Beyond Classical Search

Bùi Tiến Lên

2022

KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

Contents

Optimization problems

- In many optimization problems, **path** is irrelevant; the **goal state** itself is the solution
- "Pure optimization" problems
  - All states have an objective function
  - Goal is to find state with max (or min) objective value
  - Does not quite fit into path-cost/goal-state formulation
  - Local search can do quite well on these problems.
- Then **state space** = set of "complete" configurations
  - find *optimal* configuration, e.g., TSP
  - find configuration satisfying constraints, e.g., timetable

Local search algorithms

- Local search
  - Keep track of single current state
  - Move only to neighboring states
  - Ignore paths
- Advantages:
  - Use very little memory
  - Can often find reasonable solutions in large or infinite (continuous) state spaces.

State-space landscape

- An useful landscape has both "location" (defined by the state) and "elevation" (defined by the value of the heuristic cost function or objective function)

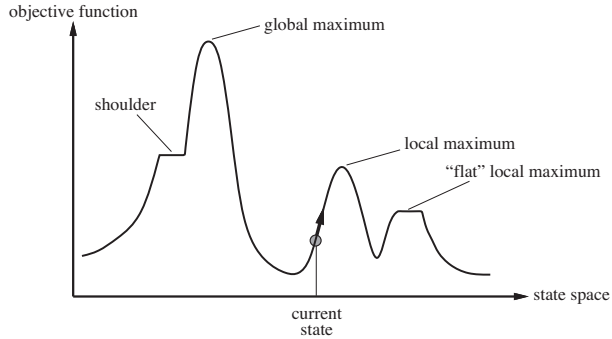


Figure 1: A one-dimensional state-space landscape in which elevation corresponds to the objective function

Hill-climbing Search

Hill-climbing Search

- The **hill-climbing** search algorithm (**steepest-ascent** version) is simply a loop that continually moves in the direction of increasing value (uphill). It terminates when it reaches a "peak" where no neighbor has a higher value.
- The algorithm does not maintain a search tree, so the data structure for the current node need only record the state and the value of the objective function.
- Hill climbing does not look ahead beyond the immediate neighbors of the current state

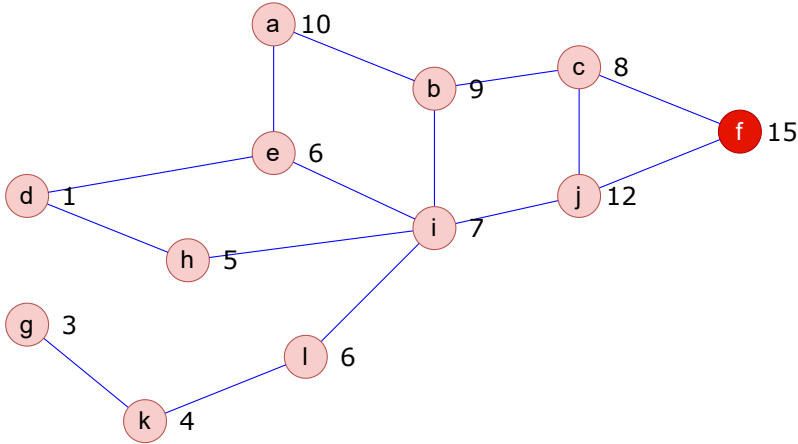
Algorithm

```
function HILL-CLIMBING(problem)
returns a state that is a local maximum
  current ← MAKE-NODE(problem.INITIAL-STATE)
  loop do
    neighbor ← a highest-valued successor of current
    if neighbor.VALUE ≤ current.VALUE then return current.STATE
    current ← neighbor
```

- At each step the current node is replaced by the best neighbor; in this version, that means the neighbor with the highest VALUE, but if a heuristic cost estimate h is used, we would find the neighbor with the lowest h.
- Hill-climbing algorithms typically choose randomly among the set of best successors if there is more than one.

Illustration

- Find a state with the highest value.

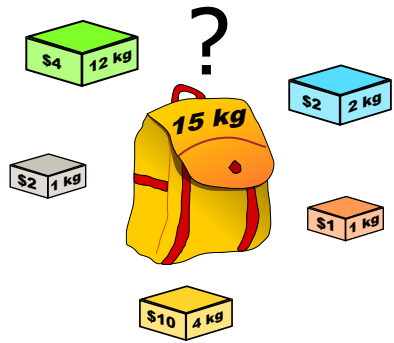


Example

- Find a state with the highest value in a grid problem.

8	10	5	8	1	9	10	8	8	8
6	2	4	2	6	5	6	5	10	10
7	5	1	1	4	10	10	9	7	6
8	2	8	9	3	7	8	7	8	10
5	7	2	9	4	7	9	8	5	9
10	7	6	9	9	15	12	14	10	12
7	9	9	9	10	14	10	11	12	10
8	10	5	7	7	11	12	14	14	13
6	8	7	9	7	12	10	11	14	13
6	8	7	9	8	15	15	12	10	14

Knapsack Problem

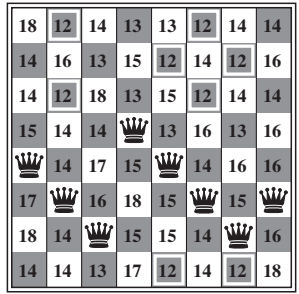


- **Configurations**: Any combination of objects inside the knapsack
- **Initial configuration**: Empty knapsack
- **Actions**: Put objects in and take objects from the knapsack
- **Best configuration**: A configuration with $\max \sum \mathit{value}_i$

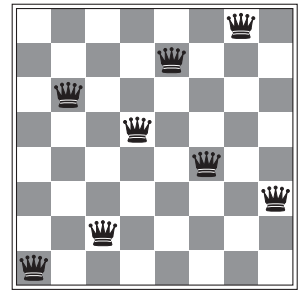
8-queens Problem

- **Complete-state formulation**
  - All 8 queens on the board, one per column
- Successor function
  - Move a single queen to another square in the same column $8 \times 7 = 56$ successors
- Heuristic cost function $h(n)$
  - The number of pairs of queens that are attacking each other
  - Global minimum has $h(n) = 0$

8-queens Problem (cont.)



(a)



(b)

Figure 2: (a) An 8-queens state with heuristic cost estimate $h = 17$, showing the value of h for each possible successor obtained by moving a queen within its column. The best moves are marked. (b) A local minimum in the 8-queens state space; the state has $h = 1$ but every successor has a higher cost. It takes just 5 steps from state (a) to state (b)

Problems

- Hill climbing often gets stuck for the following reasons:
  - **Local maxima**: a local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum
  - **Ridges**: ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.
  - **Plateaux**: a plateau is a flat area of the state-space landscape.
- In each case, the algorithm reaches a point at which no progress is being made. Starting from a randomly generated 8-queens state, hill climbing gets stuck 86% of the time, solving only 14% of problem instances.
  - It takes 4 steps on average for each successful instance and 3 for each failure

Possible solution

- If no uphill (downhill) moves, allow **sideways moves** in hope that algorithm can escape local maximum
- A limit on the possible number of sideways moves required to avoid infinite loops
- 8-queens problem
  - Allow sideways moves up to 100 $\rightarrow$ percentage of problem instances solved raises from 14 to 94%
  - It takes 21 steps on average for each successful instance and 64 for each failure

Variants of hill climbing

- **Stochastic hill climbing**
  - Choose at random from among the uphill moves with a probability of selection varied with the moves' steepness
  - Usually converge more slowly than steepest ascent, but find better solutions in some cases
- **First-choice hill climbing**
  - Generate successors randomly until better than the current state
  - Good strategy when a state has many successors (e.g., thousands)

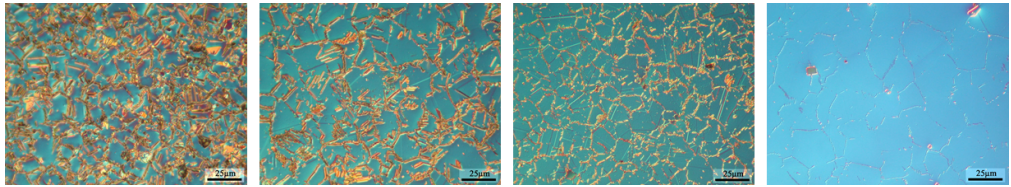
Variants of hill climbing (cont.)

- **Random-restart hill climbing**: "If at first you don't succeed, try, try again."
  - A series of hill-climbing searches from randomly generated initial states until a goal is found.
  - If each hill-climbing search has a probability p of success, then the expected number of restart $1/p$
- **Random-walk hill climbing**: At each step do one of the two
  - Greedy: With probability p move to the neighbor with largest value
  - Random: With probability $1 - p$ move to a random neighbor

Simulated Annealing

Simulated Annealing

- Simulated Annealing = physics inspired twist on random walk
- A metal alloy or dissolution is heated at high temperatures and progressively cooled in a controlled way
- If the cooling process is adequate the minimal state of energy of the system is achieved (global minimum)



Algorithm

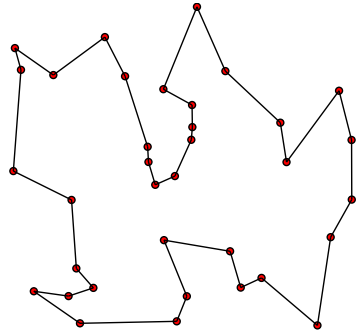
```
function SIMULATED-ANNEALING(problem, schedule)
returns a solution state
inputs: problem, a problem
        schedule, a mapping from time to "temperature"
  current ← MAKE-NODE(problem.INITIAL-STATE)
  for t = 1 to ∞ do
    T ← schedule(t)
    if T = 0 then return current
    next ← a randomly selected successor of current
    ΔE ← next.VALUE − current.VALUE
    if ΔE > 0 then current ← next
    else current ← next only with probability e^(ΔE/T)
```

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 cooling a liquid or metal until particles achieve a certain frozen crystal state
  - free variables are like particles
  - seek "low energy" (high quality) configuration
  - slowly reducing temp. T with particles moving around randomly

TSP Problem

- Given a list of cities and the distances
- What is the shortest possible route $\{(x_1, y_1), ..., (x_n, y_n)\}$ that visits each city exactly once and returns to the origin city?

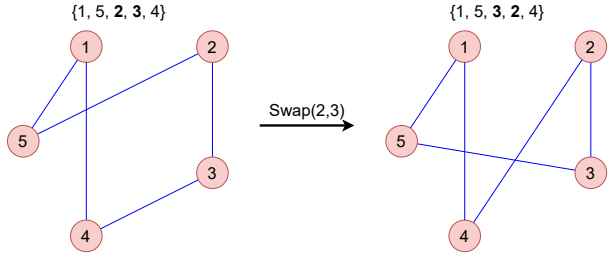


TSP Problem

- Intial a list of cities
- An energy function (sum of the distance among the cities, following the order in the list)

$$f = \sum_{i=1}^{n} \sqrt{(x_i - x_{i+1})^2 + (y_i - y_{i+1})^2} \tag{1}$$

- Action: swap two cities

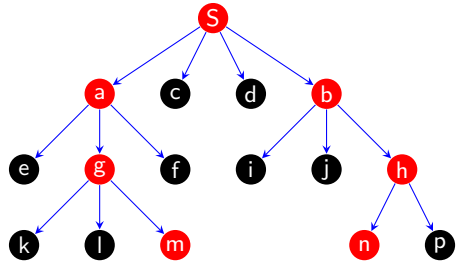


Local Beam Search

Local Beam Search

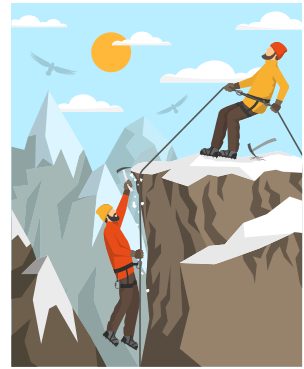
Keep track of k states rather than just one (extreme reaction to memory problems)

- Begin with k randomly generated states
- At each step, all successors of all k tates are generated
- If any of successors is goal $\rightarrow$ finished
- Else select k best from successors and repeat



Local Beam Search (cont.)

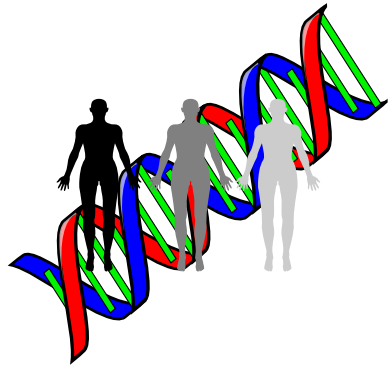
- Not the same as k random-start searches run in parallel!
- In its simplest form, local beam search can suffer from a lack of diversity among the k states
  - They can quickly become concentrated in a small region of the state space $\rightarrow$ an expensive version of hill climbing
- **Stochastic beam search**
  - Choose k successors at random, with the probability of choosing a given successor being an increasing function of its value



Genetic Algorithms

Genetic algorithms

- A **genetic algorithm** (or **GA**) is a variant of stochastic beam search in which successor states are generated by combining *two* parent states rather than by modifying a single state.
- The analogy to natural selection is the same as in stochastic beam search, except that now we are dealing with sexual rather than asexual reproduction.



Genetic algorithms (cont.)

- The GA begins with a set of k randomly generated states, called the **population**.
- Each state is rated by the objective function, called the **fitness function**.
  - Higher values for better state
- Produce the next generation by "**simulated evolution**"
  - **Random selection**
  - **Crossover**
  - **Random mutation**

Algorithm

```
function GENETIC-ALGORITHM(population,FITNESS-FN) returns an individual
inputs: population, a set of individuals
        FITNESS-FN, a function that measures the fitness of an individual
  repeat
    new_population ← ∅
    for i = 1 to SIZE(population) do
      x ← RANDOM-SELECTION(population,FITNESS-FN)
      y ← RANDOM-SELECTION(population,FITNESS-FN)
      child ← REPRODUCE(x,y)
      if (small random probability) then child ← MUTATE(child)
      add child to new_population
    population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN

function REPRODUCE(x, y) returns an individual
inputs: x,y, parent individuals
  n ← LENGTH(x);
  c ← random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c),SUBSTRING(y, c + 1, n))
```

Illustration

- Representation of Individuals
  - Each state, or individual, is represented as a string over a finite alphabet – most commonly, a string of 0s and 1s.
  - Alternatively, the state could be represented as 8 digits, each in the range from 1 to 8.

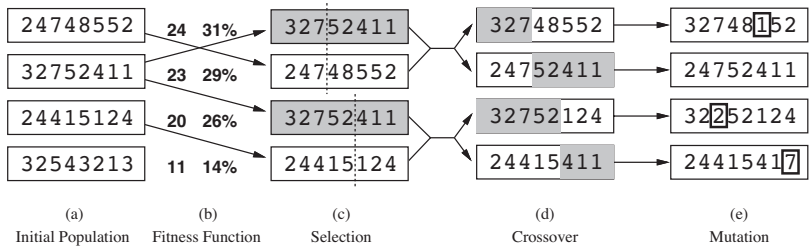


Figure 3: The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

Illustration (cont.)

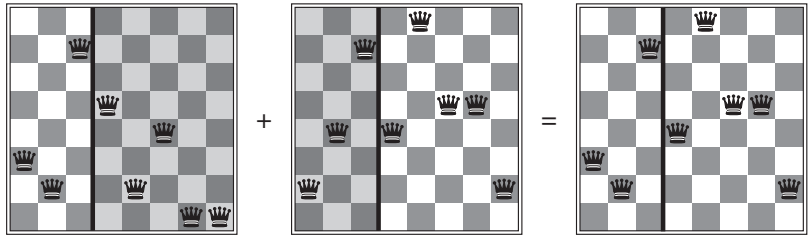
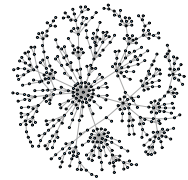


Figure 4: The 8-queens states corresponding to the first two parents in Figure 3(c) and the first offspring in Figure 3(d). The shaded columns are lost in the crossover step and the unshaded columns are retained.

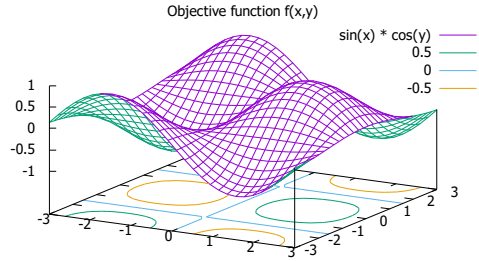
Local Search in Continuous Spaces

Optimization of Continuous Functions

- Discrete environments
  - Use hill-climbing

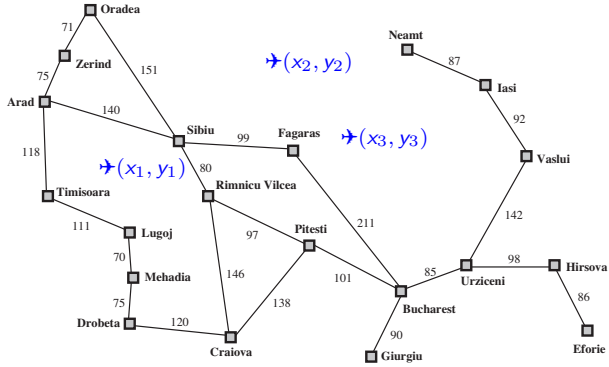


- Continuous environments
  - Use gradient descent



Example

- Suppose we want to site three airports in Romania:
  - 6-D state space defined by (x_1, y_2), (x_2, y_2), (x_3, y_3)
  - Objective function $f(\boldsymbol{x}) = f(x_1, y_2, x_2, y_2, x_3, y_3)$ = sum of squared distances from each city to nearest airport



Gradient descent

- **Discretization** methods turn continuous space into discrete space
  - E.g., **empirical gradient** considers $\pm\delta$ change in each coordinate
- **Gradient descent** methods using

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial y_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial y_3} \right)$$

- Init $\boldsymbol{x} \leftarrow \boldsymbol{x}_0$
- To reduce f, update $\boldsymbol{x}$

$$\boldsymbol{x} \leftarrow \boldsymbol{x} - \eta \nabla f$$

where η is learning rate $(0 < \eta < 1)$

Gradient descent (cont.)

- Newton–Raphson methods update $\boldsymbol{x}$

$$\boldsymbol{x} \leftarrow \boldsymbol{x} - \eta H^{-1} \nabla f$$

where H is a Hesian matrix

$$H = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1 \partial x_1} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_n} \end{pmatrix}$$

Gradient descent (cont.)

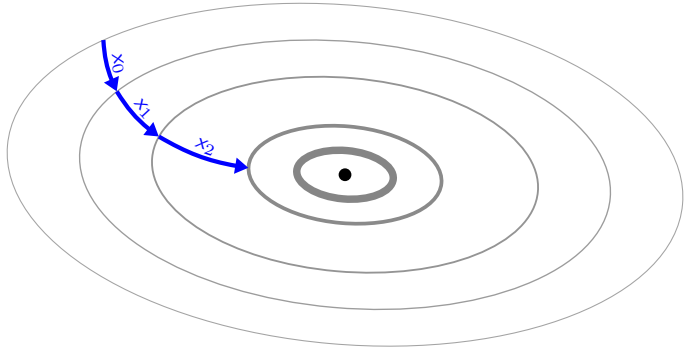


Figure 5: x_i approach to the optimal point

Searching With Nondeterministic Actions

The erratic vacuum world

In the **erratic vacuum world**, the *Suck* action works as follows:

- When applied to a dirty square the action cleans the square and sometimes cleans up dirt in an adjacent square, too.
- When applied to a clean square the action sometimes deposits dirt on the carpet.

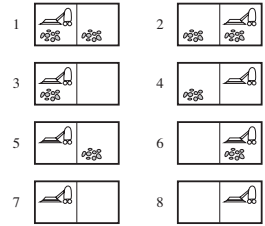


Figure 6: The eight possible states of the vacuum world; states 7 and 8 are goal states.

AND–OR search trees

- In this problem, a RESULTS function that returns a *set* of possible outcome states
- The solution to the problem is not a sequence but a **contingency plan** (also known as a **strategy**) that specifies what to do depending on what percepts are received

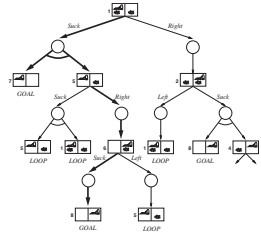


Figure 7: The first two levels of the search tree for the erratic vacuum world. State nodes are OR nodes where some action must be chosen. At the AND nodes, shown as circles, every outcome must be handled, as indicated by the arc linking the outgoing branches. The solution found is shown in bold lines.

Alogrithm

```
function AND-OR-GRAPH-SEARCH(problem) returns a conditional plan, or failure
  OR-SEARCH(problem.INITIAL-STATE, problem, ∅)

function OR-SEARCH(state, problem, path) returns a conditional plan, or failure
  if problem.GOAL-TEST(state) then return the empty plan
  if state is on path then return failure
  for each action in problem.ACTIONS(state) do
    plan ← AND-SEARCH(RESULTS(state, action), problem, [state | path])
    if plan ≠ failure then return [action | plan]
  return failure

function AND-SEARCH(states, problem, path) returns a conditional plan, or failure
  for each s_i in states do
    plan_i ← OR-SEARCH(s_i, problem, path)
    if plan_i = failure then return failure
  return [if s_1 then plan_1
          else if s_2 then plan_2
          else if ...
          else if s_{n-1} then plan_{n-1}
          else plan_n]
```

References

Goodfellow, I., Bengio, Y., and Courville, A. (2016).
Deep learning.
MIT press.

Lê, B. and Tô, V. (2014).
Cở sở trí tuệ nhân tạo.
Nhà xuất bản Khoa học và Kỹ thuật.

Nguyen, T. (2018).
Artificial intelligence slides.
Technical report, HCMC University of Sciences.

Russell, S. and Norvig, P. (2016).
Artificial intelligence: a modern approach.
Pearson Education Limited.