Introduction to Artificial Intelligence

4: Informed (Heuristic) Searches

Luca Doria, KPH Mainz

Uninformed Searches

Informed/heuristic searches: exploit the properties of a specific problem for improving the search.

Uninformed search: fixed rule for expanding a node.

Informed search: evaluate if a node n has to be expanded calculating f(n)

The evaluation function f(n) is constructed as a cost estimate, so the node with the lowest evaluation is expanded first. The choice of **f** determines the search strategy.

Many algorithms include a heuristic function h(n) which estimates the cheapest path from n to the goal. The specification of **h** is the way we input our knowledge about the problem into the search algorithm.

Heuristics

- The mathematician **G. Polya** (1887–1985) introduced the term in the context of problem solving techniques.

- In AI it has two meanings:
  1) Heuristics are methods which turn out to be fast but often incomplete for problem-solving.
  2) Heuristics are methods that improve the search in the average case.

- In all cases heuristics are problem-specific.

- "heuristic": from the Greek "heuriskein" which means "to find", "to discover". (Remember the famous "Eureka!" of Archimedes).

Best-First Search

General search problem:

```
function TREE-SEARCH(problem) returns a solution, or failure
        initialize the frontier using the initial state of the problem
        loop do
                if the frontier is empty then return failure
                choose a leaf node and remove it from the frontier
                if the node contains a goal state the return corresponding solution
                expand the chosen node, adding the resulting nodes to the frontier.
```

Best-First search uses a single evaluation function f(n) for evaluating the cost of a node n. It is a specific instance of TREE-SEARCH.

On the basis of the value of **f**, the node is pushed into a priority queue.

If f(n) makes always correct choices, there is no need to search!

Greedy Best-First Search

- Idea: estimate if it is good to expand a node estimating its path-cost to the goal.

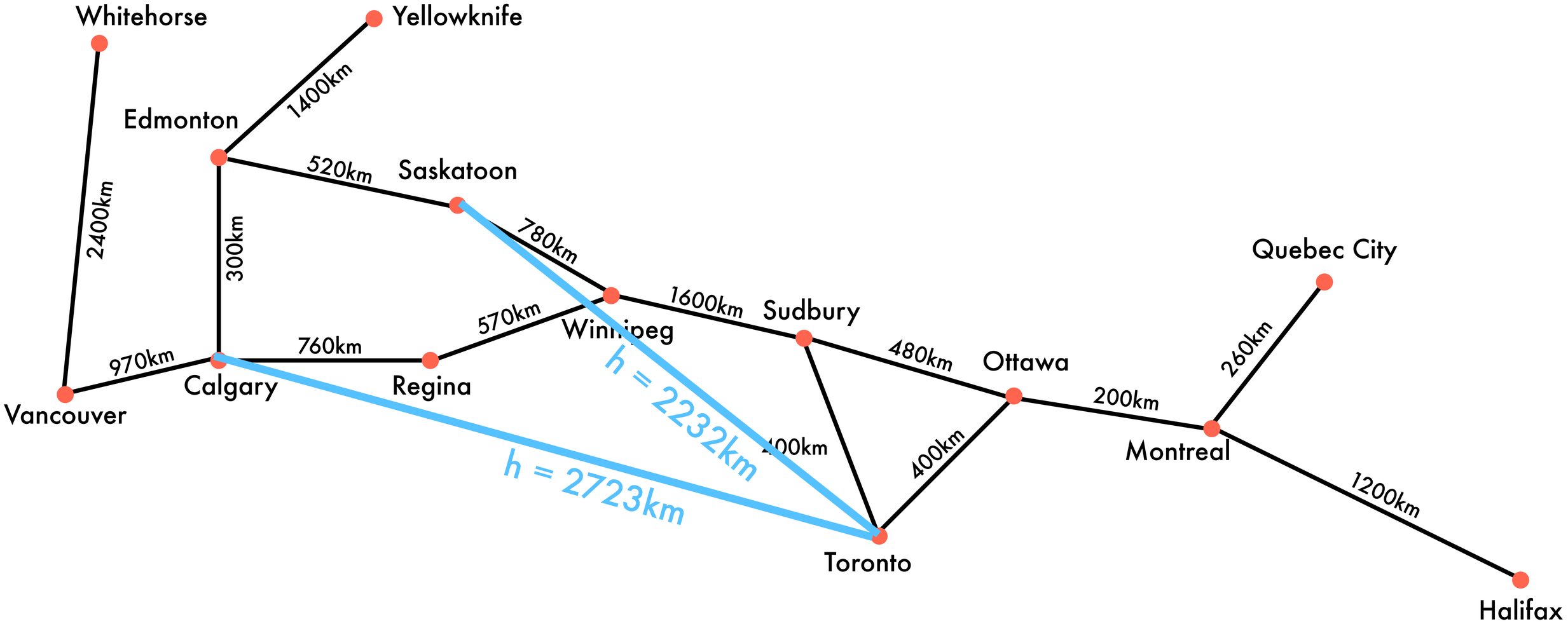
- Define the heuristic function h(n) = estimated path-cost from n to the goal.

- h(n)=0 if n is the goal.

- In a best-first search, f(n)=h(n) : greedy search.

- Example from the “Canada trip” problem: h(n)=straight-line distance from n to the goal.

Greedy Best-First Search



Greedy Best-First Search

- GBFS is incomplete in a sense very similar to Depth-First search (in the case of infinite state spaces).

- The “greediness” of the algorithm might drive the search into a dead-end.

- The worst-case time and space complexity is O(|V|)

- A good heuristic function (applied to the right problem) can strongly reduce the complexity of the average case, sometimes reaching O(bm).

- An example of improved heuristic function: **A*-Search**

A*-Search

Evaluation function:

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

estimated cost of the cheapest solution through n

estimated cost of the cheapest path from n to the goal

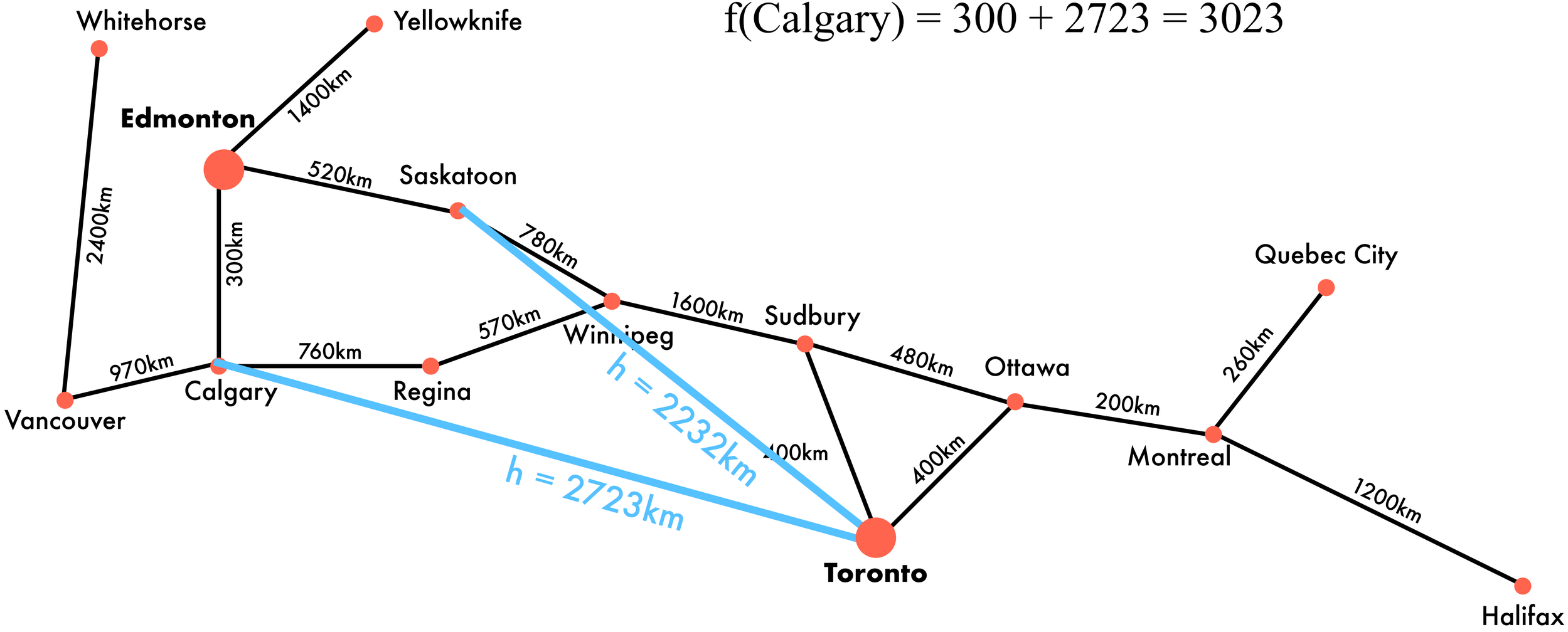
path-cost from the start-node to node n

- A*-Search can both be optimal and complete (under certain assumptions, see next)

A*-Search

f(Saskatoon) = 520 + 2232 = 2753

f(Calgary) = 300 + 2723 = 3023



A*-Search Admissibility

The **first** condition we require for optimality is that h(n) be an **admissible heuristic**.

An admissible heuristic is one that never overestimates the cost to reach the goal:

$$h(n) < h^*(n)$$

where h*(n) is the true minimal cost from n to the goal node. This is connected to the optimality of the solution.

In the example before, we took **h(n)** as the straight-line distance to the goal, which is always shorter or at most equal to the actual distance.

In underestimating the real cost, admissible heuristics can be regarded as “optimistic”.

If we instead overestimate the cost, this can “block” some ways which can contain the solution!

A*-Search Optimality

Proof of optimality (by contradiction, assuming admissibility):

Suppose the optimal path has cost C* but the algorithm returns C>C*.

Then a node n must be on the optimal path but it is unexpanded.

Denoting g*(n) the the cost of the optimal path from start to n,

and h*(n) the cost of the optimal path from n to the goal:

$f(n) = g(n) + h(n) > C^*$ otherwise n would have been expanded,

$f(n) = g^*(n) + h(n)$ because n is on the optimal path,

$f(n) \leq g^*(n) + h^*(n)$ because by admissibility, $h(n) \leq h^*(n)$

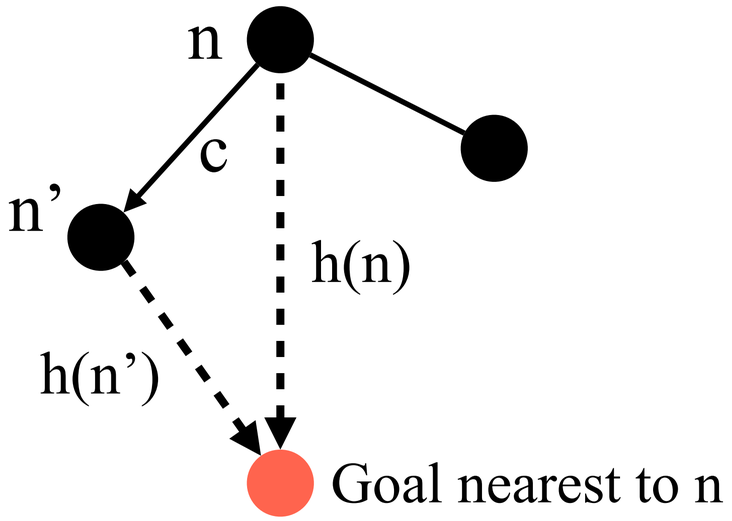
$f(n) \leq C^*$ since by definition $C^* = g^*(n) + h^*(n)$ —> contradiction.

Conclusion: A* returns only optimal paths.

A* Consistency

The **second** condition is **consistency** (or monotonicity) and applies only to graph searches.

A heuristic h(n) is consistent if, for every node n and every successor n′ of n generated by any action **a**, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n′ plus the estimated cost of reaching the goal from n′:



$$h(n) \leq c(n, a, n') + h(n')$$ Triangle inequality

Means also that the cost never decreases (monotonicity)

Consistency is a stricter requirement than admissibility:

Consistency ⇒ Admissibility

A* with a consistent heuristic is also cost-optimal (finds the absolute best solution)

Consistency implies optimality

If h(n) is consistent, then the values of f(n) along any path are nondecreasing

This property is called monotonicity and it is equivalent to consistency.

Proof:

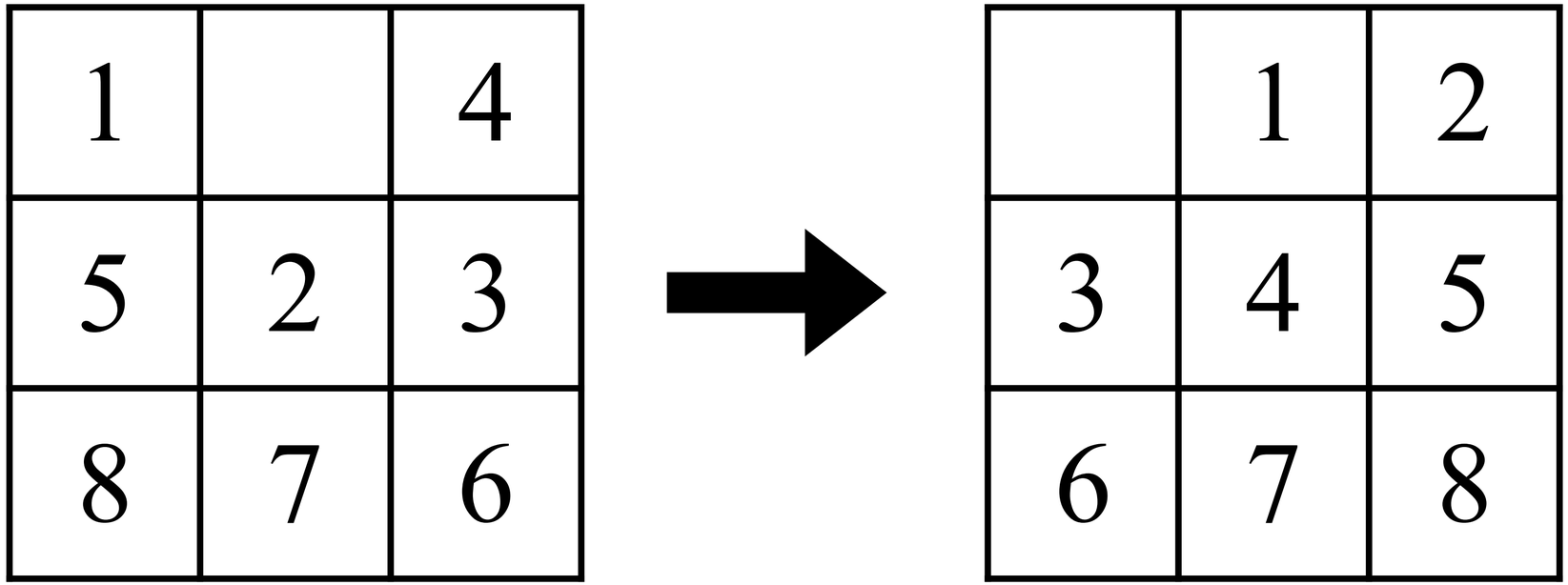
Suppose n′ is a successor of n; then g(n′) = g(n) + c(n, a, n′) for some action a, and we have

$$f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n)$$

This ensures:

- Once a node is expanded, we have already found its lowest-cost path.
- No need to re-expand nodes (efficient).
- A* will expand the goal node with the lowest cost first, ensuring optimality.

Example Heuristic Function: the 8-Puzzle



Slide the tiles from the current state to the goal state where tiles have increasing numbers.

Example Heuristic Function: the 8-Puzzle

The average moves for solving the puzzle is ~22

The branching factor of the search tree is ~3

An exhaustive search implies $3^{22} \sim 3 \times 10^{10}$ states on a tree.

The corresponding graph has 9!/2 = 181440(*) distinct states (For the 15-Puzzle, the states are $>10^9$!).

(*) Not all the configurations are reachable from one state. "Parity inversion" causes the factor /2 in the number of states.

For using A*-Search, we need a heuristic function that never overestimates the number of steps to the goal. Two commonly used heuristics are:

h_1 = # of misplaced tiles. Good because it is *admissible* (you do not misplace more…).

h_2 = Sum of all the Manhattan distances to the goal. This is also *admissible*, since you can only move one step closer to the goal.

Neither of the heuristics overestimates the true cost for reaching the solution.

Assessing the Heuristics Performance

Effective Branching Factor

If an A* heuristics generates a total number of nodes N in reaching the solution which is found at tree depth d, then the EBF b* is defined by

$$N + 1 = 1 + b^* + (b^*)^2 + (b^*)^3 + \ldots + (b^*)^d$$

which is the branching factor an uniform tree would have in order to contain N+1 nodes.

Nice property: b* is quite constant across problem instances, thus it can be estimated from a rather small amount of cases.

The better the heuristics, the closer is b* to 1 (almost a “direct” path to the solution).

Effective Depth

$O(b^d)$ —> $O(b^{d-k})$: the heuristics has the effect to reduce the tree depth by a factor b^k.

8-Puzzle Heuristics Performance

Search Cost

Effective Branching Factor

d	BFS	A*(h1)	A*(h2)	BFS	A*(h1)	A*(h2)
6	128	24	19	2.01	1.42	1.34
8	368	48	31	1.91	1.40	1.30
10	1033	116	48	1.85	1.43	1.27
12	2672	279	84	1.80	1.45	1.28
14	6783	678	174	1.77	1.47	1.31
16	17270	1683	364	1.74	1.48	1.32
18	41558	4102	751	1.72	1.49	1.34
20	91493	9905	1318	1.69	1.50	1.34
22	175921	22955	2548	1.66	1.50	1.34
24	290082	53039	5733	1.62	1.50	1.36
26	395355	110372	10080	1.58	1.50	1.35
28	463234	202565	22055	1.53	1.49	1.36

Which heuristic is better?

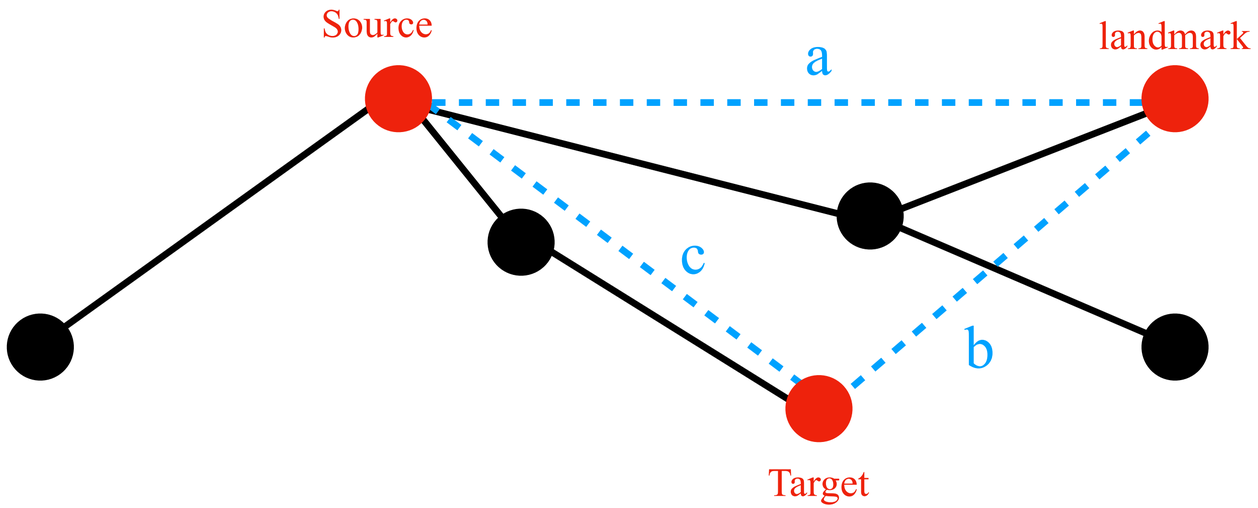
Average over 100 games.

Another Heuristics: Search with Landmarks

Some nodes of a search graph are visited more often, they are a sort of “hub” or “landmark”, using a map terminology.

Idea: pre-calculate all the distances of all the nodes from the landmarks and store them in a table.

When a minimum-cost path is required, we exploit the knowledge of the distances from the closest landmark and make an estimation using the triangular inequality.



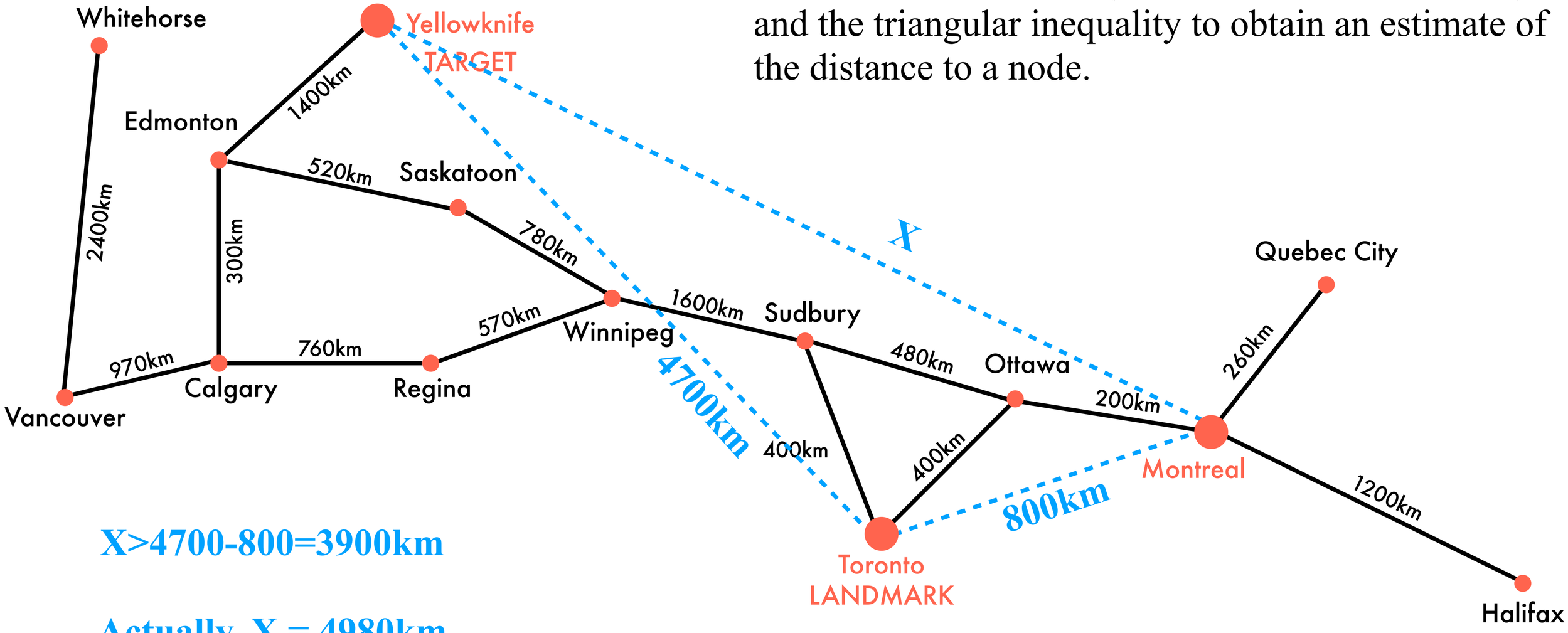
c+a>b

c+b>a

c > a-b (a>b)

Another Heuristics: Search with Landmarks

We use what we know (distances from a landmark) and the triangular inequality to obtain an estimate of the distance to a node.



X>4700-800=3900km

Actually, X = 4980km

Another Heuristics: Search with Landmarks

When the algorithm explores a node, it looks at its neighbours to add them to the frontier. This is done evaluating the cost function f(n) = g(n) + h(n) to decide the order of which nodes to explore. To estimate the cost it creates an abstract triangle between the node, the target, and the landmark with pre-calculated distances.

The triangular inequality is used to estimate the distance that should be at least covered.

We draw an abstract triangle, but the sides are (were) calculated following the graph.

Landmarks encode the graph “topography” and use it as a distance estimator.

Further variants of A* search

A* in general needs exponential memory amounts.

In order to take care of this problem, some variants have been suggested:

- Iterative-deepening A*: uses depth-first search with iteratively increasing depth : IDA*

- Recursive Best First Search (RBFS): introduces a variable f-limit to keep track of the best alternative path available from any ancestor of the current node. If current node exceeds this limit, recursion unwinds back to the alternative path.

- other alternatives:
  - memory-bounded A* (MA*)
  - simplified MA* (SMA*). …. …. … …..

Summary so far

Uninformed Search

Access only to problem definition
Build a search tree for finding the solution

Best-First search: eval.function for expanding nodes
Breadth-First search:
shallowest nodes first, complete, optimal, Exp(Space)
Uniform-Cost search: can be optimal
Depth-first search:
deepest node expanded first, not complete,
not optimal, but Lin(Space).
…

Informed Search

Heuristic function h(n) estimating the cost of a solution from the node n.

Greedy best-first search
A* search
…

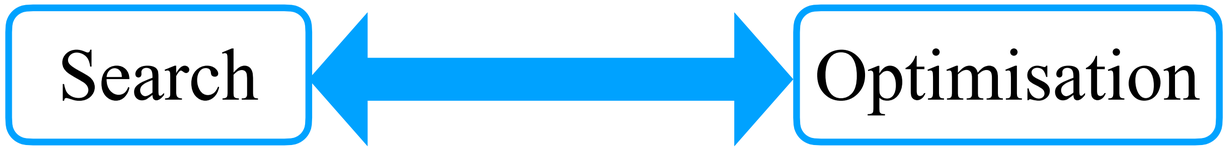
Searching in more complex spaces

Up to now the search was performed in:

- Fully observable (we have access to the full structure of the space)
- Deterministic (no randomness)
- Static (nothing in the space changes during the search)

Now we are going to abandon some of these features.

In doing so we will exploit the connection among two problems:



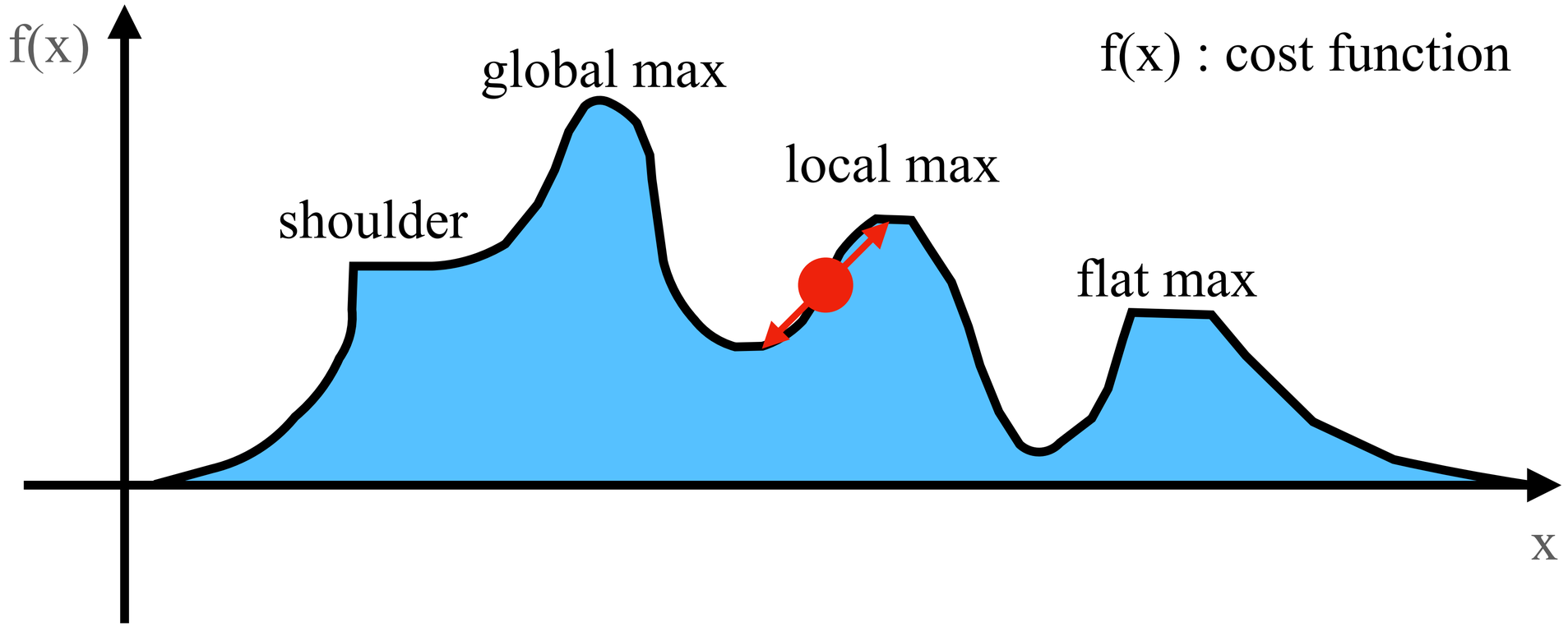
Local Search

In problems like the “map” one, often we would like to find not only a solution, but the best one (e.g. the shortest path).
In some other problems, we do not need to keep track of our path, since only reaching the solution is relevant.

One class of such algorithms is the local search one.
In local searches, the algorithm has only a partial (local) knowledge of the search space.
In some cases, indeed only the local knowledge is available but the key advantages of these algorithms are:

- the limited memory use
- the ability to explore very large spaces and find reasonable solutions.

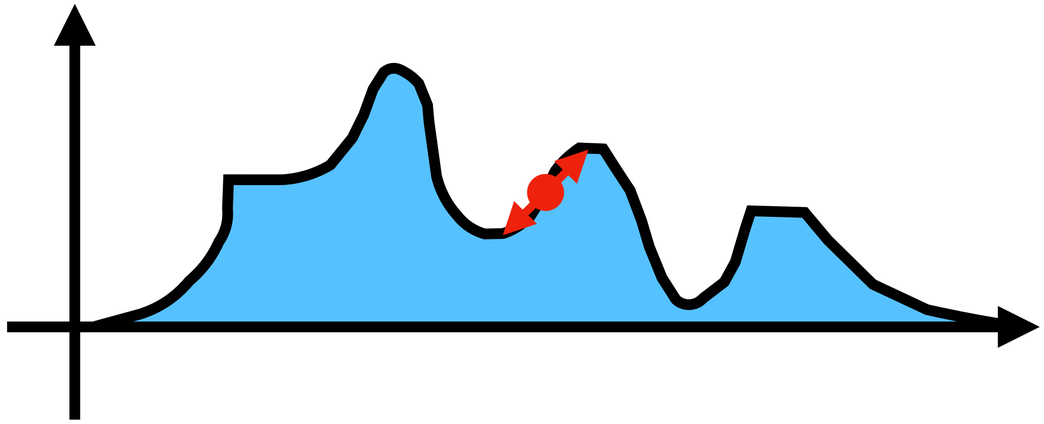
Local Search



Example: 1-dimensional search space.
Red circle: current value of f. The algorithm does not know the global shape of the function but only the "neighbourhood" and based on this local information must decide where to "move" at the next step.

Hill-Climbing Search

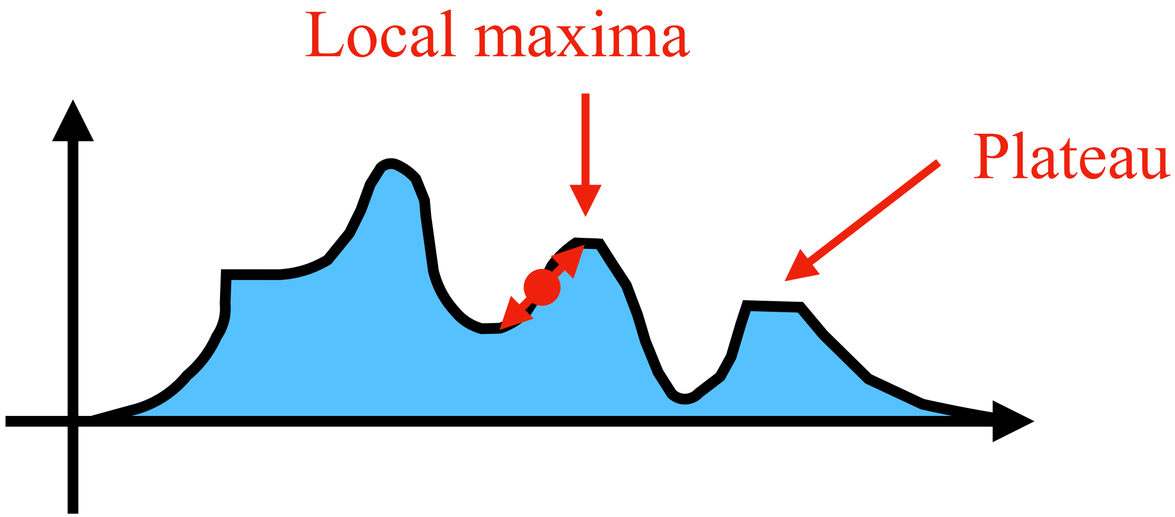
Simplest local search algorithms: move where f increases.
In numerical maths terms, this is equivalent to a gradient descend (ascend) algorithm and belongs to the family of greedy algorithms.



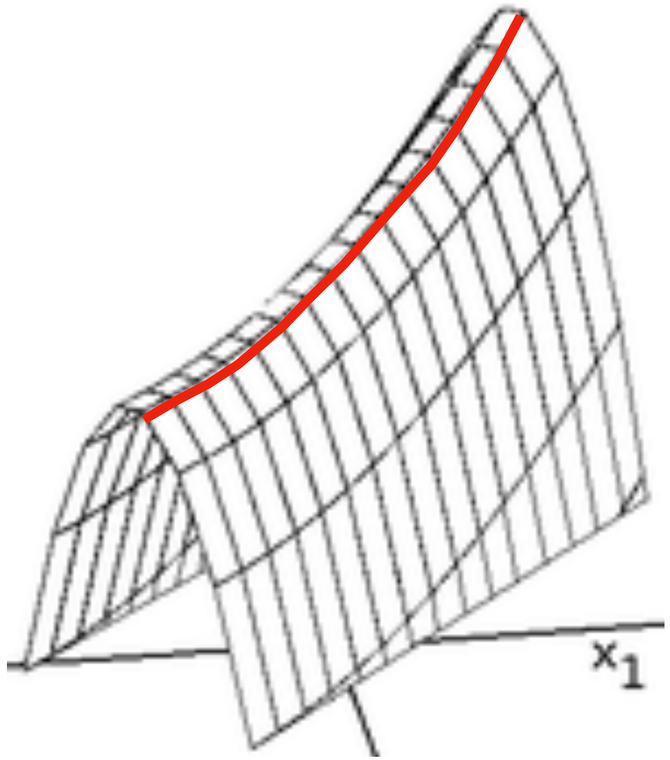
```
function HILL-CLIMBING(problem) returns loc. max. state
    current <— problem.INITIAL
    while true do
        neighbour <— highest value successor state of current
        if VALUE(neighbor) <= VALUE(current) then return current
        current <— neighbor
```

Hill-Climbing Search

Potential problems of the Hill-Climbing search:



Ridges:



adapted from Wikipedia

Hill-Climbing Variants

In order to overcome some of the previous problems, variate of the HCA were proposed:

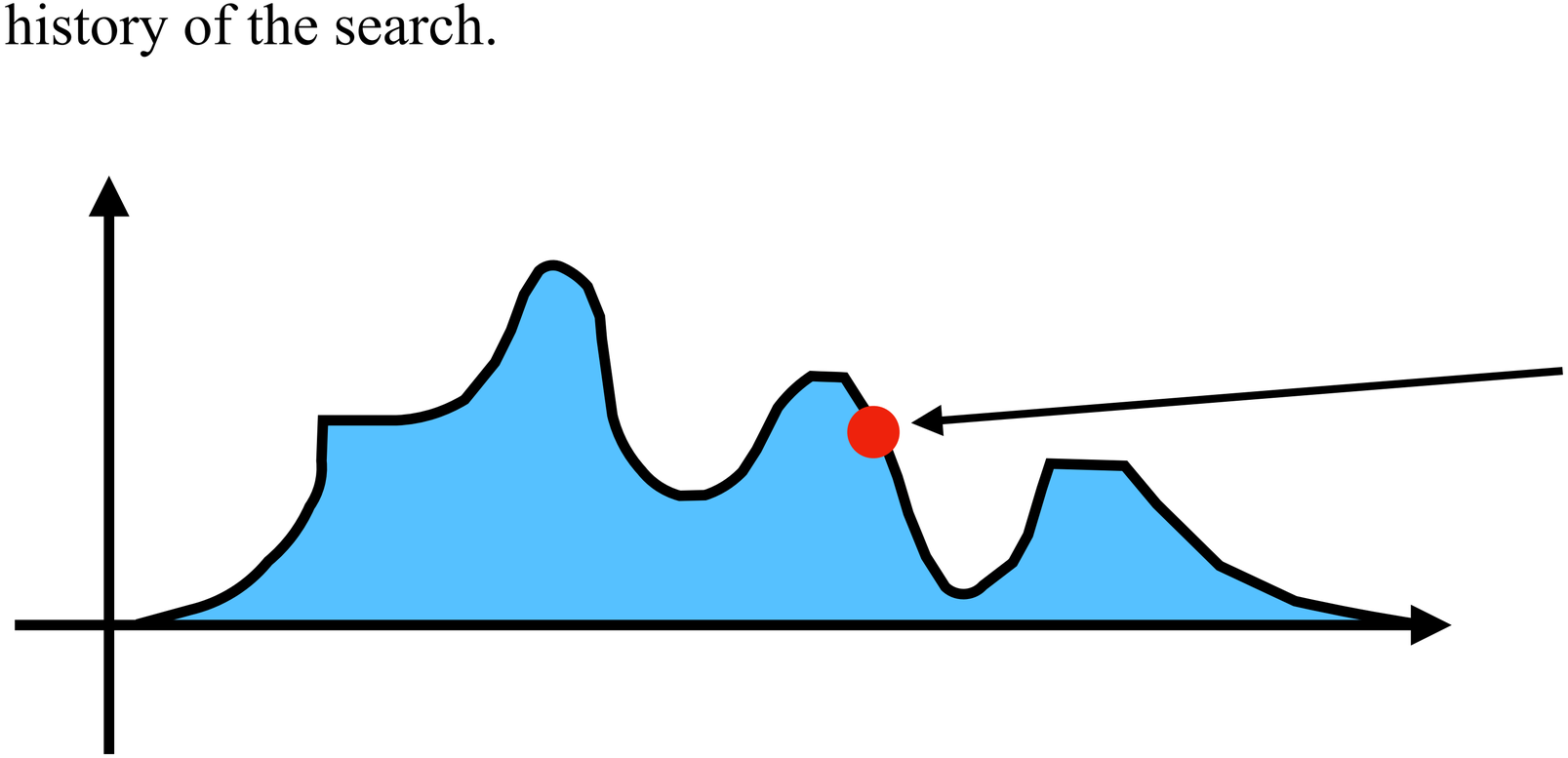
Stochastic Hill Climbing:
Among the maximising moves, it chooses one at random.

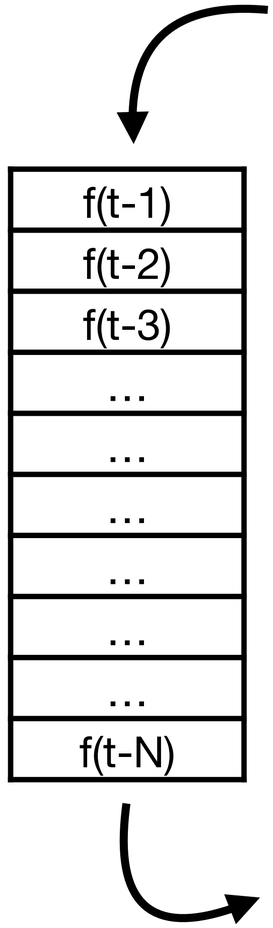
First-choice Hill Climbing:
Generates random moves until a better than the current state is found.

Random-restart Hill Climbing:
Applies HCA N-times, each time starting from a randomly chosen position.

Tabu Search

The (greedy) tabu search algorithm combines a hill-climbing search strategy with a heuristics to avoid the stops at suboptimal points (local maxima/minima) and the occurrence of “cycles”. This is obtained by using a list of forbidden moves (the tabu moves) derived from the recent history of the search.





Memory (FIFO) of “tabu”moves with fixed length N

Reactive Tabu Search

RTS is a variant of RT where the length of the tabu list is not constant but it is "reactively" changed depending from the status of the search. In particular, a "prohibition time" T is assigned to each move.

How to change?

- A too short T could not allow escaping local minima.
- A too long T could prevent reaching some solutions.

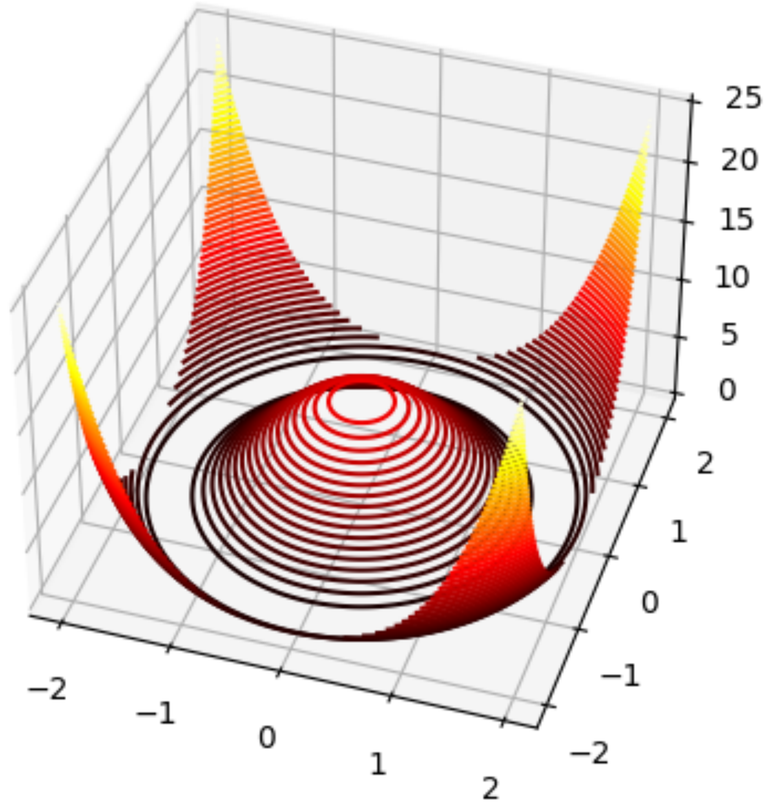
General working of the algorithm:

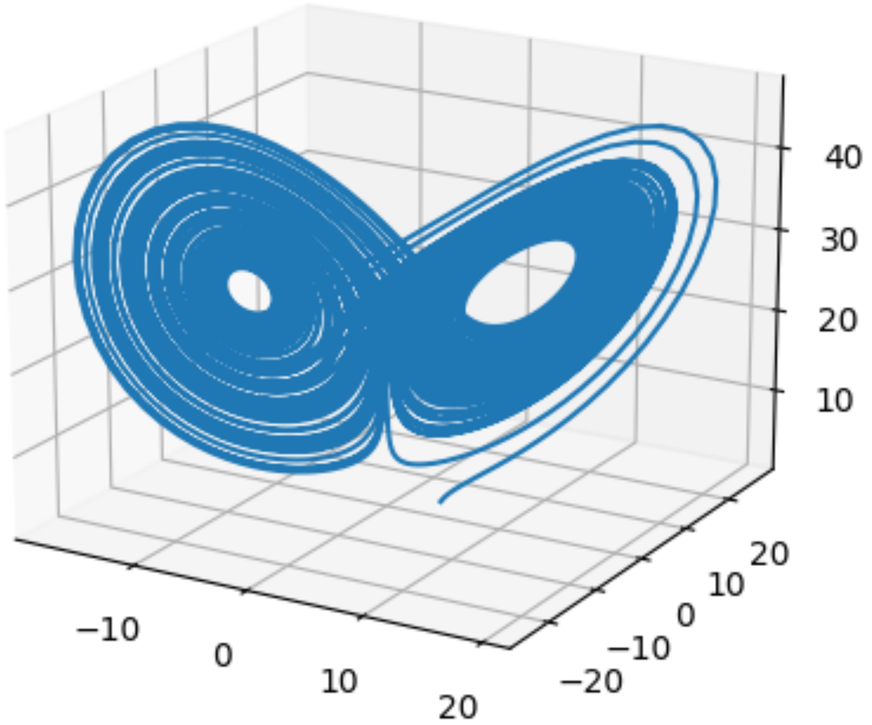
- Starts with a certain probation time T (time = # of moves).
- If a move belongs to a series of moves which keep repeating (a cycle) and T is already at its maximum (a specified parameter Tmax), then a sequence of random jumps is performed for escaping the cycle. The random steps are included in the tabu list.
- If a region was already visited, T is increased.
- If T was not increased since a while (according to a parameter), then T is reduced.

Reactive Tabu Search

Why RTS was proposed:

Limit cycles can be in principle avoided by a tabu search.
If the prohibition time is too short, you can get stuck in a cycle.
Other situation: chaotic orbits. The search is stuck in a region of space even without repetitions.





Simulated Annealing (3000 years-old idea!)

Physics-inspired algorithm: heat a material to high temperatures and then slowly cool it ("temper"). In this way, atoms/molecules relax in their lowest energy state.

Credit: HanBon Forge

Ideas:

1) If a move is maximising, accept the move
2) If not, with exponentially low probability, accept the move even if not maximising.

```
function SIMULATED-ANNEALING(problem,schedule) returns a solution state
current ← problem.INITIAL
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                                  1)
    else current ← next only with probability e^(ΔE/T.)           2)
```

Evolutionary ("Genetic") Algorithms

Background of the heuristics:

- Theory of evolution (Charles Darwin, 1859)
- Probabilistic laws of evolution (Gregor Medel, 1866)
- Discovery of the DNA (Watson, Crick, 1953)

Notes:

1) We are going to use the ideas of evolution treating strings as DNA strands which mutate, cross and copy. This is only a partial analogy with biology: in reality things are more complex. Genes in DNA not only contain information but also processes, which our strings do not.

2) There are many variants of evolutionary/genetic algorithms. We will take a look at one of them.

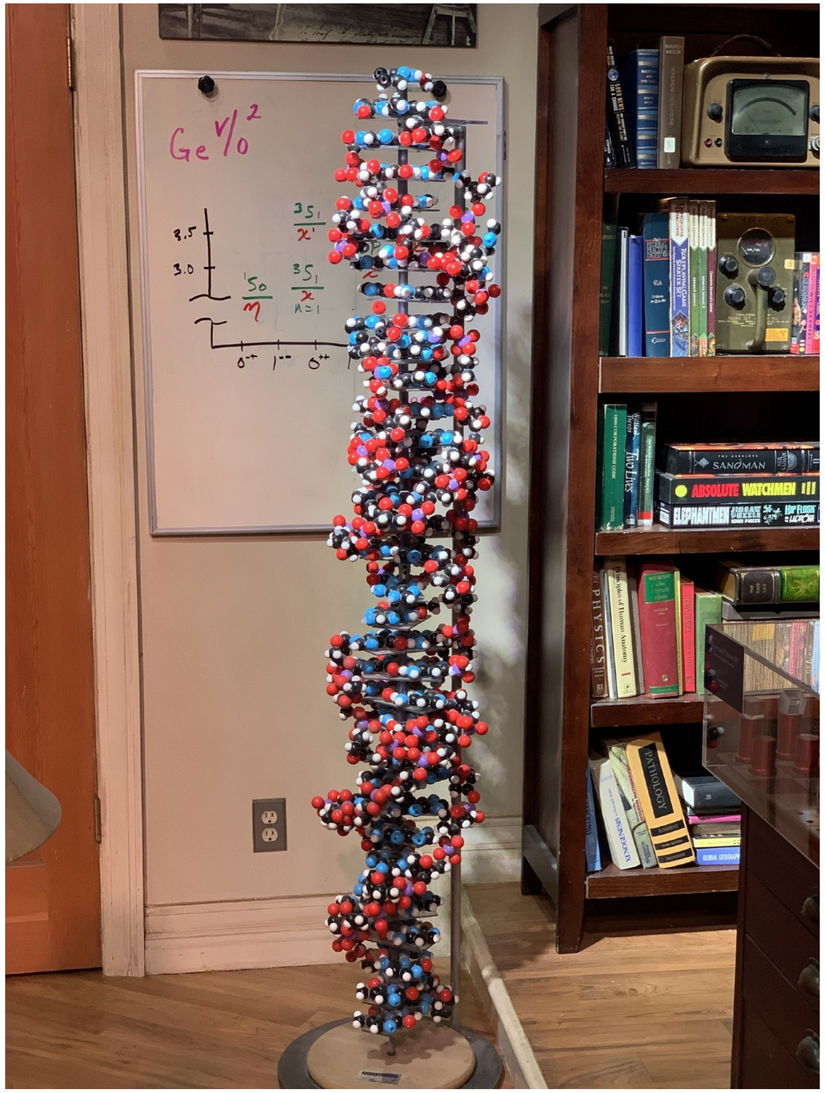
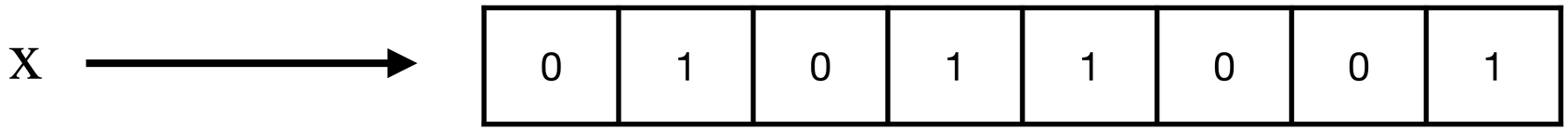


Photo from CBS

Evolutionary Algorithms

Encoding:

In general, we have an optimisation task where we look for the maximum (minimum) of a function f(x). We need to encode x in a string analogous to the ACGT strings in the DNA.
A common choice is Boolean encoding:



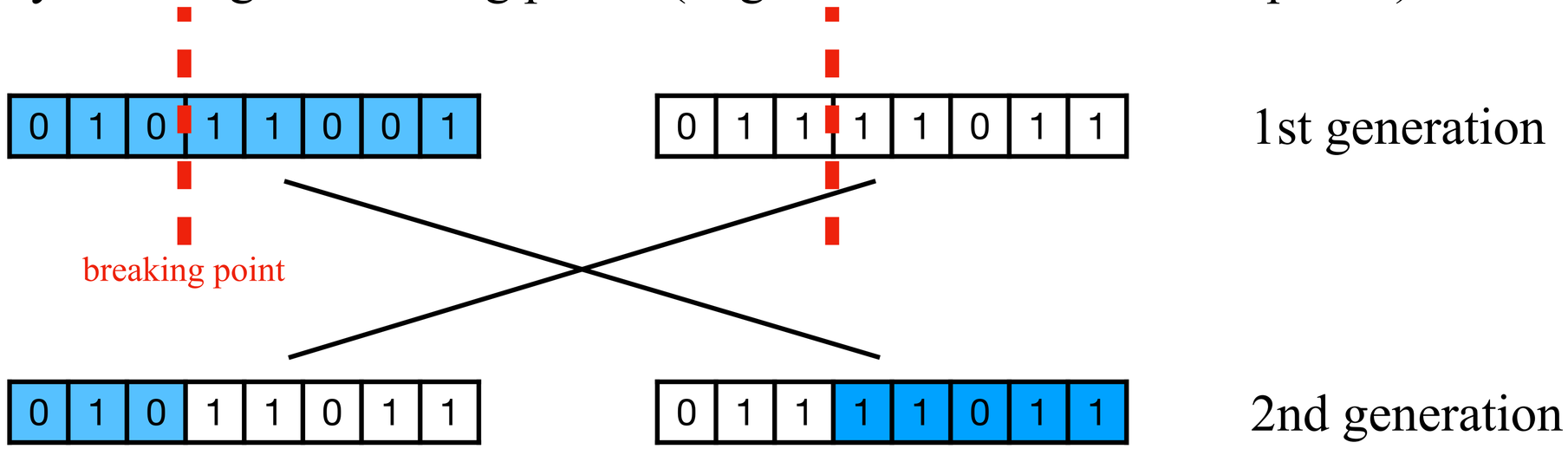
Fitness function:

Every “individual” of the species (the strings) are evaluated according to a fitness function which ranks them. The general idea is that “the fittest survives”.
The fitness function is exactly the function we would like to optimise, while the individuals are candidate solutions (function values).

Evolutionary Algorithms

Crossover:

The individuals exchange genetic material (sub-strings), giving rise to a new “generation”.
This is done randomly selecting a “breaking point” (in general there could be N points).



Mutation:

Cross-over might not be enough for discover some parts of the search space. Again, in analogy with nature, we can flip a (or more) random bit(s) of the string with some probability.

Evolutionary Algorithms

Sketch of the algorithm:

1) Define an initial population of strings
2) Evaluate the strings with the fitness function (find the “fittest” solution)
3) Apply cross-over, randomly generating the location of the crossing point
4) Apply random mutations with a certain probability (usually low)
5) Go to step 2 or stop if maximum # of iterations is reached or an acceptable solution is found.

Pros & Cons:

- Genetic algorithms are quite aggressive on the solution space, since they probe it at many places at the same time. A lot of solutions are generated and local minima can be avoided.
- Easily parallelized, gradient-less.
- Computational complexity grows fast for both, time and space.
- Convergence can be slow.
- Require fine-tuning of the parameters (population #, mutation probability, …)

Searching in Continuous Spaces

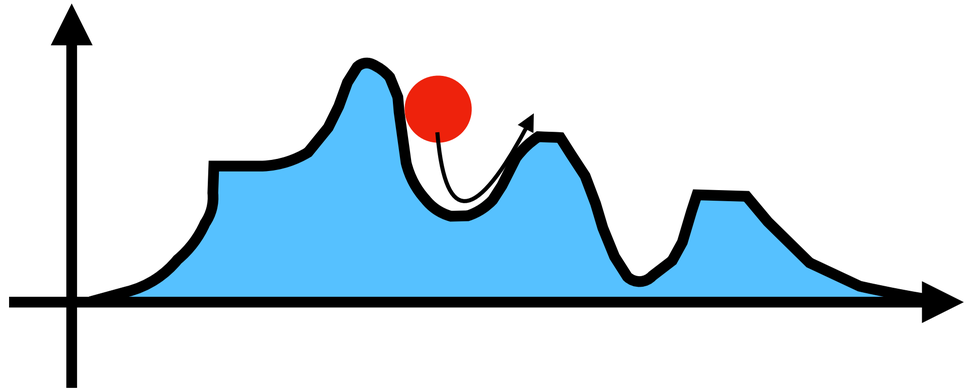
Optimisation in continuous spaces has a very long history and the starting point has been the calculation of gradients: maxima and minima correspond to points where $\nabla f(x) = 0$.

In the case we cannot calculate explicitly the derivative of the function, many variants of the gradient descent algorithm exist where the space is discretised and derivatives are approximated numerically:

$$\bar{x}_{i+1} = \bar{x}_i + \alpha \nabla f(\bar{x}_i)$$

More advanced algorithms were developed for avoiding local minima. A simple extension is the addition of the momentum term:

$$\bar{x}_{i+1} = \bar{x}_i + \alpha \nabla f(\bar{x}_i) - \beta(\bar{x}_t - \bar{x}_{t-1})$$



Other stochastic/adaptive variants exist (more on this later with machine learning).

Summary

- Heuristic algorithms incorporate “good ideas” for improving (mostly) searches.
- Best-first search expands the node with the highest score (defined by some measure) first.
- With the minimization of the evaluated costs to the goal we obtain a greedy search.
- The minimization of f(n) = g(n)+h(n) combines uniform and greedy searches.
- When h(n) is admissible (*i.e.* h* is never overestimated) we obtain the A* search, which is complete and optimal.
- Other methods: tabu search, annealing, genetic algorithms, ….