بسم اللّه الرحمن الرحیم

- تذکر۱: لطفاً یادداشت‌برداری کنید. آنچه در کلاس مطرح می‌شود ممکن است فراتر از محتوای اسلایدها باشد.
- تذکر ۲: اگر احیاناً در ترجمه یا فهم اسلایدها مشکل داشتید، لطفاً در ساعات رفع اشکال به دفتر بنده مراجعه کنید. (ساعات رفع اشکال به‌زودی تعیین خواهند شد انشاءالله.)
- تذکر ۳: اسلایدها و دیگر محتواهای مرتبط با درس روی سامانهٔ درس (lms.iut.ac.ir) آپلود می‌شوند.

سه عاملی که جلسهٔ پیش به معرفی آنها پرداختیم، یعنی بهینه‌های محلی، فلات‌ها، و Ridges، تأثیر بسیار جدی‌ای بر کارایی جستجو دارند:

In each case, the algorithm reaches a point at which no progress is being made. Starting from a randomly generated 8-queens state, steepest-ascent hill climbing gets stuck 86% of the time, solving only 14% of problem instances. It works quickly, taking just 4 steps on average when it succeeds and 3 when it gets stuck.

اگر در یک فلات به فعالیت ادامه دهیم، آیا شاهد بهبود خواهیم بود؟

In the previous session, the algorithm in page 16 halts if it reaches a plateau where the best successor has the same value as the current state. Might it not be a good idea to keep going—to allow a sideways move in the hope that the plateau is really a shoulder? The answer is usually yes, but we must take care. If we always allow sideways moves when there are no uphill moves, an infinite loop will occur whenever the algorithm reaches a flat local maximum that is not a shoulder. One common solution is to put a limit on the number of consecutive sideways moves allowed. For example, we could allow up to, say, 100 consecutive sideways moves in the 8-queens problem. This raises the percentage of problem instances solved by hill climbing from 14% to 94%. Success comes at a cost: the algorithm averages roughly 21 steps for each successful instance and 64 for each failure.

Variants of hill climbing

☞ **Stochastic hill climbing** chooses at random from among the uphill moves; the probability of selection can vary with the steepness of the uphill move. This usually converges more slowly than steepest ascent, but in some state landscapes, it finds better solutions.

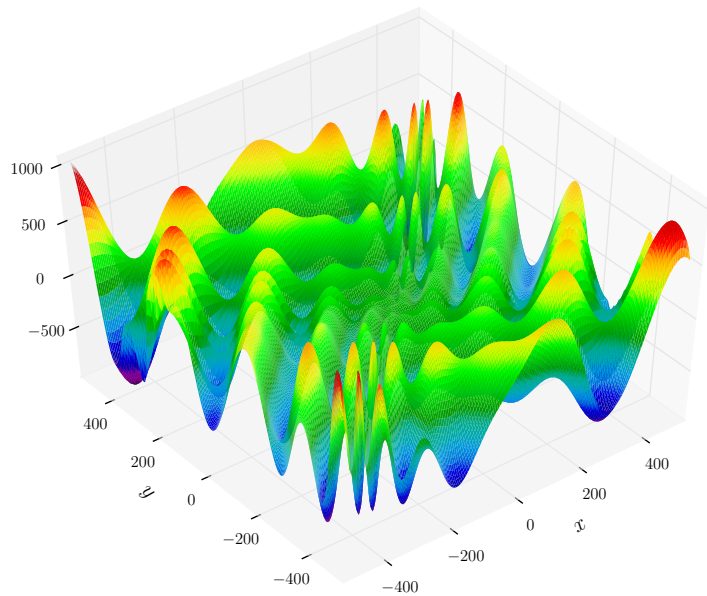
☞ **First-choice hill climbing** implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state. This is a good strategy when a state has many (e.g., thousands) of successors.

☞ **Random-restart hill climbing** adopts the well-known adage, "If at first you don't succeed, try, try again." It conducts a series of hill-climbing searches from randomly generated initial states, until a goal is found. For 8-queens, random-restart hill climbing is very effective indeed. Even for three million queens, the approach can find solutions in under a minute.

Random-restart hill climbing

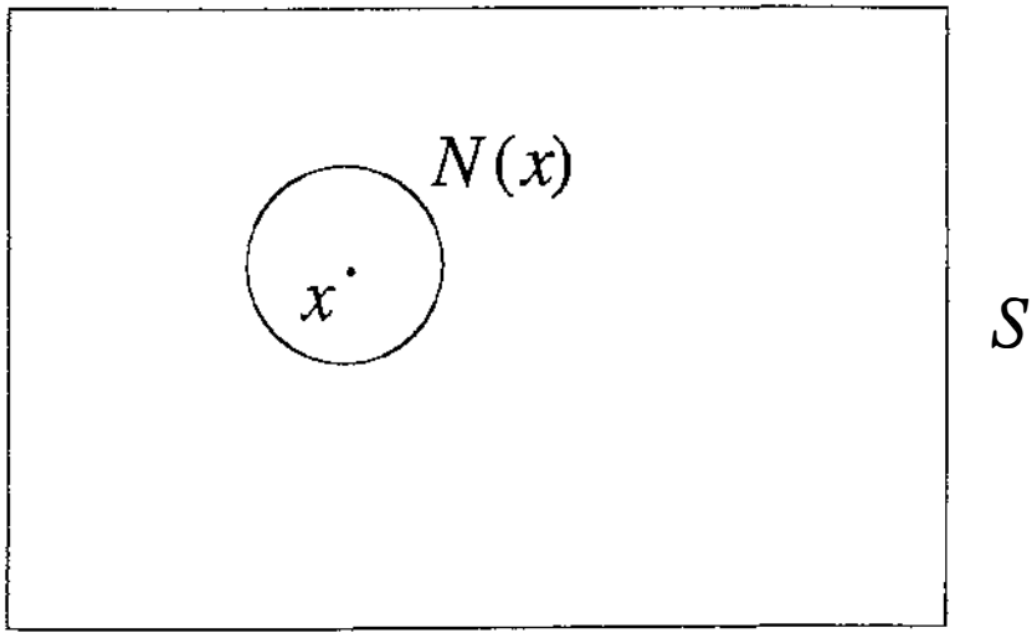
```
procedure iterated hill-climber
begin
   t ← 0
   initialize best
   repeat
      local ← FALSE
      select a current point v_c at random
      evaluate v_c
      repeat
         select all new points in the neighborhood of v_c
         select the point v_n from the set of new points
            with the best value of evaluation function eval
         if eval(v_n) is better than eval(v_c)
            then v_c ← v_n
            else local ← TRUE
      until local
      t ← t + 1
      if v_c is better than best
         then best ← v_c
   until t = MAX
end
```

The success of hill climbing depends very much on the shape of the state-space land-scape: if there are few local maxima and plateaux, random-restart hill climbing will find a good solution very quickly. NP-hard problems typically have an exponential number of local maxima to get stuck on. Despite this, a reasonably good local maximum can often be found after a small number of restarts.



الان قصد داریم کمی دربارهٔ لفظ «همسایگی» حرف بزنم:

If we concentrate on a region of the search space that's "near" some particular point in that space, we can describe this as looking at the neighborhood of that point. We can define the nearness between points in the search space in many different ways.



Example: we can define a **2-swap** mapping for the TSP that generates a new set of potential solutions from any potential solution x. Every solution that can be generated by swapping two cities in a chosen tour can be said to be in the neighbourhood of that tour under the 2-swap operator.

If we have 20 cities:

Current State: $15 - 3 - 11 - 19 - 17 - 2 - \ldots - 6$

Neighbours:

$15 - 17 - 11 - 19 - 3 - 2 - \ldots - 6$ (swapping cities at the second and fifth locations),

$2 - 3 - 11 - 19 - 17 - 15 - \ldots - 6$ (swapping cities at the first and sixth locations),

$15 - 3 - 6 - 19 - 17 - 2 - \ldots - 11$ (swapping cities at the third and last locations),

etc.

Exercise: If we have n cities, then how many neighbours each state has under the 2-swap operator?

یک مثال دیگر از شیوهٔ تعریف همسایگی

Example: For the CNF-SAT problem we could define a **1-flip** mapping that generates a set of potential solutions from any other potential solution x. These are all the solutions that can be reached by **flipping a single bit** in a particular binary vector, and these would be in the neighborhood of x under the 1-flip operator. For example, a solution x (a binary string of, say, $n = 20$ variables):
01101010001000011111
has n neighbors. These include:
11101010001000011111 (flipping the first bit),
00101010001000011111 (flipping the second bit),
01001010001000011111 (flipping the third bit),
etc.

یک سؤال مهم

برای حل یک نمونه از مسئلهٔ ۸-وزیر، آنچه باید «کمینه» شود، تعداد زوج‌هایی از وزیرها است که یکدیگر را تهدید می‌کنند. برای حل یک نمونه از مسئلهٔ TSP، آنچه باید «کمینه» شود، طول تور است. برای حل یک نمونه از مسئلهٔ SAT چه کمیتی است که باید «کمینه یا بیشینه» شود؟

دربارهٔ اندازهٔ همسایگی: بزرگ باشد؟ کوچک باشد؟

SLS methods present an interesting trade-off between the size of the neighborhood and the efficiency of the search. If the size of the neighborhood is relatively small then the algorithm may be able to search the entire neighborhood quickly. Only a few potential solutions may have to be evaluated before a decision is made on which new solution should be considered next. However, such a small range of visibility increases the chances of becoming trapped in a local optimum! This suggests using large neighborhoods: a larger range of visibility could help in making better decisions.

Perturbative vs Constructive Search

Candidate solutions for instances of combinatorial problems are composed of solution components, such as the assignments of truth values to individual propositional variables (atomic assignments) in the case of SAT. Hence, given candidate solutions can easily be changed into new candidate solutions by modifying one or more of the corresponding solution components. This can be characterised as perturbing a given candidate solution, and hence we classify search algorithms that rely on this mechanism for generating the candidate solutions to be tested as perturbative search methods.

Applied to SAT, perturbative search would start with one or more complete truth assignments and then at each step generate other truth assignments by changing the truth values of a number of variables in each such assignment.

Constructive search methods

Constructive search is regarded as the process of attaining one single solution, by iteratively adding components to a partial solution. (Generating complete candidate solutions by iteratively extending partial candidate solutions.)

Local search methods are often, but not always based on perturbative search.

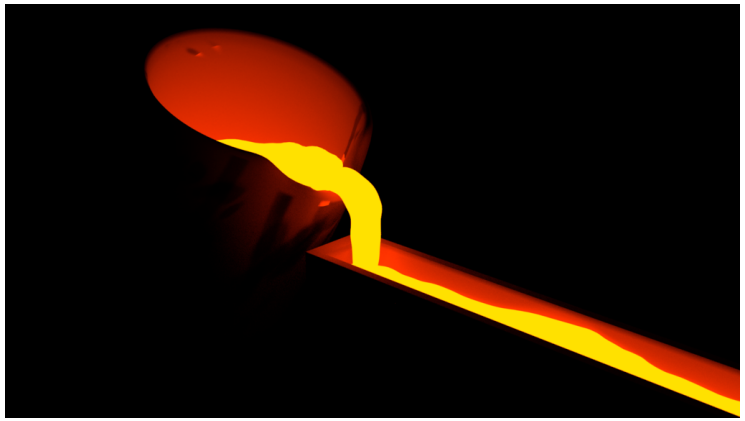
چگونه باید از بهینه‌های محلی فرار کرد؟

Some of the algorithms we study in this course guarantee finding the global solution, others don't, but they all share a common pattern. Either they guarantee discovering the global solution, but are too expensive (i.e., too time consuming) for solving typical real-world problems, or else they have a tendency of "getting stuck" in local optima. Since there is almost no chance to speed up algorithms that guarantee finding the global solution, i.e., there is almost no chance of finding polynomial-time algorithms for most real problems (as they tend to be NP-hard), the other remaining option aims at designing algorithms that are capable of escaping local optima.

Simulated Annealing

A hill-climbing algorithm that never makes "downhill" moves toward states with lower value (or higher cost) is guaranteed to be incomplete, because it can get stuck on a local maximum. In contrast, a purely random walk—that is, moving to a successor chosen uniformly at random from the set of successors—is complete but extremely inefficient. Therefore, it seems reasonable to try to combine hill climbing with a random walk in some way that yields both efficiency and completeness. Simulated annealing is such an algorithm.

سرد کردن تدریجی فلز یا شیشهٔ مذاب



In metallurgy, annealing is the process used to temper or harden metals and glass by heating them to a high temperature and then gradually cooling them, thus allowing the material to reach a low-energy crystalline state.

Annealing is a metallurgical process where molten metals are slowly cooled to allow them to reach a low energy state, making them stronger. Simulated annealing is an analogous method for optimization.

رفتار الگوریتم وقتی دما بالا و وقتی پایین است

The random movement corresponds to high temperature; at low temperature, there is little randomness. Simulated annealing is a stochastic local search algorithm where the temperature is reduced slowly, starting from a random walk at high temperature eventually becoming pure greedy descent as it approaches zero temperature. The randomness should allow the search to jump out of local minima and find regions that have a low heuristic value; whereas the greedy descent will lead to local minima. At high temperatures, worsening steps are more likely than at lower temperatures.

SA starts with high values of T making the procedure more similar to a purely random search, and then gradually decreases the value of T. Towards the end of the run, the values of T are quite small so the final stages of simulated annealing merely resemble an ordinary hill-climber. In addition, we always accept new points if they are better than the current point.

function SIMULATED-ANNEALING(*problem*, *schedule*) **returns** a solution state
  inputs: *problem*, a problem
    schedule, a mapping from time to "temperature"

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

The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. Downhill moves are accepted readily early in the annealing schedule and then less often as time goes on. The *schedule* input determines the value of the temperature T as a function of time.