CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Computing@CIT

Metaheuristic Optimization

Local Search

Dr. Diarmuid Grimes

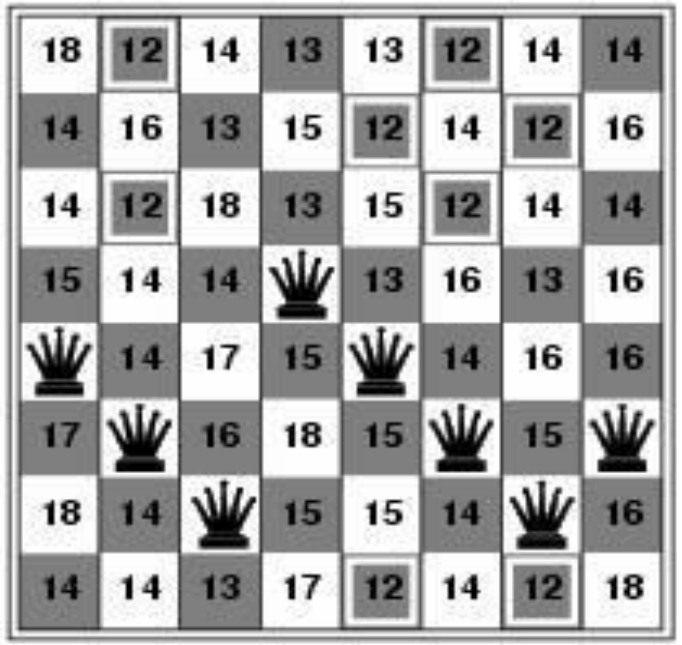
Example: n-queens

- Put n queens on an n x n board with no two queens on the same row, column, or diagonal

- First as always convert the satisfaction problem to optimization

- (Unlike most optimization problems that we will be applying metaheuristics to, in this case we know the optimal value!)

Hill-climbing search: 8-queens problem



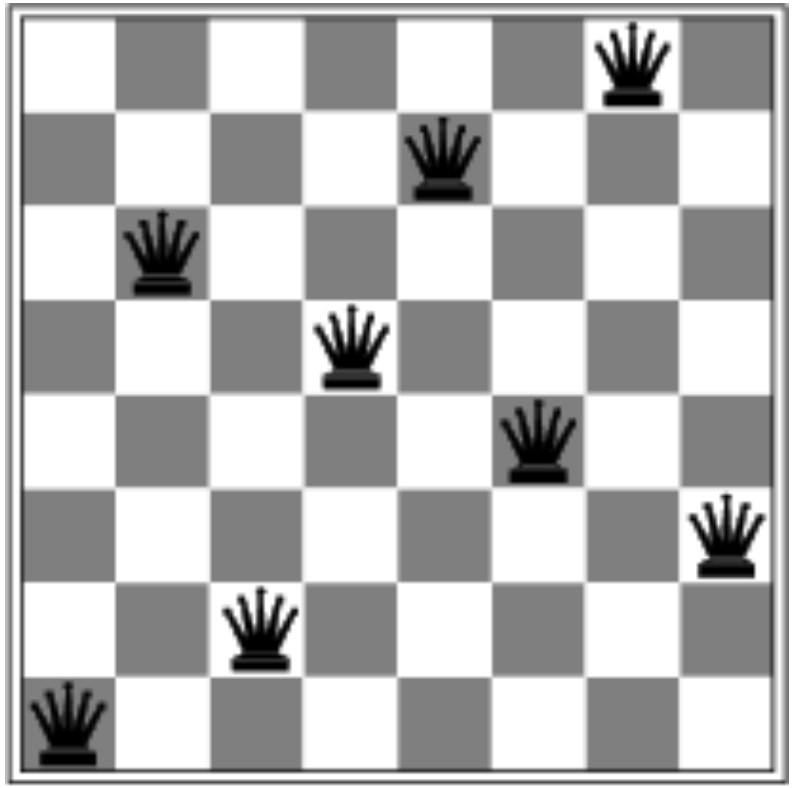
- h = number of pairs of queens that are attacking each other
- h = 17 for the above state

Search Space: 8-queens problem

- State
  - All 8 queens on the board in some configuration
- Successor function
  - move a single queen to another square in the same column.
- Example of a heuristic function h(n):
  - the number of pairs of queens that are attacking each other
  - (so we want to minimize this)

Hill-climbing search: 8-queens problem

- Is this a solution?
- What is h?

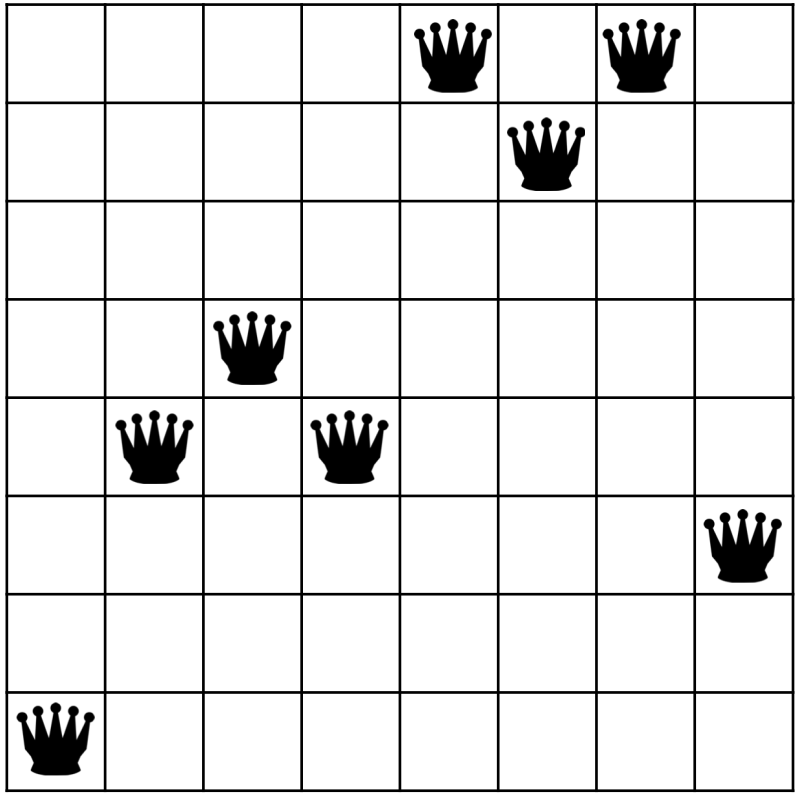


N-Queens Problem -> Satisfaction Problem

- Task: place n queens on the chess board such that they do not attack each other
- Representation: the row of the queen in each column
  - Therefore n variables each with domain of size n
- Local steps: move one of the queens in its column
- Local Search
  - start with infeasible configuration
  - move towards feasibility

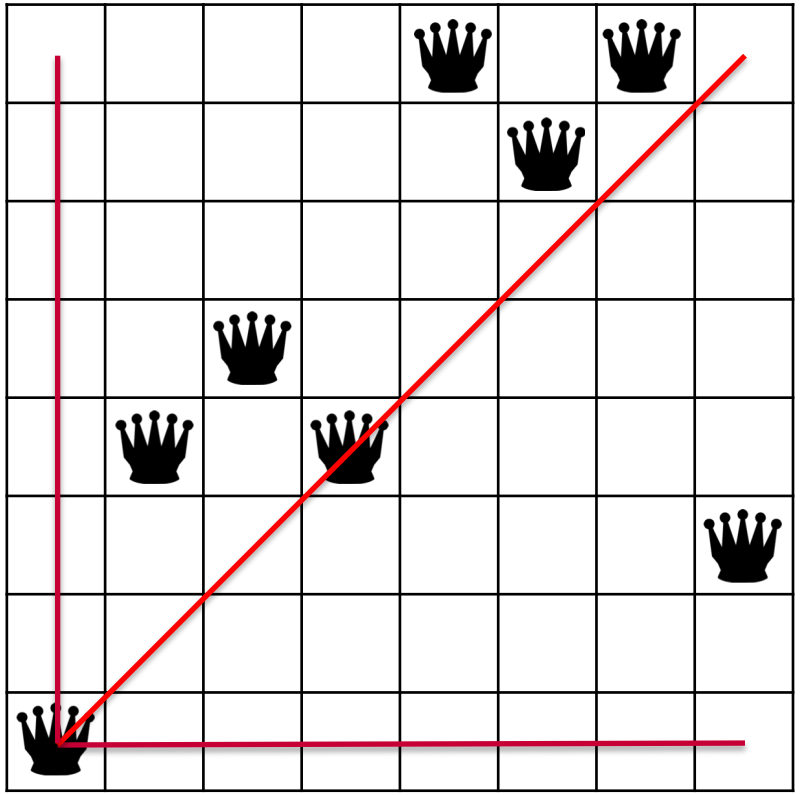
8-Queens

- Task: place 8 queens on the chess board such that they do not attack each other
- Max/Min Conflict: Count constraint violations



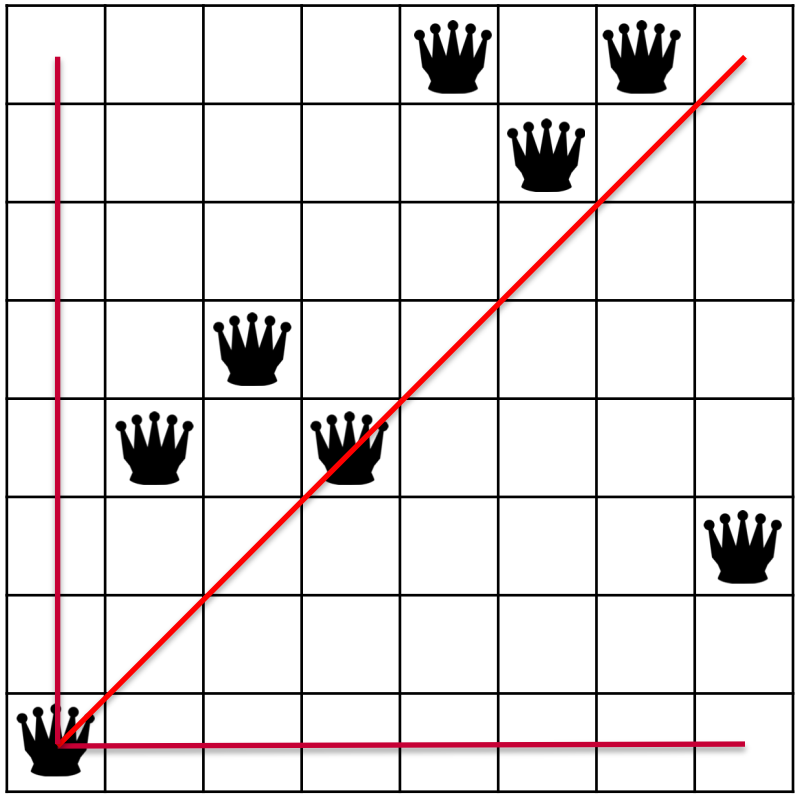
8-Queens

- Task: place 8 queens on the chess board such that they do not attack each other
- Count constraint violations



8-Queens

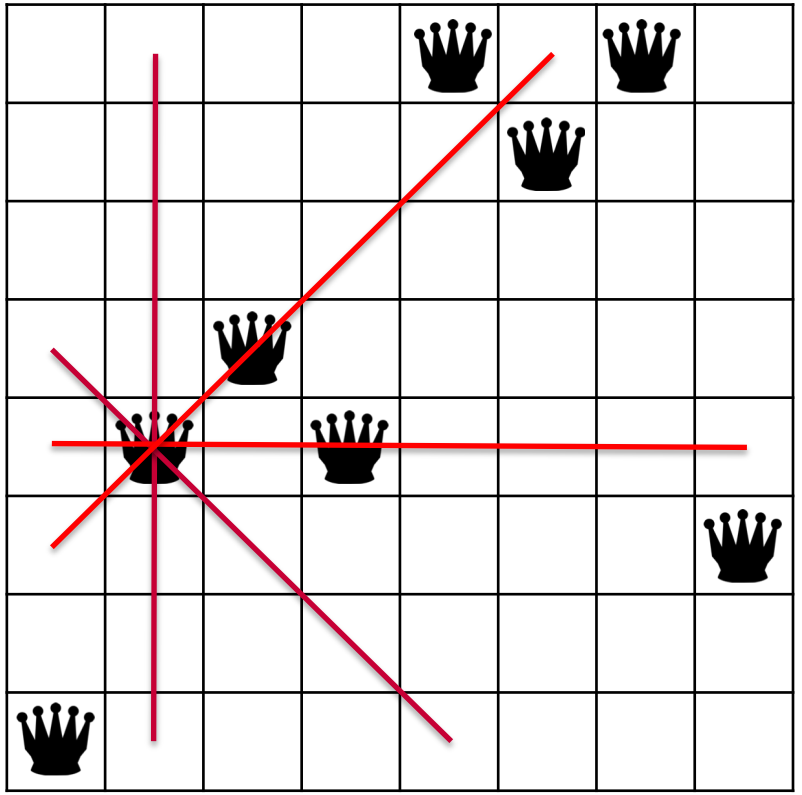
- Task: place 8 queens on the chess board such that they do not attack each other
- Count constraint violations



1

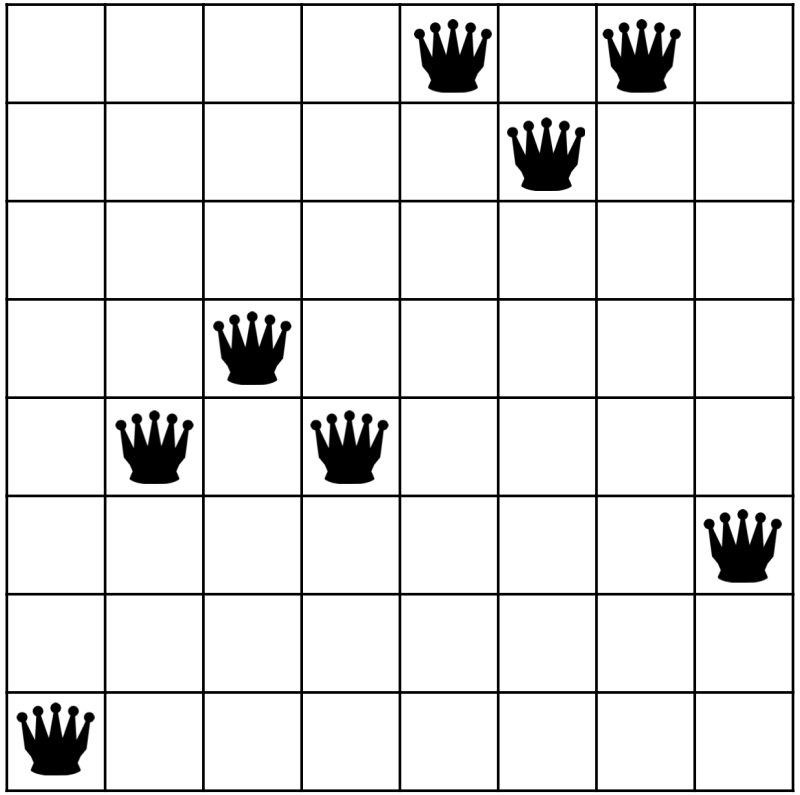
8-Queens

- Task: place 8 queens on the chess board such that they do not attack each other
- Count constraint violations



8-Queens

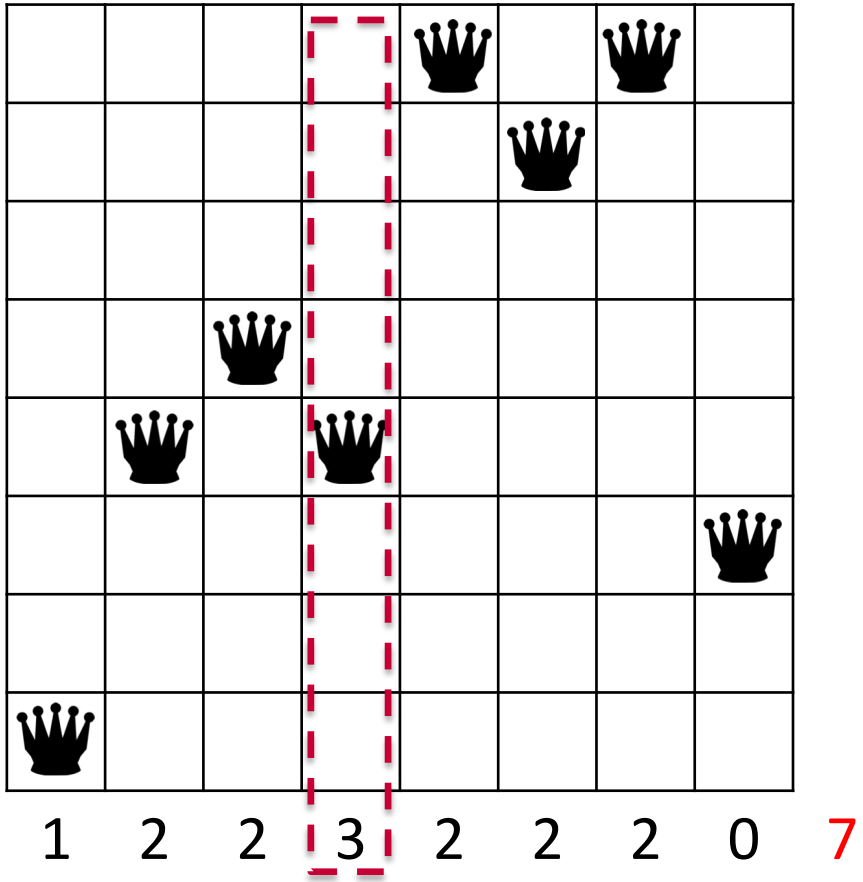
- Task: place 8 queens on the chess board such that they do not attack each other
- Count constraint violations



1 2 2 3 2 2 2 0 7

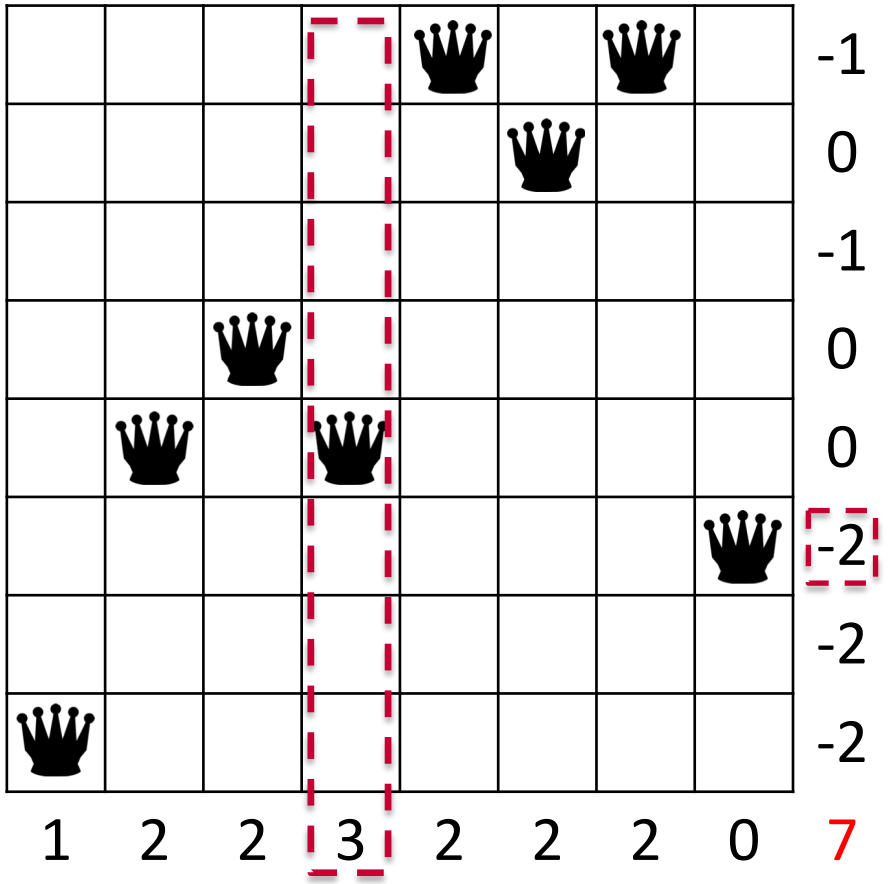
8-Queens

- Task: place 8 queens on the chess board such that they do not attack each other
- Count constraint violations
- Queen with most violations
- Possible moves



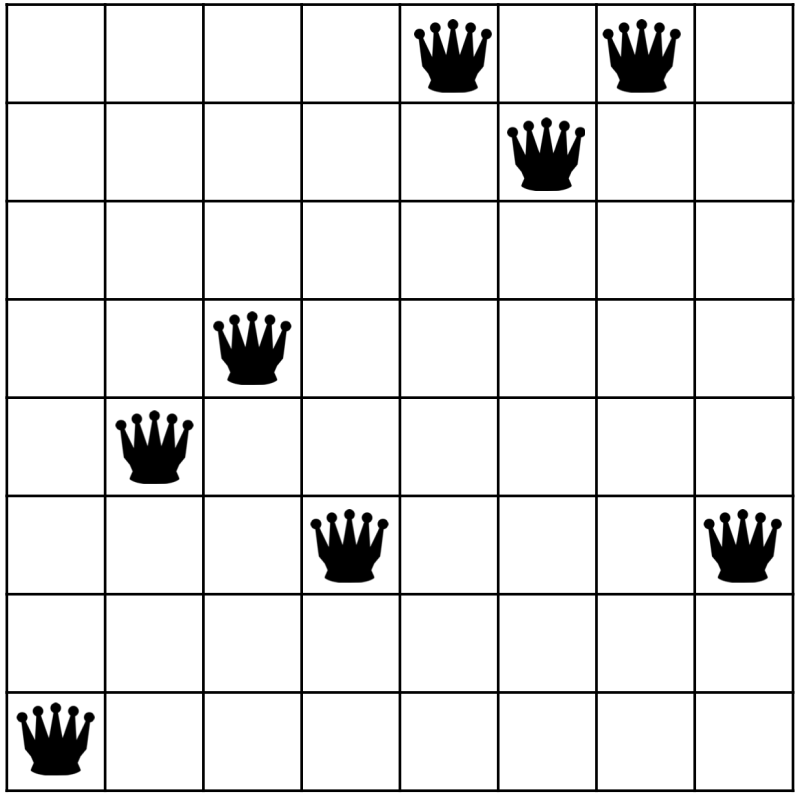
8-Queens

- Task: place 8 queens on the chess board such that they do not attack each other
- Count constraint violations
- Queen with most violations
- Possible moves
- Find the best move



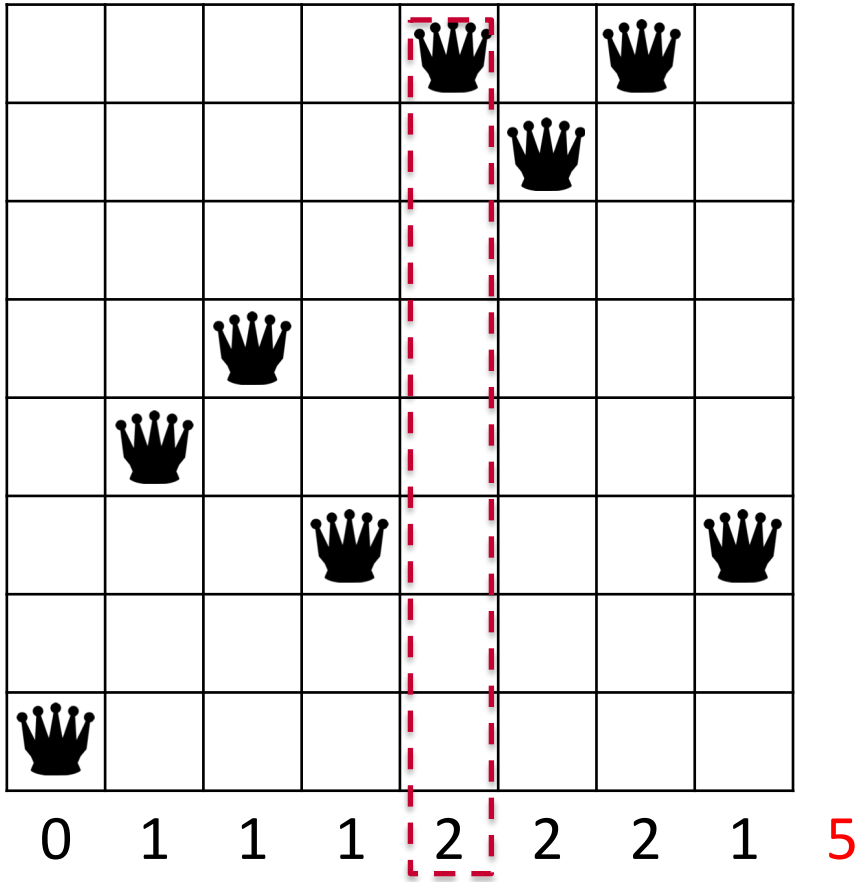
8-Queens

- Task: place 8 queens on the chess board such that they do not attack each other
- Count constraint violations
- Queen with most violations
- Possible moves
- Find the best move



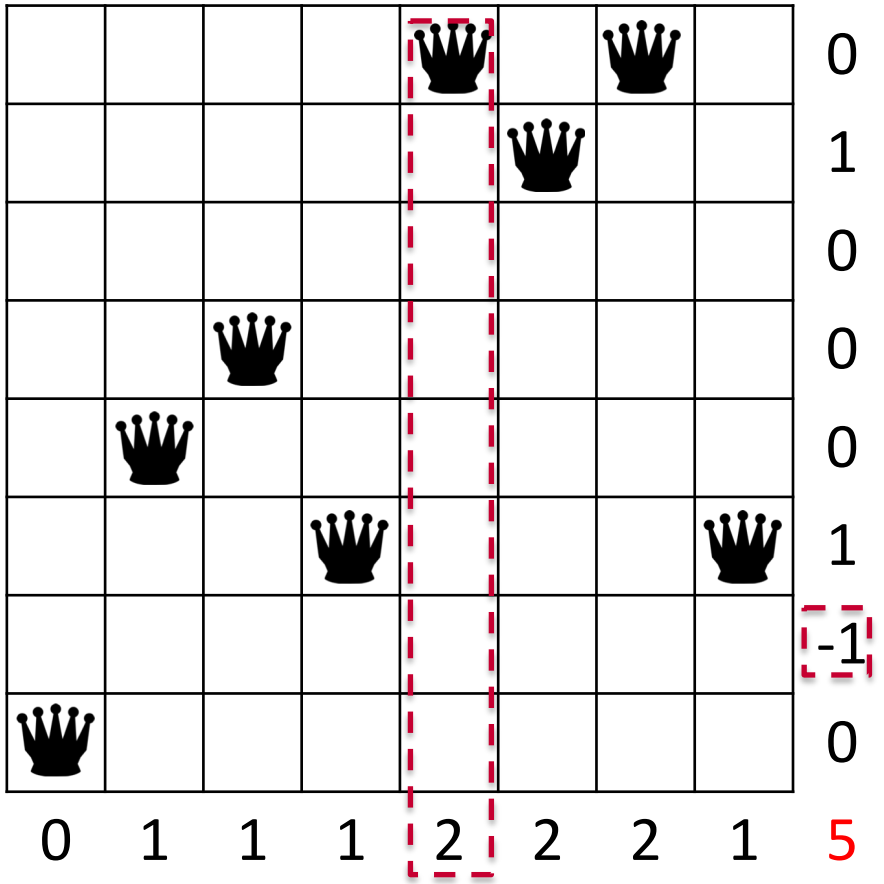
8-Queens

- Task: place 8 queens on the chess board such that they do not attack each other
- Count constraint violations
- Queen with most violations



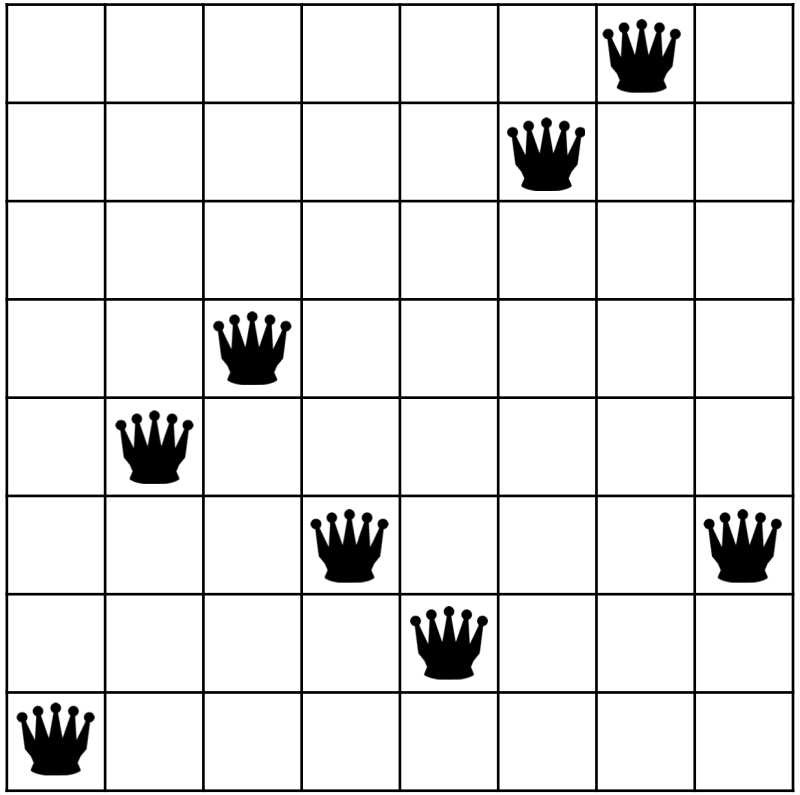
8-Queens

- Task: place 8 queens on the chess board such that they do not attack each other
- Count constraint violations
- Queen with most violations
- Possible moves
- Find the best move



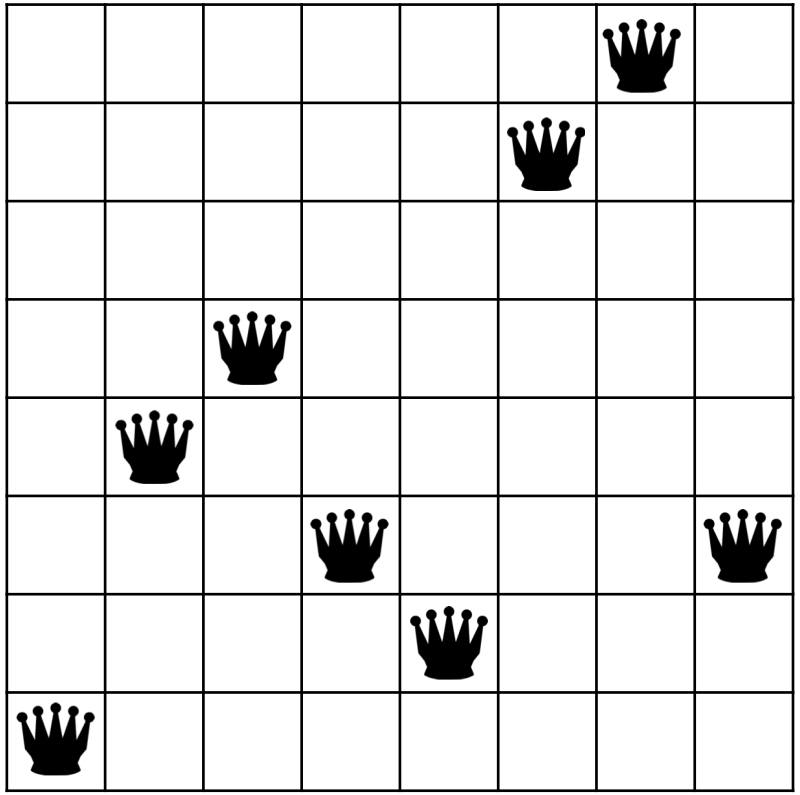
8-Queens

- Task: place 8 queens on the chess board such that they do not attack each other
- Count constraint violations
- Queen with most violations
- Possible moves
- Find the best move



8-Queens

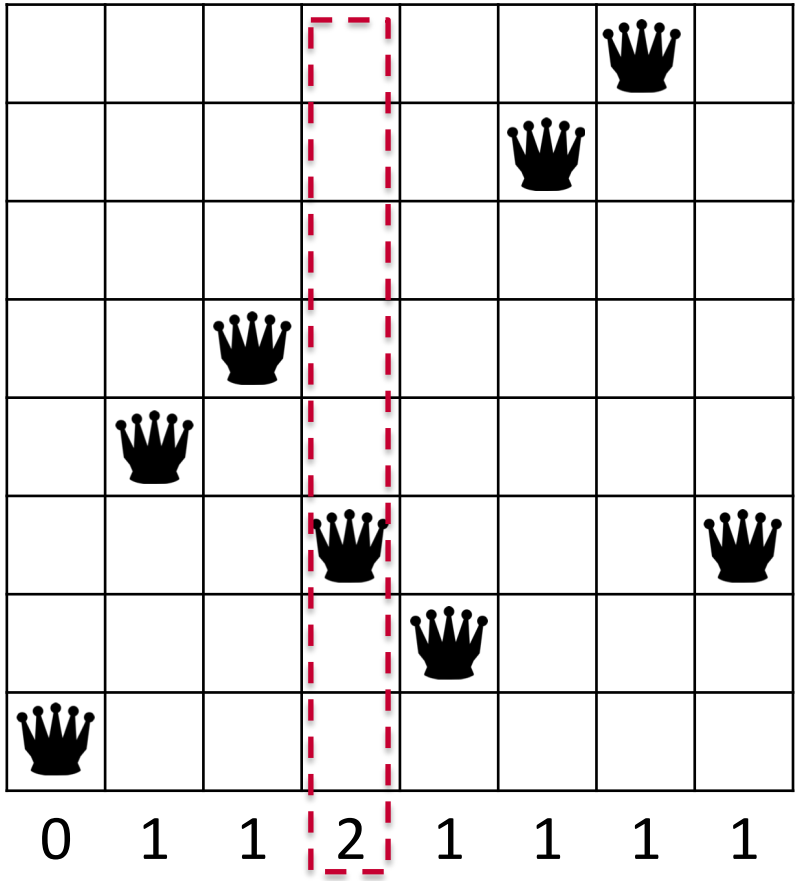
- Task: place 8 queens on the chess board such that they do not attack each other
- Count constraint violations



0 1 1 2 1 1 1 1 4

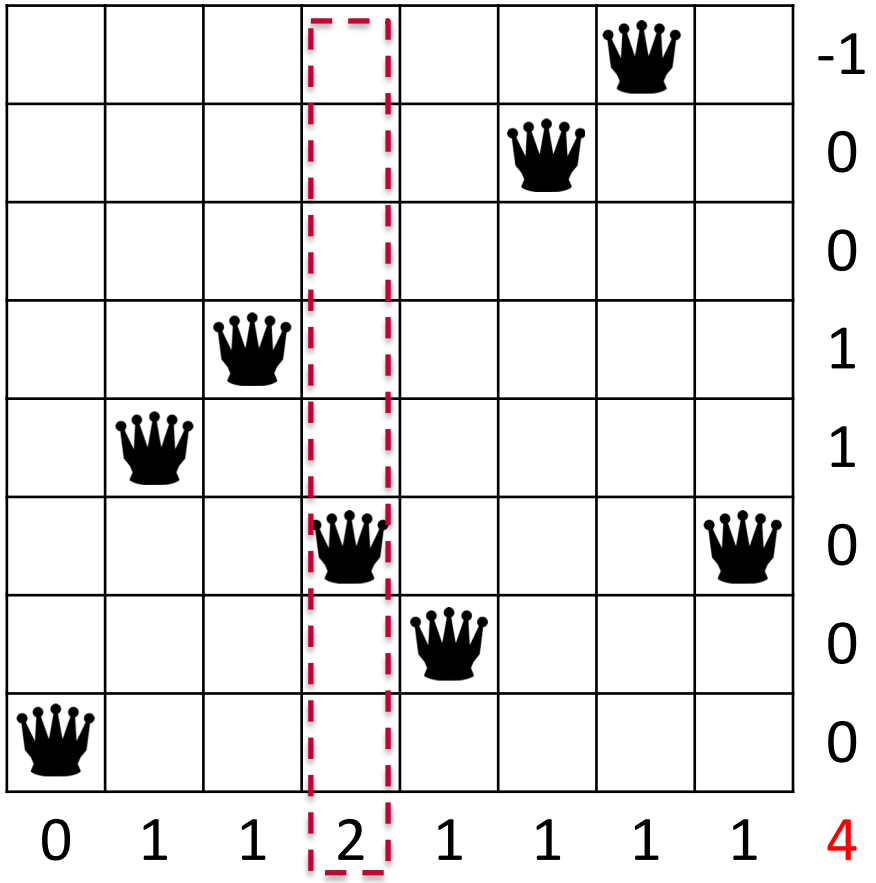
8-Queens

- Task: place 8 queens on the chess board such that they do not attack each other
- Count constraint violations
- Queen with most violations



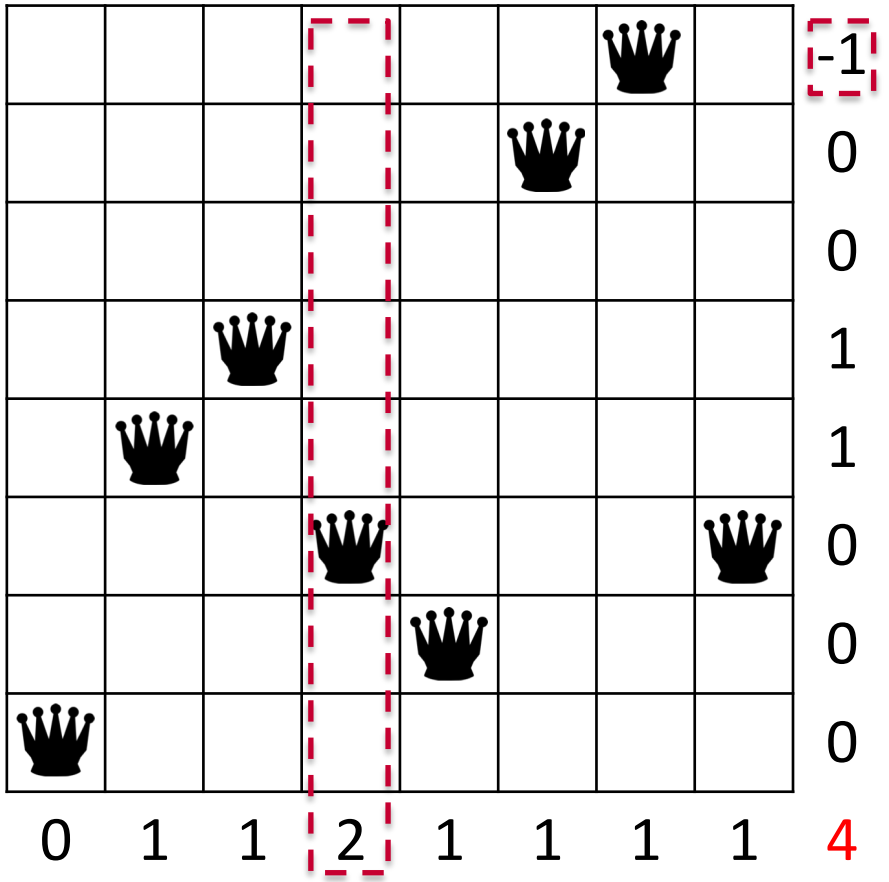
8-Queens

- Task: place 8 queens on the chess board such that they do not attack each other
- Count constraint violations
- Queen with most violations
- Possible moves



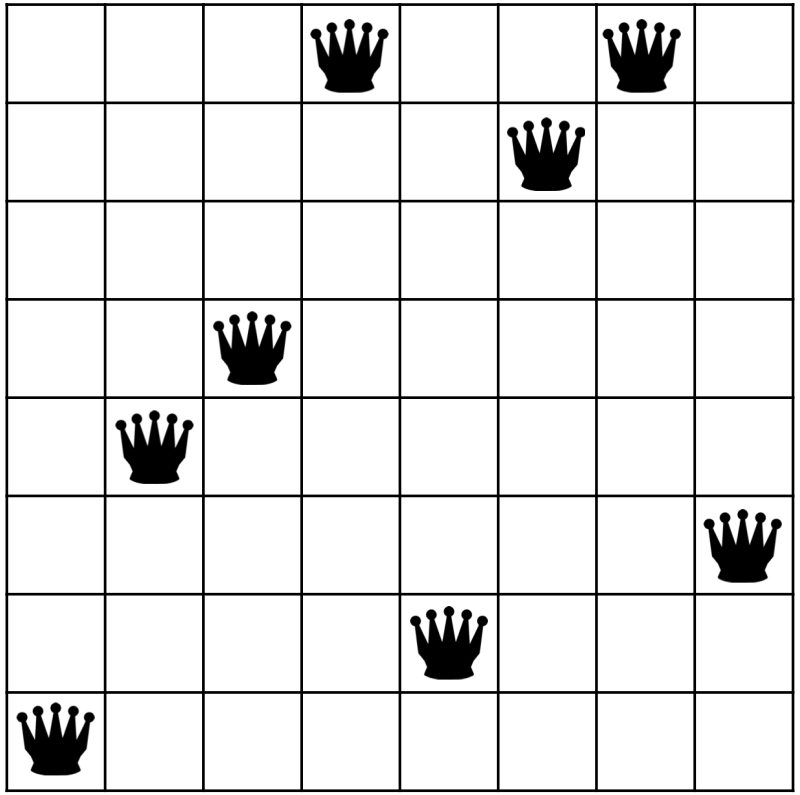
8-Queens

- Task: place 8 queens on the chess board such that they do not attack each other
- Count constraint violations
- Queen with most violations
- Possible moves
- Find the best move



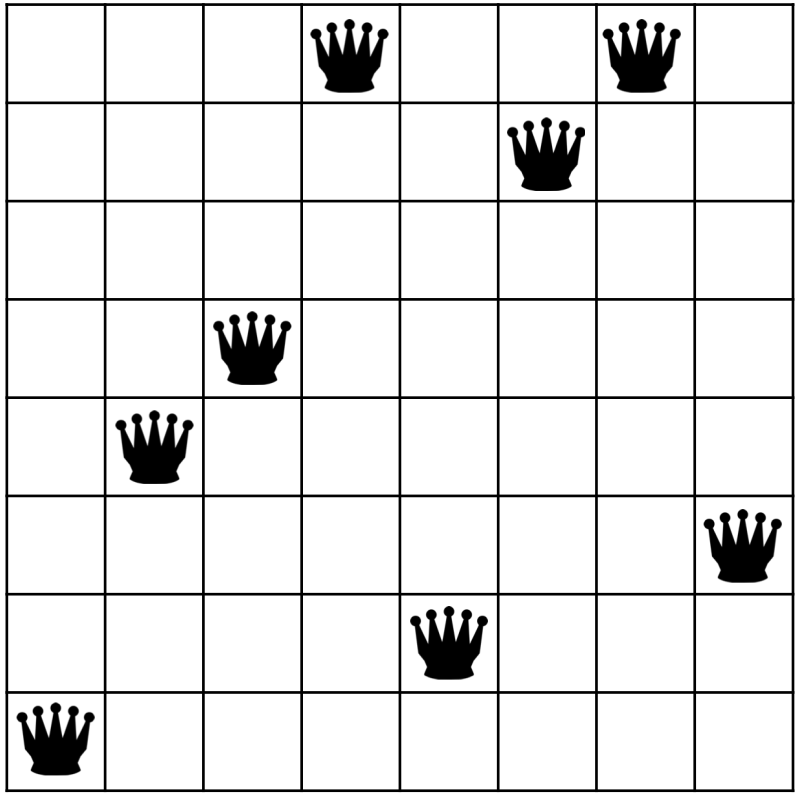
8-Queens

- Task: place 8 queens on the chess board such that they do not attack each other
- Count constraint violations
- Queen with most violations
- Possible moves
- Find the best move



8-Queens

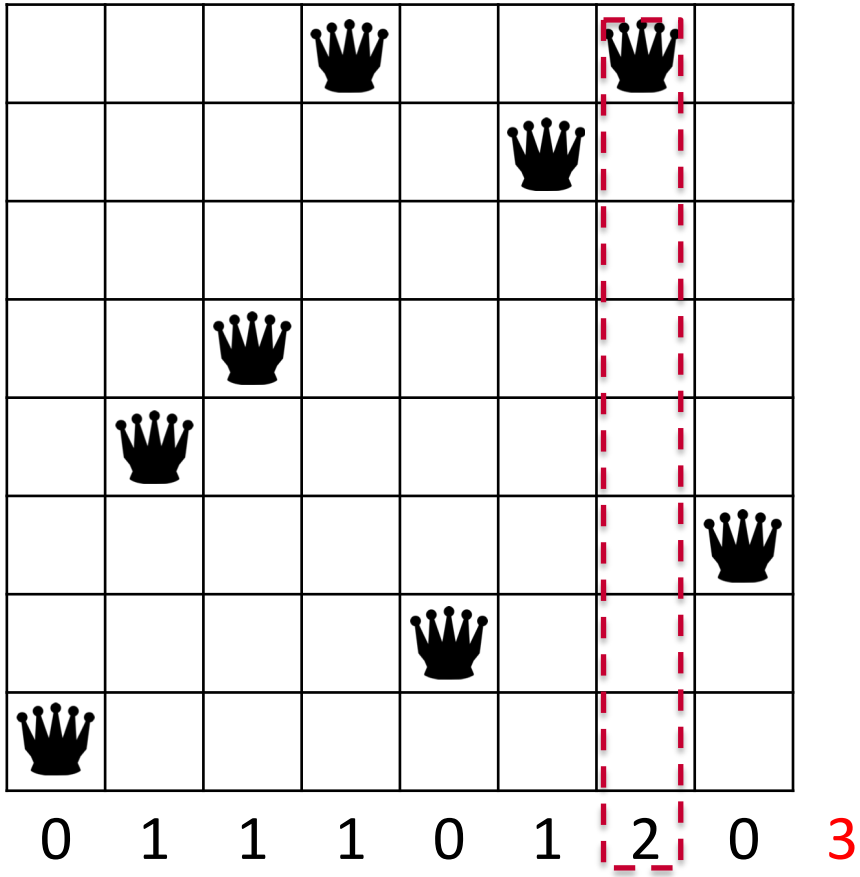
- Task: place 8 queens on the chess board such that they do not attack each other
- Count constraint violations



0 1 1 1 0 1 2 0 3

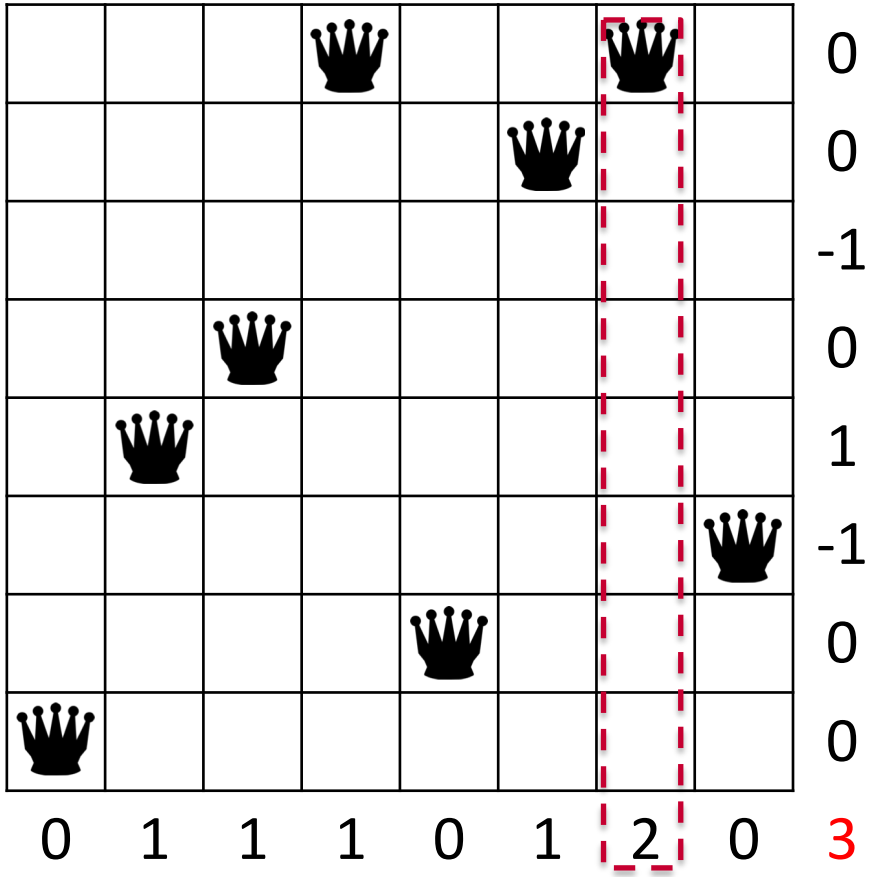
8-Queens

- Task: place 8 queens on the chess board such that they do not attack each other
- Count constraint violations
- Queen with most violations



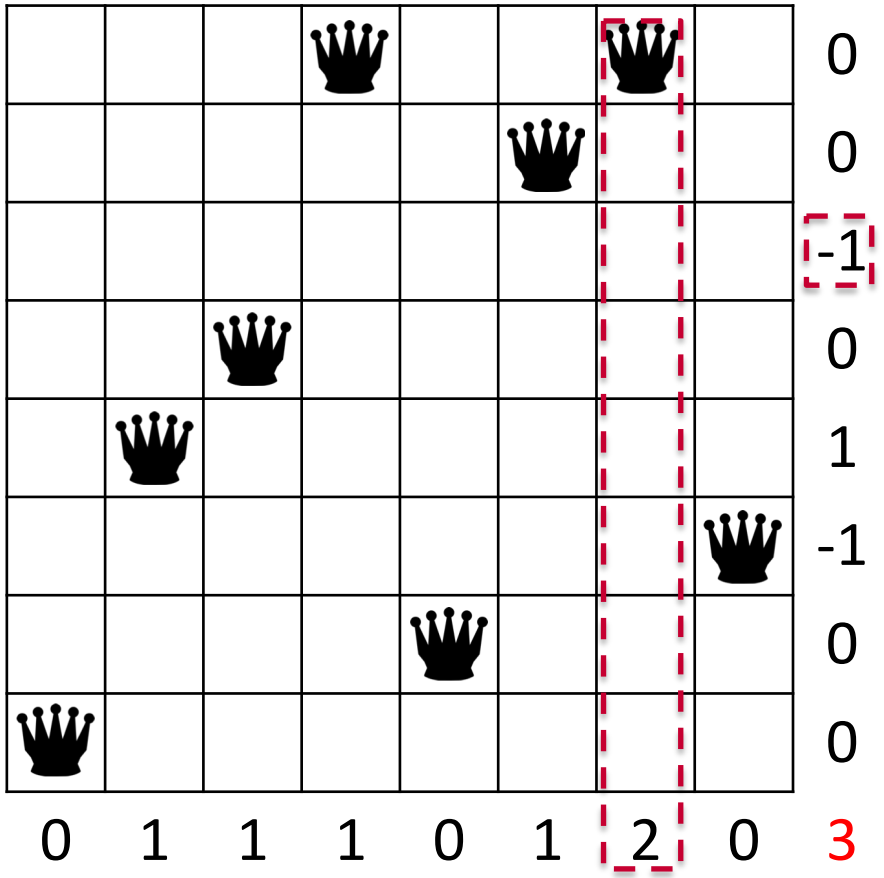
8-Queens

- Task: place 8 queens on the chess board such that they do not attack each other
- Count constraint violations
- Queen with most violations
- Possible moves



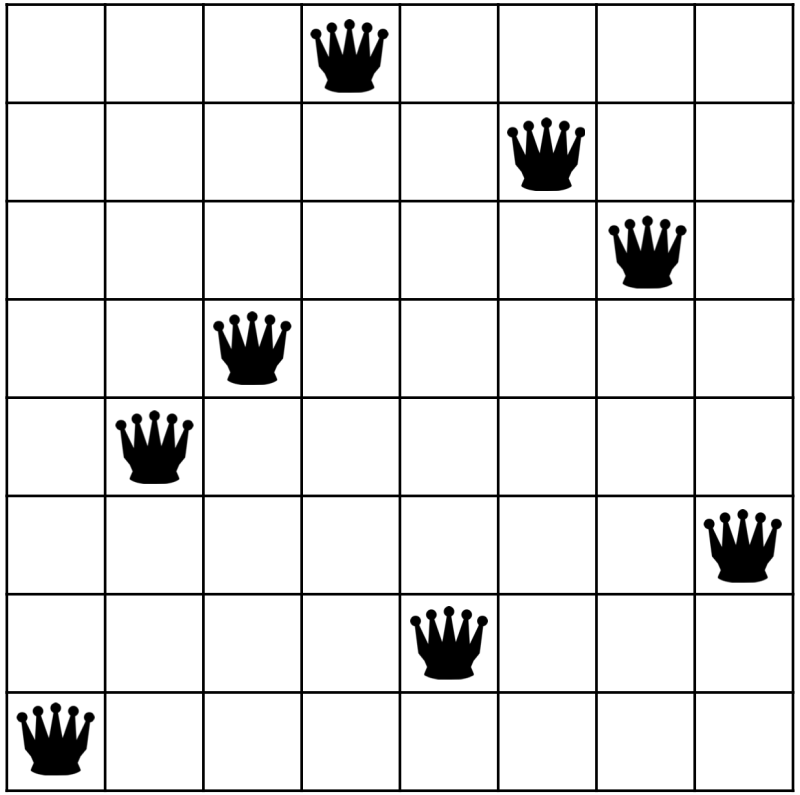
8-Queens

- Task: place 8 queens on the chess board such that they do not attack each other
- Count constraint violations
- Queen with most violations
- Possible moves
- Find the best move



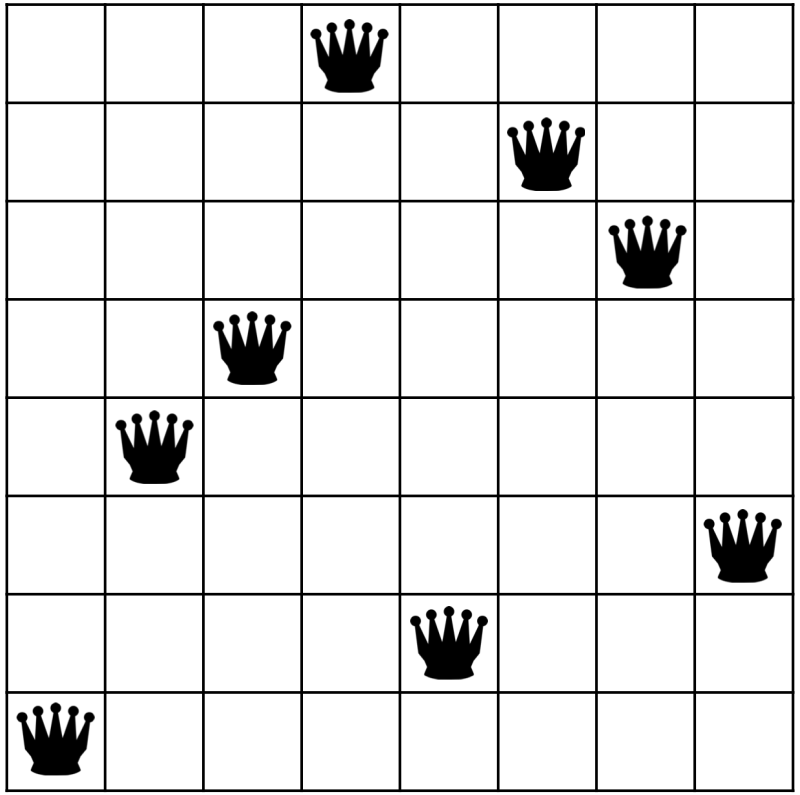
8-Queens

- Task: place 8 queens on the chess board such that they do not attack each other
- Count constraint violations
- Queen with most violations
- Possible moves
- Find the best move



8-Queens

- Task: place 8 queens on the chess board such that they do not attack each other
- Count constraint violations
- Local minimum!
  - Max conflicts is 1
  - Queen on every row
  - So moving a queen from its current row will place it in the same row as another queen



N-queens - Demo

Hill-climbing search: 8-queens problem

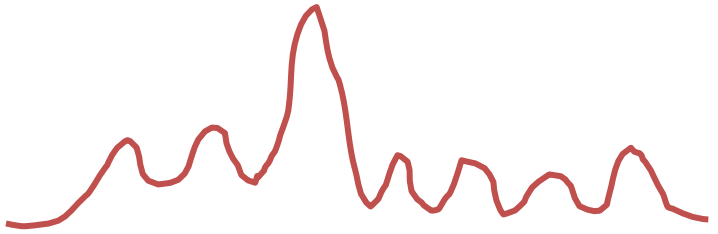
- Randomly generated 8-queens starting states…
  - 14% of the time it solves the problem
  - 86% of the time it get stuck at a local minimum

- However…
  - Takes only 4 steps on average when it succeeds
  - And 3 on average when it gets stuck

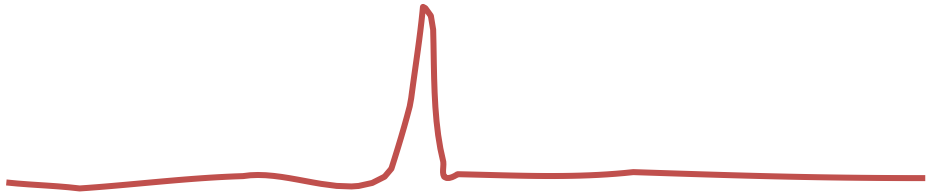
    (for a state space with 8^8 =~17 million states)

Hill Climbing Drawbacks

Local minima/maxima



Plateaus



Hill climbing & SAT → GSAT

```
procedure GSAT (F, maxSteps)
    input CNF formula F, positive integer maxSteps
    output model of F or "no solution found"
    a := randomly chosen assignment of the variables in formula F;
    for step := 1 to maxSteps do
        if a satisfies F then return a;
        v := randomly selected variable the flip of which minimises
             the number of unsatisfied clauses;
        a := a with v flipped;
    end for;
    return "no solution found";
end GSAT
```

Initial Random solution

Best move Breaking ties at random

Hill climbing & SAT → GSAT

Suppose we have a SAT problem with 20 clauses and 5 variables.

- Cost: Number of unsatisfied clauses
- a → X1 = T, X2 = T, X3 = F, X4 = F, X5= T → Cost: 10
- Potential Neighbors (Flipping one variable)

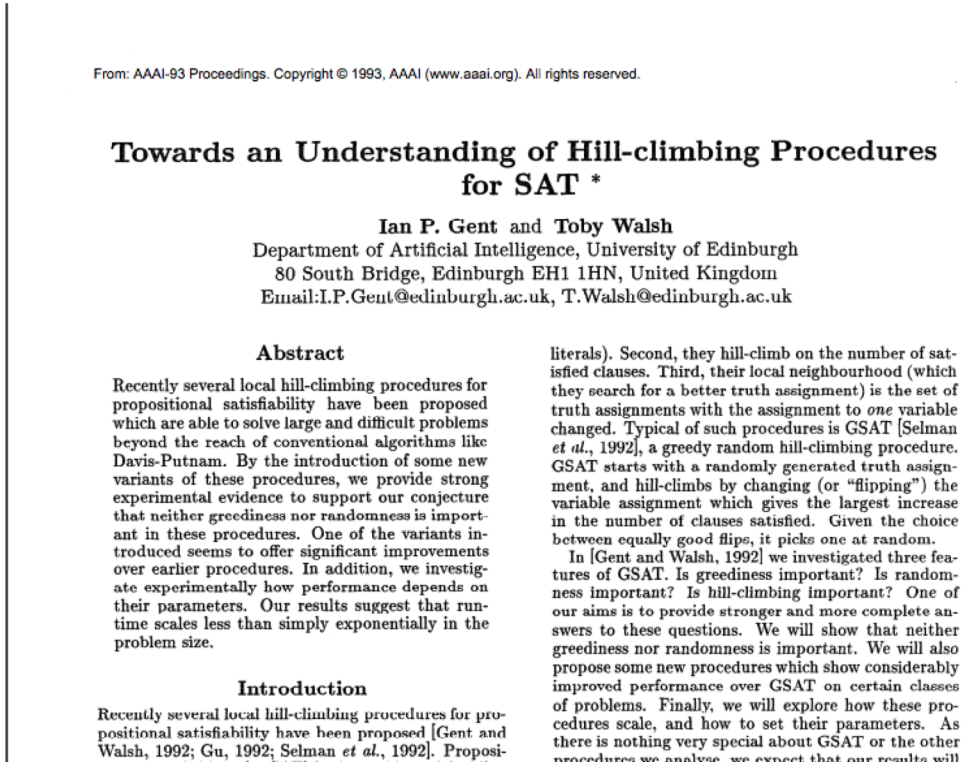
GSAT: Flip the best variable
Breaking ties at random:
i.e., random selection between X_2 and X_5

X1 = F, X2 = T, X3 = F, X4 = F, X5= T → Cost: 7
X1 = T, **X2 = F**, X3 = F, X4 = F, X5= T → Cost: 5
X1 = T, X2 = T, **X3 = T**, X4 = F, X5= T → Cost: 15
X1 = T, X2 = T, X3 = F, **X4 = T**, X5= T → Cost: 17
X1 = T, X2 = T, X3 = F, X4 = F, **X5= F** → Cost: 5

Best neighbors

HSAT

- HSAT (Gent and Walsh, 1993) – Same as GSAT, but break ties in favor of the least recently flipped variable



Towards an Understanding of Hill-climbing Procedures for SAT *

Ian P. Gent and **Toby Walsh**
Department of Artificial Intelligence, University of Edinburgh
80 South Bridge, Edinburgh EH1 1HN, United Kingdom
Email:I.P.Gent@edinburgh.ac.uk, T.Walsh@edinburgh.ac.uk

Abstract

Recently several local hill-climbing procedures for propositional satisfiability have been proposed which are able to solve large and difficult problems beyond the reach of conventional algorithms like Davis-Putnam. By the introduction of some new variants of these procedures, we provide strong experimental evidence to support our conjecture that neither greediness nor randomness is important in these procedures. One of the variants introduced seems to offer significant improvements over earlier procedures. In addition, we investigate experimentally how performance depends on their parameters. Our results suggest that runtime scales less than simply exponentially in the problem size.

Introduction

Recently several local hill-climbing procedures for propositional satisfiability have been proposed [Gent and Walsh, 1992; Gu, 1992; Selman *et al.*, 1992]. Propositional literals). Second, they hill-climb on the number of satisfied clauses. Third, their local neighbourhood (which they search for a better truth assignment) is the set of truth assignments with the assignment to *one* variable changed. Typical of such procedures is GSAT [Selman *et al.*, 1992], a greedy random hill-climbing procedure. GSAT starts with a randomly generated truth assignment, and hill-climbs by changing (or "flipping") the variable assignment which gives the largest increase in the number of clauses satisfied. Given the choice between equally good flips, it picks one at random.

In [Gent and Walsh, 1992] we investigated three features of GSAT. Is greediness important? Is randomness important? Is hill-climbing important? One of our aims is to provide stronger and more complete answers to these questions. We will show that neither greediness nor randomness is important. We will also propose some new procedures which show considerably improved performance over GSAT on certain classes of problems. Finally, we will explore how these procedures scale, and how to set their parameters. As there is nothing very special about GSAT or the other procedures we analyse, we expect that our results will

Escaping Shoulders: Sideways Move

- If no downhill (uphill) moves, allow sideways moves in hope that algorithm can escape
  - Need to place a limit on the possible number of sideways moves to avoid infinite loops
- For 8-queens
  - Now allow sideways moves with a limit of 100
  - Raises percentage of problem instances solved from 14 to 94%
  - However…
    - 21 steps for every successful solution
    - 64 for each failure

Local search: no guarantees

- Local Minima
  - No step improves the objective
- No guarantees for global optimality
- Escaping local optima is a critical issue in local search

Hill climbing – Max/Min approach

function HILL-CLIMBING(*problem*) **return** a state that is a local maximum
input: *problem*, a problem
local variables: *current*, **a node.**
neighbor, **a node.**
best, **a node.**

current ← MAKE-NODE(INITIAL-STATE[*problem*])
loop do
~~*neighbor* ← a highest valued successor of *current*~~
candidate ← variable with expected maximum impact on objective function
neighbor ← a highest valued value for *candidate*
if VALUE [*neighbor*] > VALUE[*best*] **then** *best* ← *neighbor*
current ← *neighbor*

May not be highest valued successor state of current!

THEREFORE WE DON'T KNOW WHEN WE ARE IN LOCAL MINIMUM

Iterative first-improvement

- Best improvement (aka gradient descent, greedy hill-climbing) choose the best (min or max improving neighbor)

  Requires evaluation of all neighbors in each step

- First improvement: evaluate neighbors in fixed order, chose first improving step encountered

  Can be much more efficient than Best improvement

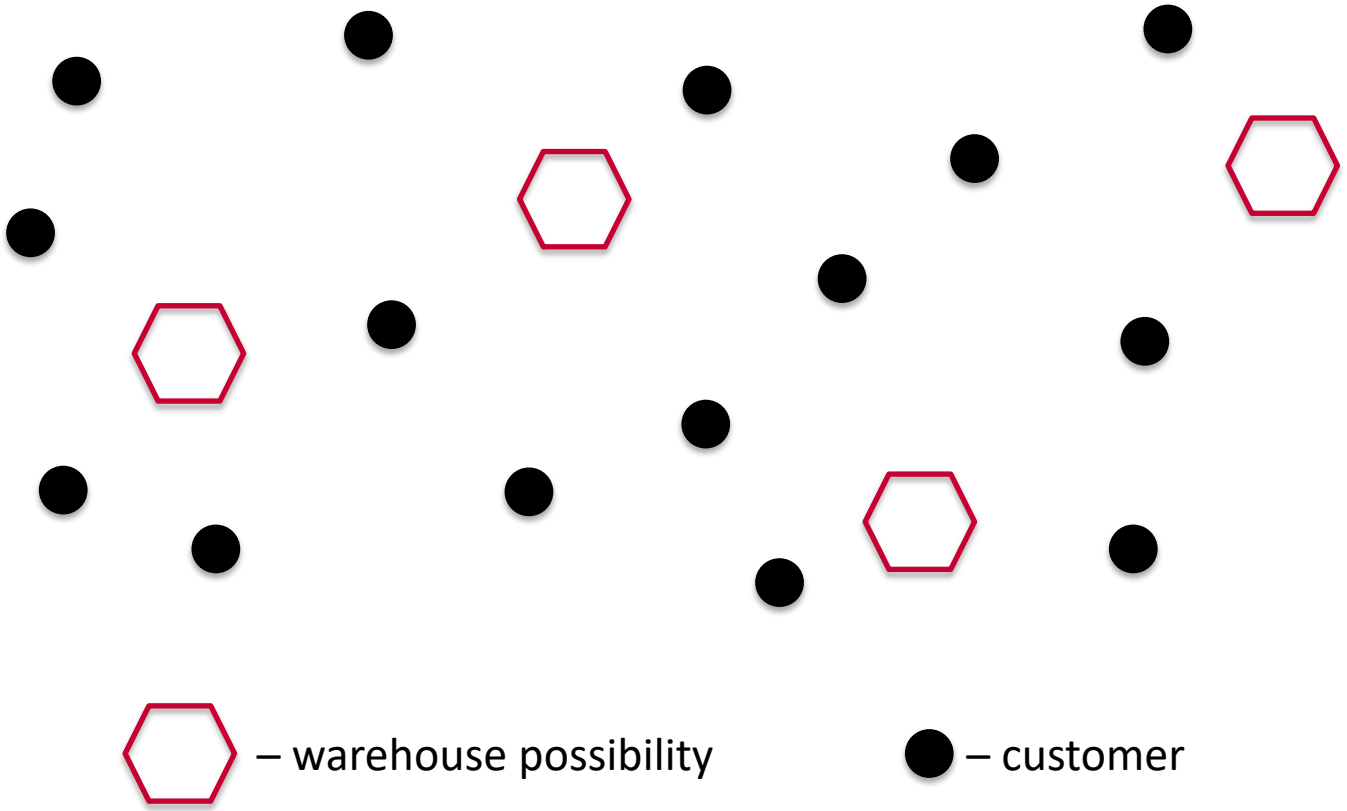
  Order of evaluation can have significant impact on performance

Iterative best/first improvement

```
procedure iterative best-improvement
    while improvement
        improvement ← false
        for i ← 1 to n do
            for j ← 1 to n do
                CheckMove(i, j);
                if move is new best improvement then
                    (k, l) ← MemorizeMove(i, j);
                    improvement ← true
            endfor
        endfor
        ApplyMove(k, l);
    endwhile
end iterative best-improvement
```

```
procedure iterative first-improvement
    while improvement
        improvement ← false
        for i ← 1 to n do
            for j ← 1 to n do
                CheckMove(i, j);
                if move improves then
                    ApplyMove(i, j);
                    improvement ← true
            endfor
        endfor
    endwhile
end iterative first-improvement
```

Example: optimization problem

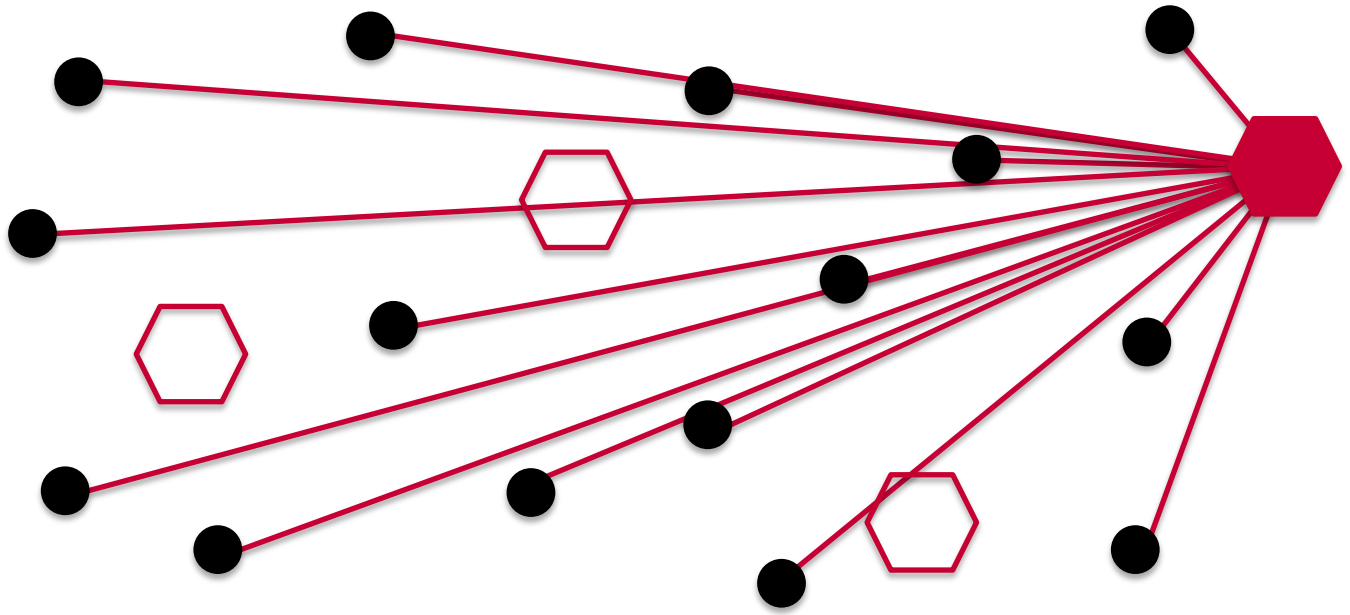


Task: Where to place warehouses?

(Fixed) cost for having warehouse at position w: f_w

Cost for transportation from w to c: $t_{w,c}$

Example: optimization problem



– warehouse possibility

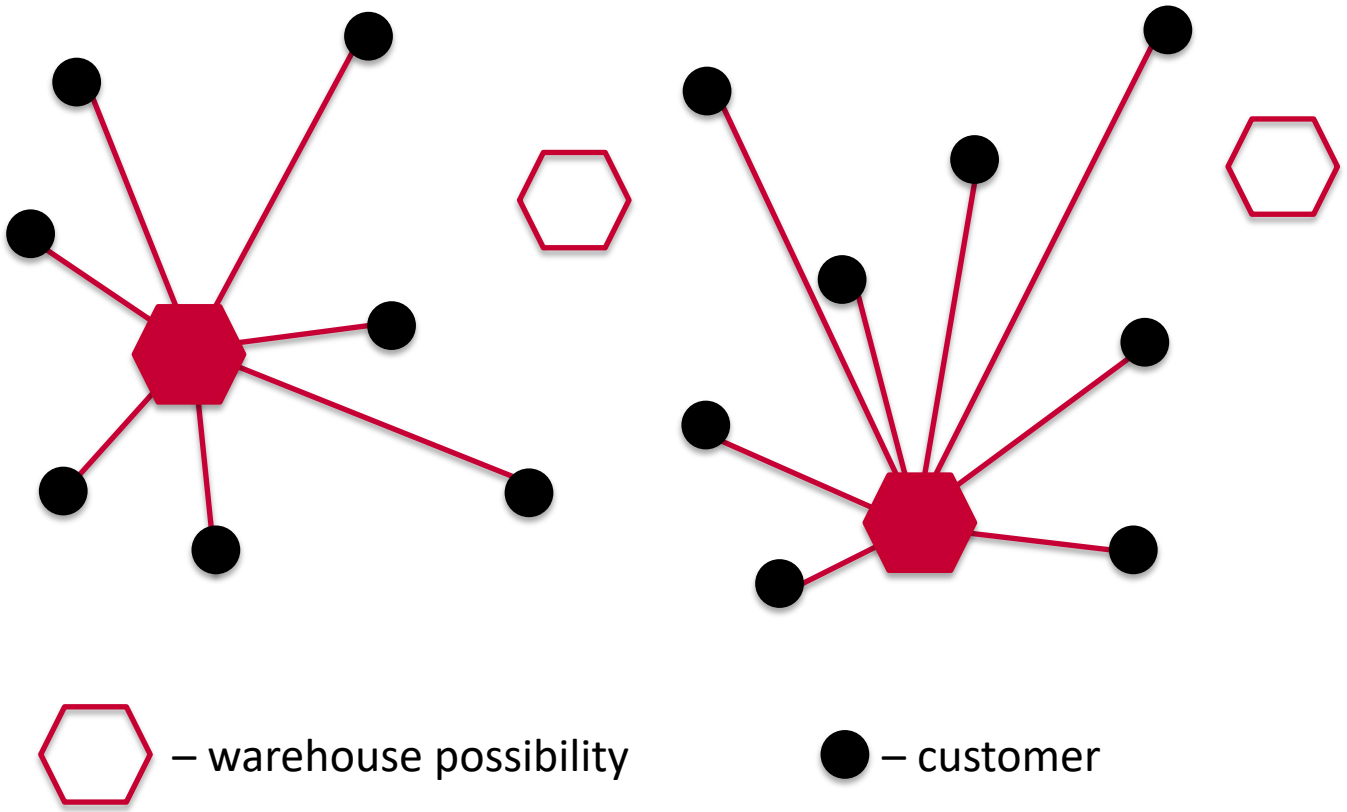
– customer

Task: Where to place warehouses?

Cost for opening warehouse at position w: f_w

Cost for transportation from w to c: $t_{w,c}$

Example: optimization problem



Task: Where to place warehouses?

Cost for opening warehouse at position w: f_w

Cost for transportation from w to c: $t_{w,c}$

Warehouse location

- Task: where to place warehouses? (input: f_w , $t_{w,c}$)

- Decision Variables:
  - $o_w \in \{0, 1\}$: whether warehouse w is open
  - $\alpha[c] \in \{1, \ldots, W\}$: the warehouse assigned to customer c

- Constraints: customers can be assigned only to open warehouses

- Objective:

$$\min \quad \sum_{w \in W} f_w o_w + \sum_{c \in C} t_{a[c],c}$$

Warehouse location

- Task: where to place warehouses? (input: f_w , $t_{w,c}$)

- Objective

$$\min \quad \sum_{w \in W} f_w o_w + \sum_{c \in C} t_{a[c],c}$$

- Key observations:
  - Once the warehouse locations have been chosen, the problem is easy
  - It's enough to assign customers to warehouses minimizing the transportation costs
  - Note the difference between testing different values for the o_w variables and the $\alpha[c]$ variables

K-medoids: Partitioning around medoids algorithm

Source https://en.wikipedia.org/wiki/K-medoids

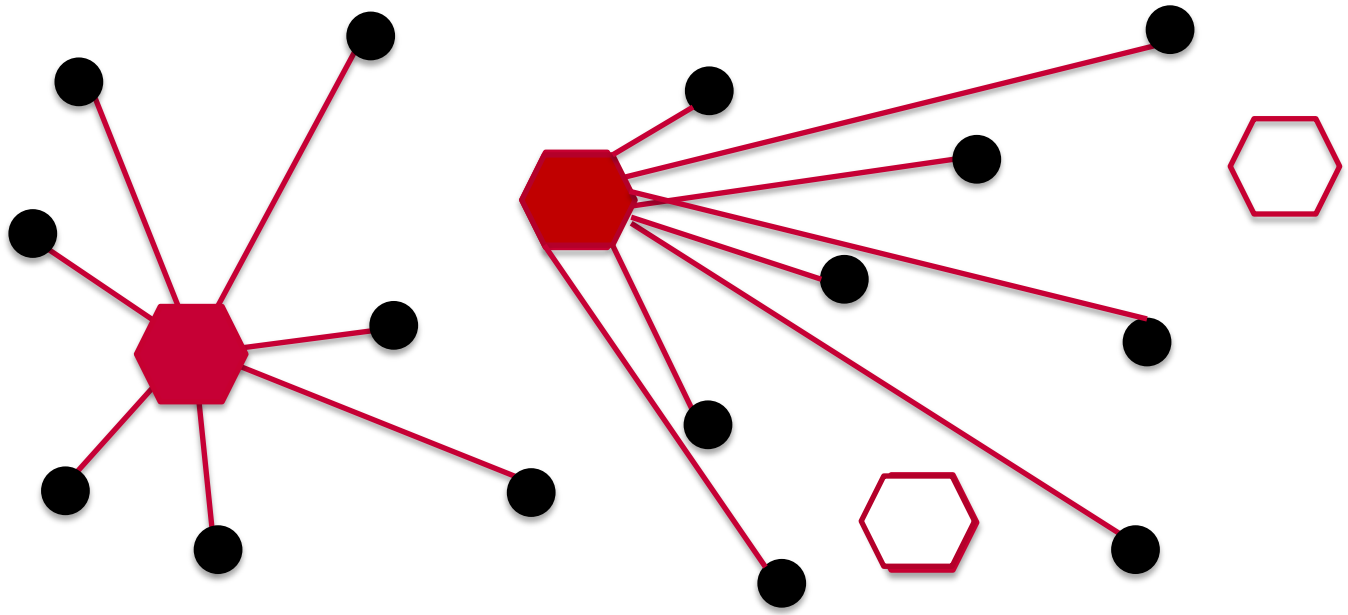
1. Initialize: greedily select *k* of the *n* data points as the medoids to minimize the cost
2. Associate each data point to the closest medoid.
3. While the cost of the configuration decreases:
    1. For each medoid *m*, and or each non-medoid data point *o*:
        1. Consider the swap of *m* and *o*, and compute the cost change
        2. If the cost change is better than the current best, remember this *m* and *o* combination as the best swap and cost change as current best
    2. Perform the best swap if it decreases the cost function. Otherwise, the algorithm terminates.

Warehouse location

- Task: where to place warehouses? (input: f_w , $t_{w,c}$)
- Neighborhood:
  - Open/close warehouses (flip the value of o_w)
  - Swap warehouses (close one and open the other)
  - Simultaneous swap of k warehouses
  - Or even combine 1st and 2nd neighborhoods, so with some probability one chooses a neighborhood of type 1 for an iteration or otherwise a neighborhood of type 2
- Only 1 move not allowed?

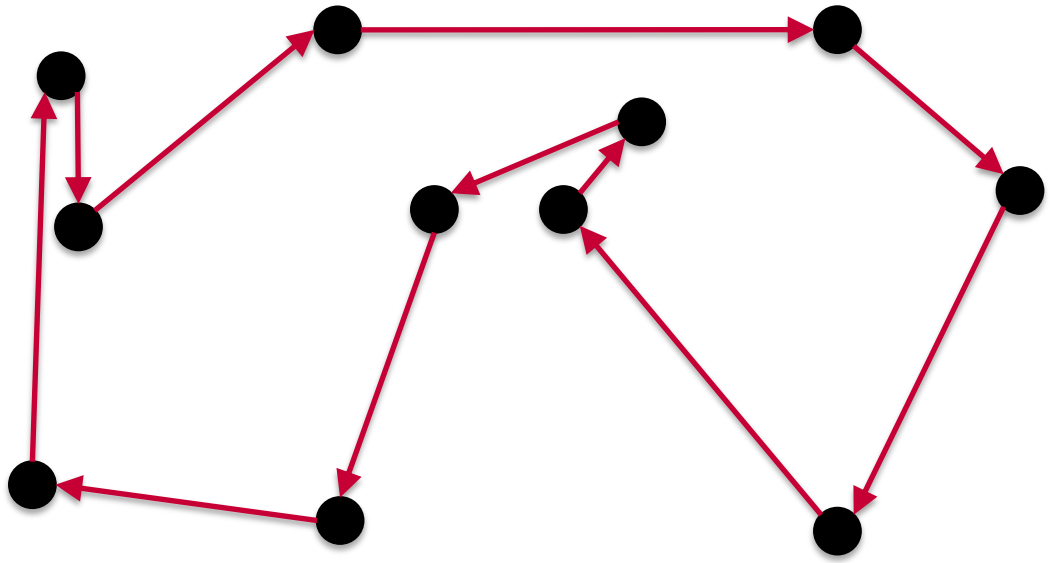
Example: optimization problem

Swap/Flip



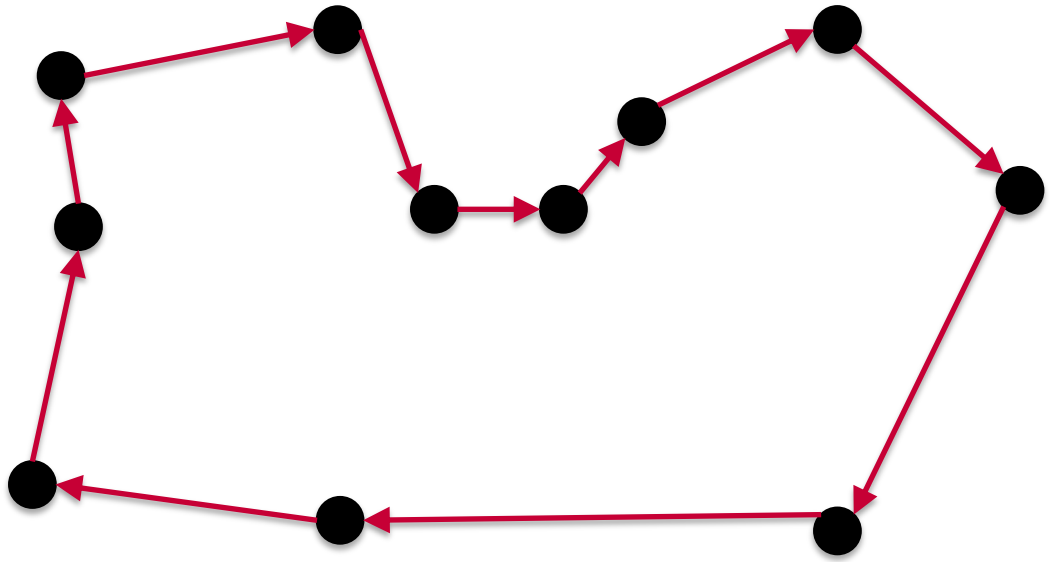
Travelling salesman problem (TSP)

- Task: find the shortest path to visit all cities exactly once
  - Hamiltonian cycle in a graph
  - Simplification: cities are points in 2D



Travelling salesman problem (TSP)

- Task: find the shortest path to visit all cities exactly once
  - Hamiltonian cycle in a graph
  - Simplification: cities are points in 2D



Travelling salesman problem (TSP)

- Task: find the shortest path to visit all cities exactly once
  - Hamiltonian cycle in a graph
  - Simplification: cities are points in 2D

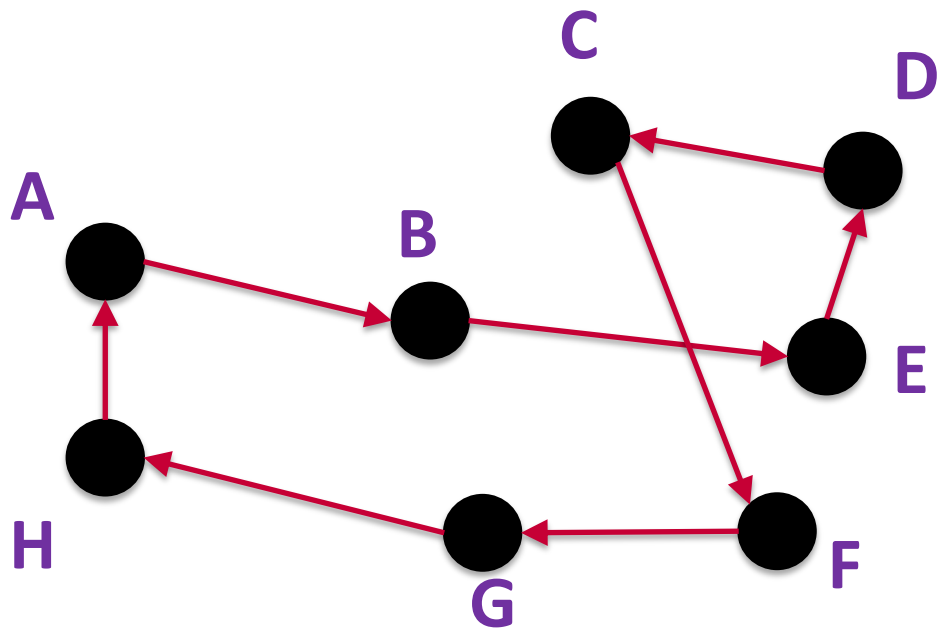
- Decision Variables:
  - $c_i \in \{0, \ldots, n\}$: which city is in position *i* of the route

- Neighborhood?

Local search for TSP: 2-OPT

- Task: find the shortest path to visit all cities exactly once
- Local Move:
  - Select two edges and replace them by two other edges

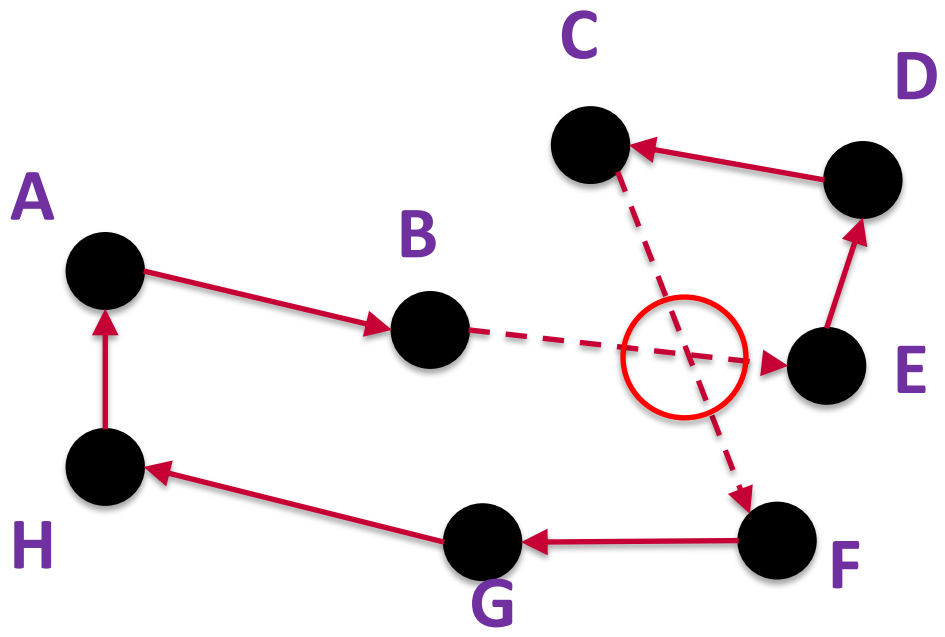
Route:
ABEDCFGHA



Local search for TSP: 2-OPT

- Task: find the shortest path to visit all cities exactly once
- Local Move:
  - Select two edges and replace them by two other edges

B-E and C-F

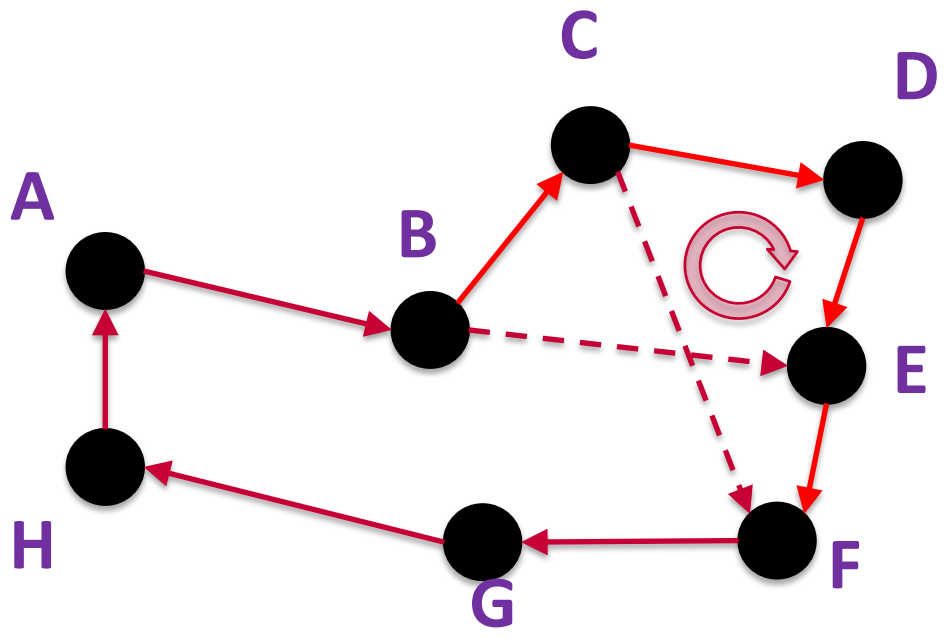


Local search for TSP: 2-OPT

- Task: find the shortest path to visit all cities exactly once
- Local Move:
  - Select two edges and replace them by two other edges

B-C and E-F

Route:
ABCDEFGHA



Local search for TSP: 2-OPT

- A TSP tour T is called 2-optimal if there is no 2-adjacent tour to T with lower cost than T.

- 2-opt heuristic:

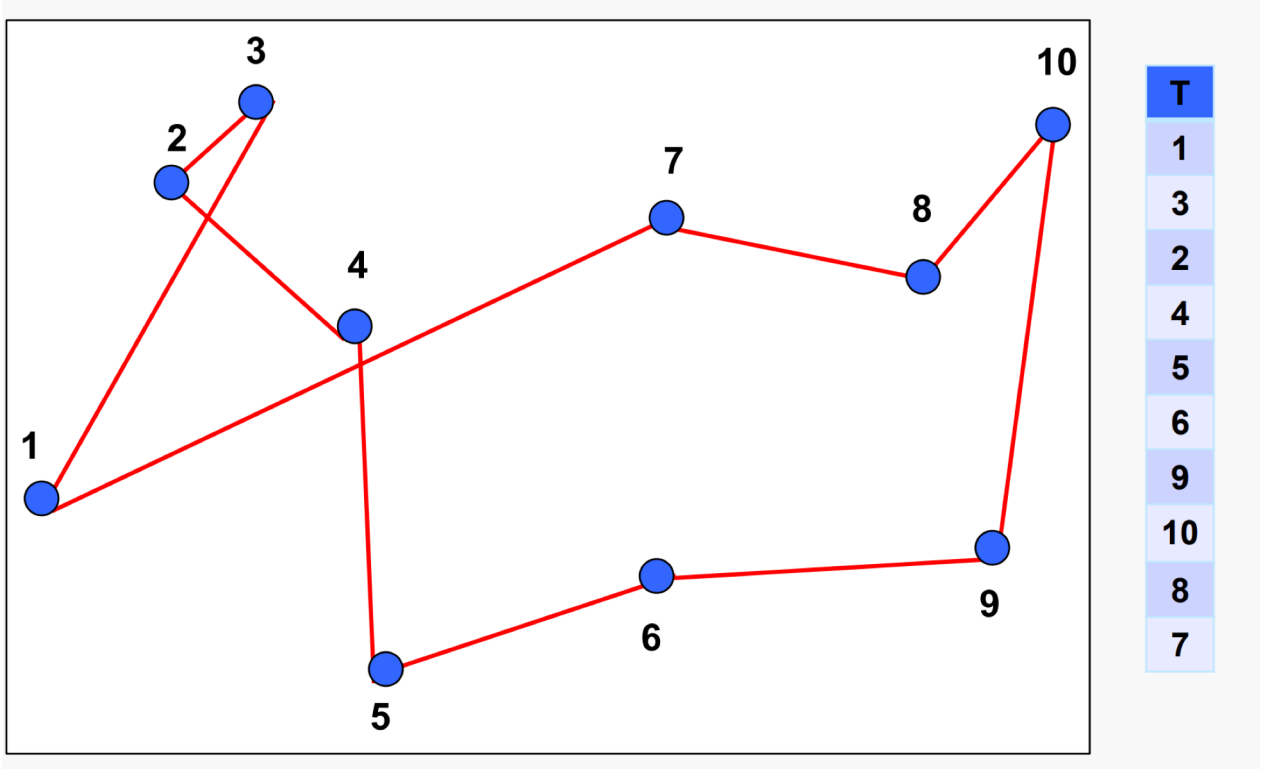
  Look for a 2-adjacent tour with lower cost than the current tour.

  If one is found then it replaces the current tour.

  This continues until there is a 2-optimal tour.

2-opt

Look for an improvement obtained by deleting two edges and adding two edges

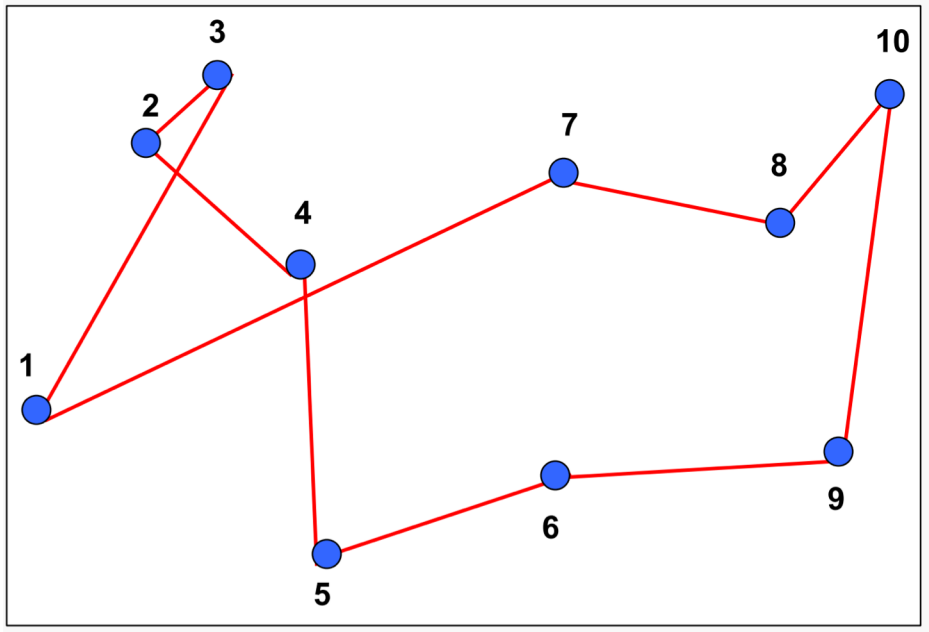


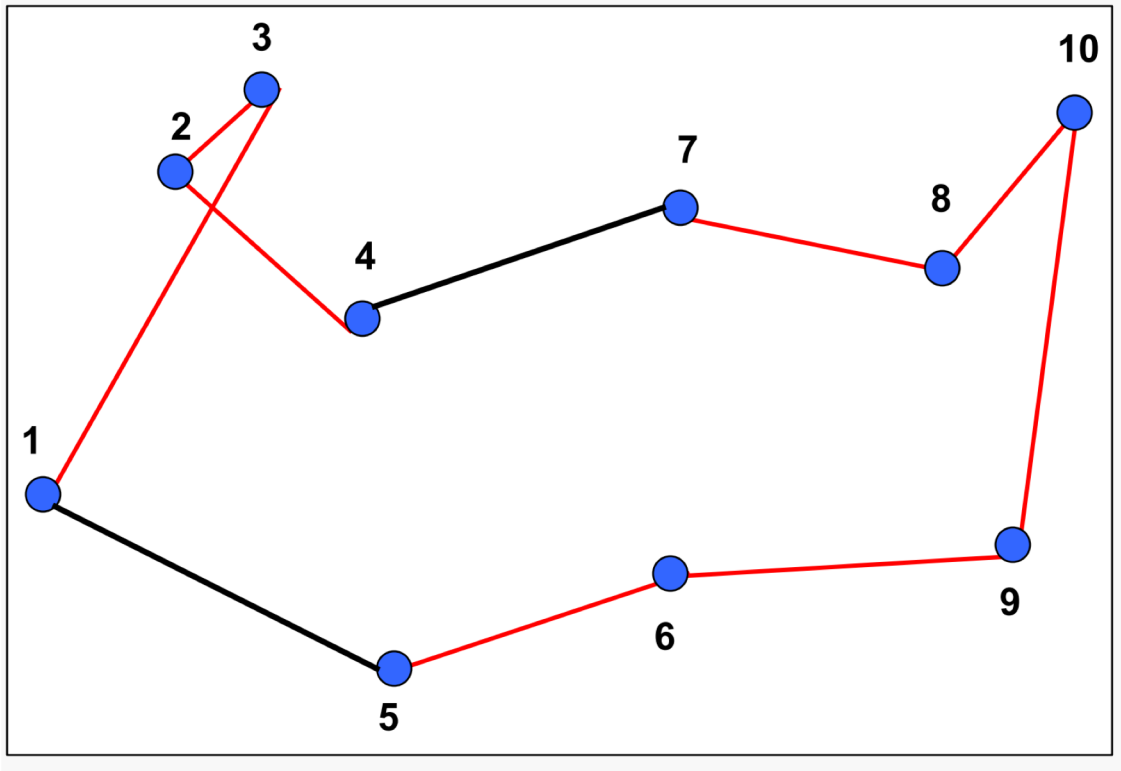
T
1
3
2
4
5
6
9
10
8
7

After the two exchange

Deleting arcs (1, 7) and (4,5) flips the subpath from node 7 to node 5

Note we don't need to recompute the full distance every time, just subtract removed edge distances and add new edge distances (since symmetric distances)

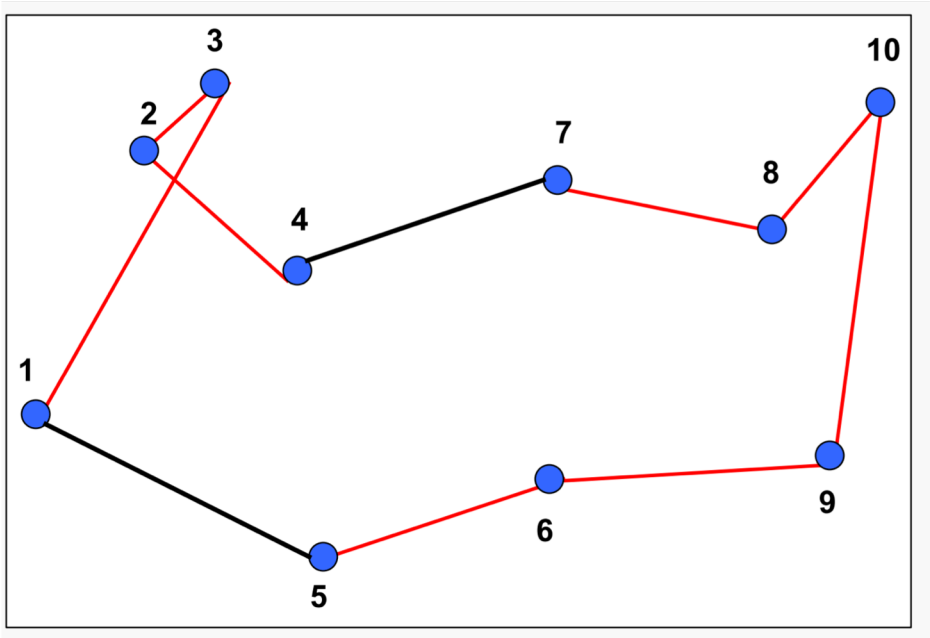


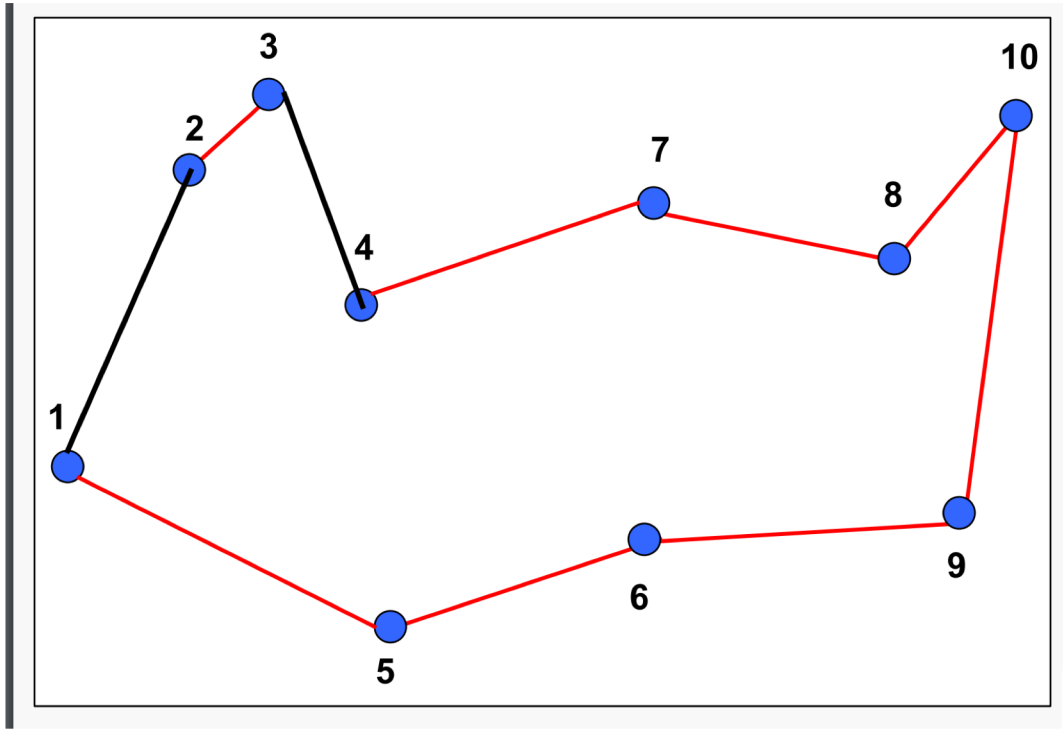


T_1	T_2
1	1
3	3
2	2
4	4
5	7
6	8
9	10
10	9
8	6
7	5

After the two exchange

Deleting arcs (1, 3) and (2,4) flips the subpath from node 3 to node 2

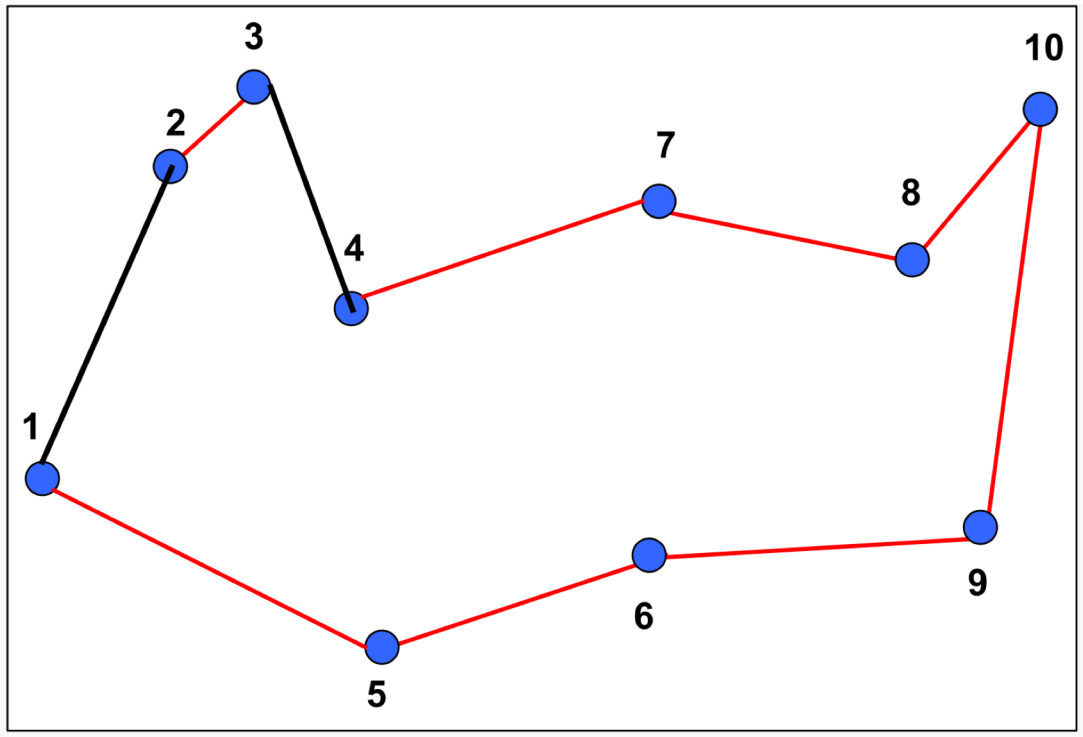


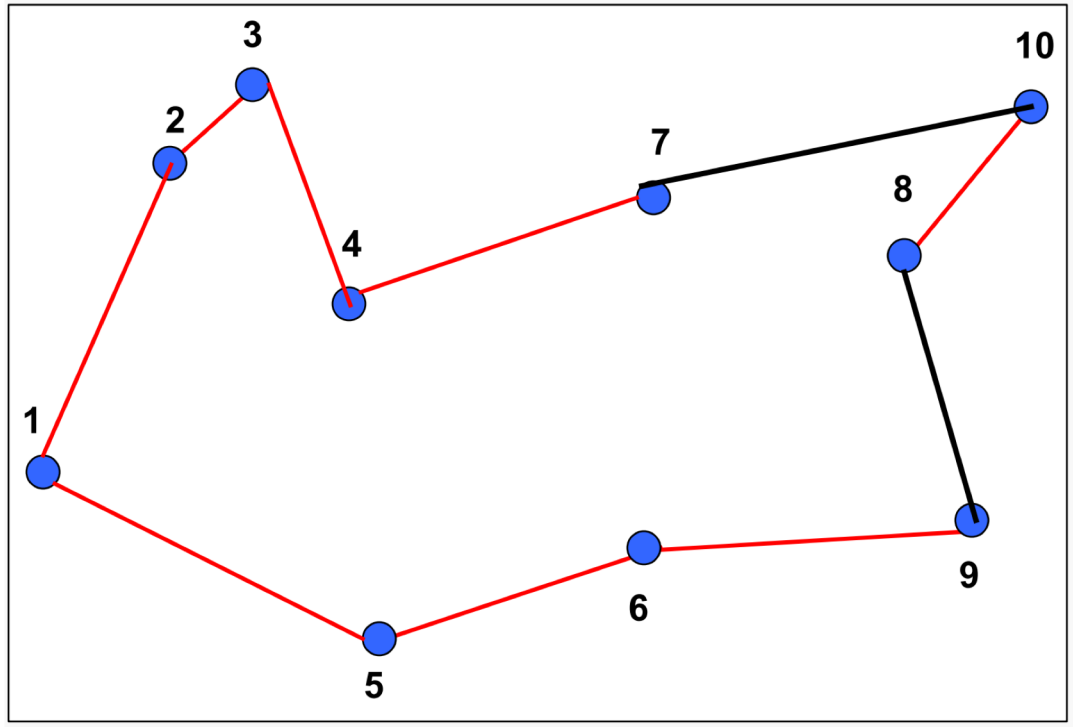


T_2	T_3
1	1
3	2
2	3
4	4
7	7
8	8
10	10
9	9
6	6
5	5

After the final improving two-exchange

Deleting arcs (7, 8) and (10,9) flips the subpath from node 8 to node 10

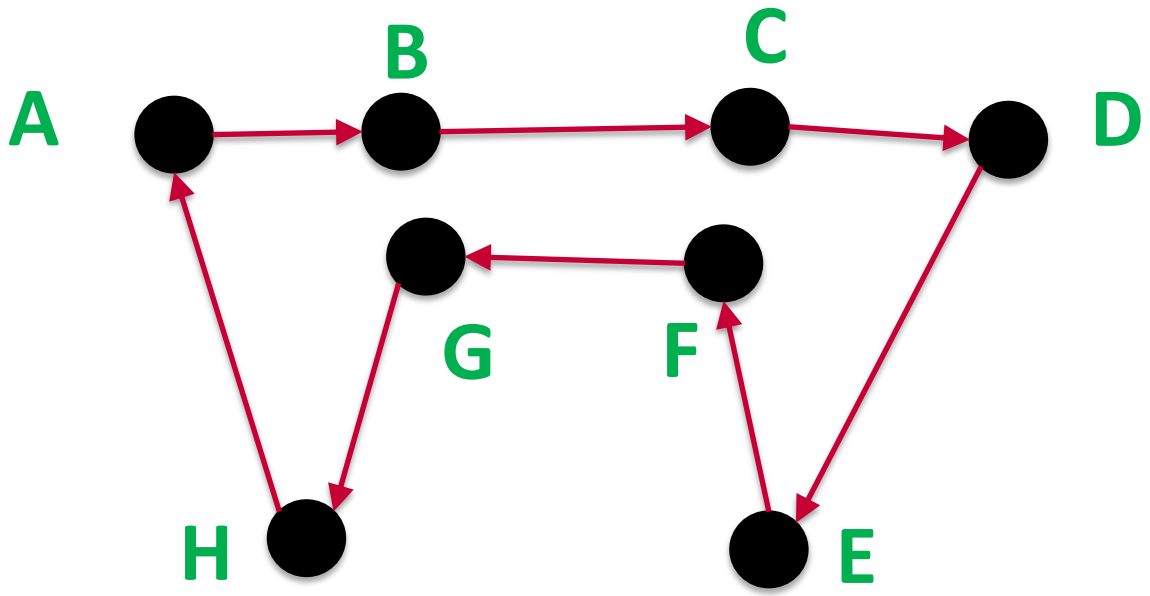




T_3	T_4
1	1
2	2
3	3
4	4
7	7
8	10
10	8
9	9
6	6
5	5

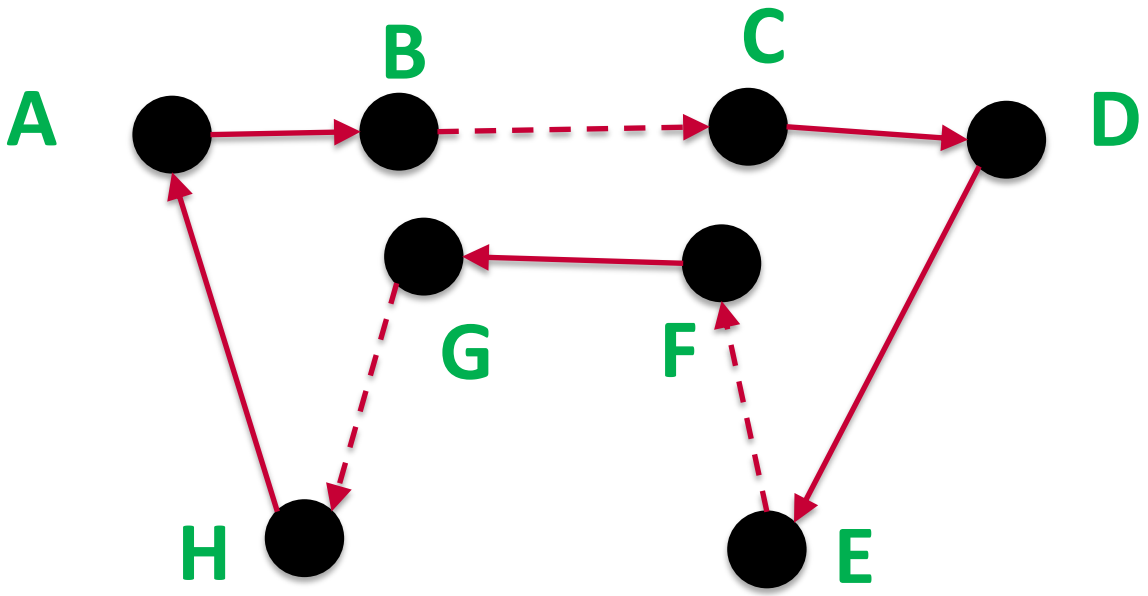
Local search for TSP: 3-OPT

- Task: find the shortest path to visit all cities exactly once
- Local Move:
  - Select three edges and replace them by three other edges
  - 7 different possibilities for replacement for valid tour (if including 3 2-opt moves!)



Local search for TSP: 3-OPT

- Task: find the shortest path to visit all cities exactly once
- Local Move:
  - Select three edges and replace them by three other edges
  - 7 different possibilities for replacement for valid tour (including 3 2-opt moves!)



Local search for TSP: 3-OPT

ABCDEGFHA

AHCDEFGBA

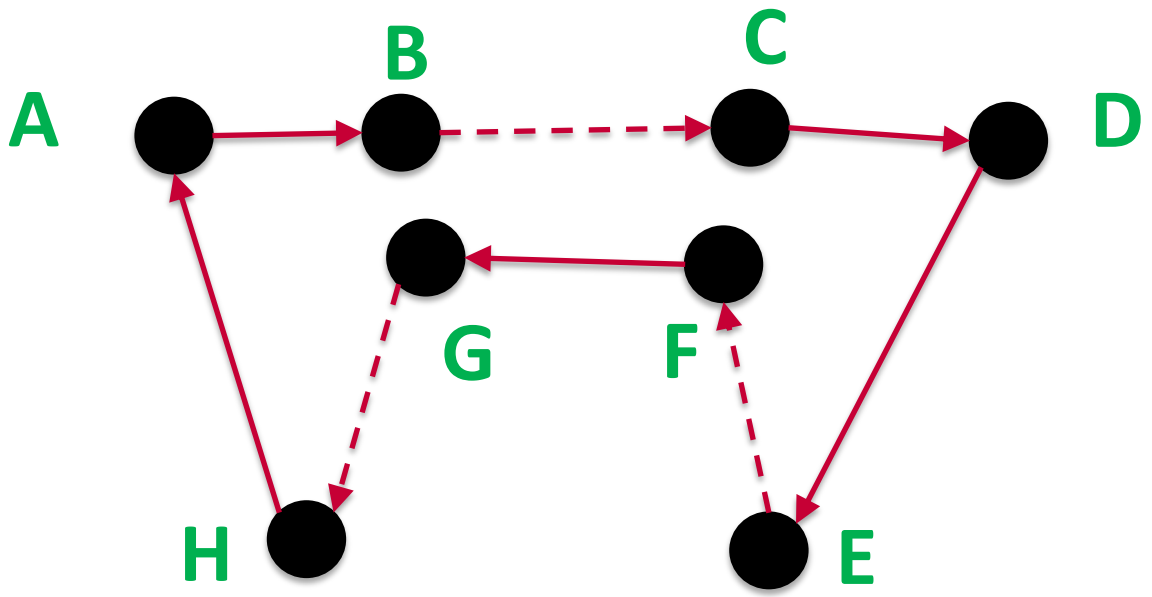
ABEDCFGHA

ABEDCGFHA

ABGFCDEHA

ABFGEDCHA

ABFGCDEHA



Local search for TSP: 3-OPT

ABCDEGFHA

ABEDCGFHA

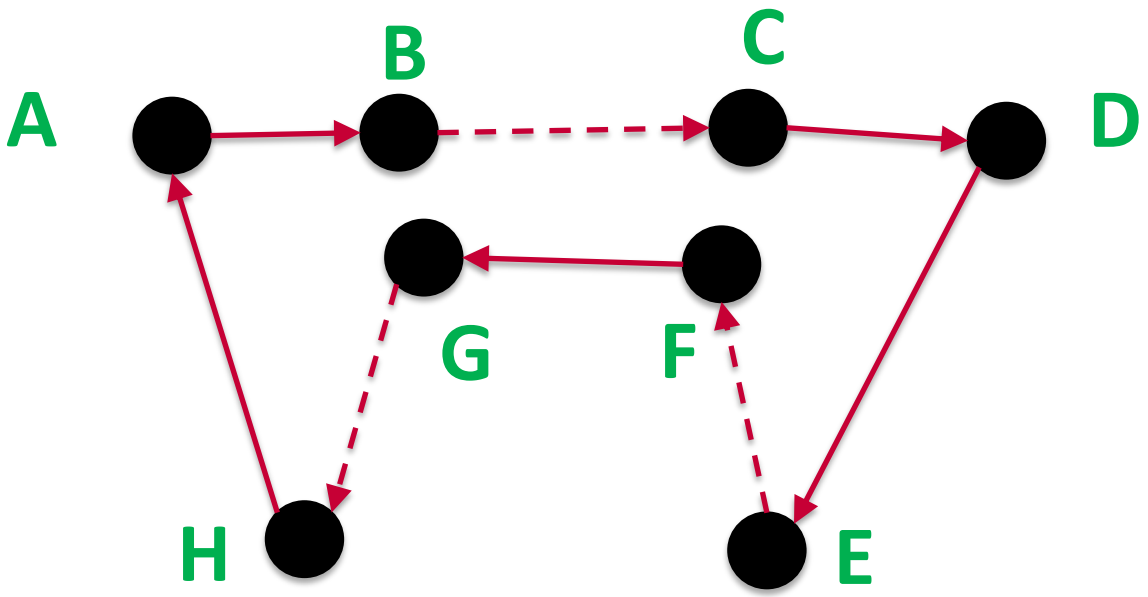
ABFGEDCHA

ABGFEDCHA

ABGFCDEHA

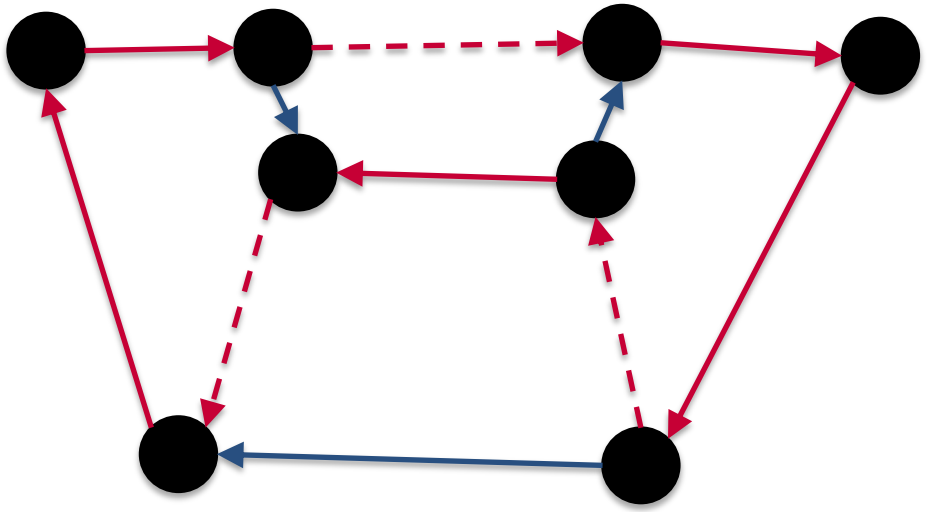
ABFGCDEHA

ABEDCFGHA



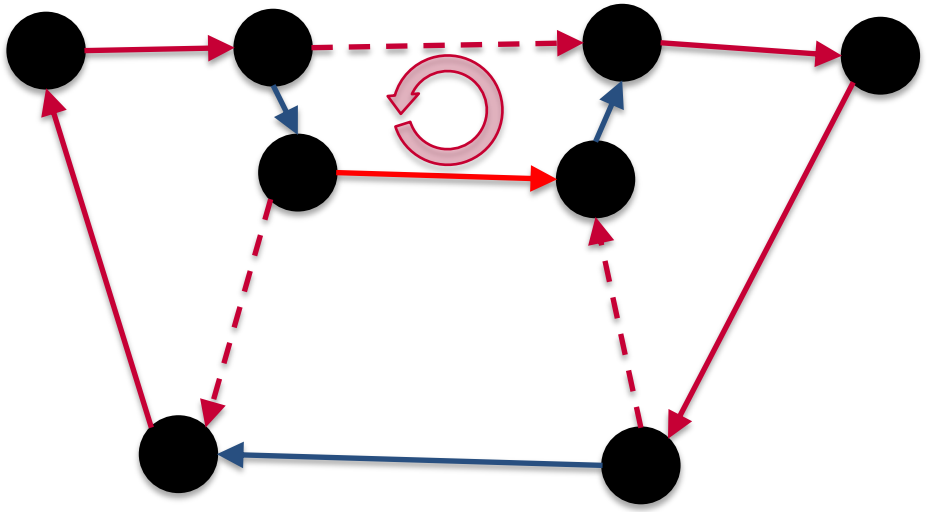
Local search for TSP: 3-OPT

- Task: find the shortest path to visit all cities exactly once
- Local Move:
  - Select three edges and replace them by three other edges
  - 7 different possibilities for replacement for valid tour (including 3 2-opt moves!)



Local search for TSP: 3-OPT

- Task: find the shortest path to visit all cities exactly once
- Local Move:
  - Select three edges and replace them by three other edges
  - 7 different possibilities for replacement for valid tour (including 3 2-opt moves!)



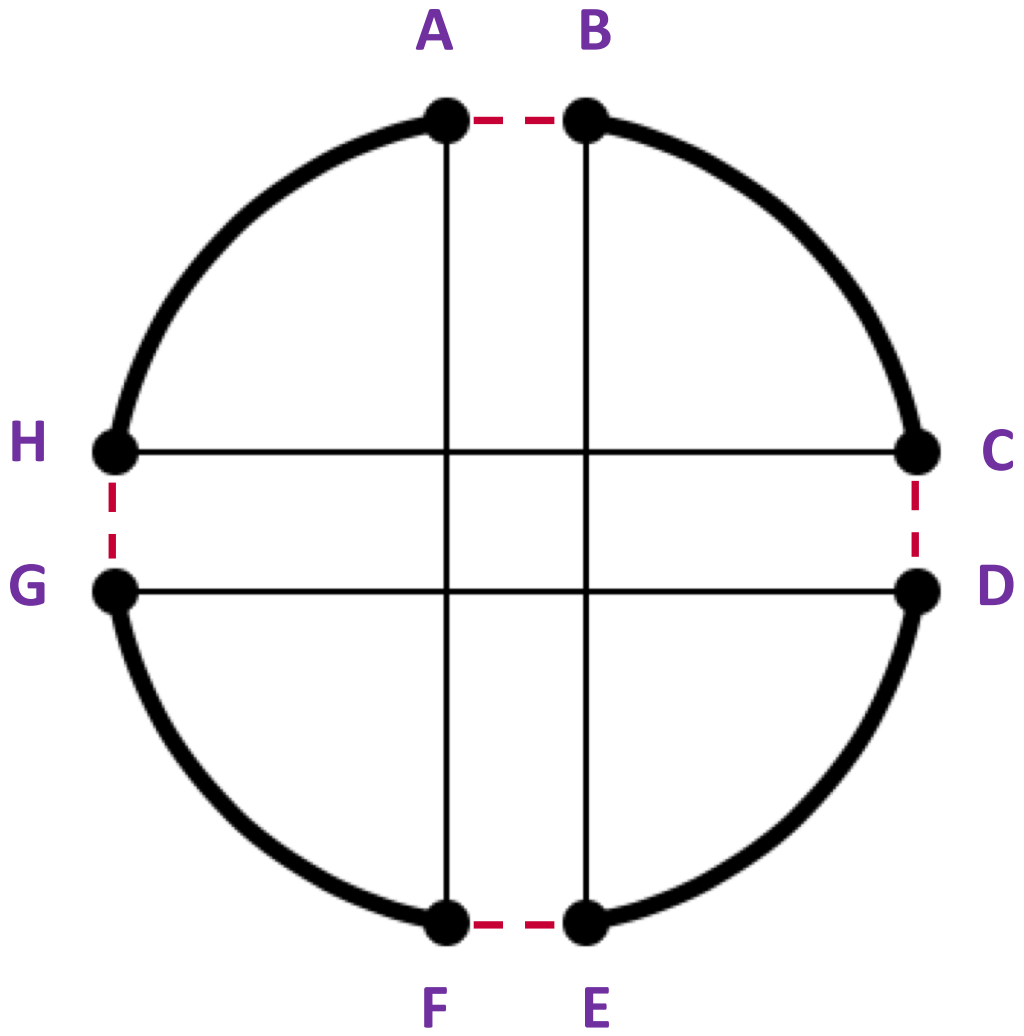
Local search for TSP: 2-Opt vs 3-Opt

- 3-opt move is same as sequence of 2-opt moves.
- However, it can find solutions that 2-opt wouldn’t!
- In particular, if one of the (non-last) sequence of 2-opt moves for our 3-opt move resulted in worse objective value.

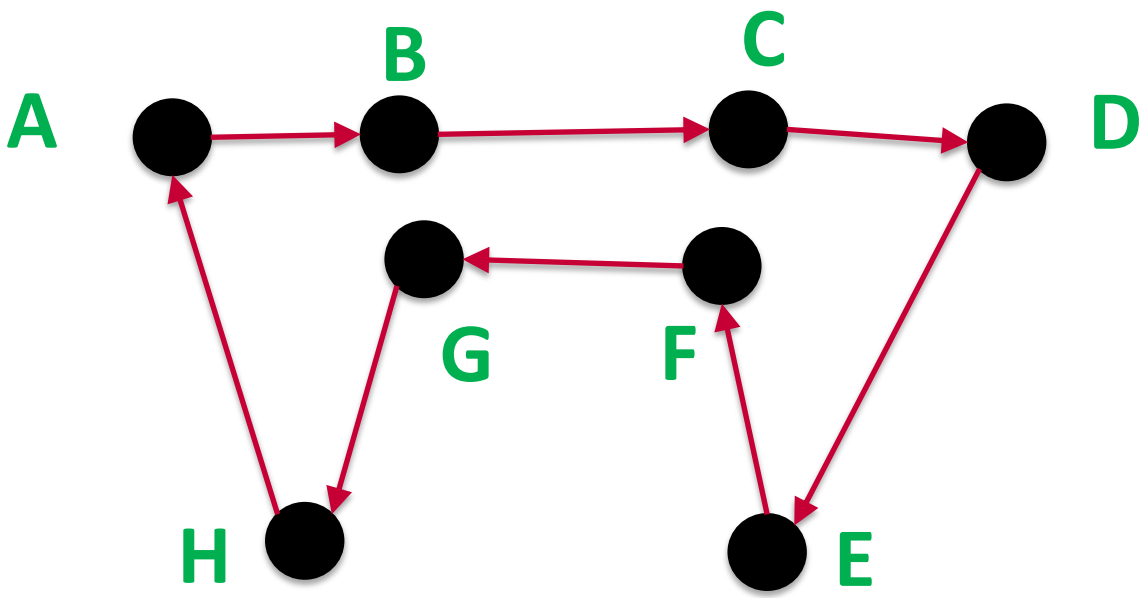
Local search for TSP

- 2-OPT:
  - Neighborhood is set of all tours reached by swapping two edges
- 3-OPT:
  - Neighborhood is set of all tours reached by swapping three edges
  - Much better than 2-OPT in quality but more expensive: $O(N^2)$ vs $O(N^3)$
- 4-OPT?
  - Neighborhood is set of all tours reached by swapping four edges
  - Marginally better but much more expensive $O(N^4)$
  - *Double bridge* move + 3-opt very popular

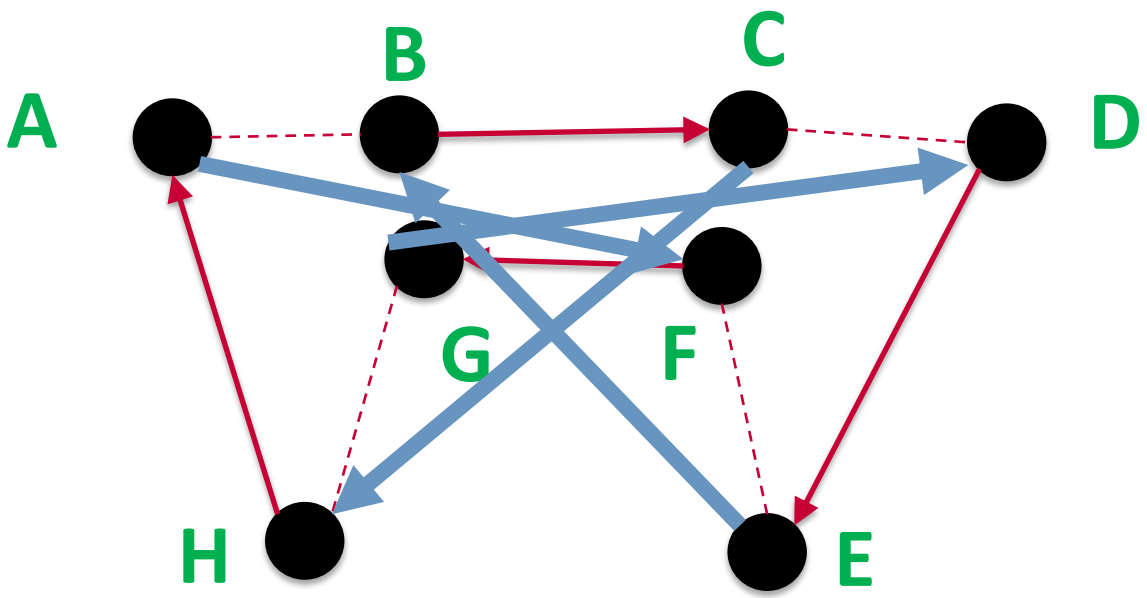
Local search for TSP: Double Bridge



Local search for TSP: Double bridge



Local search for TSP: Double Bridge



AFGDEBCHA