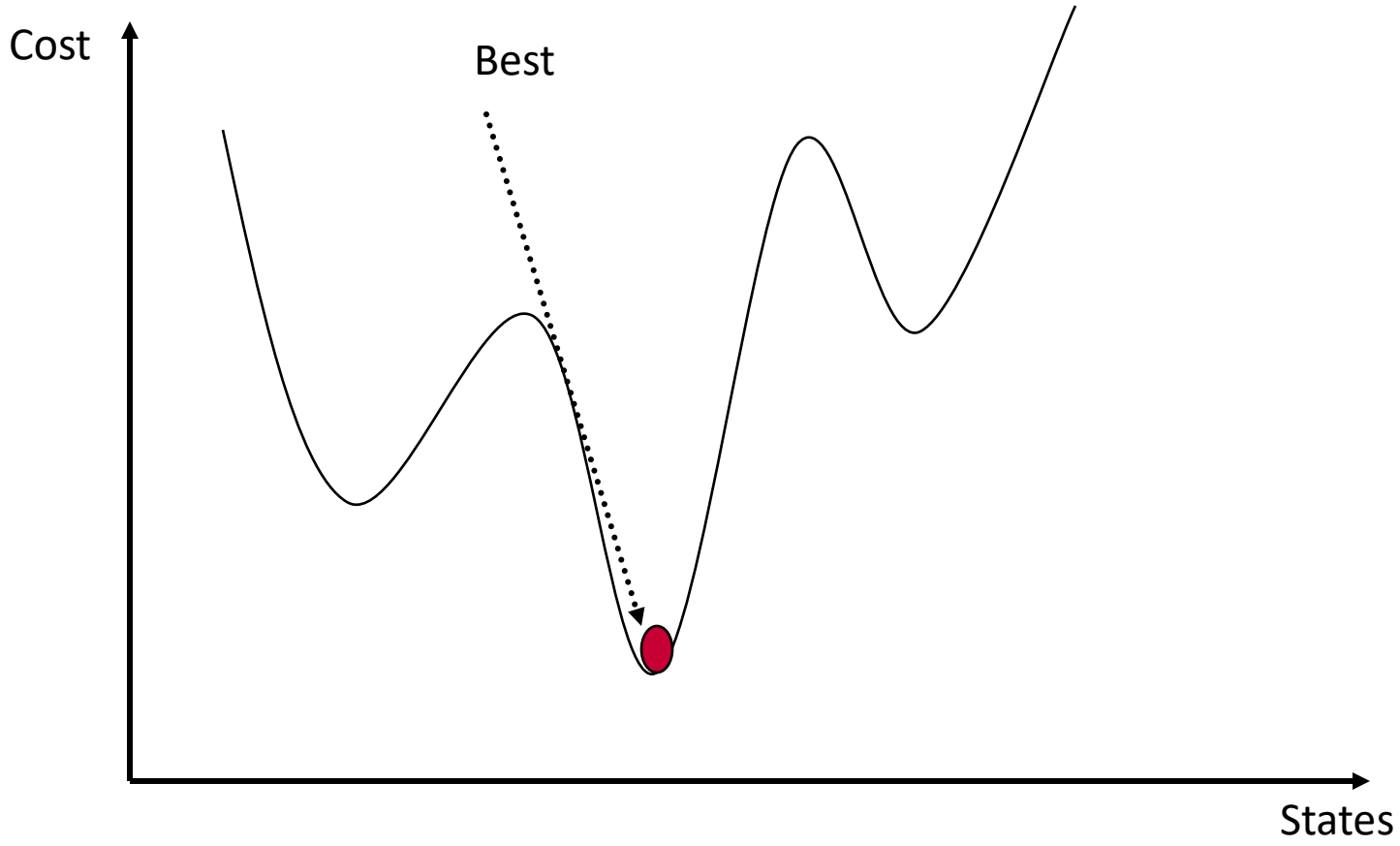
Simulated annealing



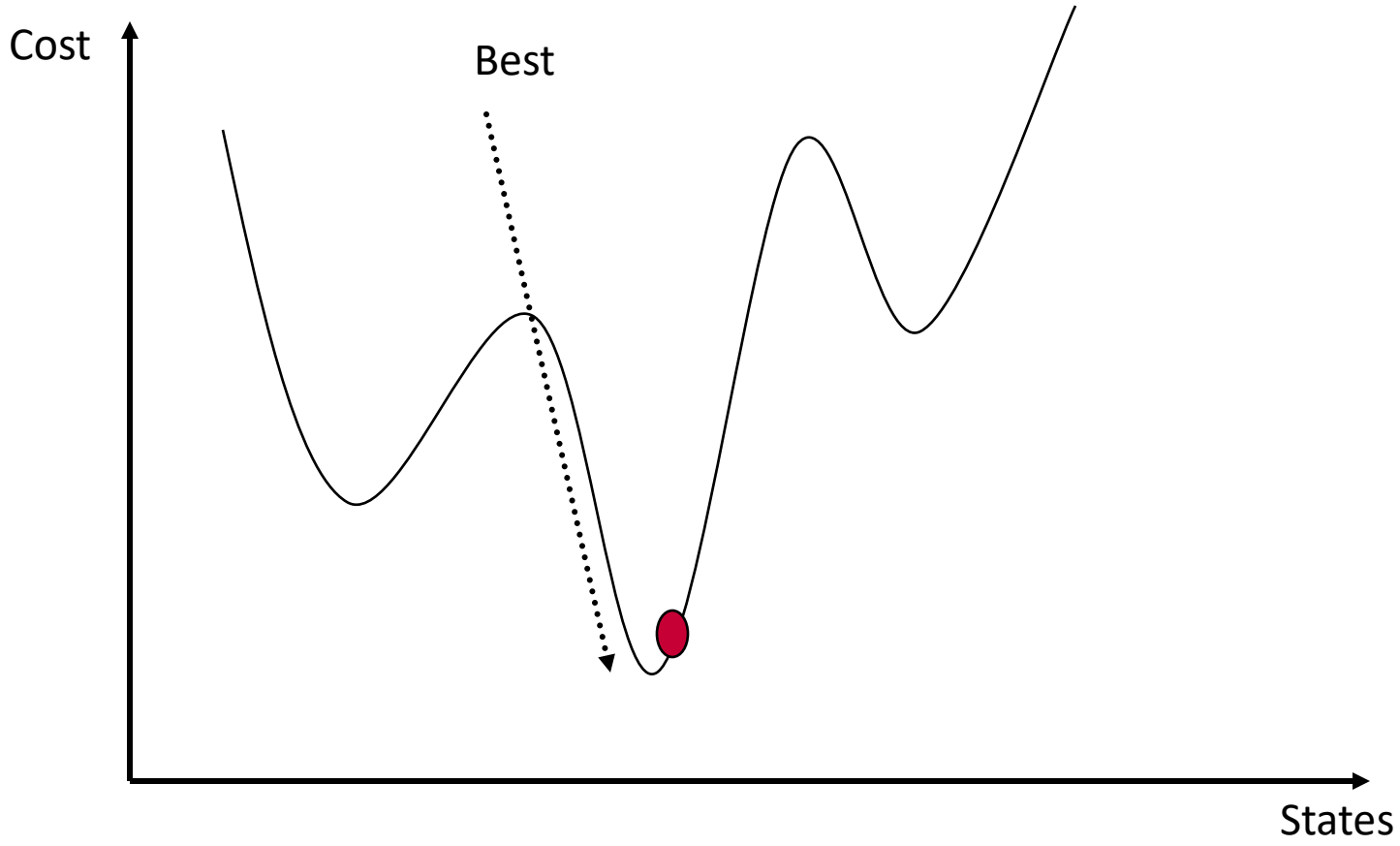
Simulated annealing



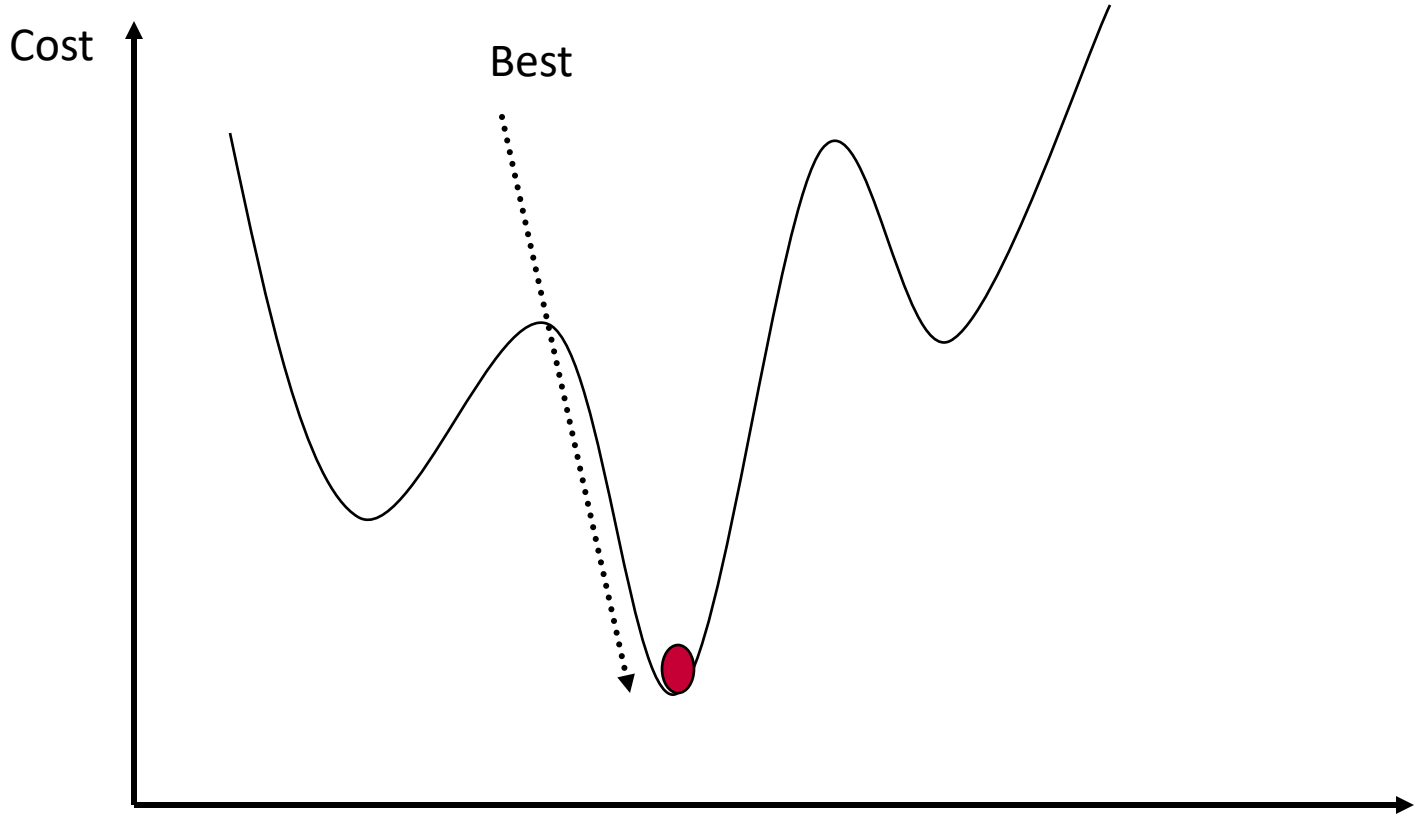
Simulated annealing



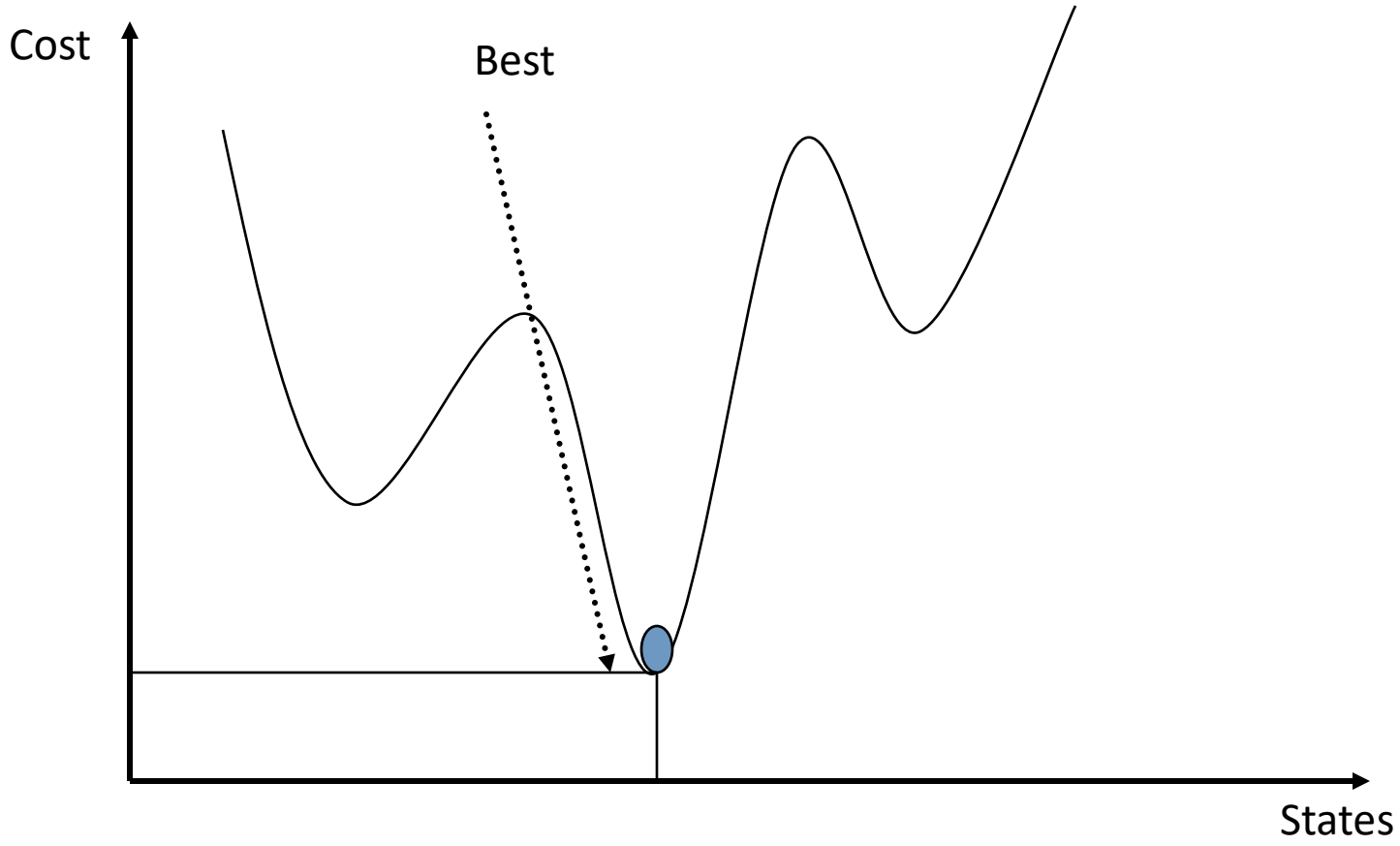
Simulated annealing



Simulated annealing



Simulated annealing



Simulated annealing

- Lets say T=1 and there are 3 moves available, with changes in the objective function of

$$\delta_1 = -0.1 \qquad \delta_2 = 0.5 \qquad \delta_3 = -5$$

Pick a move randomly:

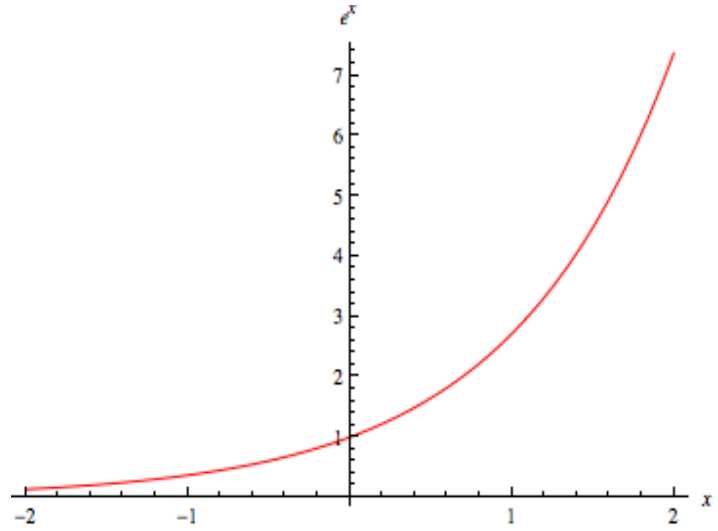
- If δ_2 is picked, move there.
- If δ_1 or δ_3 are picked, probability of move = $\exp(\delta/T)$
  - Move δ_1 : prob1 = exp(-0.1) = 0.9, i.e., 90% of the time we will accept this move
  - Move δ_3: prob3 = exp(-5) = 0.05, i.e., 5% of the time we will accept this move

Evaluating a Proposed Move

- The distribution used to decided if we accept a bad move is based on a probability distribution (E is the quality of the new state and E_o the old, T is the current temperature)

Eq: prob = exp(- (**E-E$_o$**) /T)

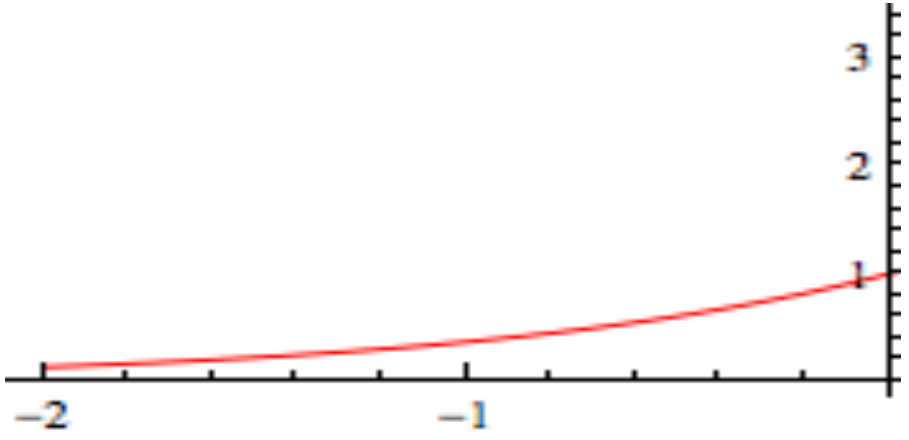
The result is always negative, because the proposed state is always worse than the current solution and temperature is positive



The exponential of any of any negative number will have a value between 0 and 1.

Evaluating a Proposed Move

- Relationship between the probability value and the difference between the current neighbor state
  - The greater the difference between the neighbor state and the current state the lower the probability of accepting the move
- What about the temperature?
  - The higher the temperature the more we reduce the difference between the neighbor and the current state and thereby increase the probability
    - exp(-45/1000) = 0.955
    - exp(-45/100) = 0.637
    - exp(-45/10) = 0.011

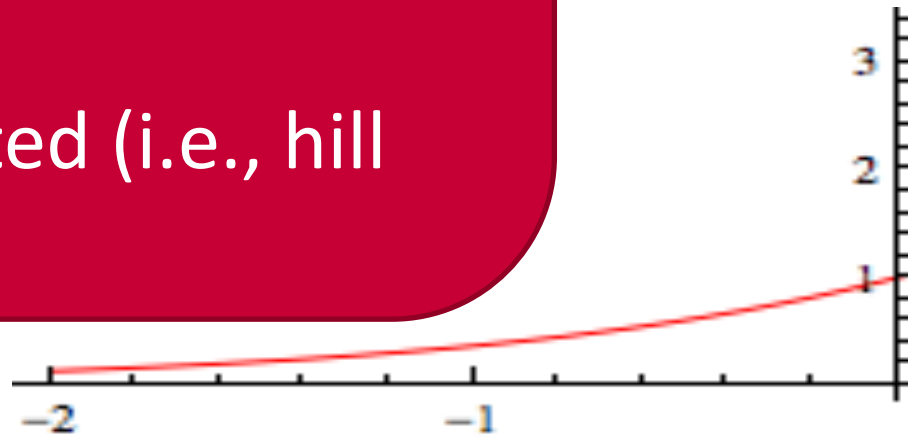


Evaluating a Proposed Move

Simulated Annealing

- Relationship between the probability value and the difference between the curre
  - Th lower the pr
- Wh
  - Th ighbor and th
  - ex
  - ex
  - ex

- The probability of accepting a worse state is a function of both the temperature of the systems and the change in the cost function
- As the temperature decreases, the probability of accepting worse moves decreases
- If T=0, no worse moves are accepted (i.e., hill climbing)



Temperature T

- High T: probability of “locally bad” move is higher
- Low T: probability of “locally bad” move is lower

- Typically, T is decreased as the algorithm runs longer
  - There is a “temperature schedule”

Starting T

- Must be hot enough to allow moves to almost every neighbor state
- Must not be so hot that we conduct a random search for a long period of time
- Problem is finding a suitable starting temperature

Temperature schedule → $T = \alpha t \ (0 < \alpha < 1)$

Simulated Annealing in Practice

- Method proposed in 1983 by IBM researchers for solving VLSI layout problems (Kirkpatrick et al, *Science*, 220:671-680, 1983).
  - Theoretically will always find the global optimum
- Other applications: Traveling salesman, Graph partitioning, Graph coloring, Scheduling
- Useful for some problems, but can be very slow
  - slowness comes because T must be decreased very gradually to retain optimality

Metaheuristics in Practice on TSP

https://www.youtube.com/watch?v=rYjxtmt8g9A

Iterative Local Search

Iterative Local Search

Use two types of Stochastic Local Search steps:

Intensification

- Subsidiary **local search steps** (e.g. hill climbing) for reaching local optimal as efficient as possible
- **Perturbation steps** for efficiently escaping from local optima

Diversification

Also use **acceptance criterion** to control diversification vs. intensification behavior

Iterative Local Search

Iterated Local Search (ILS):

determine initial candidate solution s

perform *subsidiary local search* on s

While termination criterion is not satisfied:

- $r := s$
- perform *perturbation* on s
- perform *subsidiary local search* on s
- based on *acceptance criterion*,
  - keep s or revert to $s := r$

Iterative Local Search

- Subsidiary local search results in a local minimum
- ILS trajectories can be seen as walks in the space of local minima of the given evaluation function
- Perturbation phase and acceptance criterion may use aspects of search history (i.e., limited memory)
- In a high-performance ILS algorithm, subsidiary local search, perturbation mechanism and acceptance criterion need to complement each other well

Iterative Local Search (detailed)

procedure *Iterated Local Search*
    $s_0 \leftarrow$ GenerateInitialSolution
    $s^* \leftarrow$ LocalSearch(s_0)
    repeat
        $s' \leftarrow$ Perturbation(s^*, history)
        $s^{*\prime} \leftarrow$ LocalSearch(s')
        $s^* \leftarrow$ AcceptanceCriterion($s^*, s^{*\prime}$, history)
    until *termination condition met*
end

Iterative Local Search

- Basic version
  - Initial solution: random or construction heuristic
  - Subsidiary local search: often readily available
  - Perturbation: random moves in higher order neighborhoods
  - Acceptance criterion: force cost to decrease
- Advantages
  - Often leads to very good performance
  - Only requires few lines of additional code to existing local search algorithm
  - State-of-the-art results with further optimizations

Iterative Local Search

- Intelligent Perturbation
- Local optima is specific to neighborhood definition
- By changing the neighborhood function we can escape local minima

(Basic) Iterative Local Search for TSP

- GenerateInitialSolution: greedy heuristic
- LocalSearch: 2-opt, 3-opt
- Perturbation: double bridge move
- AcceptanceCriterion: accept s*’ only if f(s*’) <= f(s’)

(Basic) Iterative Local Search for SAT

- GenerateInitialSolution: random initial solution
- LocalSearch: short tabu search runs based on 1-flip neighborhood
- Perturbation: random k-flip move, k >> 2
- AcceptanceCriterion: accept s*’ only if f(s*’) <= f(s’)

Iterative Local Search -- Perturbation

- Important: strength of perturbation
  - **Too strong:** close to random restart.
  - **Too weak**: LocalSearch may undo perturbation easily, e.g. return to local minima.
    - Choice of *k* for SAT
    - Could make perturbation size dynamic
- Random perturbations are simplest but not necessarily best
- Perturbation should be complementary to LocalSearch

Iterative Local Search -- Speed

- On many problems, small perturbations are sufficient
- LocalSearch in such a case will execute very fast; very few improvement steps
- Sometimes access to LocalSearch in combination with Perturbation increases strongly speed

Iterative Local Search – Acceptance criterion

- AcceptanceCriterion has strong influence on nature and effectiveness of walk in S*
- Controls balance between intensification and diversification
- Extreme intensification: Accept only if improves best solution
- Extreme diversification: Always accept
- Many intermediate choices possible

Iterative Local Search

- Based on simple principles
- Easy to understand
- Basic versions are easy to implement
- Flexible, allowing for many additional optimizations if needed
- Highly effective in many applications

Dynamic Local Search

Dynamic Local Search

- Modify the evaluation function whenever a local optimum is encountered in such a way that further improvement steps become possible
- Associate penalty weights (penalties) with solution components; these determine impact of components on evaluation function value
- Perform iterative improvement (e.g., hill-climbing); when in local minimum, increase penalties of some solution components until improving steps become available
  - E.g. in SAT could take the weighted sum of the clauses where each clause has a weight associated with it based on a function of how often it has been unsat

Dynamic Local Search

Dynamic Local Search (DLS):

determine *initial candidate solution* s

initialise penalties

While *termination criterion* is not satisfied:

- compute *modified evaluation function* g' from g based on *penalties*
- perform *subsidiary local search* on s using *evaluation function* g'
- *update penalties* based on s

Dynamic Local Search

Modified evaluation function:

$$g'(\pi, s) = g(\pi, s) + \sum_{i \in SC(\pi,s)} penalty(i)$$

Where $SC(\pi, s)$ is the set of solution components of problem instance π used in candidate solution s.

- **Penalty initialisation:** For all i : *penalty* (i) := 0.
- **Penalty update** in local minimum s: Typically involves *penalty increase* of some or all solution components of s; often also occasional *penalty decrease* or *penalty smoothing*.
- **Subsidiary local search:** Often *Iterative Improvement*.

Dynamic Local Search (SAT)

Procedure `Dynamic Local Search(`*CNF formula* F, maxTries, maxSteps)

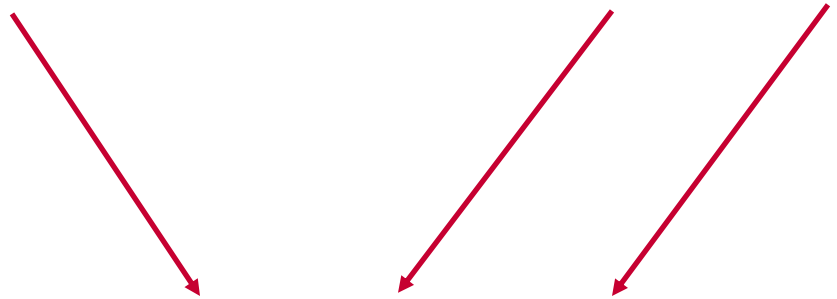
Input: CNF formula F, positive integers *maxTries* and *MaxSteps*
Output: model of F or 'no solution found'
for try := *1 to* maxTries **do**
    a:= randomly chosen assignment of the variable in formula F ;
    initialize *clause-weight* for each clause ;
    for *step :=1 to maxSteps* **do**
        if a *satisfies* F **then**
            return a ;
        x := variable selected using a scoring function that considers *clause-weights*;
        a := a with x flipped;
        update *clause-weight* for each clause ;
return 'no solution found'

Increase the clause penalty unsatisfied clauses

Dynamic Local Search (SAT)

Clause (weights or penalty)
Score: sum of all clause penalties

Clauses → **C1** (1), C2 (1), C3 (1), **C4** (1), **C5** (1) , C6 (1)



Unsatisfied clauses

Increase the clause penalty unsatisfied clauses

Clauses → **C1** (2), C2 (1), C3 (1), **C4** (2), **C5** (2) , C6 (1)

Dynamic Local Search (SAT)

- Smoothing step: Prevent an unbounded increase of clause weights

- Every x iterations, divide all weights by some constant c.

Dynamic Local Search: SAPS - Scaling and Probabilistic Smoothing

Scaling and Probabilistic Smoothing: Efficient Dynamic Local Search for SAT

Frank Hutter, Dave A. D. Tompkins, and Holger H. Hoos*

Department of Computer Science
University of British Columbia
Vancouver, B.C., V6T 1Z4, Canada
mail@fhutter.de,{davet,hoos}@cs.ubc.ca
WWW home page: http://www.cs.ubc.ca/labs/beta

Abstract. In this paper, we study the approach of dynamic local search for the SAT problem. We focus on the recent and promising Exponentiated Sub-Gradient (ESG) algorithm, and examine the factors determining the time complexity of its search steps. Based on the insights gained from our analysis, we developed Scaling and Probabilistic Smoothing (SAPS), an efficient SAT algorithm that is conceptually closely related to ESG. We also introduce a reactive version of SAPS (RSAPS) that adaptively tunes one of the algorithm's important parameters. We show that for a broad range of standard benchmark problems for SAT, SAPS and RSAPS achieve significantly better performance than both ESG and the state-of-the-art WalkSAT variant, Novelty$^+$.

1 Introduction and Background

http://www.cs.ubc.ca/labs/lci/papers/docs2002/hutter-cp02-saps.pdf

Dynamic Local Search: SAPS - Scaling and Probabilistic Smoothing

procedure UpdateWeights(F, x, W, α, ρ,P_{smooth})
  input:
    propositional formula F, variable assignment x, clause weights $W = (w_i)$,
    scaling factor α, smoothing factor ρ, smoothing probability P_{smooth}
  output:
    clause weights W
  $C = \{\text{clauses of } F\}$
  $U_c = \{c \in C \mid c \text{ is unsatisfied under } x\}$
  for each i s.t. $c_i \in U_c$ **do**
    $w_i := w_i \times \alpha$
  end
  with probability P_{smooth} **do**
    for each i s.t. $c_i \in C$ **do**
      $w_i := w_i \times \rho + (1 - \rho) \times \bar{w}$
    end
  end
  return (W)
end