Combinatorial optimization

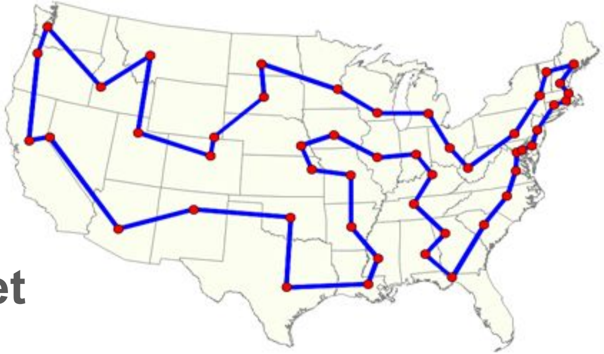
Stanislav Protasov

Agenda

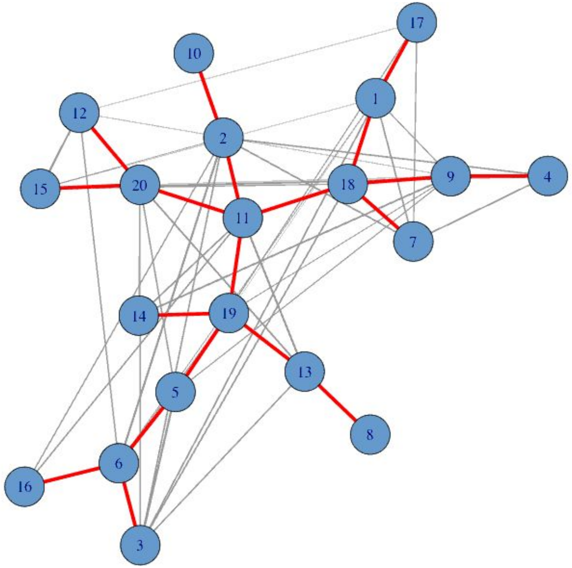
- Problem area overview
- Brute force and optimizations
- Matroids
    - What is this
    - Rado-Edmonds greedy algorithm and theorem
    - Matroid types and applications
- Optimization that is not optimal

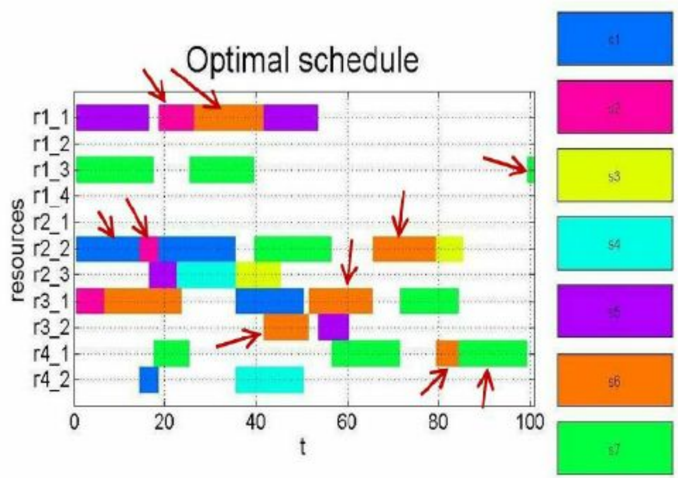
Combinatorial optimization approaches

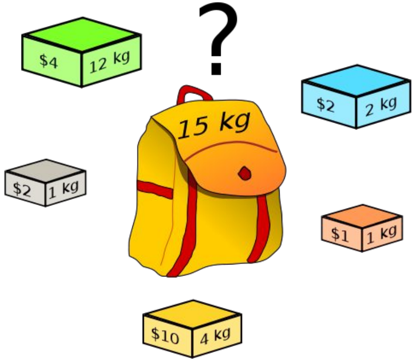
What is combinatorial optimization



Finding optimal (for some metric) element of a **finite set**







0	a_7
1	a_8
00	a_9
01	a_1
10	..
11	a_1

Brute force: subsets

- Subset (combinations) generation
  - *visitor* processing
    - Backtracking (DFS): binary tree of “take/don’t take” of K-depth
  - stream processing
    - Bit arrays

Subsets: knapsack

```
def visit(b):
    return sum(i[0] for i in b), sum(i[1] for i in b)

def depth_search(loot, bag, depth):
    global nodes_count, best
    nodes_count += 1
    if depth == len(loot):
        w, c = visit(bag)
        wb, cb = visit(best)
        if c > cb and w <= limit:
            best = list(bag)
    else:
        depth_search(loot, bag, depth + 1)
        bag = bag + [loot[depth]]
        depth_search(loot, bag, depth + 1)
```

Branches-and-bounds

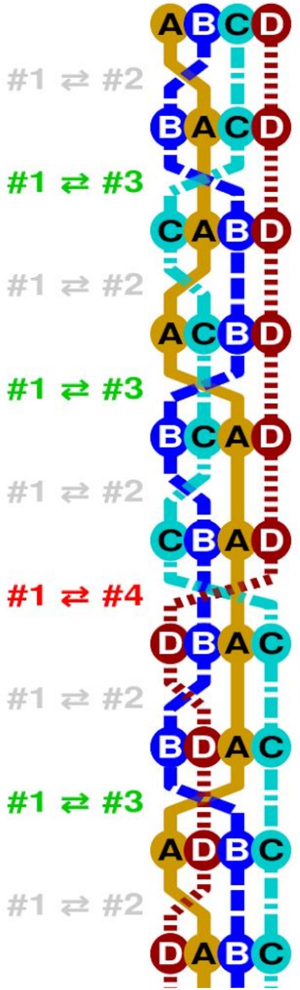
```
def visit(b):
    return sum(i[0] for i in b), sum(i[1] for i in b)

def depth_search(loot, bag, depth):
    global nodes_count, best
    nodes_count += 1
    if depth == len(loot):
        w, c = visit(bag)
        wb, cb = visit(best)
        if c > cb and w <= limit:
            best = list(bag)
    else:
        depth_search(loot, bag, depth + 1)
        bag = bag + [loot[depth]]
        w, c = visit(bag)
        if w > limit: return          # optimization 1
        # optimization 2: branch-and-bound, require sorted loot
        if c + (limit - w) / bag[-1][0] * bag[-1][1] <= visit(best)[1]: return
        depth_search(loot, bag, depth + 1)
```

Brute force: permutation generation

Heap's algorithms: generate next permutation by swapping 2 elements

```
procedure generate(k : integer, A : array of any):
    if k = 1 then
          output(A)
    else
        for i := 0; i < k; i += 1 do
            generate(k - 1, A)
            if k is even then
                swap(A[i], A[k-1])
            else
                swap(A[0], A[k-1])
            end if
        end for
    end if
```



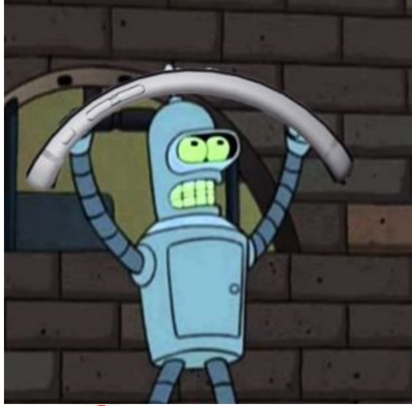
Dynamic programming: integer programming

If we search for a solution in discrete space of **values** (knapsack cost is integer)

Then, instead of thinking about the problem as a ***combinatorial task for input***, consider search space of possible ***integer outputs*** (which is much smaller)

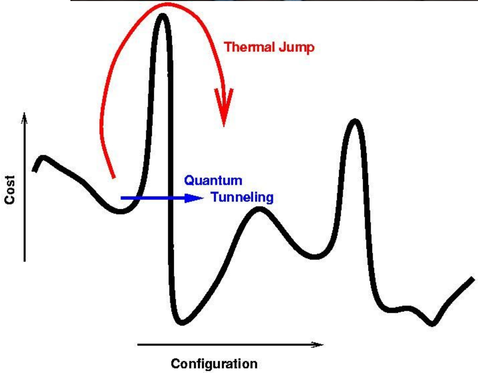
Knapsack: O(N * sum(cost))

Simulated annealing, Quantum annealing



Annealing: you bend metal, then you want to *relax tension*

Idea: atoms are faster and mobile in hotter metal. *Slowly* cool down the metal to let them settle in energetically optimal places



Simulated annealing: *probabilistically* decide to move to neighbouring state based on the idea of energy minimization

Quantum annealing: the same idea, but *tunneling field strength* is used instead of temperature

D:WAVE
The Quantum Computing Company™

General quantum computers

`f(x) = y` - satisfaction of **L**-digits Boolean function (y = 1), where `f(x)` is a *black box*; inversion of f(x)

Problem statement: Search for **x** among possible inputs

Classic solution: brute force in $O(2^L)$ iterations

Grover quantum algorithm idea: iteratively increase amplitude of “correct” quantum state. Achieves result in $O(2^{L/2})$ iterations with **L** qubits.

Matroids

Matroid definition (1)

Matroid = ordered pair (E, I)

E - finite set called **ground set**

I - subset of $\mathbf{2}^{E}$ called
"independent" sets

Matroid definition (2)

1) *Empty set is independent* $\emptyset \in I$
2) Any *subset of independent set* is also *independent*
   $M \in I \rightarrow \forall (M' \subset M)\ M' \in I$
3) *All biggest independent sets are of the same size (called **rank**)*
   $A, B \in I,\ |A| > |B| \rightarrow$
   $\rightarrow \exists x \in A \setminus B,\ B \cup \{x\} \in I$

Matroid theory terms

X is a **dependent set**, if ***X*** is a subset of ***E***, but not in ***I***.

Maximal independent set ***M*** (means $M \cup \{x\}$ - dependent) is called **basis**.

Circuit *C* is a dependent set such that $\forall (C' \subset C)\ C' \in I$

Rado-Edmonds Theorem (preparation)

Let's assign weight ***w*(*x*)** to each ***x*** in ***E***.

Then weight ***w*(*M*)** of $\boldsymbol{M \in \mathcal{I}}$ is $w(M) = \Sigma_{x \in M} w(x)$

Rado-Edmonds Theorem (**greedy** algorithm)

```
Sorted = sort x in E by w(x) [asc|desc]
A = ∅
for i from 1 to |E|:
   if A ∪ {Sorted[i]} ∈ I:
      A = A ∪ {Sorted[i]}
return A
```

Rado-Edmonds Theorem proof notes

Theorem:
Algorithm finds a **basis** *A* **of minimal** (maximal) **weight** ***w*(*A*)**

$A = \{a_1, a_2, \dots, a_n\}, w(a_1) \le w(a_2) \le \dots \le w(a_{\mathbf{n}})$
$B = \{b_1, b_2, \dots, b_k\}, w(b_1) \le w(b_2) \le \dots \le w(b_{\mathbf{k}}), k \le n$
$X = \{a_1, a_2, \dots, a_{\mathbf{i-1}}\}$
$Y = \{b_1, b_2, \dots, b_{\mathbf{i-1}}, b_{\mathbf{i}}\}, i \le k$

$w(b_j) \le w(b_i)$
$w(a_j) \le w(b_j) \quad \Rightarrow \quad \mathbf{w(a_i) \le w(b_i)}$

Matroid method idea

1. Show that a **problem model** is a **matroid** (apply to definition)
2. This allows you to **apply** Rado-Edmonds **theorem** to your problem
3. **Implement** Rado-Edmonds **greedy** algorithm for your case as an **optimal** solution

Matroid types

Graphic interpretation

For weighted undirected graph G=(V, E) with *no loops and no parallel edges* with defined `w(e)` for e ∈ E:

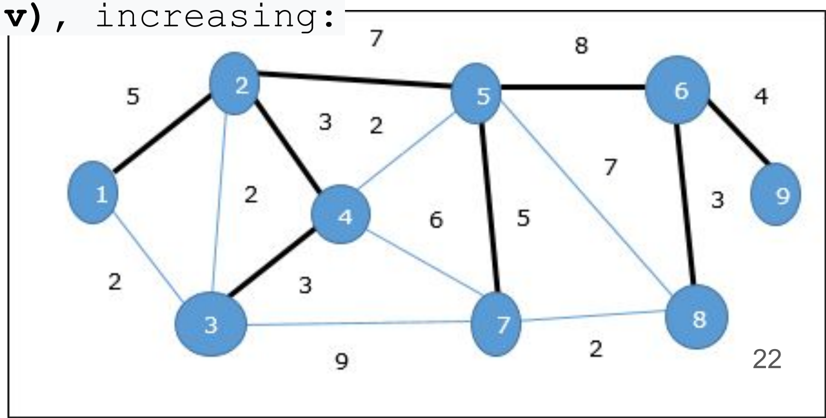
1. Let E be a ground set
2. Let a **set of all possible forests** in G be I (independent sets). In other words, “independent” = “acyclic subgraph”
    a. Empty set of edges is acyclic
    b. Any subset of forest (acyclic graph) is a forest
    c. If for a forest A there is a bigger forest B:
        i. $A \subset B \Rightarrow$ take any x from B\A
        ii. $A \not\subset B \Rightarrow$ there is as least one edge with a vertex not present in A. Attach it.

… consequence

The biggest forest of maximal weight can be found using greedy approach.

Kruskal's algorithm is exactly Rado-Edmonds algorithm applied to trees

```
KRUSKAL(G):
A = ∅
foreach v ∈ G.V:
    MAKE-SET(v)
foreach (u, v) in G.E ordered by weight(u, v), increasing:
    if FIND-SET(u) ≠ FIND-SET(v):
        A = A ∪ {(u, v)}
        UNION(FIND-SET(u), FIND-SET(v))
return A
```

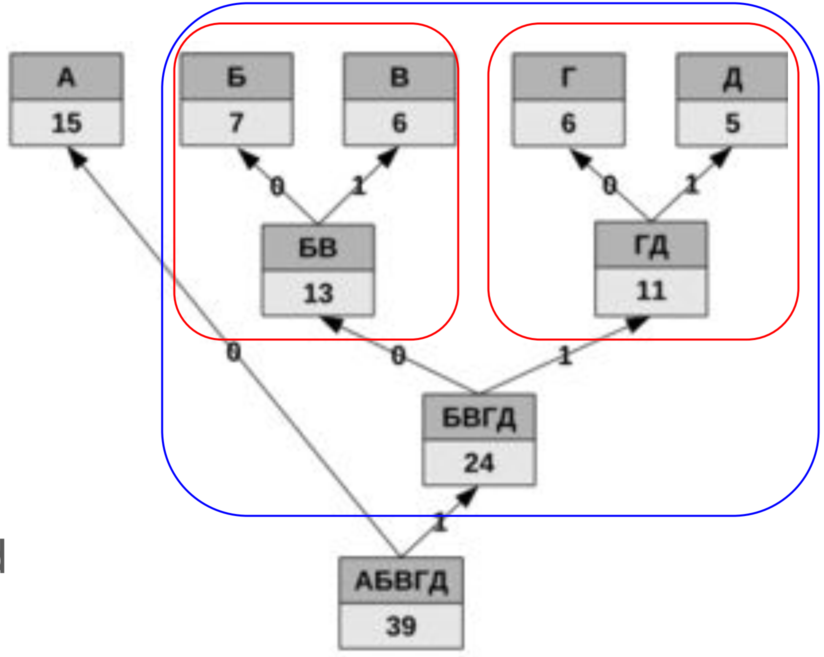


Unique prefix interpretation

Define alphabet A

Let both E and I be all valid **binary prefix (full) trees** on A:

1. **Empty binary prefix tree** is a trivial tree
2. Any subtree of **binary prefix tree** is a valid tree
3. There are always **trivial subtrees** (letters) to attach to a smaller tree



Итого:

А	Б	В	Г	Д
0	100	101	110	111

… consequence

Huffman coding is exactly Rado-Edmonds algorithm for finding minimal cost prefix tree

Let weight function be:

$$w(tree) = \Sigma_{letter} w(letter) * 2^{level(letter)}$$

Greedy Huffman encoding

```python
def encode(symb2freq):
    """Huffman encode the given dict mapping symbols to weights"""
    heap = [[wt, [sym, ""]] for sym, wt in symb2freq.items()]
    heapify(heap)
    while len(heap) > 1:
        lo = heappop(heap)
        hi = heappop(heap)
        for pair in lo[1:]:
            pair[1] = '0' + pair[1]
        for pair in hi[1:]:
            pair[1] = '1' + pair[1]
        heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])
    return sorted(heappop(heap)[1:], key=lambda p: (len(p[-1]), p))
```

Definition: vector matriods

Let **ground set** be finite **subset of vector space** V

Let **independent sets** be … **sets of linearly independent vectors (matrices)**

Steinitz exchange lemma shows that two bases for a finite-dimensional vector space have the same number of elements

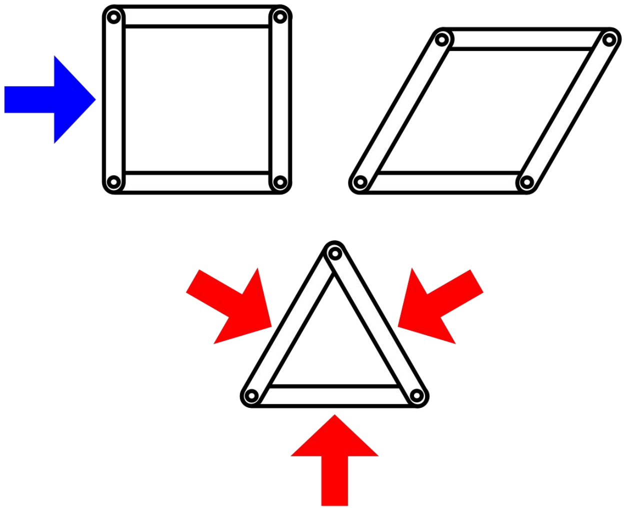
… consequence

Matrix rank search can be done in a greedy way by making the matrix diagonal

$$\begin{bmatrix} 3 & 2 & -1 \\ 2 & -3 & -5 \\ -1 & -4 & -3 \end{bmatrix} \rightsquigarrow \begin{bmatrix} 1 & 4 & 3 \\ 3 & 2 & -1 \\ 2 & -3 & -5 \end{bmatrix} \rightsquigarrow \begin{bmatrix} 1 & 4 & 3 \\ 0 & -10 & -10 \\ 0 & -11 & -11 \end{bmatrix} \rightsquigarrow \begin{bmatrix} 1 & 4 & 3 \\ 0 & 1 & 1 \\ 0 & -11 & -11 \end{bmatrix} \rightsquigarrow \begin{bmatrix} 1 & 4 & 3 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

See also…

Rigidity matroid

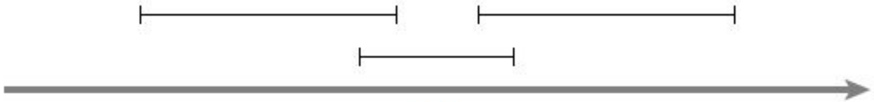


Other greedy optimal algorithms

Interval scheduling: statement

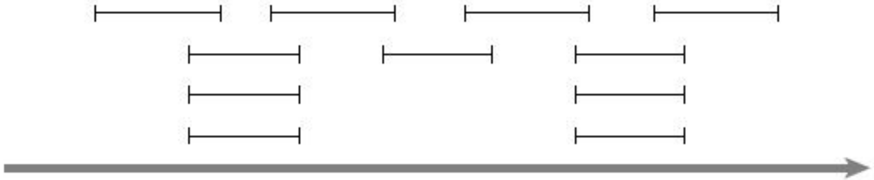
(a)

take which starts earlier



(b)

take shortest



(c)

take with fewer conflicts

Interval scheduling: algorithm

```
Initially let R be the set of all requests, and let A be empty
While R is not yet empty
  Choose a request i ∈ R that has the smallest finishing time
  Add request i to A
  Delete all requests from R that are not compatible with request i
EndWhile
Return the set A as the set of accepted requests
```

Interval colouring: algorithm

```
Sort the intervals by their start times, breaking ties arbitrarily
Let I_1, I_2, ..., I_n denote the intervals in this order
For j = 1, 2, 3, ..., n
  For each interval I_i that precedes I_j in sorted order and overlaps it
     Exclude the label of I_i from consideration for I_j
  Endfor
  If there is any label from {1, 2, ..., d} that has not been excluded then
    Assign a nonexcluded label to I_j
  Else
    Leave I_j unlabeled
  Endif
Endfor
```

And even more

Function `stMinCut`(G)

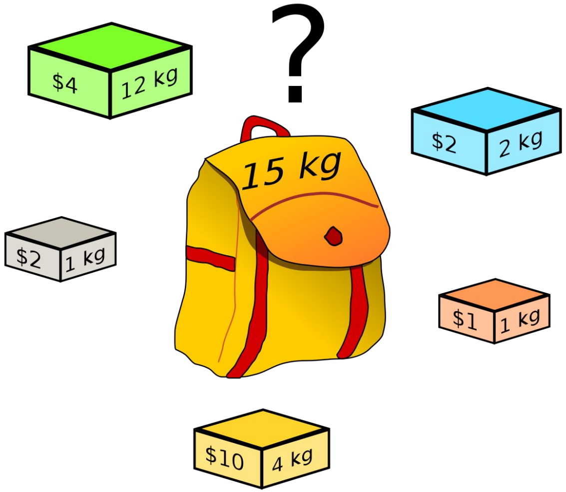
> $A \leftarrow \{a\}$
> **while** $A \neq V$ **do**
>   Let $v \notin A$ be such that $w(A, \{v\})$ is maximized
> $A \leftarrow A \cup \{v\}$
> Let s and t be the last two vertices added to A
> **return** $\langle (V - \{t\}, \{t\}), s, t \rangle$

In *Global Min Cut* problem `stMinCut()` function is greedy

Unbounded knapsack problem (unlimited supply) is solved with greedy algorithm

Biggest maximal matching problem is solved greedy

Interval scheduling and **interval colouring**



When greedy works, but not optimally

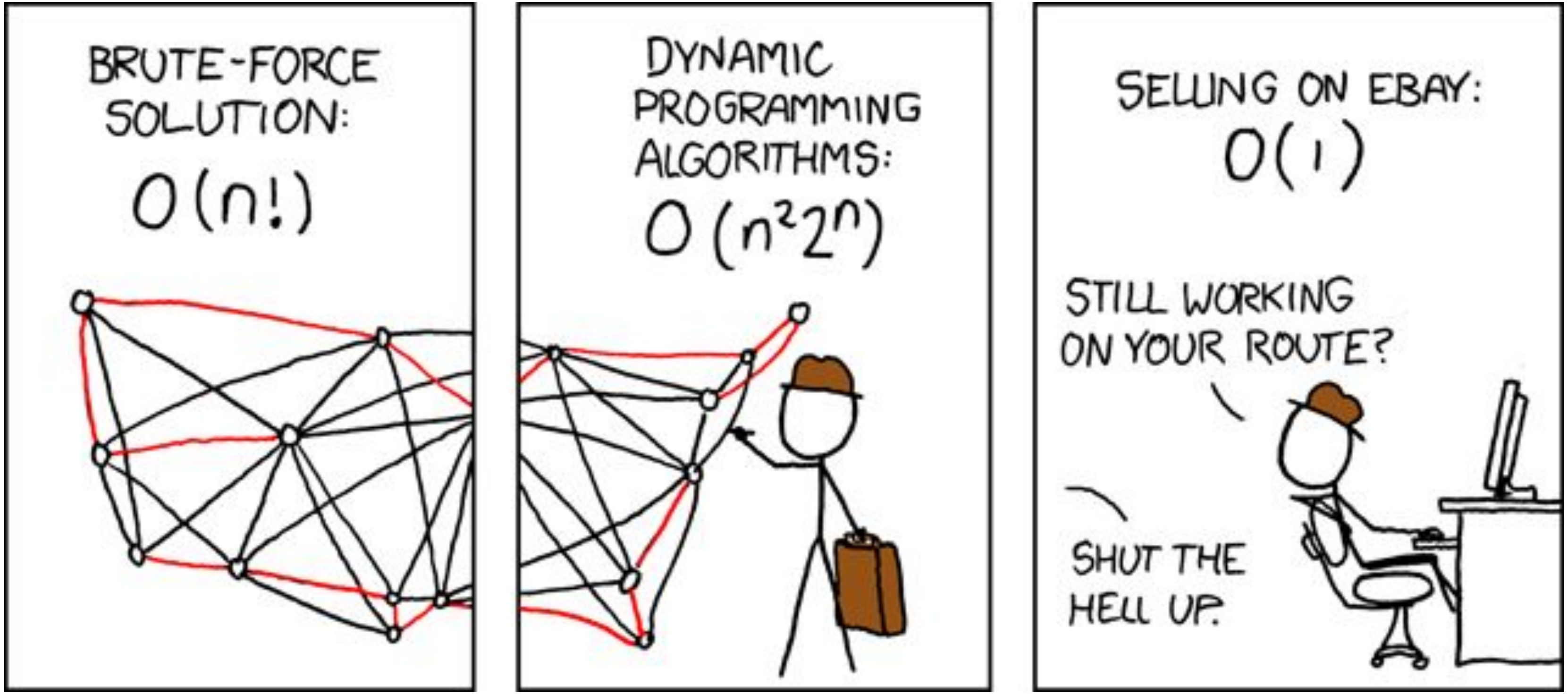
A* is greedy

Graph clustering

Clique problem

Travelling salesman

Travelling salesman: statement



Travelling salesman: nearest neighbour

```
path = [point]
remaining = {... all vertices ...}
sum = 0
while remaining:
   closest, dist = closestpoint(path[-1], remaining)
   path.append(closest)
   remaining.remove(closest)
   sum += dist
# Go back the the beginning when done.
closest, dist = closestpoint(path[-1], [point])
path.append(closest)
sum += dist
```