CHAPTERS 3–4: MORE SEARCH ALGORITHMS

DIT410/TIN174, Artificial Intelligence

Peter Ljunglöf

28 March, 2017

TABLE OF CONTENTS

HEURISTIC SEARCH (R&N 3.5–3.6)

GREEDY BEST-FIRST SEARCH (3.5.1)

A* SEARCH (3.5.2)

ADMISSIBLE AND CONSISTENT HEURISTICS (3.6–3.6.2)

THE GENERIC TREE SEARCH ALGORITHM

Tree search: Don’t check if nodes are visited multiple times

function Search(*graph*, *initialState*, *goalState*):
    initialise *frontier* using the *initialState*
    while *frontier* is not empty:
        select and **remove** *node* from *frontier*
        if *node*.state is a *goalState* **then return** *node*
        for each *child* **in** ExpandChildNodes(*node*, *graph*):
            add *child* to *frontier*
    return failure

DEPTH-FIRST AND BREADTH-FIRST SEARCH

THESE ARE THE TWO BASIC SEARCH ALGORITHMS

Depth-first search (DFS)

- implement the frontier as a Stack
- space complexity: $O(bm)$
- incomplete: might fall into an infinite loop, doesn't return optimal solution

Breadth-first search (BFS)

- implement the frontier as a Queue
- space complexity: $O(b^m)$
- complete: always finds a solution, if there is one
- (when edge costs are constant, BFS is also optimal)

COST-BASED SEARCH

IMPLEMENT THE FRONTIER AS A PRIORITY QUEUE, ORDERED BY $\mathbf{f(n)}$

Uniform-cost search (this is not a heuristic algorithm)

- expand the node with the lowest path cost
- $f(n) = g(n)$
- complete and optimal

Greedy best-first search

- expand the node which is closest to the goal (according to some heuristics)
- $f(n) = h(n)$
- incomplete: might fall into an infinite loop, doesn't return optimal solution

A* search

- expand the node which has the lowest estimated cost from start to goal
- $f(n) = g(n) + h(n)$ = estimated cost of the cheapest solution through n
- complete and optimal (if $h(n)$ is admissible/consistent)

A* TREE SEARCH IS OPTIMAL!

A* always finds an optimal solution first, provided that:

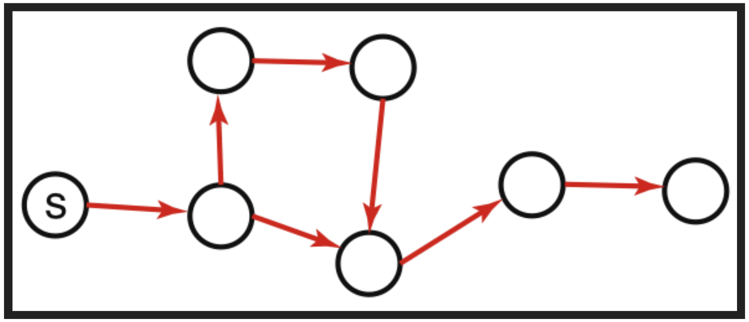
- the branching factor is finite,
- arc costs are *bounded above zero* (i.e., there is some $\epsilon > 0$ such that all of the arc costs are greater than ϵ), and
- $h(n)$ is ***admissible***
- i.e., $h(n)$ is *nonnegative* and an *underestimate* of the cost of the shortest path from n to a goal node.

THE GENERIC GRAPH SEARCH ALGORITHM

Tree search: Don't check if nodes are visited multiple times
Graph search: Keep track of visited nodes

function Search(*graph*, *initialState*, *goalState*):
    initialise *frontier* using the *initialState*
    initialise *exploredSet* to the empty set
    while *frontier* is not empty:
        select and **remove** *node* from *frontier*
        if *node*.state is a *goalState* **then return** *node*
        add *node* to *exploredSet*
        for each *child* **in** ExpandChildNodes(*node*, *graph*):
            if *child* is not in *frontier* or *exploredSet*:
                add *child* to *frontier*
    return failure

GRAPH-SEARCH = MULTIPLE-PATH PRUNING



Graph search keeps track of visited nodes, so we don’t visit the same node twice.

- Suppose that the first time we visit a node is not via the most optimal path

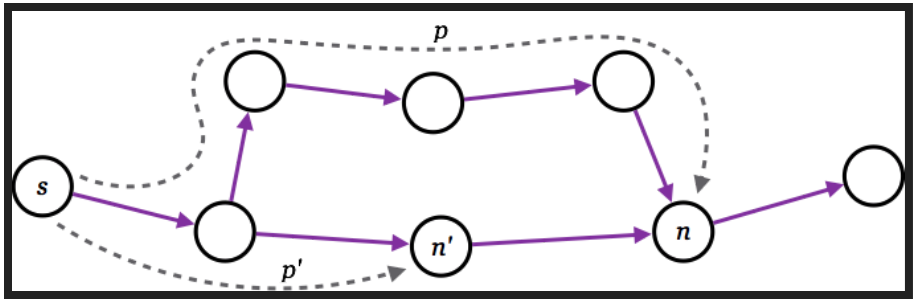
  $\Rightarrow$ then graph search will return a suboptimal path

- Under which circumstances can we guarantee that A* graph search is optimal?

WHEN IS A* GRAPH SEARCH OPTIMAL?

If h is *consistent*, then A* graph search is optimal:

- Consistency is defined as: $h(n') \leq cost(n', n) + h(n)$ for all arcs (n', n)
- **Lemma**: the f values along any path $[\ldots, n', n, \ldots]$ are nondecreasing:
    - **Proof**: $g(n) = g(n') + cost(n', n)$, therefore:
    - $f(n) = g(n) + h(n) = g(n') + cost(n', n) + h(n) \geq g(n') + h(n')$;
    - therefore: $f(n) \geq f(n')$, i.e., f is nondecreasing
- **Theorem**: whenever A* expands a node n, the optimal path to n has been found
    - **Proof**: Assume this is not true;
    - then there must be some n' still on the frontier, which is on the optimal path to n;
    - but $f(n') \leq f(n)$;
    - and then n' must already have been expanded $\Longrightarrow$ *contradiction*!



STATE-SPACE CONTOURS

The f values in A* are nondecreasing, therefore:

first	A* expands all nodes with $f(n) < C$
then	A* expands all nodes with $f(n) = C$
finally	A* expands all nodes with $f(n) > C$

A* will not expand any nodes with $f(n) > C*$, where $C*$ is the cost of an optimal solution.

SUMMARY OF OPTIMALITY OF A*

A* *tree search* is optimal if:

- the heuristic function $h(n)$ is **admissible**
- i.e., $h(n)$ is nonnegative and an underestimate of the actual cost
- i.e., $h(n) \leq cost(n, goal)$, for all nodes n

A* *graph search* is optimal if:

- the heuristic function $h(n)$ is **consistent** (or monotone)
- i.e., $|h(m) - h(n)| \leq cost(m, n)$, for all arcs (m, n)

SUMMARY OF TREE SEARCH STRATEGIES

Search strategy	Frontier selection	Halts if solution?	Halts if no solution?	Space usage
Depth first	Last node added	*No*	*No*	*Linear*
Breadth first	First node added	*Yes*	*No*	*Exp*
Greedy best first	Minimal $h(n)$	*No*	*No*	*Exp*
Uniform cost	Minimal $g(n)$	*Optimal*	*No*	*Exp*
A*	$f(n) = g(n) + h(n)$	*Optimal**	*No*	*Exp*

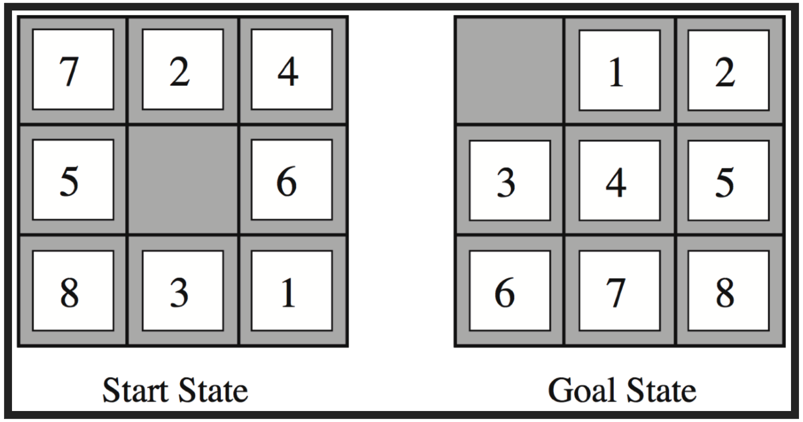
**Provided that $h(n)$ is admissible.*

Halts if: If there is a path to a goal, it can find one, even on infinite graphs.
Halts if no: Even if there is no solution, it will halt on a finite graph (with cycles).
Space: Space complexity as a function of the length of the current path.

RECAPITULATION: HEURISTICS FOR THE 8 PUZZLE

$h_1(n)$ = number of misplaced tiles

$h_2(n)$ = total Manhattan distance
(i.e., no. of squares from desired location of each tile)



$h_1(StartState)$ = *8*

$h_2(StartState)$ = *3+1+2+2+2+3+3+2 = 18*

DOMINATING HEURISTICS

If (admissible) $h_2(n) \geq h_1(n)$ for all n,
then h_2 **dominates** h_1 and is better for search.

Typical search costs (for 8-puzzle):

depth = 14	DFS ≈ 3,000,000 nodes
	A*(h_1) = 539 nodes
	A*(h_2) = 113 nodes
depth = 24	DFS ≈ 54,000,000,000 nodes
	A*(h_1) = 39,135 nodes
	A*(h_2) = 1,641 nodes

Given any admissible heuristics h_a, h_b, the **maximum** heuristics $h(n)$ is also admissible and dominates both:

$$h(n) = \max(h_a(n), h_b(n))$$

HEURISTICS FROM A RELAXED PROBLEM

Admissible heuristics can be derived from the exact solution cost of a relaxed problem:

- If the rules of the 8-puzzle are relaxed so that a tile can move anywhere, then $h_1(n)$ gives the shortest solution
- If the rules are relaxed so that a tile can move to any adjacent square, then $h_2(n)$ gives the shortest solution

Key point: the optimal solution cost of a relaxed problem is never greater than the optimal solution cost of the real problem

NON-ADMISSIBLE (NON-CONSISTENT) A* SEARCH

A* search with admissible (consistent) heuristics is optimal

But what happens if the heuristics is non-admissible?

- i.e., what if $h(n) > c(n, goal)$, for some n?
- the solution is not guaranteed to be optimal…
- …but it will find *some* solution!

Why would we want to use a non-admissible heuristics?

- sometimes it's easier to come up with a heuristics that is almost admissible
- and, often, the search terminates faster!

EXAMPLE DEMO

Here is an example demo of several different search algorithms, including A*. Furthermore you can play with different heuristics:

http://qiao.github.io/PathFinding.js/visual/

Note that this demo is tailor-made for planar grids, which is a special case of all possible search graphs.

MORE SEARCH STRATEGIES (R&N 3.4–3.5)

ITERATIVE DEEPENING (3.4.4–3.4.5)

BIDIRECTIONAL SEARCH (3.4.6)

MEMORY-BOUNDED HEURISTIC SEARCH (3.5.3)

ITERATIVE DEEPENING

BFS is guaranteed to halt but uses exponential space.
DFS uses linear space, but is not guaranteed to halt.

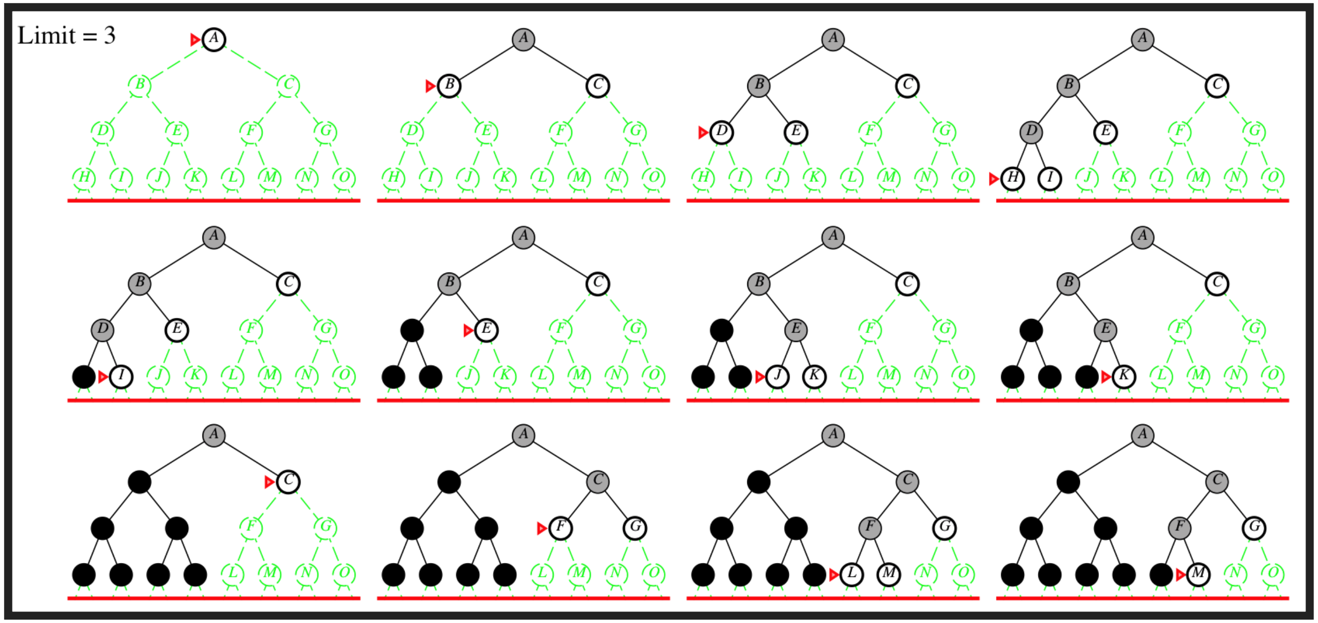
Idea: take the best from BFS and DFS — recompute elements of the frontier rather than saving them.

- Look for paths of depth 0, then 1, then 2, then 3, etc.
- Depth-bounded DFS can do this in linear space.

Iterative deepening search calls depth-bounded DFS with increasing bounds:

- If a path cannot be found at *depth-bound*, look for a path at *depth-bound* + 1.
- Increase *depth-bound* when the search fails unnaturally (i.e., if *depth-bound* was reached).

ITERATIVE DEEPENING EXAMPLE



Depth bound = *3*

ITERATIVE-DEEPENING SEARCH

function IDSearch(*graph*, *initialState*, *goalState*)
    for *limit* **in** 0, 1, 2, …:
        result := DepthLimitedSearch([*initialState*], *limit*)
        if *result* ≠ cutoff **then return** *result*

function DepthLimitedSearch([$n_0, \ldots, n_k$], *limit*):
    if n_k is a *goalState* **then return** path [$n_0, \ldots, n_k$]
    else if *limit* = 0 **then return** cutoff
    else:
        failureType := failure
        for each *neighbor* n of n_k:
            result := DepthLimitedSearch([$n_0, \ldots, n_k, n$], *limit*–1)
            if *result* is a path **then return** *result*
            else if *result* = cutoff **then** *failureType* := cutoff
        return *failureType*

ITERATIVE DEEPENING COMPLEXITY

Complexity with solution at depth k and branching factor b:

level	breadth-first	iterative deepening	# nodes
1	1	k	b
2	1	$k-1$	b^2
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$k-1$	1	2	b^{k-1}
k	1	1	b^k
total	$\geq b^k$	$\leq b^k \left(\frac{b}{b-1}\right)^2$	

Numerical comparison for $k = 5$ and $b = 10$:

BFS = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,110
IDS = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450

Note: IDS recalculates shallow nodes several times, but this doesn't have a big effect compared to BFS!

BIDIRECTIONAL SEARCH (3.4.6)

DIRECTION OF SEARCH

The definition of searching is symmetric: find path from start nodes to goal node or from goal node to start nodes.

Forward branching factor: number of arcs going out from a node.

Backward branching factor: number of arcs going into a node.

Search complexity is $O(b^n)$. Therefore, we should use forward search if forward branching factor is less than backward branching factor, and vice versa.

Note: when a graph is dynamically constructed, the backwards graph may not be available.

BIDIRECTIONAL SEARCH

Idea: search backward from the goal and forward from the start simultaneously.

- This can result in an exponential saving, because $2b^{k/2} \ll b^k$.
- The main problem is making sure the frontiers meet.

One possible implementation:

- Use BFS to gradually search backwards from the goal, building a set of locations that will lead to the goal.
    - this can be done using *dynamic programming*
- Interleave this with forward heuristic search (e.g., A*) that tries to find a path to these interesting locations.

DYNAMIC PROGRAMMING

Idea: for statically stored graphs, build a table of the actual distance $dist(n)$, of the shortest path from node n to a goal.

- This can be built backwards from the goal:

$$\begin{aligned} dist(n) \quad &= \quad \textbf{if } isGoal(n) \textbf{ then } 0 \\ & \qquad \textbf{else } \min_{(n,m)\in G}(|(n,m)| + dist(m)) \end{aligned}$$

The calculation of $dist$ can be interleaved with a forward heuristic search.

MEMORY-BOUNDED A* (3.5.3)

The biggest problem with A* is the space usage.
Can we make an iterative deepening version?

- IDA*: use the f value as the cutoff cost
  - the cutoff is the smalles f value that exceeded the previous cutoff
  - often useful for problems with unit step costs
  - **problem**: with real-valued costs, it risks regenerating too many nodes
- RBFS: recursive best-first search
  - similar to DFS, but continues along a path until $f(n) > limit$
  - $limit$ is the f value of the best *alternative path* from an ancestor
  - if $f(n) > limit$, recursion unwinds to alternative path
  - **problem**: regenerates too many nodes
- SMA* and MA*: (simplified) memory-bounded A*
  - uses all available memory
  - when memory is full, it drops the worst leaf node from the frontier

LOCAL SEARCH (R&N 4.1)

HILL CLIMBING (4.1.1–4.1.2)

POPULATION-BASED METHODS (4.1.3–4.1.4)

ITERATIVE BEST IMPROVEMENT

In many optimization problems, the path is irrelevant

- the goal state itself is the solution

Then the state space can be the set of “complete” configurations

- e.g., for 8-queens, a configuration can be any board with 8 queens (it is irrelevant in which order the queens are added)

In such cases, we can use *iterative improvement* algorithms;
we keep a single “current” state, and try to improve it

- e.g., for 8-queens, we start with 8 queens on the board, and gradually move some queen to a better place

The goal would be to find an optimal configuration

- e.g., for 8-queens, where no queen is threatened

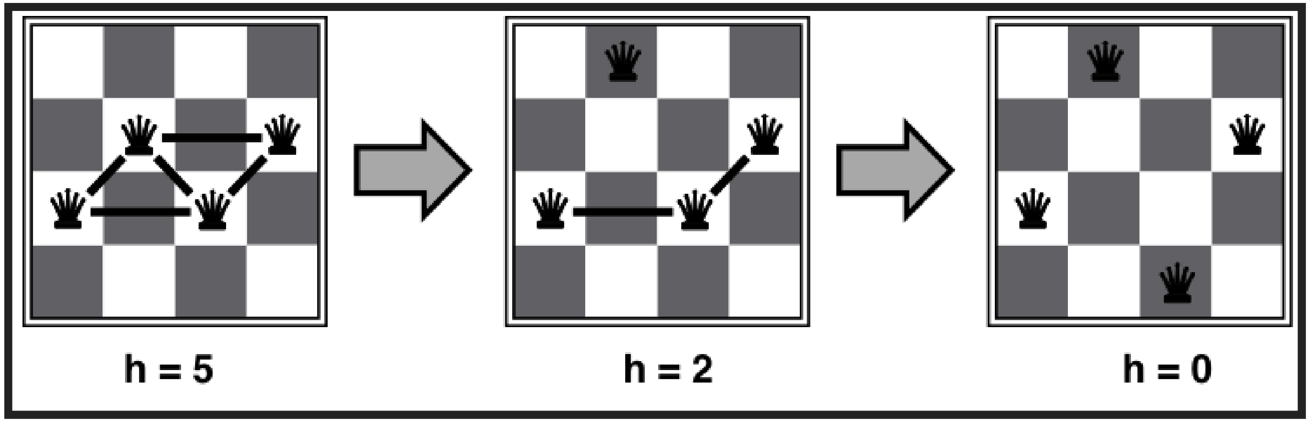
This takes constant space, and is suitable for online and offline search

EXAMPLE: n-QUEENS

Put n queens on an $n \times n$ board, in separate columns

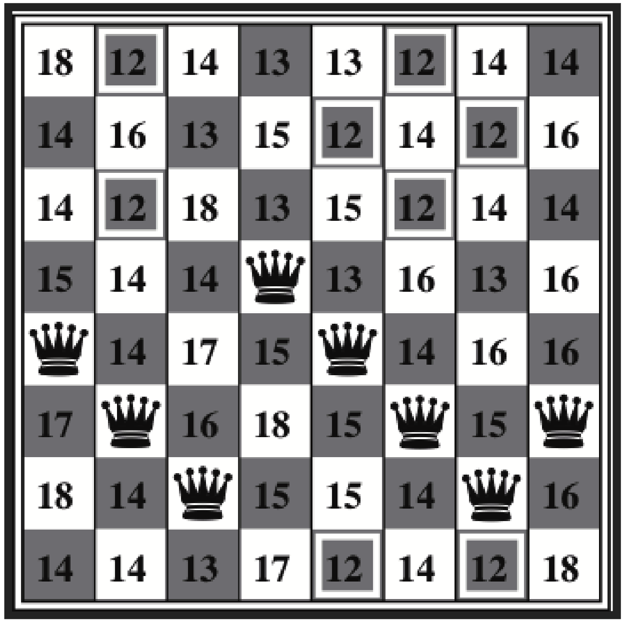
Move a queen to reduce the number of conflicts;
repeat until we cannot move any queen anymore

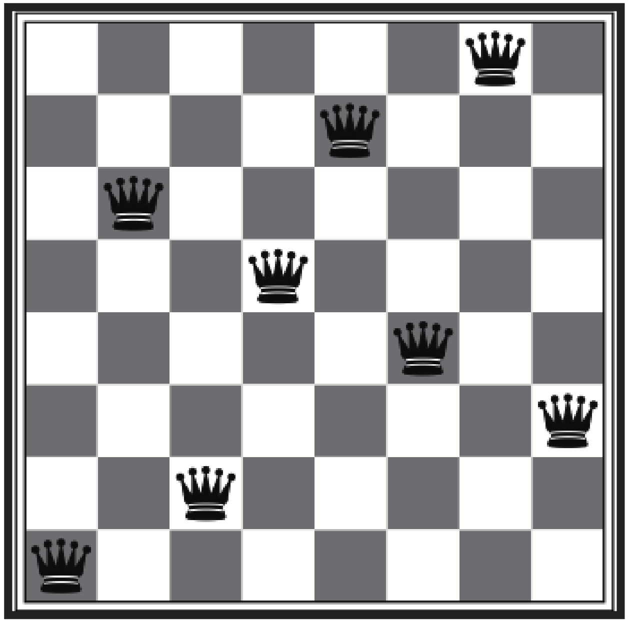
$\Rightarrow$ then we are at a local maximum, hopefully it is global too



This almost always solves n-queens problems
almost instantaneously for very large n (e.g., n = 1 million)

EXAMPLE: 8-QUEENS



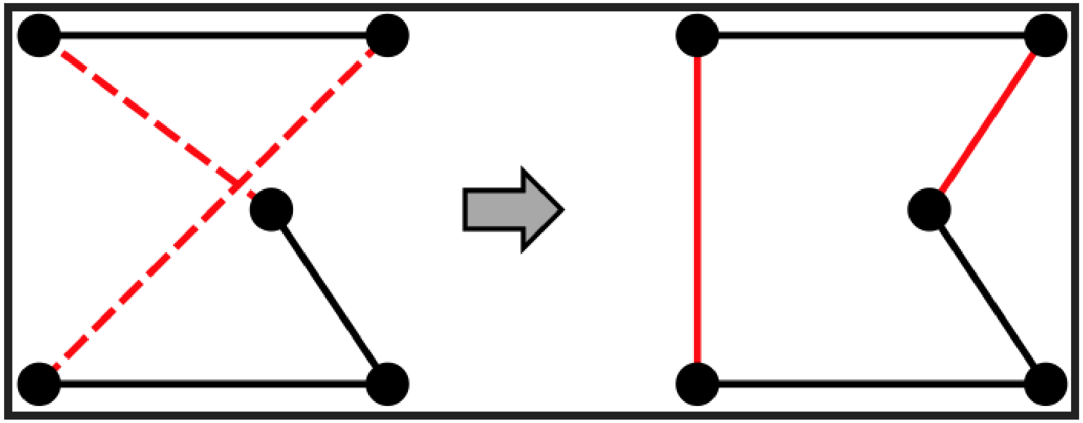


Move a queen within its column, choose the minimum n:o of conflicts

- the best moves are marked above (conflict value: 12)
- after 5 steps we reach a local minimum (conflict value: 1)

EXAMPLE: TRAVELLING SALESPERSON

Start with any complete tour, and perform pairwise exchanges



Variants of this approach get within 1% of optimal very quickly with thousands of cities

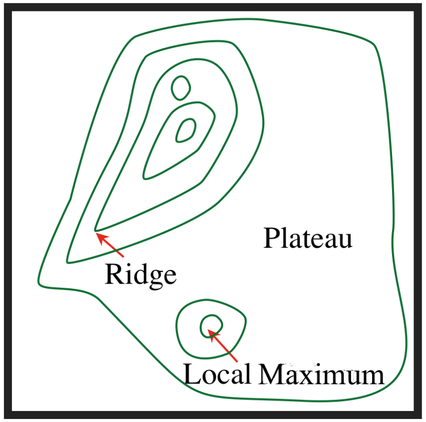
HILL CLIMBING SEARCH (4.1.1–4.1.2)

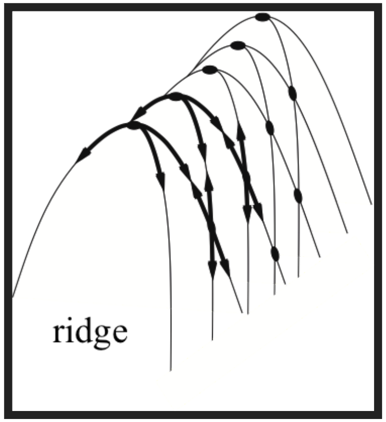
Also called gradient/steepest ascent/descent,
or greedy local search.

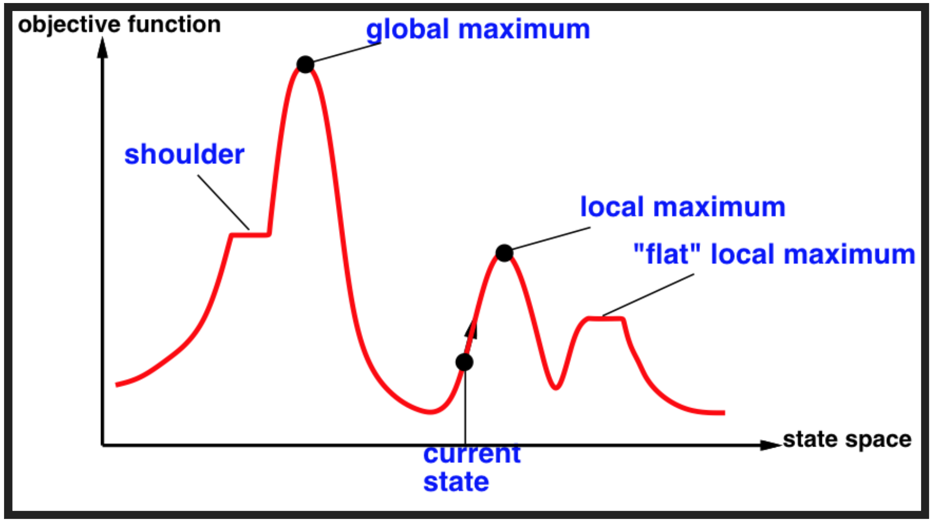
function HillClimbing(*graph*, *initialState*):
    current := *initialState*
    loop:
        neighbor := a highest-valued successor of *current*
        if *neighbor*.value ≤ *current*.value **then return** *current*
        current := *neighbor*

PROBLEMS WITH HILL CLIMBING

Local maxima — Ridges — Plateaux







RANDOMIZED ALGORITHMS

Consider two methods to find a minimum value:

- Greedy ascent: start from some position,
  keep moving upwards, and report maximum value found
- Pick values at random, and report maximum value found

Which do you expect to work better to find a global maximum?

Can a mix work better?

RANDOMIZED HILL CLIMBING

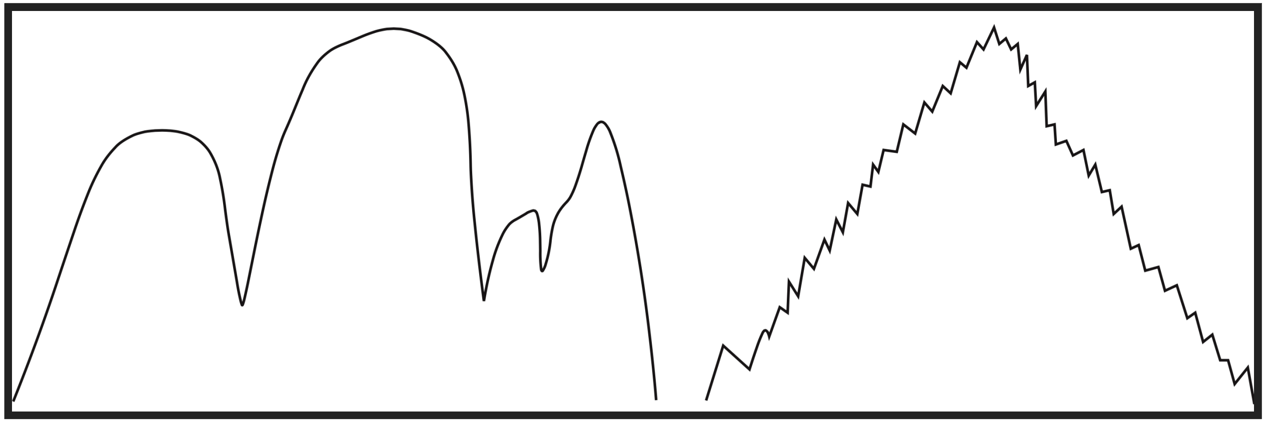
As well as upward steps we can allow for:

- *Random steps:* (sometimes) move to a random neighbor.
- *Random restart:* (sometimes) reassign random values to all variables.

Both variants can be combined!

1-DIMENSIONAL ILLUSTRATIVE EXAMPLE

Two 1-dimensional search spaces; you can step right or left:



Which method would most easily find the global maximum?

- random steps or random restarts?

What if we have hundreds or thousands of dimensions?

- …where different dimensions have different structure?

SIMULATED ANNEALING

Simulated annealing is an implementation of random steps:

```
function SimulatedAnnealing(problem, schedule):
    current := problem.initialState
    for t in 1, 2, …:
        T := schedule(t)
        if T = 0 then return current
        next := a randomly selected neighbor of current
        ΔE := next.value – current.value
        if ΔE > 0 or with probability e^(ΔE/T):
            current := next
```

T is the “cooling temperature”, which decreases slowly towards 0

The cooling speed is decided by the *schedule*

POPULATION-BASED METHODS (4.1.3–4.1.4)

LOCAL BEAM SEARCH

Idea: maintain a population of k states in parallel, instead of one.

- At every stage, choose the k best out of all of the neighbors.
  - when $k = 1$, it is normal hill climbing search
  - when $k = \infty$, it is breadth-first search
- The value of k lets us limit space and parallelism.
- *Note*: this is not the same as k searches run in parallel!
- *Problem*: quite often, all k states end up on the same local hill.

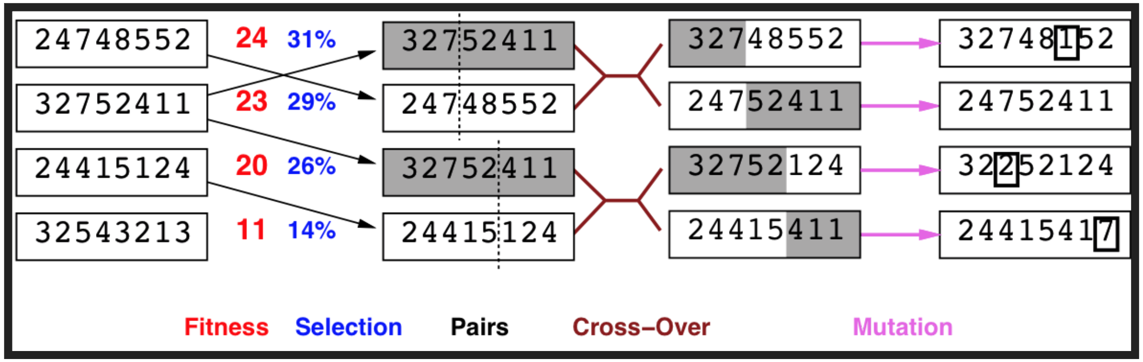
STOCHASTIC BEAM SEARCH

Similar to beam search, but it chooses the next k individuals *probabilistically*.

- The probability that a neighbor is chosen is proportional to its heuristic value.
- This maintains diversity amongst the individuals.
- The heuristic value reflects the fitness of the individual.
- Similar to natural selection:
  each individual mutates and the fittest ones survive.

GENETIC ALGORITHMS

Similar to stochastic beam search,
but *pairs* of individuals are combined to create the offspring.



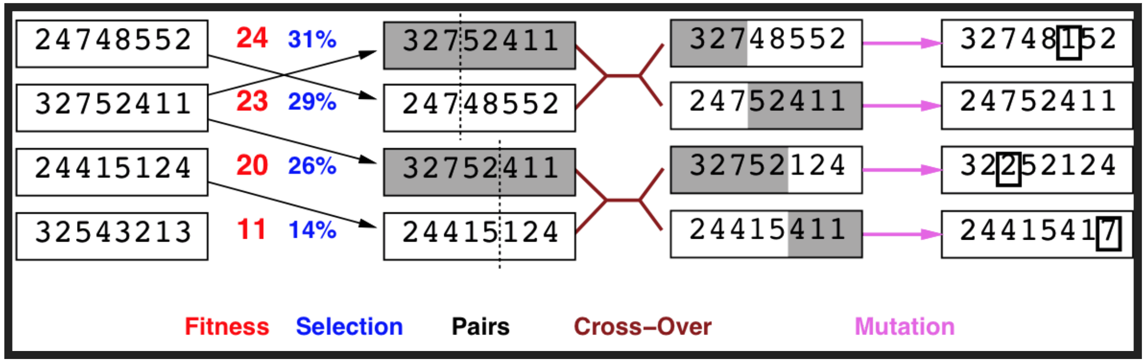
For each generation:

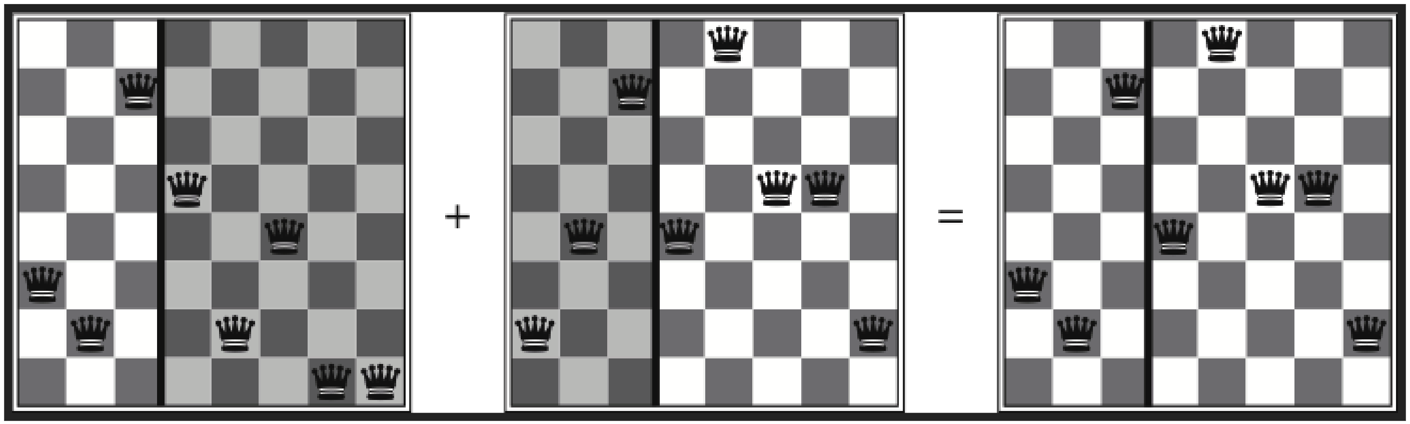
- Randomly choose pairs of individuals where
  the fittest individuals are more likely to be chosen.
- For each pair, perform a cross-over:
  form two offspring each taking different parts of their parents:
- Mutate some values.

Stop when a solution is found.

n QUEENS ENCODED AS A GENETIC ALGORITHM

The n queens problem can be encoded as n numbers $1 \ldots n$:





EVALUATING RANDOMIZED ALGORITHMS (NOT IN R&N)

How can you compare three algorithms A, B and C, when

- A solves the problem 30% of the time very quickly but doesn’t halt for the other 70% of the cases
- B solves 60% of the cases reasonably quickly but doesn’t solve the rest
- C solves the problem in 100% of the cases, but slowly?

Summary statistics, such as mean run time or median run time don’t make much sense.

RUNTIME DISTRIBUTION

Plots the runtime and the proportion of the runs that are solved within that runtime.