PG4200: Algorithms And Data Structures

Lesson 10: Decision and Optimization Problems

Prof. Andrea Arcuri

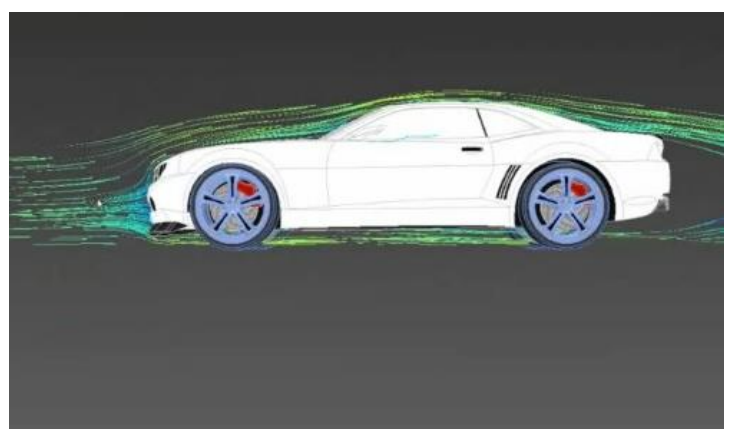
Runtime Of Algorithms

- Depending on input size N of the addressed problem
- *Polynomial* $O(N^k)$: usually fine, for small *k*
- *Exponential* $O(10^N)$: **hopeless**, unless tiny N
  - Eg, number of atoms in the whole universe is estimated to be no more than 10^{82}

Complex Problems

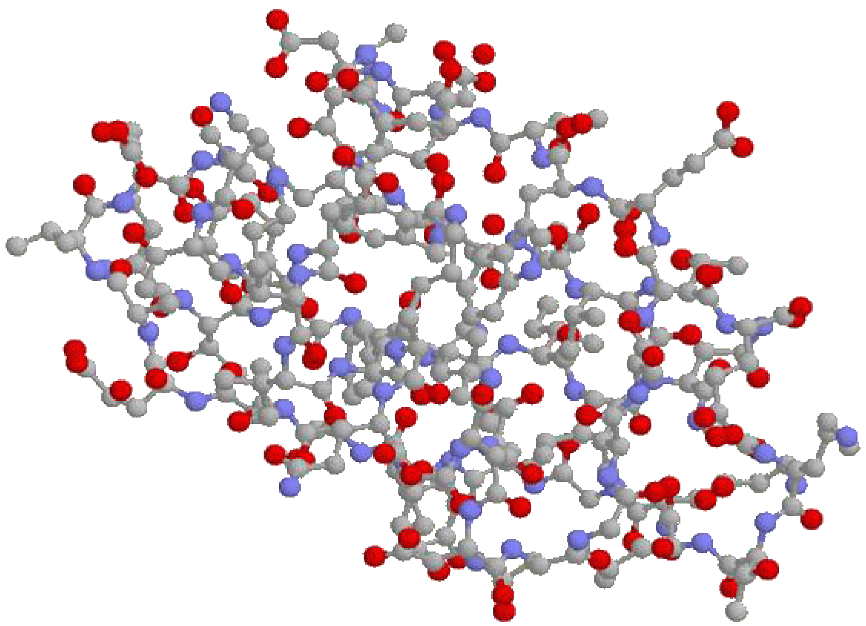
- There are a lot of problems in science and engineering for which we do not know any algorithm that can solve them in *polynomial* time
  - Such algorithms might exist, but we do not know them yet
- *Brute Force*: try all possible combinations, until find valid solution… but that is *exponential!!!*
- We need some *heuristics* to address these problems
  - But **no guarantee** that we can find a solution in reasonable time

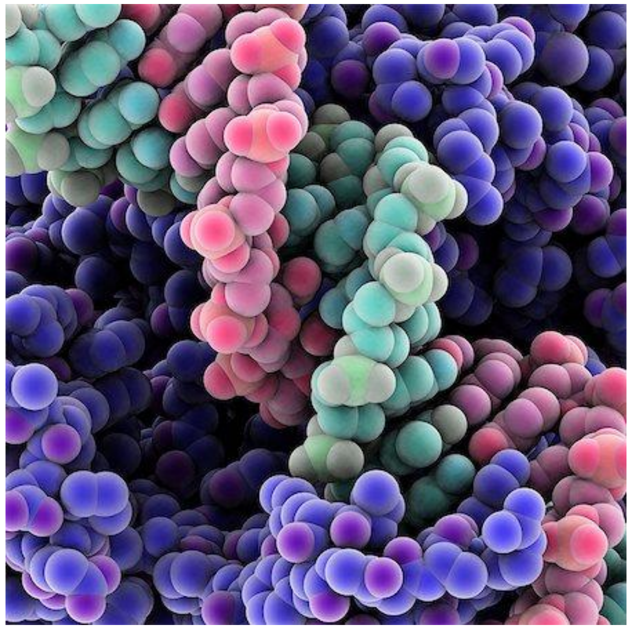
Vehicle Design



- How to find best shape to reduce air resistance?
- Can have different designs, and then test them in a wind tunnel

Protein Design



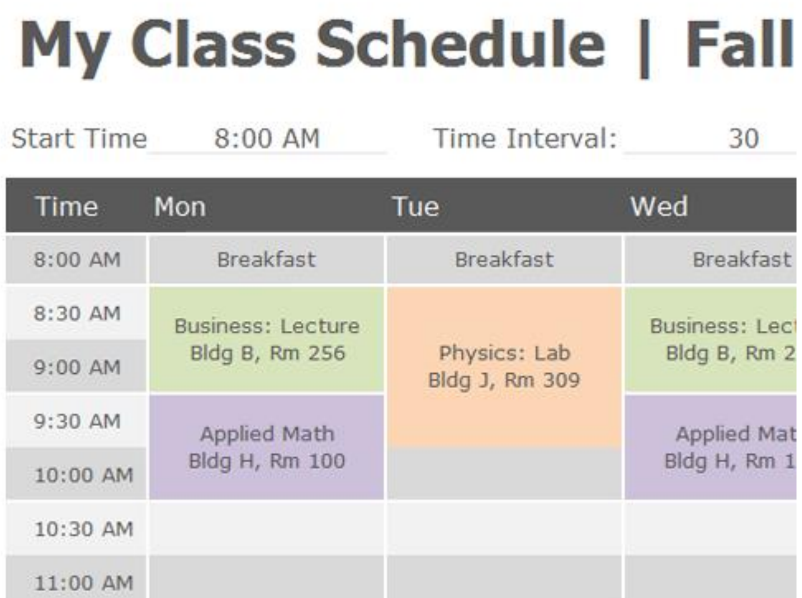


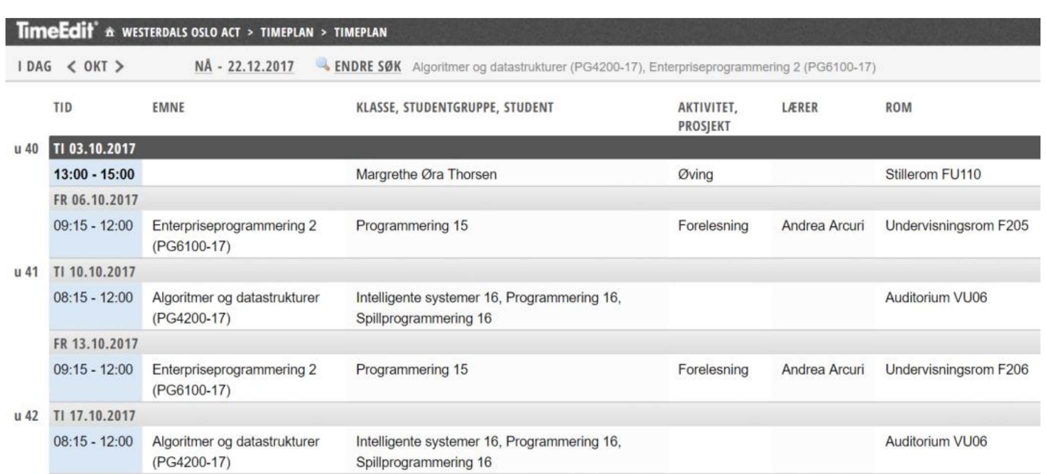
- How to find the right sequence of amino acids which will result in a protein with some sought properties?

Stock Market

- How to find best investment portfolio to maximize profit?

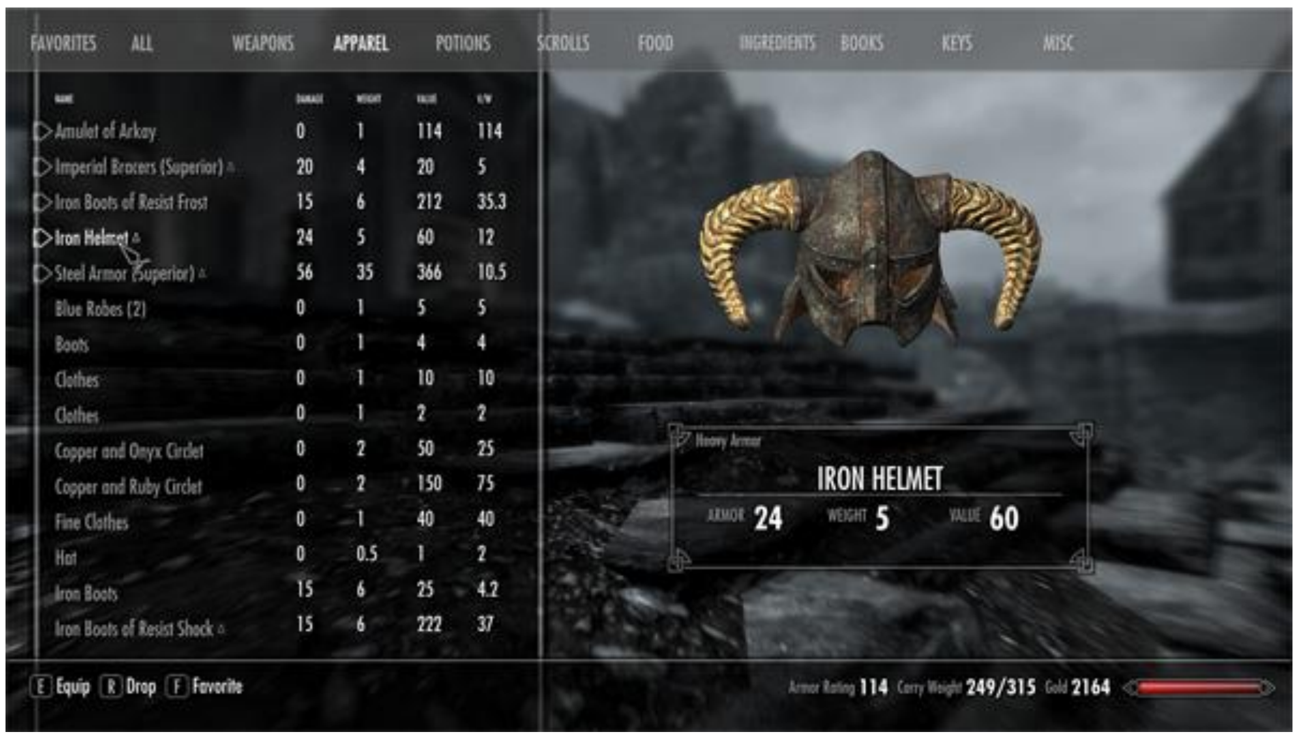
Class Schedule





- How to find best class schedule for which:
  - There is time for all classes
  - Classes in same year are not in parallel (ie conflicting)
  - Preferences of lectures are taken into account
  - Etc.
  - ?

RPG Equipment



- In RPGs, how find best combination of wearable items to maximize attack and defense under the constraints of maximum weight and item slots available?

Optimization Problem

- 2 main components: *Search Space* and *Fitness Function*
- **Goal**: find the best solution from the search space such that the fitness function is minimized/maximized

Search Space

- Set X of all possible solutions for the problem
- If a solution can be represented with 0/1 bit sequence of length N, then search space is all possible bit strings of size N
- Search space is usually huge, eg 2^N
  - Otherwise use brute force, and so would not be a problem

Fitness Function

- $f(x)=h$
- Given a solution *x* in X, calculate an heuristic *h* that specifies how good the solution is
- Problem dependent, to minimize or maximize:
  - Minimize air resistance
  - Maximize protein structure properties
  - Maximize Return Of Investment
  - etc.

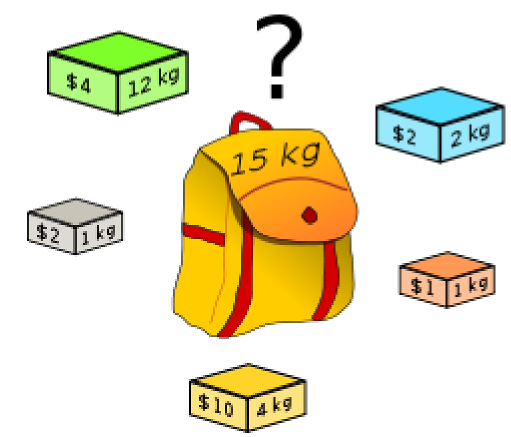
Optimization Algorithms

- Algorithm that explores the search space X
- Only a tiny sample of X can be evaluated
- Use fitness f(x) to guide the exploration to fitter areas of the search space with better solutions
- Stopping criterion: after evaluating K solutions (or K amount of time is passed), return best *x* among the evaluated solutions
- Many different kinds of optimization algorithms…
  - But as a user, still need to provide the representation and f(x)

Search Operator

- $s(x) = x'$
- An operator that, from a solution *x*, gives a new one *x'*
- Still need to evaluate its fitness, ie $f(x')$
- The optimization algorithm will use the search operators to choose which new *x'* in X to evaluate
- The search operator will depend on the problem representation
- Example: flip a bit in a bit-sequence representation

Example:
Knapsack Problem (KP)



- Insert N items to a knapsack
- Each item has a weight *w* and a value *v*
- The knapsack has a maximum load of weight L
- Goal: find the selection of items that can be inserted within limit L, and for which the total value is maximized
- Note: many real-world problems are instances of the knapsack problem

Details

- Each unique item has an index from 0 to N-1
- A solution can be represented as an array *x* of 0s (item not present) and 1s (item present)
- Maximize: $f(x) = \sum_{i=o}^{N-1} v[i] * x[i]$
- Constraint: $\sum_{i=o}^{N-1} w[i] * x[i] \ \leq L$
- Can have f(x)=0 if constraint is not satisfied

Brute Force

- Given size N, enumerate all possible bit arrays
- Return the one with maximum f(x)
- Astronomically expensive, but for tiny N
  - $2^{10} = 1024$
  - $2^{20} =$ 1,048,576
  - $2^{50} \cong 1{,}120{,}000{,}000{,}000{,}000$
  - etc

Greedy Algorithm

- Build a solution as quickly as possible
- Don’t explore the search space, but rather focus on the most promising path in it
- Actual implementation is problem dependent
- For example on KP:
  - Start from empty selection *x*
  - Add 1 item at a time to *x* (but how to choose???)
  - Stop when not possible to add any item

KP Example

- L = 26
- W = [12, 7, 11, 8, 9]
- V = [24, 13, 23, 15, 16]
- From https://people.sc.fsu.edu/~jburkardt/datasets/knapsack_01/knapsack_01.html

Greedy: Heaviest First

X	1	0	1	0	0
W	12	7	11	8	9
V	24	13	23	15	16

- First choose (12,24)
- Then choose (11,13)
- Weight becomes 12 + 11 = 23
- Cannot add any other element without exceeding L=26
- f(x) = 24 + 23 = 47
- Is 47 the best score we can achieve???

Greedy: Lightest First

X	0	1	0	1	1
W	12	7	11	8	9
V	24	13	23	15	16

- Choosing (7,13), (8,15) and (9,16)
- Weight becomes 7 + 8 + 9 = 24 < 26
- Cannot add any other element without exceeding L=26
- f(x) = 13 + 15 + 16 = 44
- Worse solution 44 than previous 47

Greedy: Best Ratio

X	1	0	1	0	0
W	12	7	11	8	9
V	24	13	23	15	16
Ratio	2.00	1.85	2.09	1.87	1.77

- Consider first the best ratio v/w, ie which item gives best return for unit of weight
- Choose (11,23) and then (12,24)
- Cannot add any other element without exceeding L=26
- f(x) = 24 + 23 = 47
- Different order of insertion, but still 47

Best Solution

X	0	1	1	1	0
W	12	7	11	8	9
V	24	13	23	15	16

- Weight is 7 + 11 + 8 = 26
- f(x) = 13 + 23 + 15 = 51
- Better than the previous 47
- Greedy algorithms can be fast, but can give poor results
- Need something more general, with better results

General Optimization Algorithms

- **Random Search** (in this class)
- **Hill Climbing** (in this class)
- Simulated Annealing
- **Genetic Algorithms** (next class)
- Ant Colony Algorithms
- Particle Swarm Algorithms
- Etc. etc. (there are many)

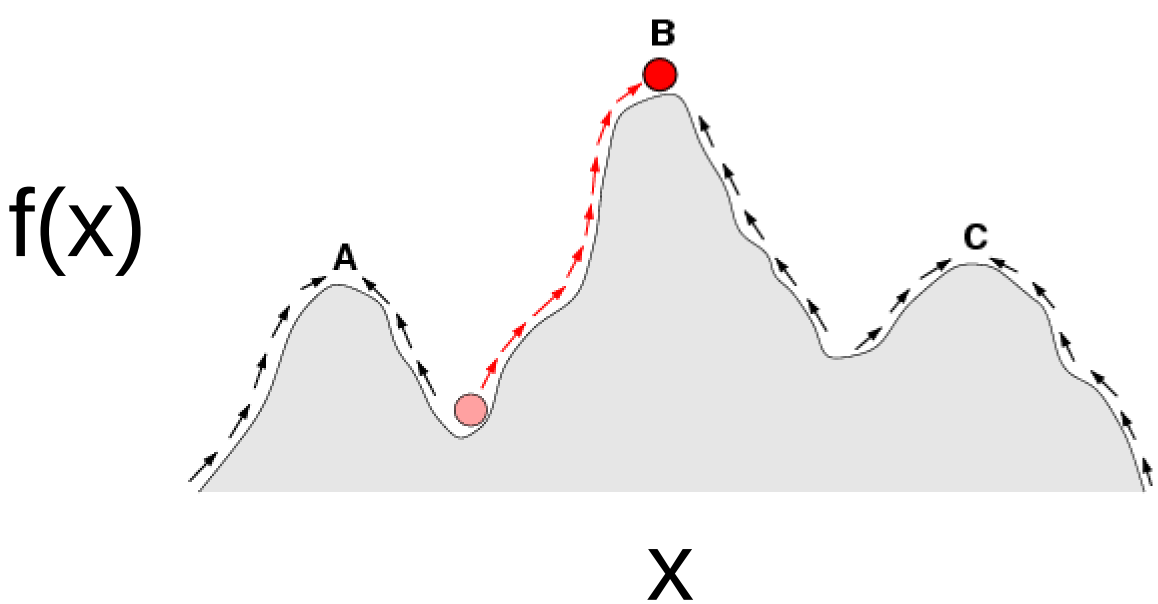
Random Search (RS)

- Easiest of the optimization algorithms
- Sample a random solution from search space X
- Keep sampling until run out of time
- Keep track of best solution found so far

Hill Climbing (HC)

- Start from a random solution
- Keep track of / store one solution
- Use search operator to do small modifications
- If better solution, *move* to it, and repeat
- If no better solution in the *neighborhood*, restart from a random solution

Very Simplified Example



Fitness Landscape

- Consider problem in which *x* is a number
- Search operator is +- 1 on such *x*
- “Climb” up to maximize *f(x)*
- *B* is *global optimum*
- *A* and *C* are *local optima*
- Final result depends from starting point

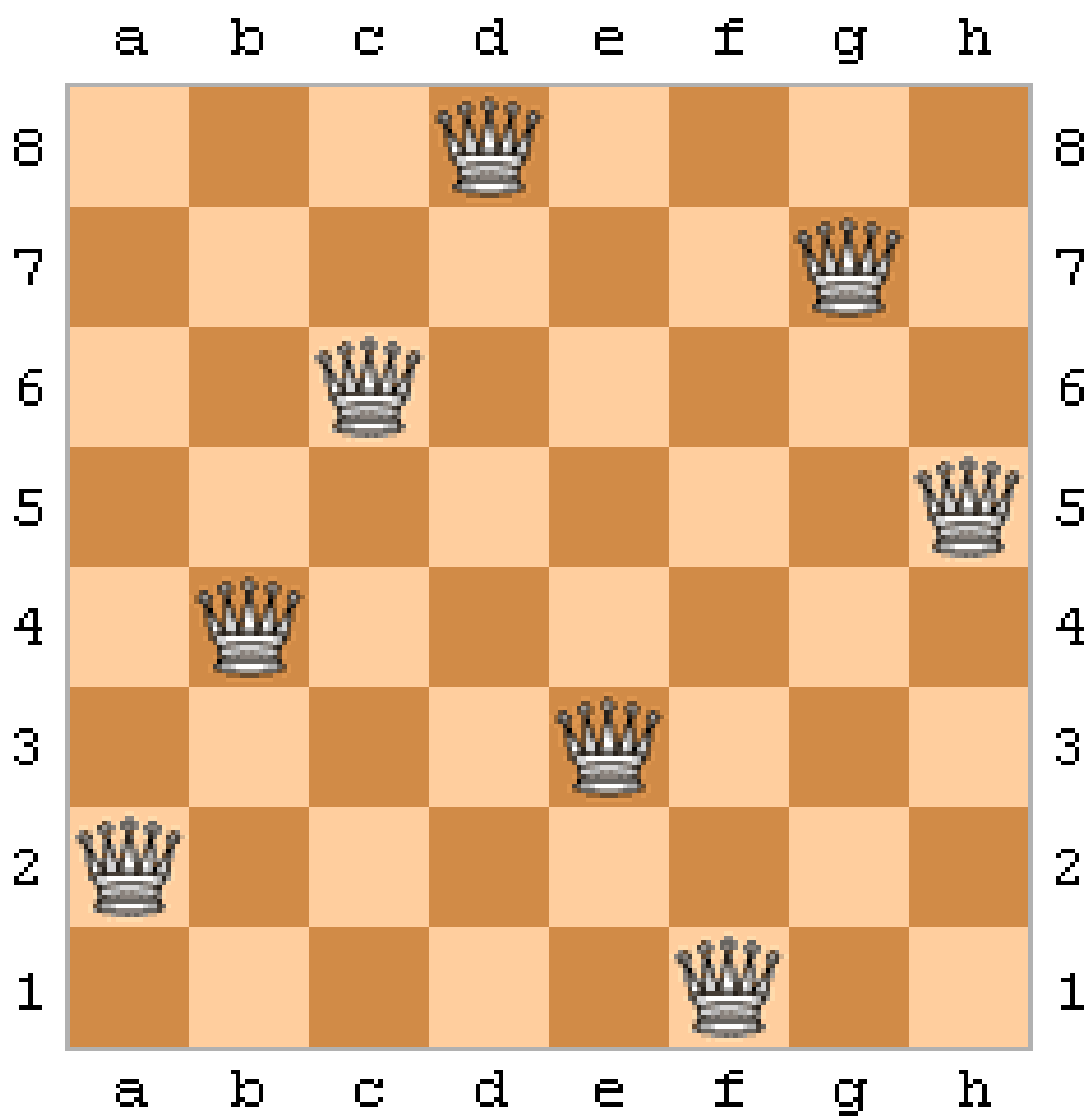
HC for KP

- HC is a general algorithm
- But still need to define a proper search operator for each problem domain
- Example:
  - add/remove 1 item (small neighborhood)
  - remove K items, add J different items (larger neighborhood)
- The larger the neighborhood, the slower the ascent, so the less restart we can do within same time
  - Ie, it is not necessarily better

No Free Lunch (NFL) Theorem

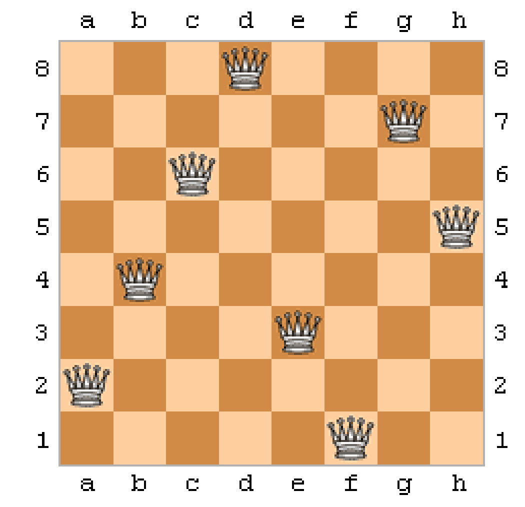
- What is the best optimization algorithm?
  - Which best variants / choice of parameters?
- Considering *all* optimization problems, mathematically proved (NFL) that **all optimization algorithms perform on average the same**
  - Yes, it follows that, on some problems, RS is the best
- There exist no best algorithm
- But on *specific* problems, you can have some algorithms that are better than others, especially by exploiting *domain knowledge*
- It follows that a general algorithm will perform worse than a specialized variant for a specific problem

Queen Puzzle (QP)



- Position 8 queens such that no 2 queens threaten each other
- Generalization: N queens into a N*N board

QP As Optimization



- Search Space: matrix N*N of bits
  - 1 for a queen in that position, 0 otherwise
- Search operator: flip bits in the matrix
- Fitness: need to reward having N queens, and minimize number of threatened queens

00010000
00000010
00100000
00000001
01000000
00001000
10000000
00000100

QP: Better Representation

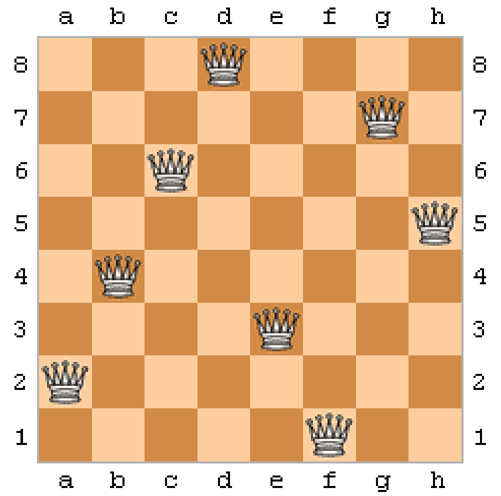
- Binary matrix N*N would allow for any number of queens on the board (eg all 1s)

- Domain knowledge: no 2 queens on same row, no 2 queens on the same column
  - Choose representation and search operator to only explore solution for which these constraints are satisfied

- New representation: array of size N, with column indices from 0 to N-1 (position *i* is for queen in row *i*)

- New operator: swap 2 column indices

QP Cont.



- Representation: [f,a,e,b,h,c,g,d]
  - Eg, queen in row 1 is in column *“f”,* row 2 is in column *“a”*, etc
- Search operator: swap two elements
  - Eg, swap “f” with “c”, [c,a,e,b,h,f,g,d]
- By construction, I am only exploring board configurations that do not clash on columns/rows
  - But still a problem, as need to handle threating on diagonals

Decision Problem

- Technically, QP is a *decision* problem
- Once we find a solution with N queens no threating any other, we know we have found a global optimum
- In optimization problems, usually we cannot know if we found the best
- Decision Problem: can say “yes” or “no” about if a solution is optimal or not
- For decision problems, still doing the same as optimization, only difference is that we can know when best solution is found
- Practically all optimization problems have a decision variant for some metric K, eg “find knapsack configuration with total item values of at least K”

NP

- **NP**: “**N**ondeterministic **P**olynomial time”
- **NP** is “the set of all *decision* problems that can be solved in polynomial time on a theoretical non-deterministic Turing machine”
- Equivalent, easier definition: “*set of all decision problems whose solutions can be verified in polynomial time*”
  - Ie, can answer “yes” or “no” in polynomial time
- KP and QP are in **NP**:
  - QP: can quickly verify if N queens do not threaten each other
  - KP: can quickly verify in linear time if a solution has at least a certain value

P

- **P** is the set of all decision problems that can be *solved* in polynomial time
- **P** is at least a subset of **NP**… but is it a strict subset???
- **P = NP ???**
- **P != NP ???**
- *This is arguably the most important question in computer science for which we do not have an answer (yet)*
- Consequence: there might be undiscovered, efficient algorithms to solve today’s complex problems, or those might be impossible to design… we simply have no clue ☹

Homework

- Study Book pages 910-921
  - Note: details of optimization algorithm are not in the book
- Study code in the *org.pg4200.les10* package
- Do exercises in *exercises/ex10*