CS440/ECE448: Intro to Artificial Intelligence

Lecture 4: Heuristic search and local search

Prof. Julia Hockenmaier
juliahmr@illinois.edu

http://cs.illinois.edu/fa11/cs440

Tuesday's key concepts

Problem solving as search:

Solution = a finite sequence of actions

State graphs and search trees

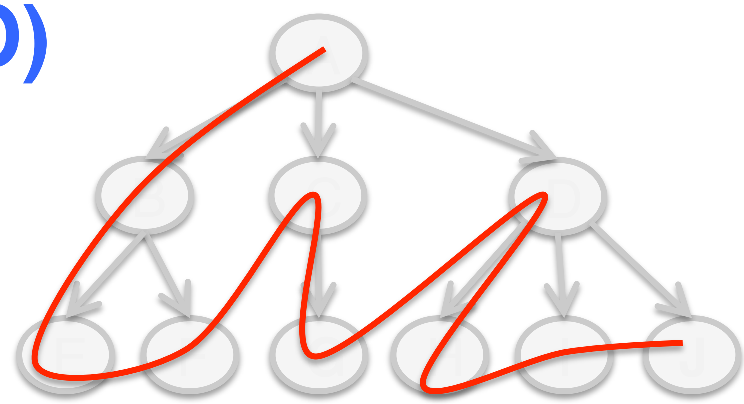
Which one is bigger/better to search?

Systematic (blind) search algorithms

Breadth-first vs. depth-first; properties?

Blind search:
deterministic queuing functions

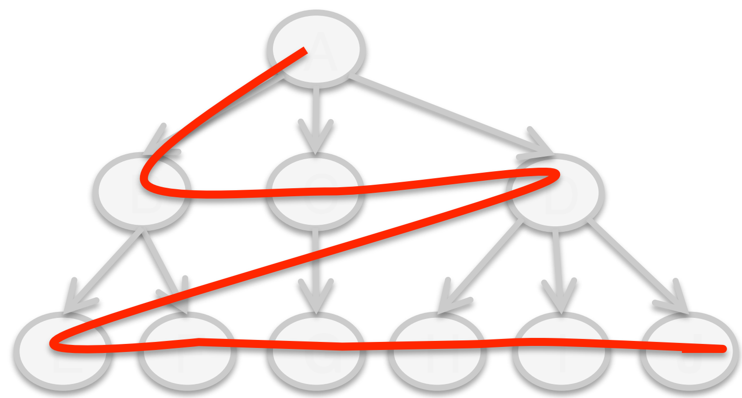
Depth-first search (LIFO)



Expand deepest node first

```
QF(old, new):
    Append(new, old)
```

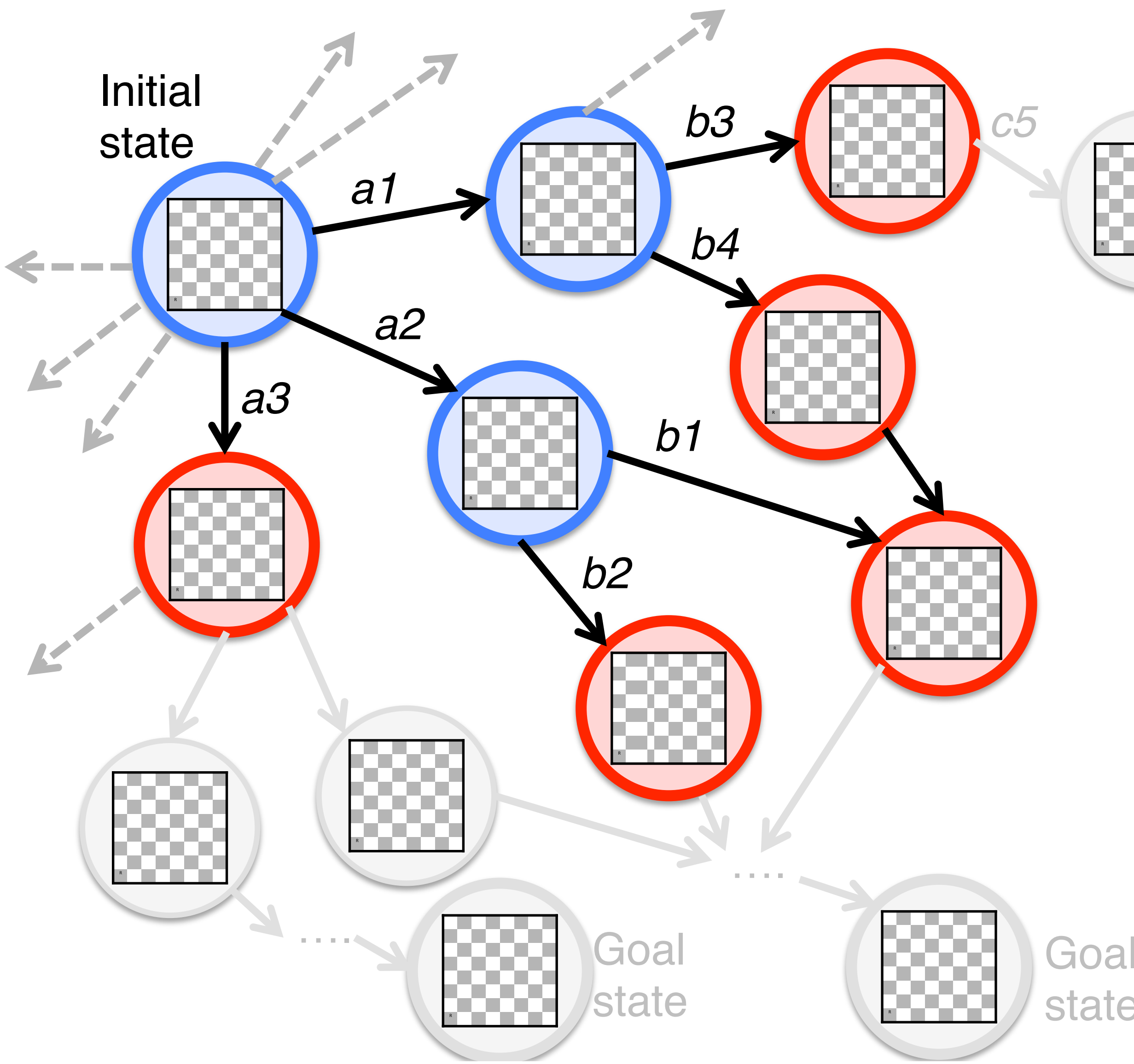
Breadth-first (FIFO)



Expand nodes level by level

```
QF(old, new):
    Append(old, new);
```

Graph search



Explored set
Frontier
(Queue)
Unexplored set

Today’s key questions

How can we find the *optimal* solution?
We need to assign values to solutions

How can we find a *better* solution if we can only foresee the effect (=value) of the next action?
This is local search.

Informed (heuristic) search

Considering the cost of solutions

We may not just want to find *any* solution, but the *cheapest* solution, if:

- Each action has a (positive, finite) cost
- Some solutions may be cheaper than others

Heuristic search: priority queue

Heuristic search algorithms sort the nodes on the queue according to a **cost function**:

```
QF(a,b):
   sort(append(a,b), CostFunction)
```

The cost function is an estimate of the true cost. Nodes with the lowest estimated cost have the highest priority.

Heuristic graph search

```
SEARCH(Problem P, Queuing Function QF):
  local: n, q, e;
  q ← new List(Initial_State(P));
  Loop:
    if q == () return failure;
    n ← Pop(q);
    if n solves P return n;
    add n.STATE to e
    for m in Expand(n):
      if m is not in e or q:
           q ← QF(q,{m});
/*NEW: we want to find the cheapest goal!*/
     else if m.STATE in q with higher cost:
                q ← replace(q, m.STATE, m);
  end
```

Cost from root to node: $g(n)$

$g^*(n)$: Minimum cost from root to n

$g^*(n)$ is the sum of the costs for each action from the root to node n.

This requires a cost function for actions

$g(n)$: Computable approximation to $g^*(n)$

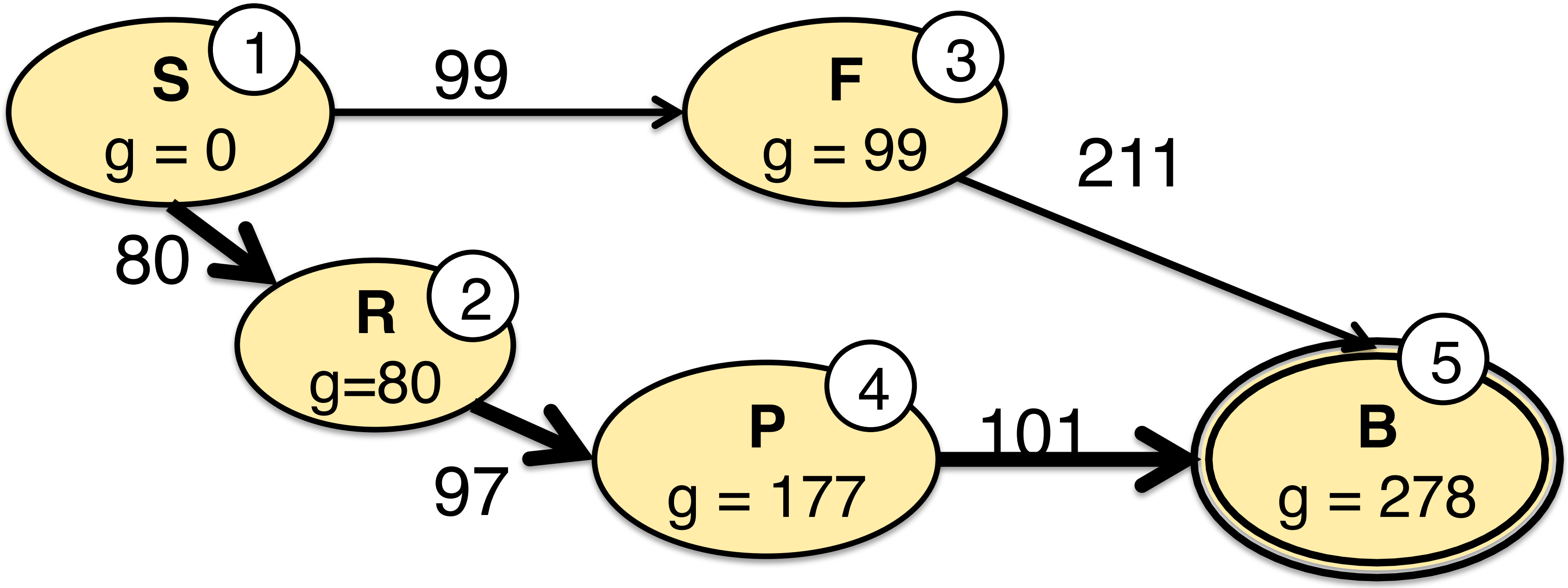
For trees: $g(n) == g^*(n)$

Uniform-cost search

Sort the queue by **path cost g(n):**
First expand the node with lowest g(n)

```
QF(a,b): sort(Append(a,b), g)
```

Uniform-cost search illustrated



S:0 [R:80, F:99]

R:80 [F:99, P:177]

F:99 [P:177, B:310]

P:177 [B:278, B:310]

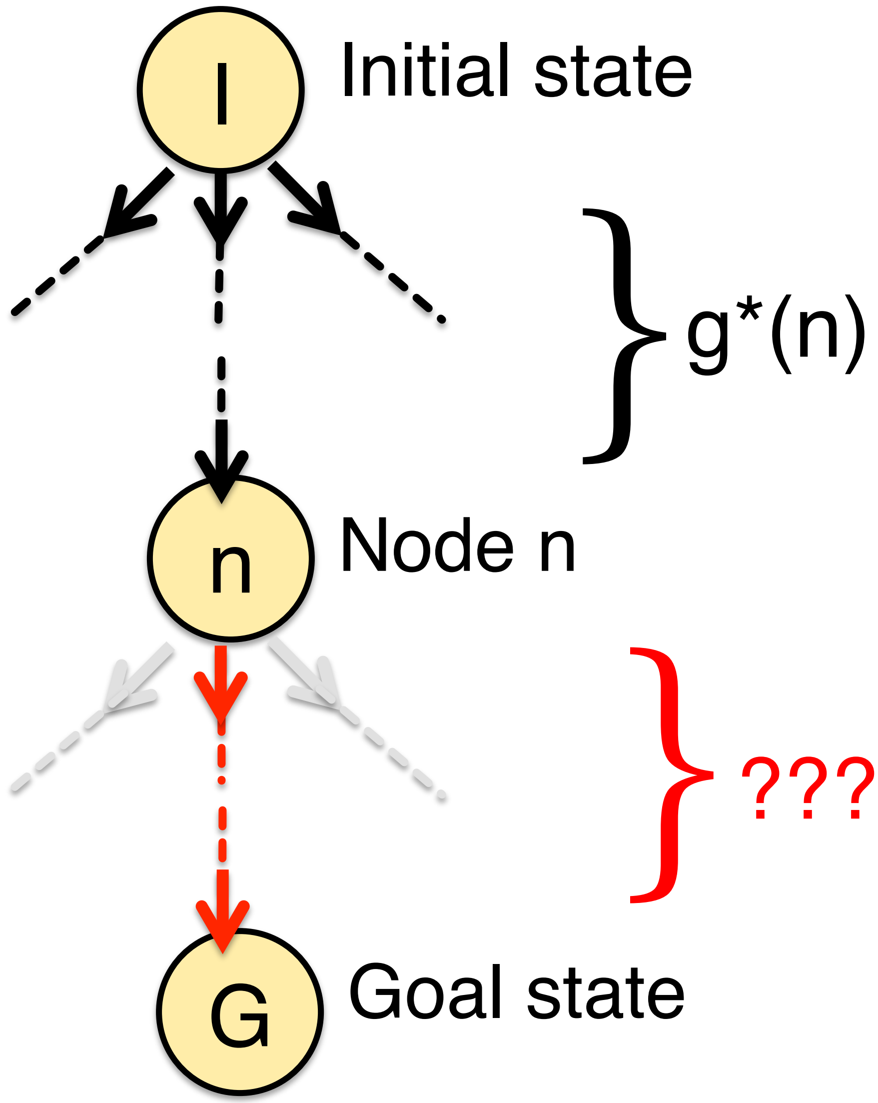
B:278 [B:310]

Properties of uniform-cost search

Complete if *b* is finite, and each action has positive (non-zero) cost
(gets stuck in loops of zero-cost actions)
Optimal.

Time and space complexity similar to breadth-first search if costs are uniform
(possibly much worse otherwise)

How close are we to the goal?



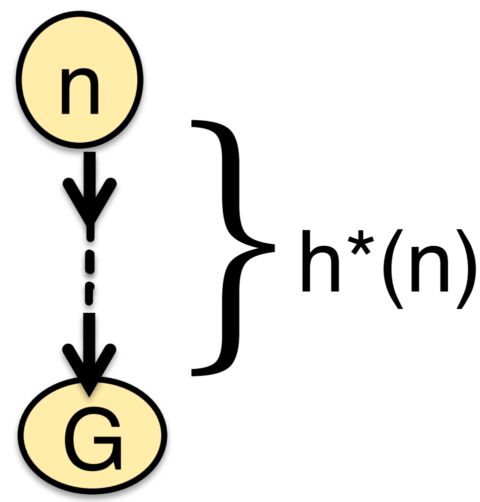
Cost from node to goal: h(n)

h*(n): the minimum cost from n to any goal

h*(n) is generally unknown

h(n): computable approximation to h*

h(n) is called the heuristic function



Greedy best-first search

Sort the queue by **heuristic function h(n):**
First expand the node with lowest h(n)

```
QF(a,b): sort(Append(a,b), h)
```

Problem: This ignores g(n)

Properties of greedy best-first search

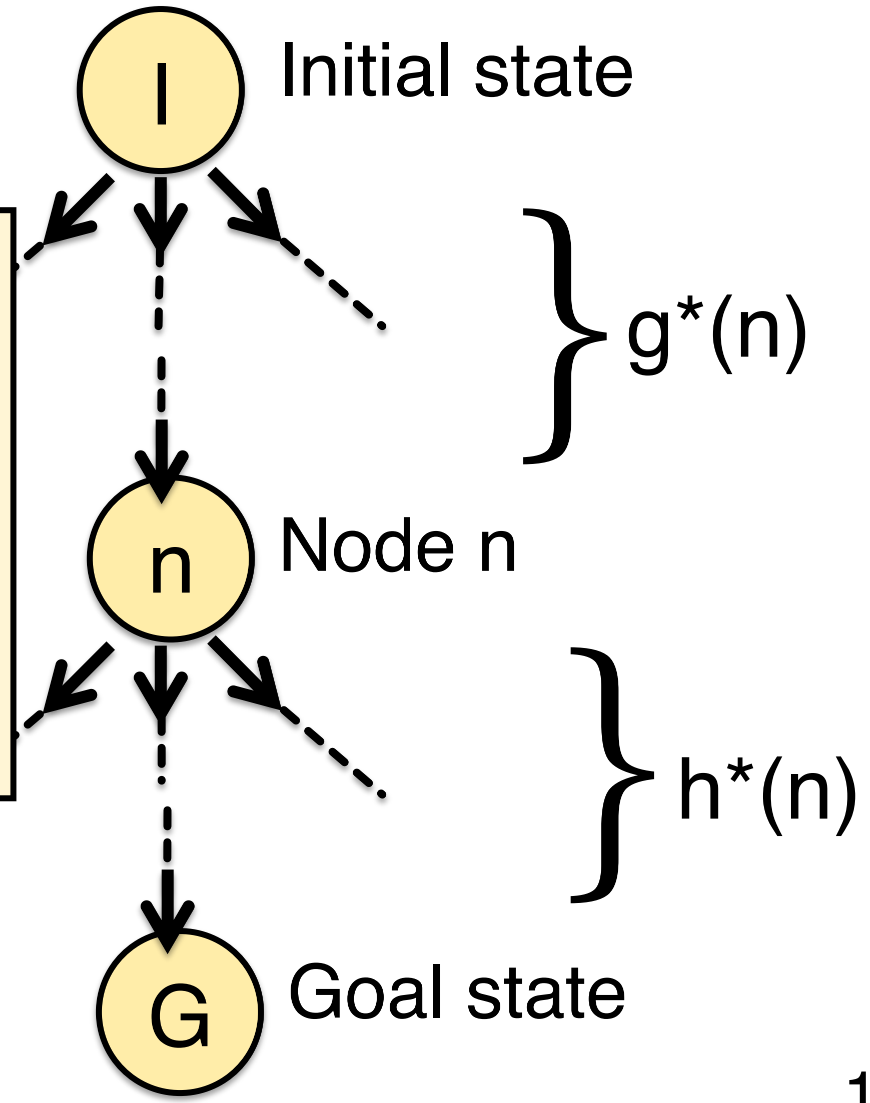
Similar to DFS:
Tree-search version is **incomplete.**
Graph-search version is **complete**
in finite state spaces. Both are **not optimal.**

Worst-case **time and space complexity**
similar to DFS
(actual complexity depends on h)

Total cost: f*(n)

f*(n) = g*(n) + h*(n)

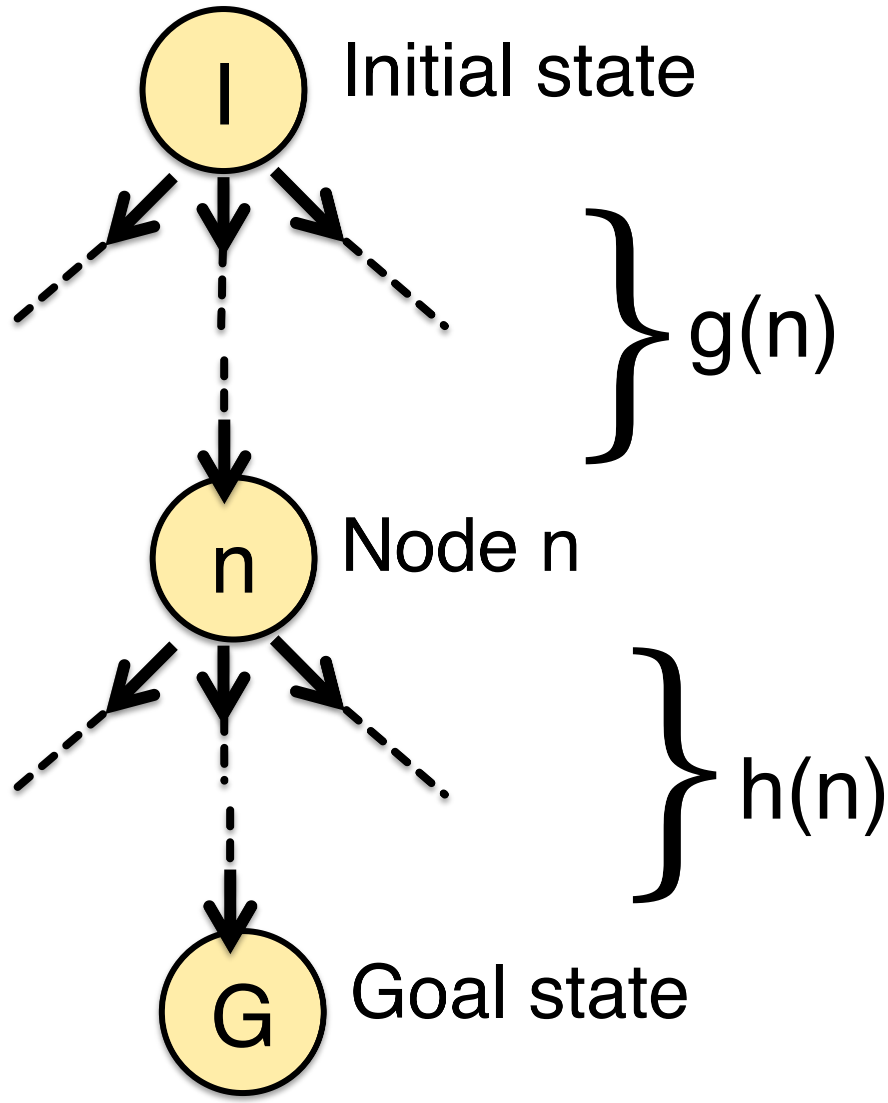
f*(n) is the lowest cost solution from the initial state to a goal *constrained through n*



Total *estimated* cost: f(n)

f(n) = g(n) + h(n)

f(n) *approximates* the lowest cost solution from the initial state to a goal constrained through n



A* search

Sort the queue by **total estimated cost f(n):**
First expand the node with lowest f(n)

```
QF(a,b): sort(Append(a,b), f)
```

Properties of A* search

A* is **complete** if *b* is finite, and each action has positive (non-zero) cost

Its **complexity** depends on the heuristic function h(n)

Is A* optimal?

Tree-search A* is optimal if…

…h(n) is an **admissible heuristic**

Admissible heuristics never overestimate the future cost.

Definition

$h(n)$ *is admissible:* $\forall n \in nodes$: $h(n) \leq h^*(n)$

Tree-search A* is optimal if *h(n)* is admissible

Confused? This is **optional** material

In tree-search: $g(n) = g^*(n)$

If n is **a goal**: $f(n) = f^*(n)$

➜The goal that is returned is the cheapest on the queue!

If n is **not a goal**: $f(n) \leq f^*(n)$

$$g(n) + h(n) \leq g^*(n) + h^*(n)$$

$f^*(n)$: true cost of *cheapest goal* that can be reached from n.

If g_{no} is a non-optimal goal, and n_{opt} is **an ancestor of the optimal goal** g_{opt}: $f(n_{opt}) \leq f(g_{opt}) < f(g_{no})$

➜n_{opt} will be explored first!

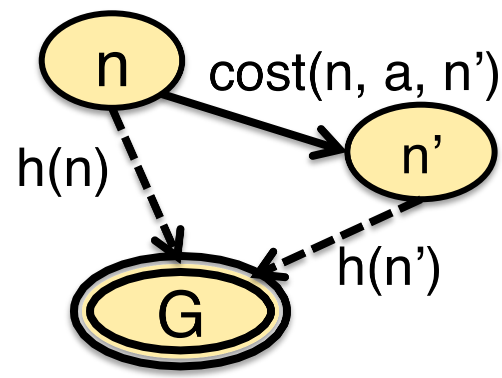
Graph-search A* is optimal if…

Confused?
This is **optional**
material

…h(n) is **monotonic** (=consistent)

Monotonic heuristics obey the **triangle inequality**:
Going from *n* to the goal via *n'* is at least as expensive as going from *n* to the goal.

$$\forall n\, \forall n' \in nodes,$$
$$\forall a \in actions \text{ with } a(n) \rightarrow n'$$
$$h(n) \leq cost(n, a, n') + h(n')$$



NB.: every monotonic heuristic is also admissible

Graph-search A* is optimal if…

…h(n) is **monotonic** (=consistent)

$$h(n) \leq cost(n, a, n') + h(n')$$

Confused?
This is **optional**
material

Proof sketch:

1. If *h* is monotonic, the values of *f(n)* along any path are non-decreasing. (true by definition)
2. When *n* is expanded, *g(n) = g*(n)*
   [we have found the cheapest path to *n*.]

Thus, sorting by *f* finds the cheapest goal first

Possible Heuristic Functions

Often simplify by relaxing some constraint

h_1: 3
h_2 (8-puzzle):
count # tiles out of place
h_3 (route-finding):
sum Manhattan metric distances

Uniform-cost: $h_{\text{uniform-cost}} = 0$

Informed heuristics

Given: A_1^* with heuristic function h_1
and A_2^* with heuristic function h_2
Both A_1^* and A_2^* are admissible

A_1^* is ***more informed*** than A_2^* iff
for all non-goal nodes n $h_1(n) > h_2(n)$

"More informed": "guaranteed not to search more"

Local search

Motivation for local search

How can we find the goal when:

- we can’t keep a queue?
  (because we don’t have the memory)

- we don’t want to keep a queue?
  (because we just need to find the goal state,)

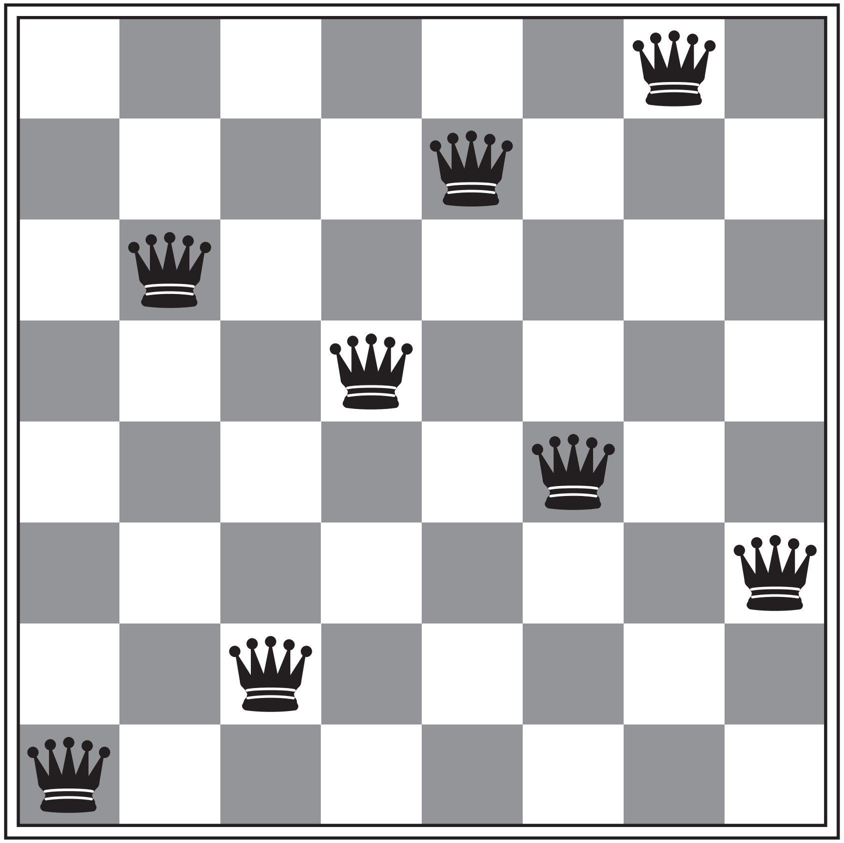
- we can’t enumerate the next actions?
  (because there’s an infinite number)

Local **search algorithms:**

Consider only the current node
and the next action

Also useful for **optimization:**
finding the best state according to
an **objective function**

8-queens



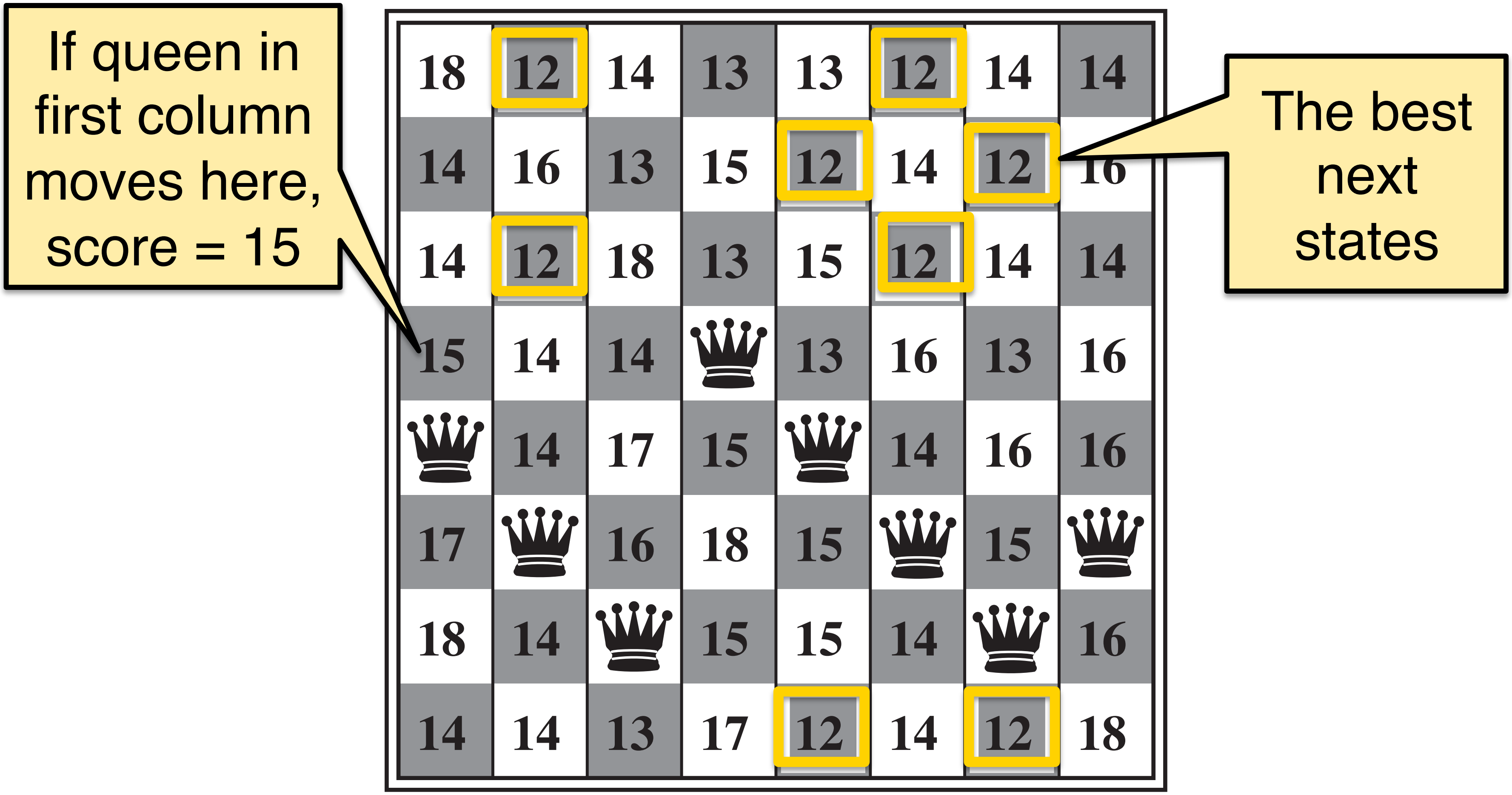
8-queens by local search

Initial state:
A board with 8 queens, one in each column.
(we use a *complete-state formulation*).

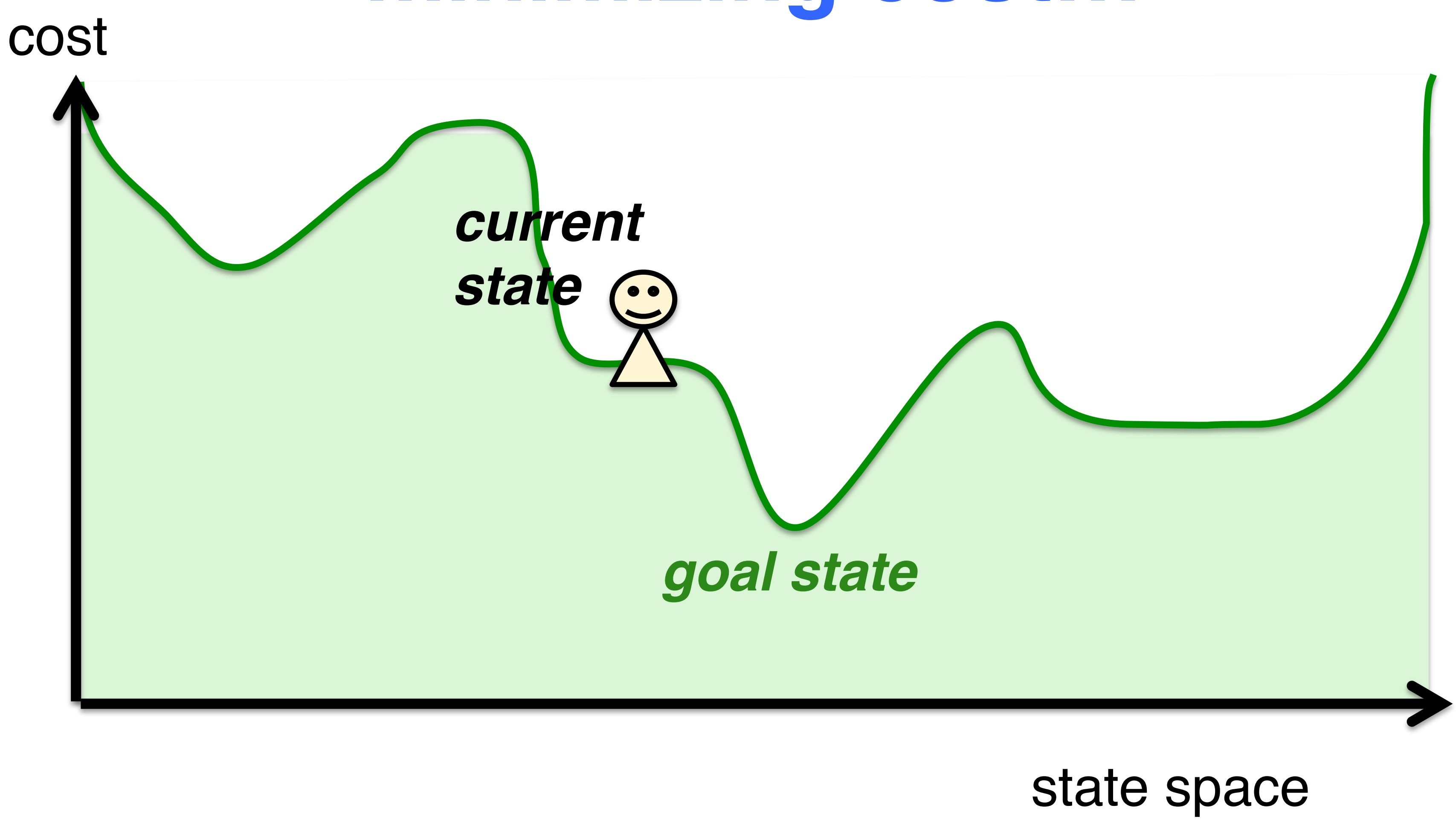
Action: Move one queen to another square in its column.

Heuristic function (score):
How many queens attack each other?

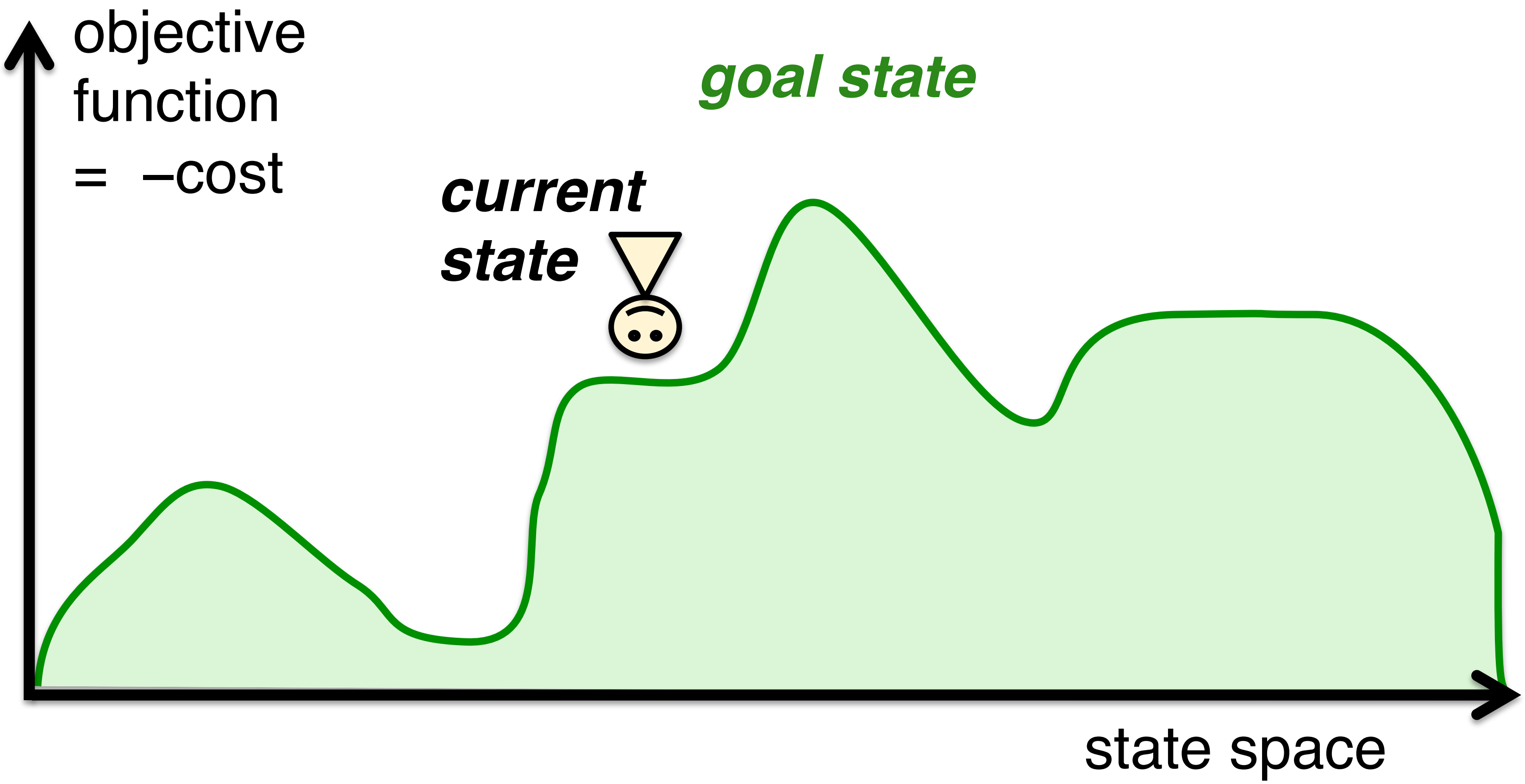
Possible successor states



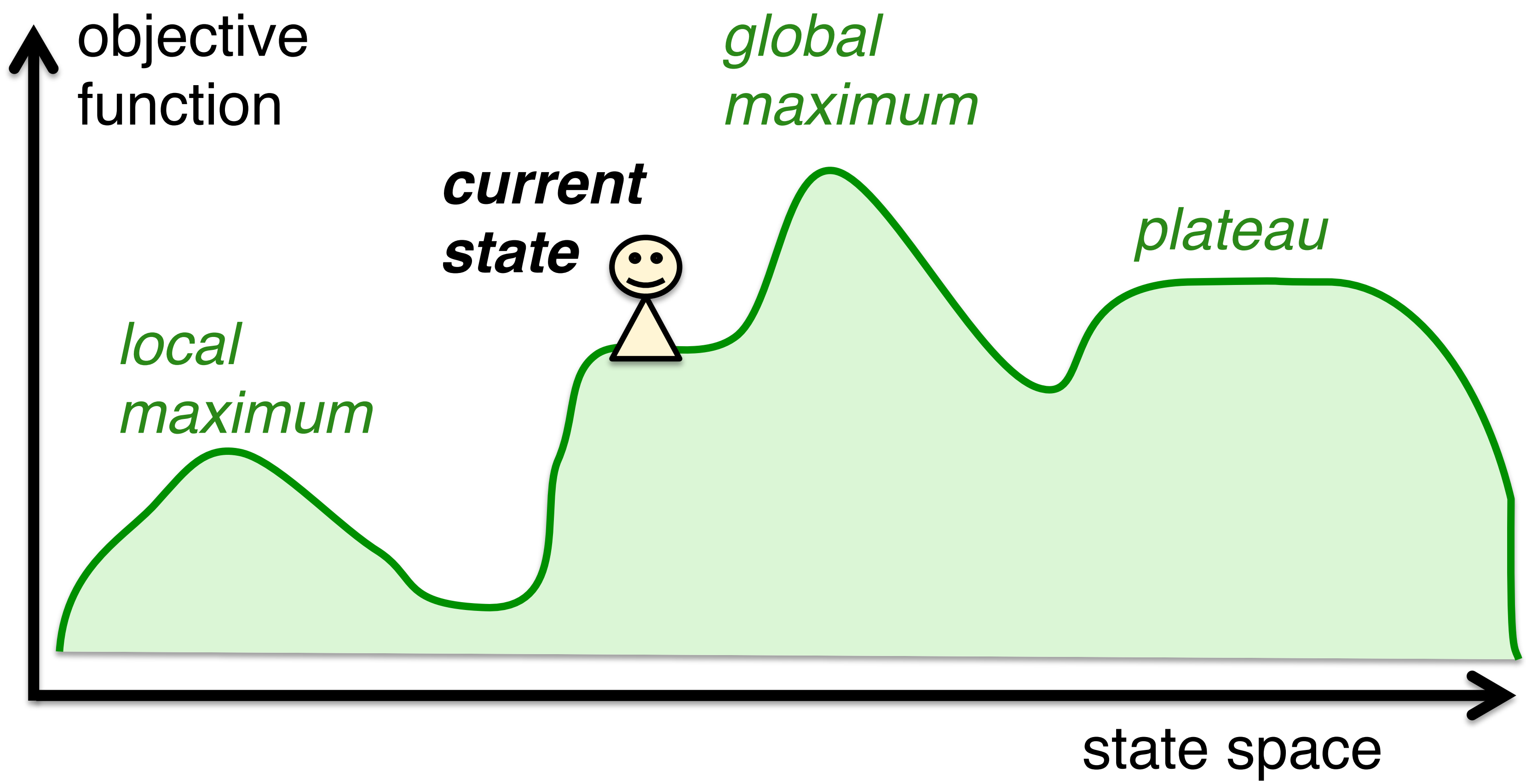
The state space landscape: minimizing cost…



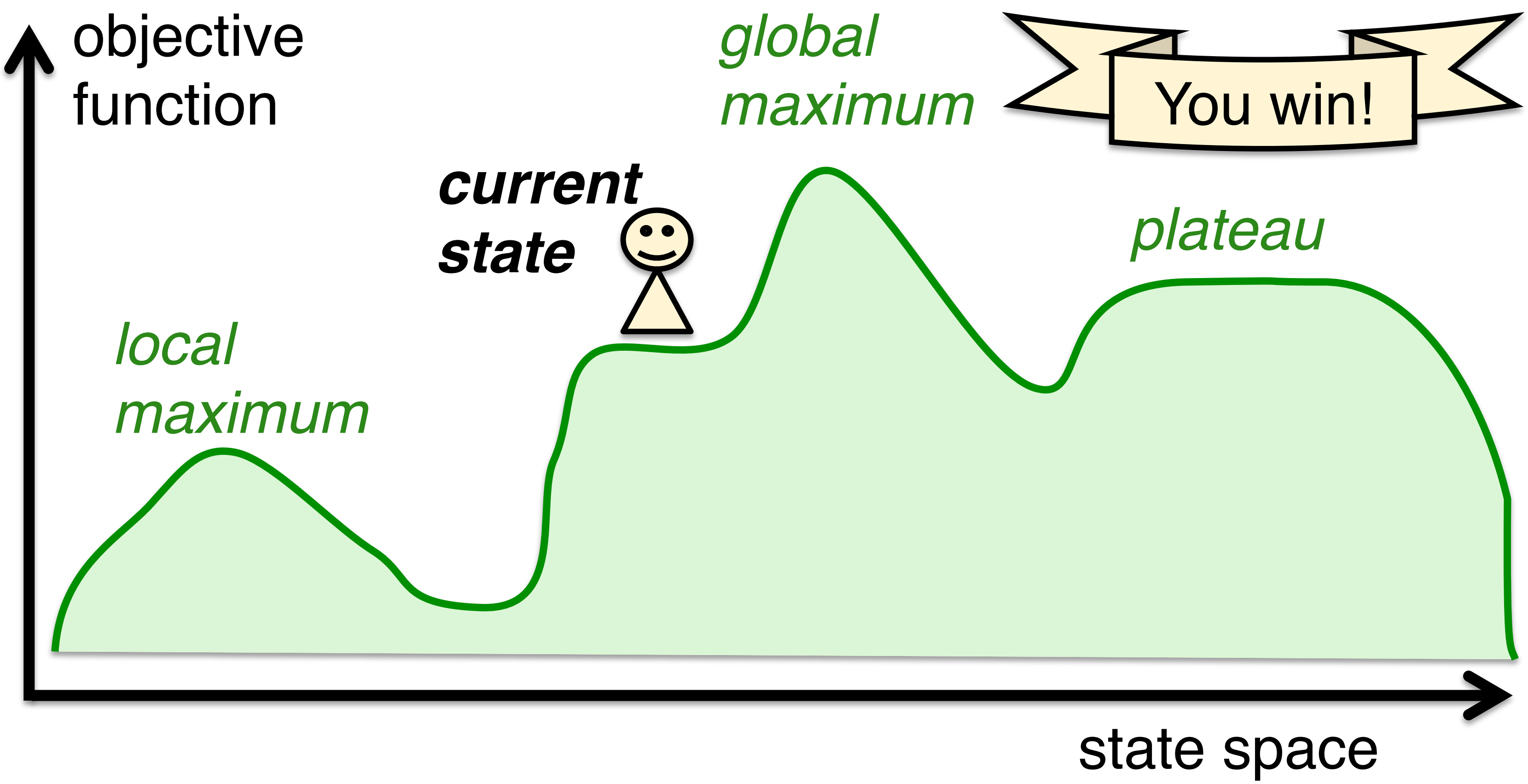
…or *maximizing* an objective function



The state space landscape



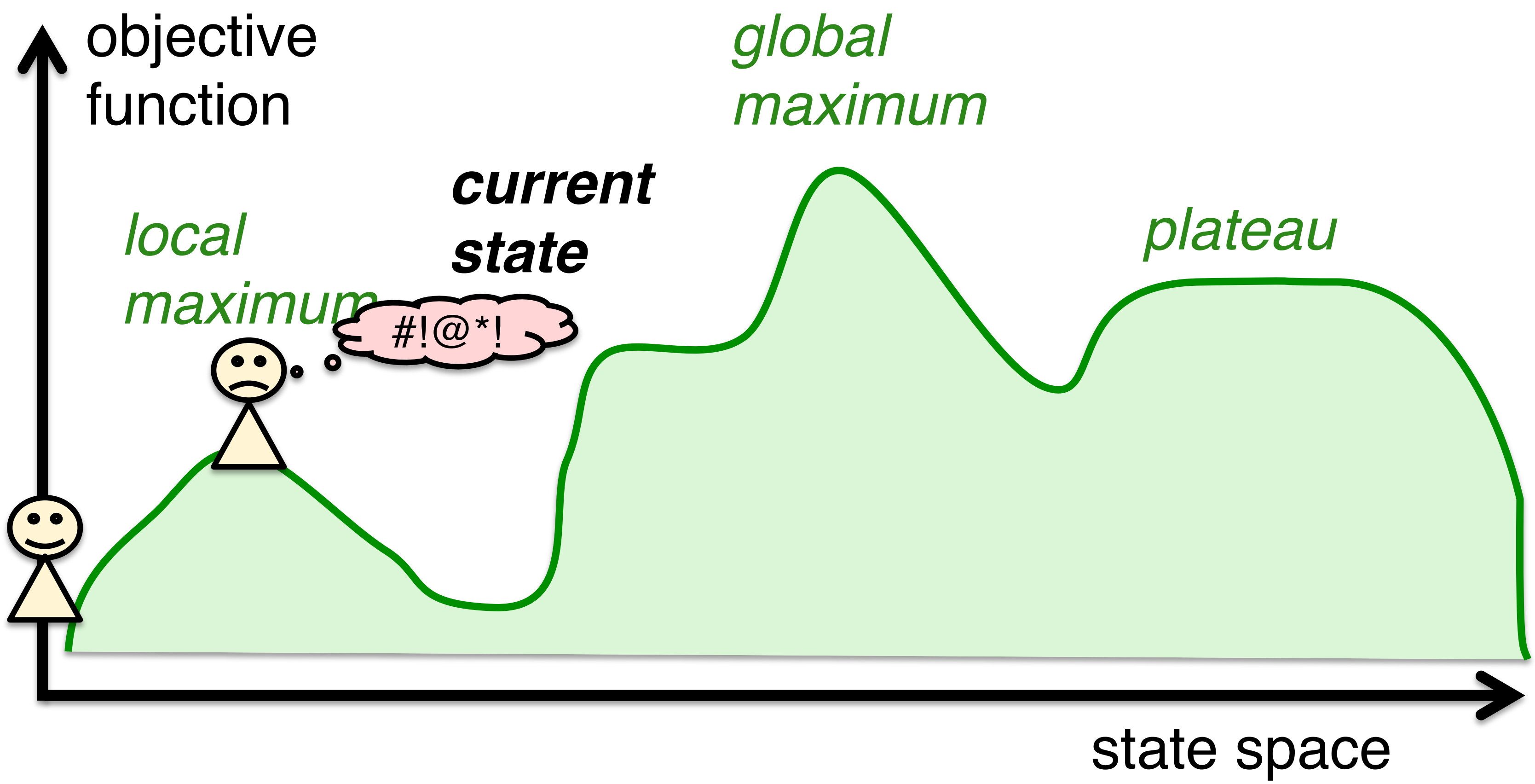
Goal: reaching the global maximum



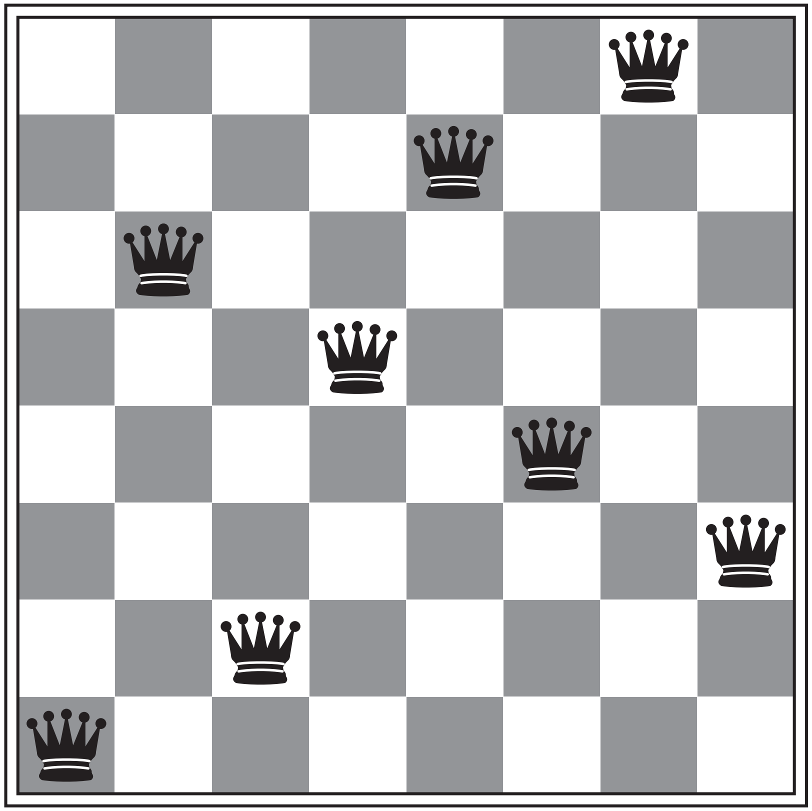
Hill-climbing search

```
HillClimb(Problem P):
  local: n   /*current node*/
  n ← InitialState(P);
Loop:
   n′ ← highestValueNeighborOf(n);
  /* Are we at the peak yet? */
   if n′.VALUE ≤ n.VALUE return n;
   else n = n′ /* Keep climbing up to n′ */
end
```

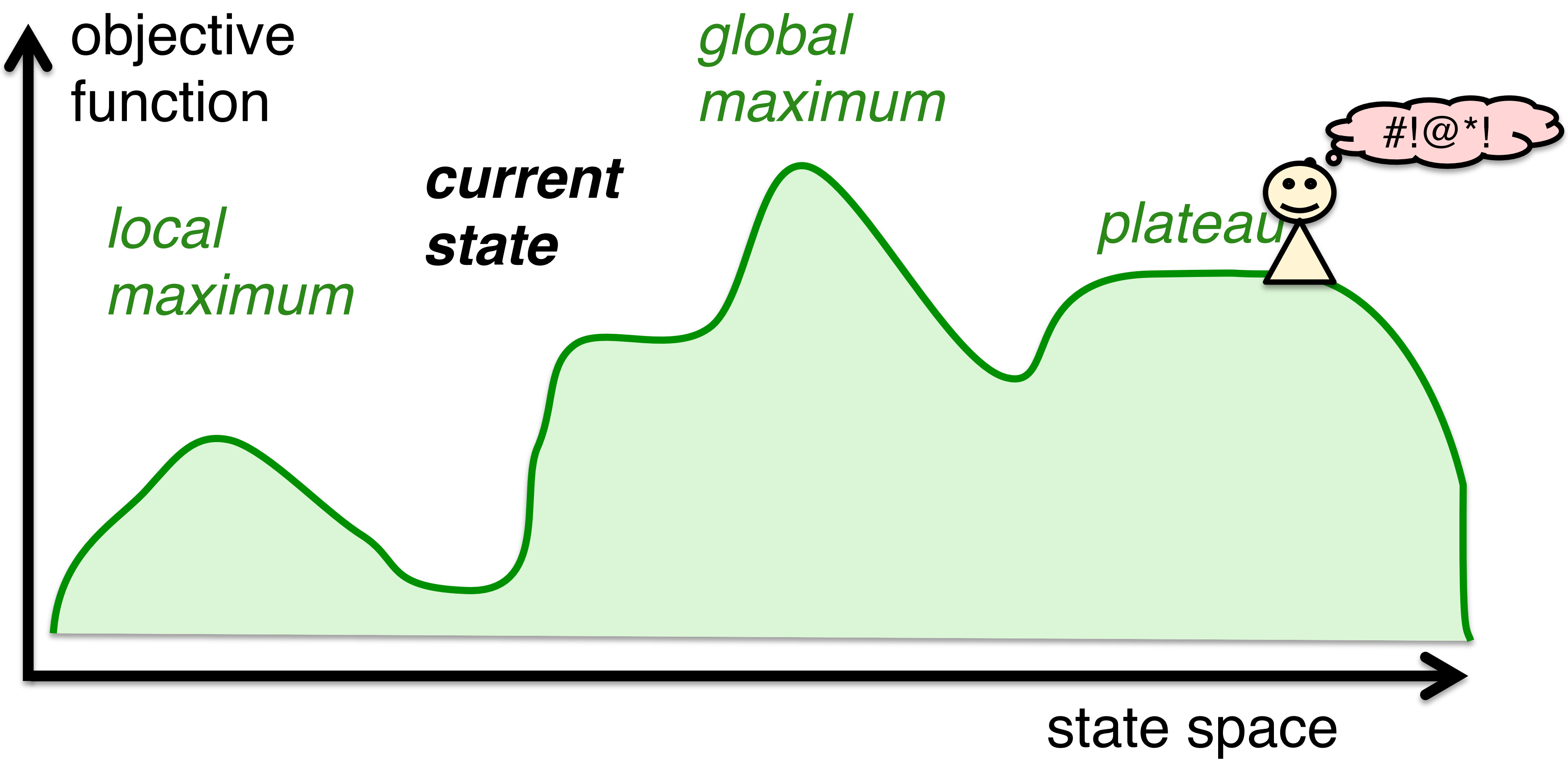
Problem: local maxima are difficult to avoid



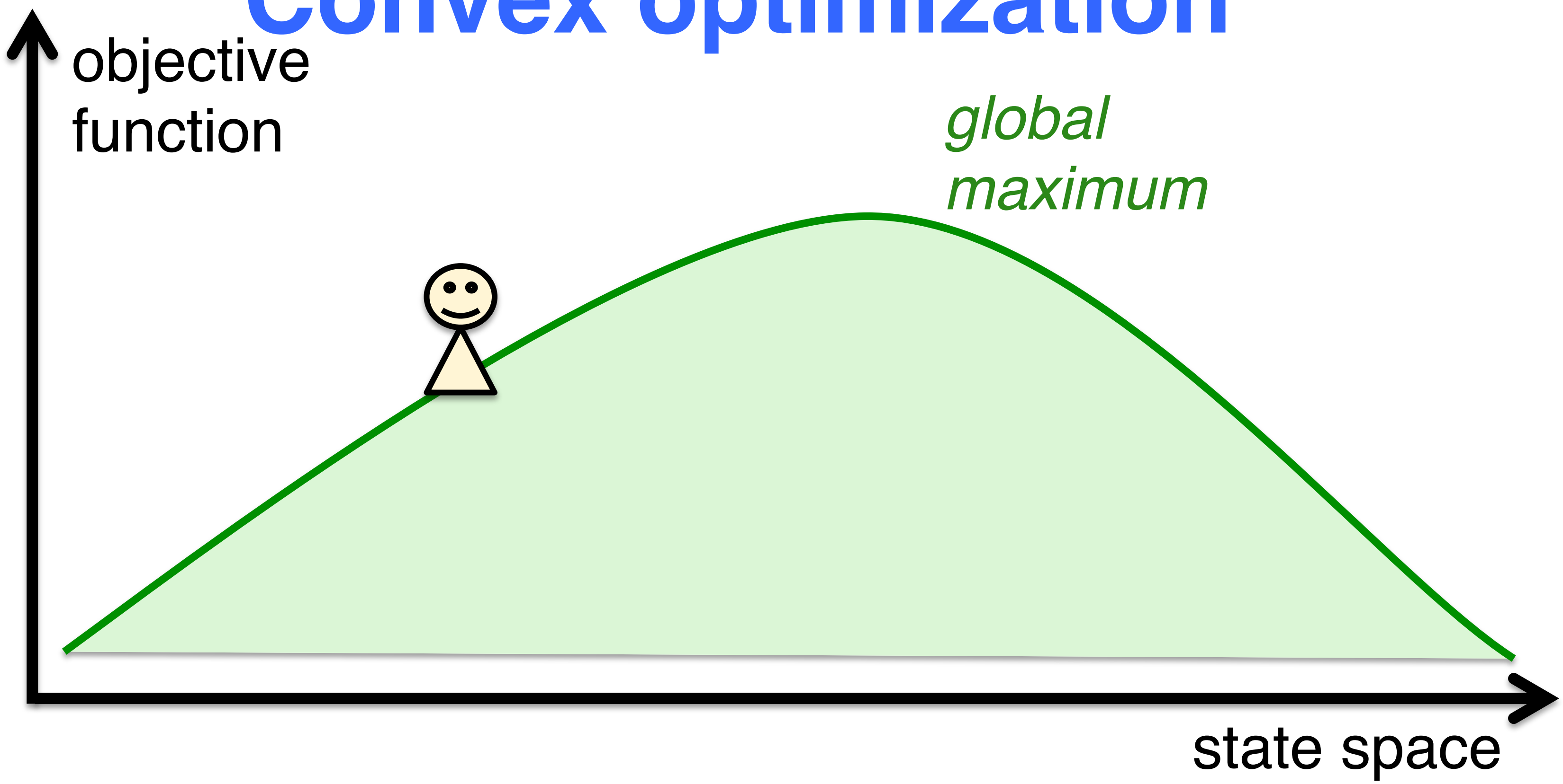
A local maximum for 8 queens



Problem: plateaus are difficult to navigate



Ideal case: Convex optimization



When the objective function is **convex**, search/optimization becomes vastly simplified.

Dealing with local maxima

Random restart hill-climbing:
k trials starting from random positions

k-best (beam) hill climbing:
Pursue k trials in parallel; only keep the k best successors at each time step

Simulated annealing:
Accept downhill moves with non-zero prob.

Random restart hill-climbing search

```
RandomRestartHillClimb(Problem P):
  local: n, nMax; /* current best node*/
  For r = 1 … R:  /* try R times*/
    n ← RandomInitialState(P);
    nMax ← n;
    Loop (climb):
      n′ ← highestValueNeighborOf(n);
      if n′.VALUE > nMax.VALUE nMax ← n′;
      if n′.VALUE > n.VALUE n ← n′;
      else exit Loop;
  return nMax;
```

k-best (beam) hill climbing

```
K-bestHillClimb(Problem P, int k):
 local: N = Array[k] /*vector of K node*/
 for i = 0…k-1:
    N[i] ← RandomInitialState(P);
 N ← sort(N)
 Loop:
   nMax = N[0];
   local: N’ = Array[k], M = Array[2k]
   for i = 0…k-1:
      N’[i] ← highestValueNeighborOf(n[i]);
   M ← sort(append(N, N’));
    N ← M[0…k-1];
   if N[0].VALUE ≤ nMax.VALUE return nMAx;
  end
```

Simulated annealing

A version of *stochastic* hill-climbing:
go downhill with non-zero probability p_{down}

p_{down} depends on **step size:**
p_{down} is greater for smaller steps.

p_{down} depends on current **temperature:**
p_{down} is greater at high temperature.
Temperature is a **function of time:**
Start with high temperature, lower gradually.

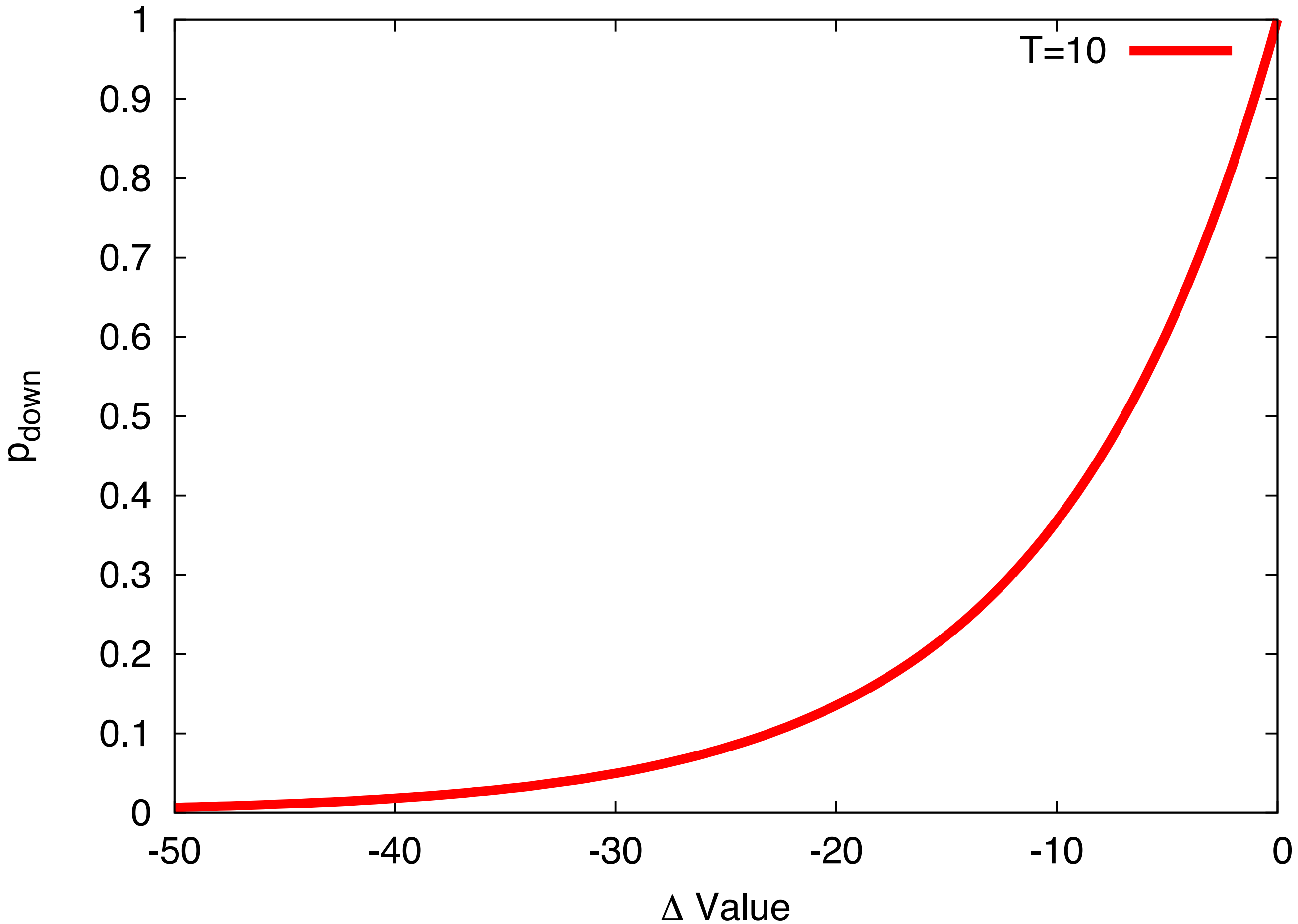
Computing p_{down}

***ΔValue:* neighbor.VALUE – this.VALUE**
In a downhill move, ΔValue is negative.

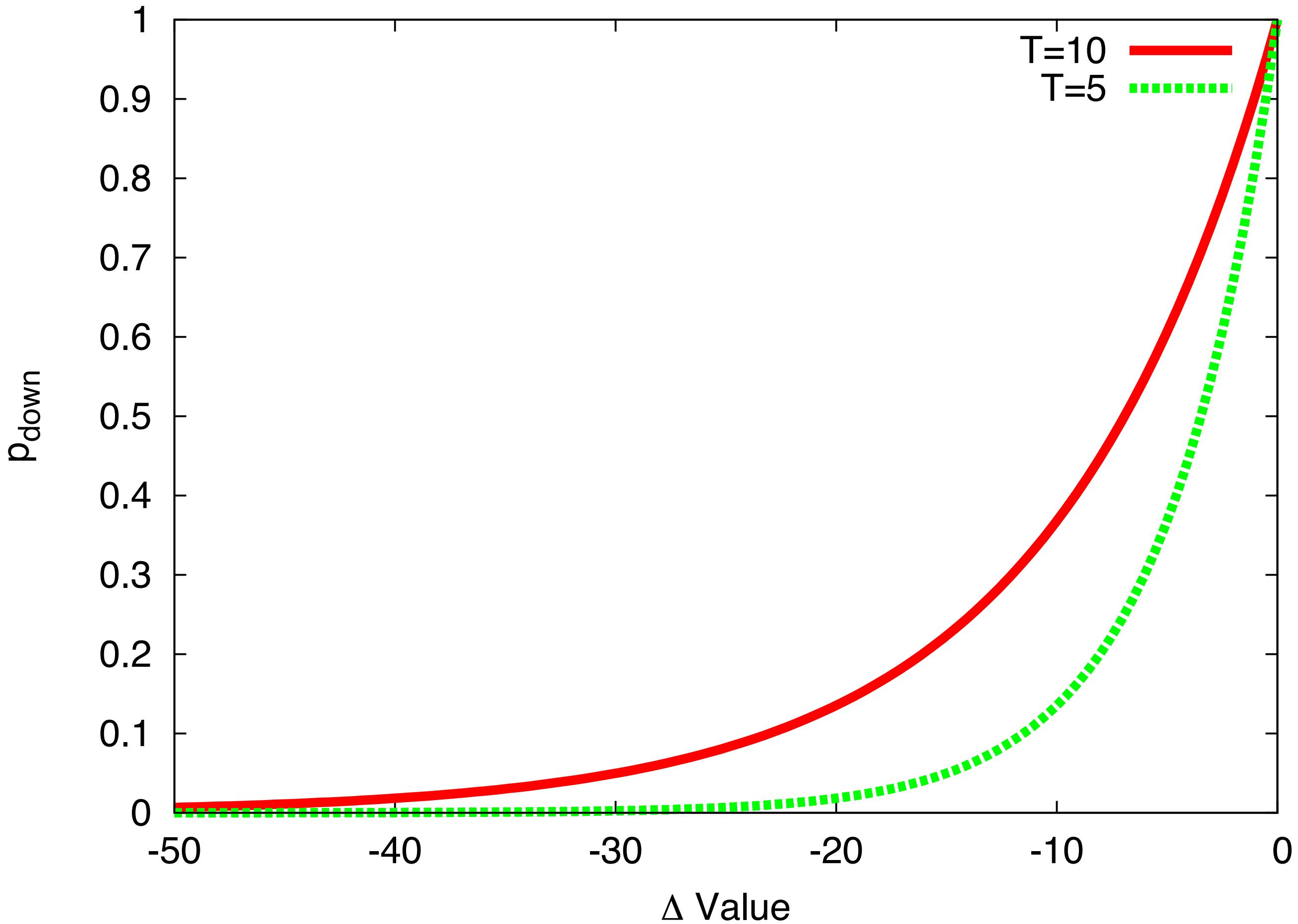
Temperature:
We define temperature to be a decreasing function of time.

$$p_{down} = e^{\Delta Value/Temperature}$$

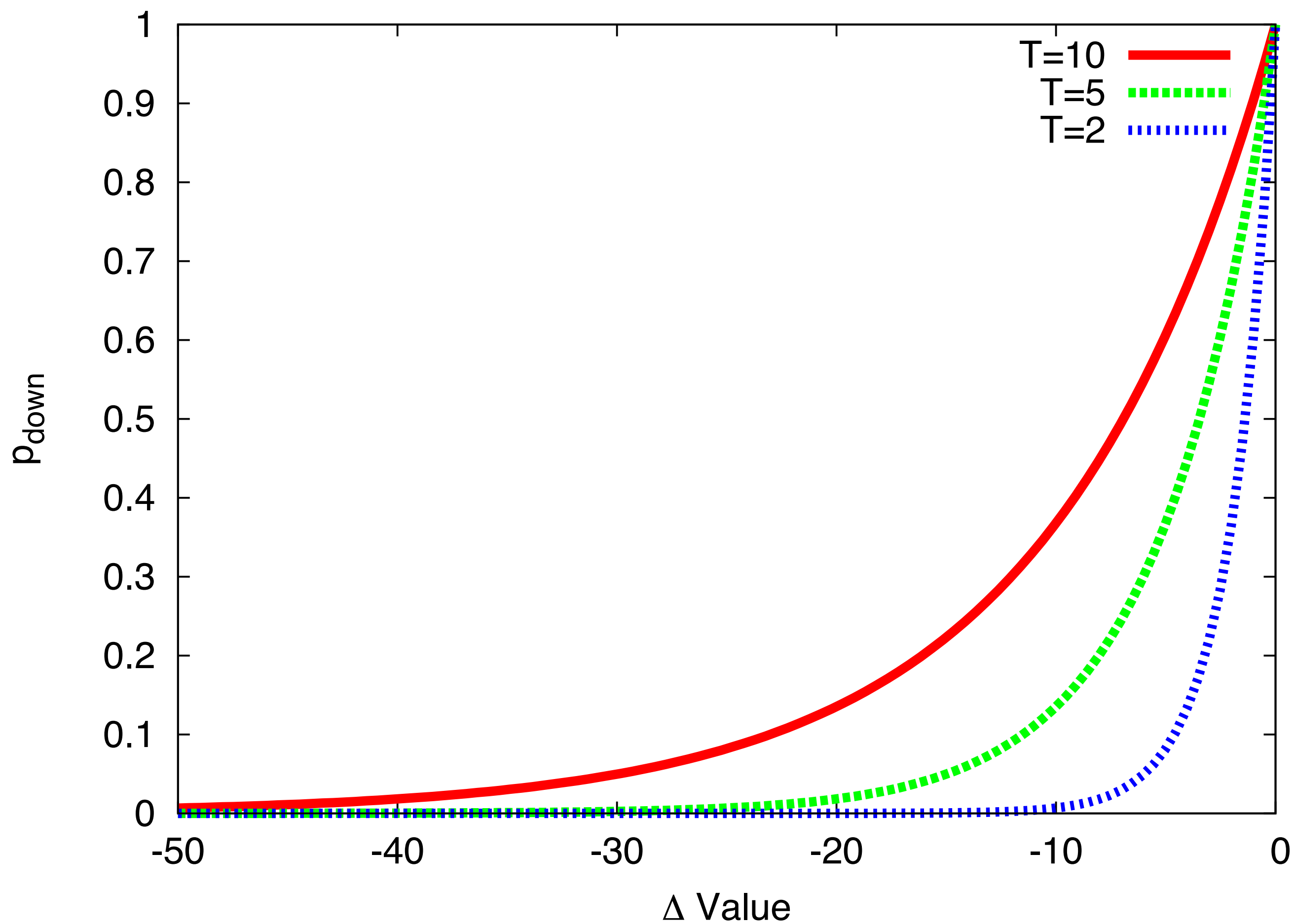
Temperature = 10



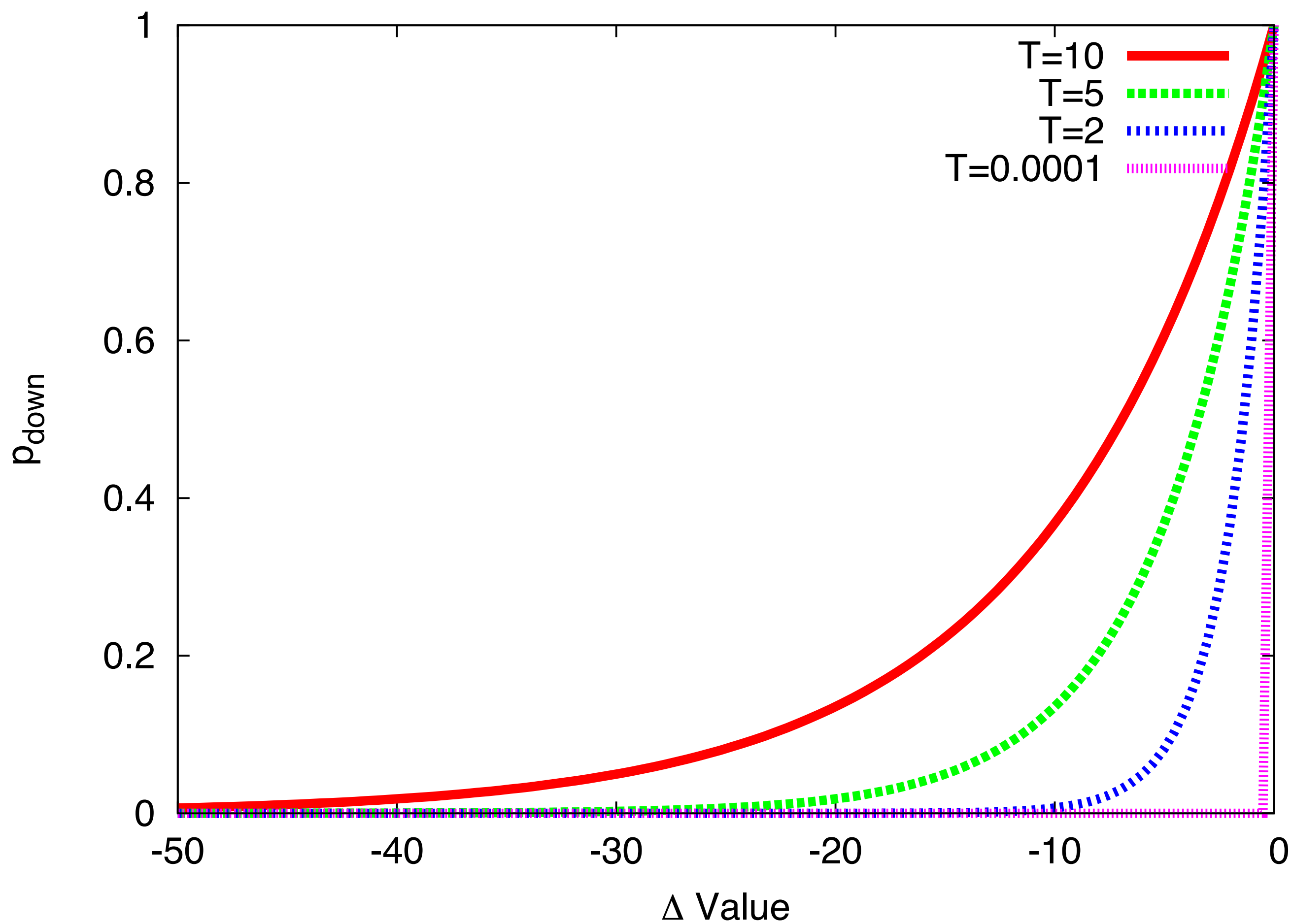
Temperature = 5



Temperature = 2



Temperature = 0.0001



Simulated annealing

```
SimAnnealing(Problem P):
  local: n
  n ← InitialState(P);
For time = 1 to ∞ do:
  Temperature = temp(time);
  if Temperature == 0 return n;
  n′ ← randomNeighborOf(n);
   ΔValue = n′.VALUE - n.VALUE;
  if ΔValue > 0  n ← n′
  else n ← n′ with prob. e^(ΔValue/Temperature)
end
```

To conclude…

Today's key concepts

Heuristic search:
Actions and solutions have costs
Heuristic function: estimate of future cost
Uniform cost, best-first, A*

Local search:
Agent only sees the next steps.
Features of the state space landscape
Hill-climbing, random restart, beam, simulated annealing

Your tasks

Reading:
Chapter 3.5, Chapter 4.1

Compass quiz:
Up at 2pm

Assignments:
MP 1 will go out this afternoon.