Heuristic Algorithms

Master's Degree in Computer Science/Mathematics

Roberto Cordone

DI - Università degli Studi di Milano

Schedule:	Wednesday 13.30 - 16.30 in classroom Alfa
	Thursday 09.30 - 12.30 in classroom Alfa
Office hours:	on appointment
E-mail:	roberto.cordone@unimi.it
Web page:	https://homes.di.unimi.it/cordone/courses/2026-ae/2026-ae.html
Ariel site:	https://myariel.unimi.it/course/view.php?id=7439

Efficiency of a heuristic algorithm

A heuristic algorithm is useful if it is

1. efficient: it "costs" much less than an exact algorithm
2. effective: it "frequently" returns a solution "close to" an exact one

Both aspects can be investigated with a

- theoretical analysis (*a priori*), proving that the algorithm finds always or with a given frequency solutions with a given guarantee of quality
- experimental analysis (*a posteriori*), measuring the performance of the algorithm on sampled benchmark instances
  to show that a guarantee of quality is respected in practice

We here discuss

- the theoretical analysis
- of efficiency

Problems

Informally, a problem is a question on a system of mathematical objects

The same question can often be asked on many similar systems

- an instance $I \in \mathcal{I}$ is each specific system concerned by the question
- a solution $S \in \mathcal{S}$ is an answer corresponding to one of the instances

Example: "*is n a prime number?*" is a problem with infinite instances and two solutions ($\mathcal{I} = \mathbb{N}^+ \setminus \{1\}$ and $\mathcal{S} = \{ \texttt{yes}, \texttt{no} \}$)
instance $I = 7$ corresponds to solution $S_I = \texttt{yes}$
instance $I' = 10$ corresponds to solution $S_{I'} = \texttt{no}$
...

Formally, a problem is the function which relates instances and solutions

$$P : \mathcal{I} \to \mathcal{S}$$

Defining a function does not mean to know how to compute it

Algorithms

An algorithm is a formal procedure, composed by elementary steps, in finite sequence, each determined by an input and by the results of the previous steps

An algorithm for a problem P is an algorithm which, given in input $I \in \mathcal{I}$, returns in output $S_I \in \mathcal{S}$

$$A : \mathcal{I} \to \mathcal{S}$$

An algorithm defines a function plus the way to compute it; it is

- exact if its associated function coincides with the problem
- heuristic otherwise

A heuristic algorithm is useful if it is

1. efficient: it "costs" much less than an exact algorithm
2. effective: it "frequently" provides a solution "close" to the right one

This lesson deals with efficiency

Cost of a heuristic algorithm

The “cost” of an (exact or heuristic) algorithm denotes

- not the monetary cost to buy or implement it
- but the computational cost of running it
  - time required to terminate the finite sequence of elementary steps
  - space occupied in memory by the results of the previous steps

The time is much more discussed because

- the space is a renewable resource, the time is not
- using space requires to use at least as much time
- it is technically easier to distribute the use of space than of time

Space and time are partly interchangeable:
it is possible to reduce the use of one by increasing the use of the other

A useful measure of time

The time required to solve a problem depends on several aspects

- the specific instance to solve
- the algorithm used
- the machine running the algorithm
- ...

Our measure of the computational time should be

- unrelated to technology, that is the same for different machines
- concise, that is summarised in a simple symbolic expression
- ordinal, that is sufficient to compare different algorithms

The computational time in seconds for each instance violates all requisites

Worst-case asymptotic time complexity

The worst-case asymptotic complexity of an algorithm (nearly) provides such a measure through the following passages

1. define time as the number T of elementary operations performed (that is a value independent from the specific computer)
2. define the size of an instance as a suitable value n (e.g., the number of elements of the ground set, variables or clauses of the CNF, rows or columns of the matrix, nodes or arcs of the graph)
3. find the worst-case, i. e. the maximum of T on all instances of size n

$$T(n) = \max_{I \in \mathcal{I}_n} T(I) \qquad n \in \mathbb{N}$$

(*now time complexity is only a function* $T : \mathbb{N} \to \mathbb{N}$)

4. approximate $T(n)$ from above and/or below with a simpler function $f(n)$, considering only their asymptotic behaviour (for $n \to +\infty$)
   (*the algorithm should be efficient on instances of large size*)
5. collect the functions in classes with the same approximating function
   (*the approximation relation is an equivalence relation*)

The Θ functional spaces

$$T(n) \in \Theta(f(n))$$

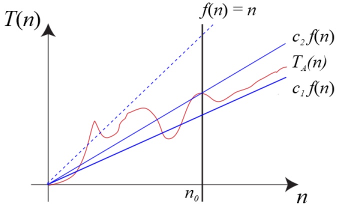
formally means that

$$\exists c_1, c_2 \in \mathbb{R}^+, n_0 \in \mathbb{N} : c_1 f(n) \leq T(n) \leq c_2 f(n) \text{ for all } n \geq n_0$$

where c_1, c_2 and n_0 are independent from n

$T(n)$ is *"enclosed"* between $c_1 f(n)$ and $c_2 f(n)$

- for some *"small"* value of c_1
- for some *"large"* value of c_2
- for some *"large"* value of n_0
- for some definition of *"small"* *and "large"*



Asymptotically, $f(n)$ estimates $T(n)$ up to a multiplying factor:

- for large instances, the computational time is at least and at most proportional to the values of function $f(n)$

The O functional spaces

$$T(n) \in O(f(n))$$

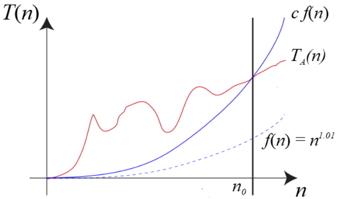
formally means that

$$\exists c \in \mathbb{R}^+, n_0 \in \mathbb{N} : T(n) \leq c\, f(n) \text{ for all } n \geq n_0$$

where c, and n_0 are independent from n

$T(n)$ is *"dominated"* by $c\, f(n)$

- for some *"large"* value of c
- for some *"large"* value of n_0
- for some definition of *"small"* *and* *"large"*



Asymptotically, $f(n)$ overestimates $T(n)$ up to a multiplying factor:

- for large instances, the computational time is at most proportional to the values of function $f(n)$

The Ω functional spaces

$$T(n) \in \Omega(f(n))$$

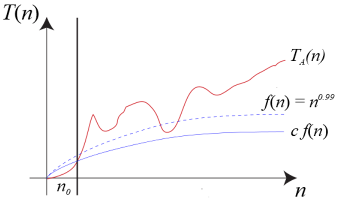
formally means that

$$\exists c > 0, n_0 \in \mathbb{N} : T(n) \geq c\, f(n) \text{ for all } n \geq n_0$$

where c and n_0 are independent from n

$T(n)$ *"dominates"* $c\, f(n)$

- for some *"small"* value of c
- for some *"large"* value of n_0
- for some definition of *"small" and "large"*



Asymptotically, $f(n)$ underestimates $T(n)$ up to a multiplying factor:

- for large instances, the computational time is at least proportional to the values of function $f(n)$

The exhaustive algorithm

For Combinatorial Optimisation problems the size of an instance can be measured by the cardinality of the ground set

$$n = |B|$$

The exhaustive algorithm

- considers each subset $x \subseteq B$, that is each $x \in 2^{|B|}$
- tests its feasibility $(x \in X)$ in time $\alpha(n)$
- in the positive case, it evaluates the objective $f(x)$ in time $\beta(n)$
- if necessary, it updates the best value found so far

The time complexity of the exhaustive algorithm is

$$T(n) \in \Theta\left(2^n \left(\alpha(n) + \beta(n)\right)\right)$$

that is at least exponential, even if $\alpha(n)$ and $\beta(n)$ are small polynomials (which is the most frequent case)

Most of the time, the exhaustive algorithm is impractical

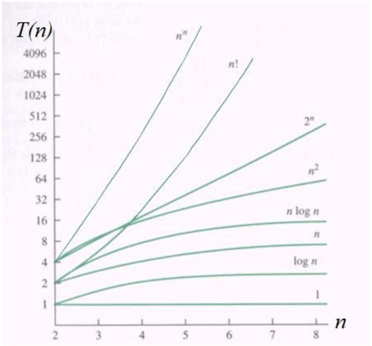
Polynomial and exponential complexity

In Combinatorial Optimisation, the main distinction is between

- polynomial complexity: $T(n) \in O\left(n^d\right)$ for a constant $d > 0$
- exponential complexity: $T(n) \in \Omega\left(d^n\right)$ for a constant $d > 1$

The first family includes efficient algorithms, the second inefficient ones

In general, the heuristic algorithms are polynomial algorithms
for problems whose known exact algorithms are all exponential



Assuming 1 operation/μsec

n	n^2 op.	2^n op.
1	1μ sec	2μ sec
10	0.1 msec	1 msec
20	0.4 msec	1 sec
30	0.9 msec	17.9 min
40	1.6 msec	12.7 days
50	2.5 msec	35.7 years
60	3.6 msec	366 centuries

Problem transformations and reductions

A relation between problems allows to design algorithms (*Interlude 5*):

- by transformation:
  1. given I_P, (instance of P) build I_Q (instance of Q)
  2. given I_Q, apply algorithm A_Q to obtain S_Q (solution of I_Q)
  3. given S_Q, build S_P (solution of I_P)
- by reduction: repeat the transformation 1-2-3 several times correcting I_Q based on the solutions $\{S_Q\}$ already obtained

If A_Q is exact/heuristic, the overall algorithm A_P is exact/heuristic

The two algorithms often have a similar complexity:
if A_Q is polynomial/exponential and

1. building I_Q takes polynomial time
2. the number of iterations is polynomial
3. building S_P takes polynomial time

then A_P is polynomial/exponential

Beyond the worst-case complexity

The worst-case complexity

- cancels all information on the easier instances
  (*how are they made? how many are they?*)
- gives a rough overestimate of the computational time,
  in some (rare) cases useless
  (*see the simplex algorithm for Linear Programming*)

What if the hard instances are rare in the practical applications?

To compensate, one can investigate

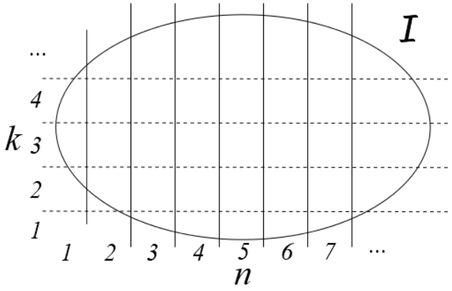
- the parameterised complexity, that is introduce some other relevant parameter k (besides the size n) and express the time as $T(n, k)$
- the average-case complexity, that is assume a probability distribution on $\mathcal{I}$ and express the time as the expected value

$$T(n) = E\left[T(I) \,|\, I \in \mathcal{I}_n\right]$$

Parameterised complexity

Some algorithms are exponential in k and polynomial in n, and therefore

- efficient on instances with low k
- inefficient on instances with large k



Nature of the additional parameter

If the additional parameter k is a part of the input, such as

- a numerical constant (e. g., the capacity in the *KP*)
- the maximum number of literals per clause in logic function problems
- the number of nonzero elements in numerical matrix problems
- the maximum degree, the diameter, etc. . . in graph problems

one knows *a priori* whether the algorithm is efficient on a given instance

If the additional parameter k is a part of the solution, such as

- its cardinality (as in the *VCP*)

one will only find out *a posteriori*

(*but an a priori estimate could be available*)

An example: the *VCP*

Exhaustive algorithm: for each of the 2^n subsets of vertices, test if it covers all edges, compute its cardinality and keep the smallest one

$$T(n,m) \in \Theta\left(2^n (m+n)\right)$$

(m *can be removed observing that* $m \leq n(n-1)/2$)

But if we already know a solution with $f(x) = |x| = k+1$,
we can look for a solution of k vertices, and progressively decrease k
(*even better, use binary search on* k)

Naive algorithm: for each subset of k vertices, test if it covers all edges

$$T(n,m,k) \in \Theta\left(n^k m\right)$$

For fixed k, this algorithm is polynomial (*but in general very slow*)

Bounded tree search for the *VCP*

A better algorithm can be based on the following useful property

$$x \cap (u, v) \neq \emptyset \text{ for all } x \in X, (u, v) \in E$$

Any feasible solution includes at least one extreme vertex for each edge

Bounded tree search algorithm to find x with $|x| \leq k$:

1. choose any (u, v): either $u \in x$ or $u \notin x$ and $v \in x$
2. for each open case, remove the vertices of x and edges they cover

$$V := V \setminus x \qquad E := E \setminus \{e \in E : e \cap x \neq \emptyset\}$$

(*The edges covered by vertices in x are no longer constraining*)

3. if $|x| \leq k$ and $E = \emptyset$, x is the required solution
4. if $|x| = k$ and $E \neq \emptyset$, there is no solution
5. otherwise go to step 1

The complexity is $T(n, m, k) \in \Theta\left(2^k m\right)$, polynomial in n $(m < n^2)$

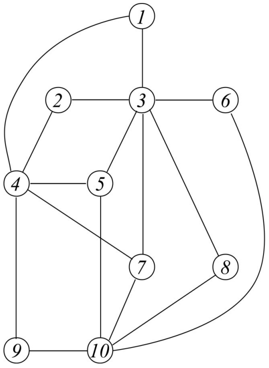
For $n \gg 2$, this algorithm is much more efficient than the naive one

Example

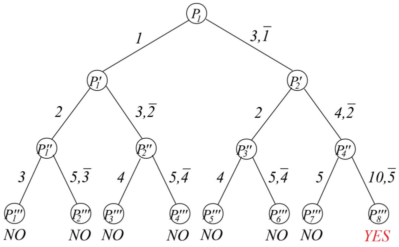
In the following graph $n = 10$, $m = 16$: is there a solution with $|x| \leq 3$?

Exhaustive algorithm: $\Theta\left(2^n\left(m+n\right)\right)$, with $2^n\left(m+n\right) = 1024 \cdot (16+10)$

Naive algorithm: $\Theta\left(n^k m\right)$, with $n^k m = 1\,000 \cdot 16$



Bounded tree search algorithm: $\Theta\left(2^k m\right)$ with $2^k m = 8 \cdot 16$



(*edges selected in lexicographic order*)

Kernelisation (“problem reduction”)

Kernelisation transforms all instances of P into simpler instances of P, instead of instances of another problem Q

This is also known as problem reduction

Quite often, in fact, useful properties allow to prove that

- there exists an optimal solution not including certain elements of B ($\Rightarrow$ *such elements can be removed*)
- there exists an optimal solution including certain elements of B ($\Rightarrow$ *such elements can be set apart and added later*)

In short, remove elements of B without affecting the solution

Possible useful outcomes are

- an exact algorithm polynomial in n (parameterised complexity)
- faster exact and heuristic algorithms
- better heuristic solutions
- heuristic kernelisation: apply relaxed conditions sacrificing optimality

Kernelisation of the *VCP*

If $\delta_v \geq k+1$, vertex v belongs to any feasible solution of value $\leq k$
(v *has* $k+1$ *incident edges that should be covered by as many vertices*)

Kernelisation algorithm to keep only vertices of solutions x with $|x| \leq k$:

- start at step $t = 0$ with $k_0 = k$ and an empty vertex subset $x_t := \emptyset$
- set $t = t + 1$ and add to the solution the vertices of degree $\geq k_t + 1$

$$\delta_v \geq k_t + 1 \Rightarrow x_t := x_{t-1} \cup \{v\}$$

- update k_t: $k_t := k_0 - |x_t|$
- remove the vertices of zero degree, those of x and the covered edges

$$V := \{v \in V : \delta_v > 0\} \setminus x_t \qquad E := \{e \in E : e \cap x_t = \emptyset\}$$

- if $|E| > k_t^2$, there is no feasible solution (k_t *vertices are not enough*)
- if $|E| \leq k_t^2 \Rightarrow |V| \leq 2k_t^2$; apply the exhaustive algorithm

The complexity is $T(n,k) \in \Theta\left(n + m + 2^{2k^2}k^2\right) = \Theta\left(n + m + 2^{|V|}|E|\right)$

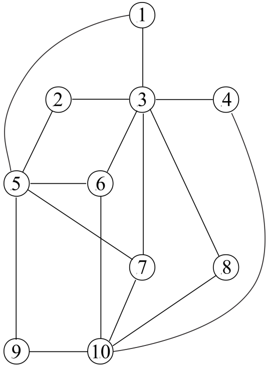
Example

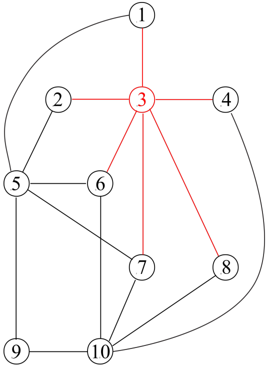
Given the following graph, is there a solution with $|x| \leq k_0 = 5$?
$(n = 10,\ m = 16)$

Exhaustive algorithm: $\Theta\left(2^n (m+n)\right) \Rightarrow T \approx 2^{10}(10+16) = 26\,624$

Naive algorithm: $\Theta\left(n^k m\right) \Rightarrow T \approx 10^5 \cdot 16 = 16\,000\,000$

$\delta_3 = 6 \geq k_0 + 1 \Rightarrow x_1 := \{3\}$, remove the incident edges and $k_1 = 4$





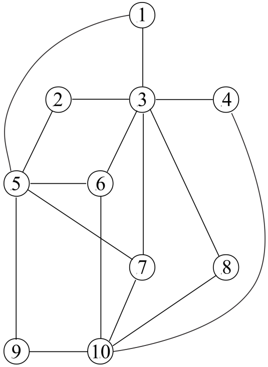
Example

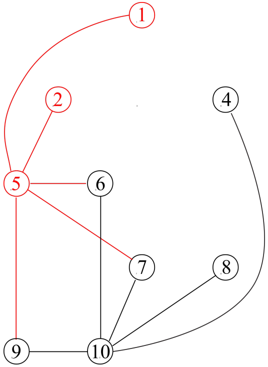
Given the following graph, is there a solution with $|x| \leq k_0 = 5$?
$(n = 10,\ m = 16)$

Exhaustive algorithm: $\Theta\left(2^n\left(m+n\right)\right) \Rightarrow T \approx 2^{10}\left(10+16\right) = 26\,624$

Naive algorithm: $\Theta\left(n^k m\right) \Rightarrow T \approx 10^5 \cdot 16 = 16\,000\,000$

$\delta_5 = 5 \geq k_1 + 1 \Rightarrow x_2 := \{3, 5\}$, remove the incident edges and $k_2 = 3$





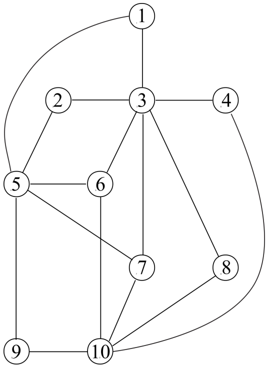
Example

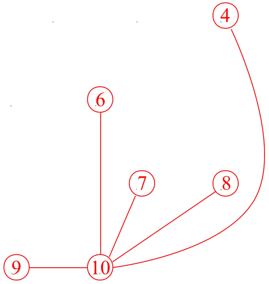
Given the following graph, is there a solution with $|x| \leq k_0 = 5$? ($n = 10,\ m = 16$)

Exhaustive algorithm: $\Theta\left(2^n\left(m+n\right)\right) \Rightarrow T \approx 2^{10}\left(10+16\right) = 26\,624$

Naive algorithm: $\Theta\left(n^k m\right) \Rightarrow T \approx 10^5 \cdot 16 = 16\,000\,000$

$\delta_{10} = 5 \geq k_2 + 1 \Rightarrow x_3 := \{3, 5, 10\}$, remove the incident edges and $k_3 = 2$





Kernelisation: $\Theta\left(n+m\right) \Rightarrow T \approx 10 + 16 = 26$

Average-case complexity

The worst-case description of complexity might be not very significant: some algorithms are efficient on nearly all instances

(*see the simplex algorithm for Linear Programming*)

A theoretical study of the average-case complexity

- defines a probabilistic model of the problem, assuming a (usually simple) probability distribution on $\mathcal{I}_n$ for each $n \in \mathbb{N}$
- estimates the expected value of $T(I)$

$$T(n) = E\left[T(I) \mid I \in \mathcal{I}_n\right]$$

How do we define a probability distribution on $\mathcal{I}_n$ for each $n \in \mathbb{N}$?

Let us see some examples

Probabilistic models for numerical matrices

Binary random matrix with a given size (m rows and n columns)

1. equiprobability: list all 2^{mn} binary matrices and select one of the matrices with uniform probability
2. uniform probability: set each cell to 1 with a given probability p

$$Pr\left[a_{ij} = 1\right] = p \qquad (i = 1, \dots, m;\ j = 1, \dots, n)$$

   If $p = 0.5$, it coincides with the equiprobability model, for other values some instances are more likely than others

3. fixed density: extract δmn cells out of mn with uniform probability and set them to 1

   If $\delta = p$, it resembles the uniform probability model, but some instances cannot be generated

Probabilistic models for graphs

Random graph with a given number of vertices n

1. equiprobability: list all $2^{\frac{n(n-1)}{2}}$ graphs and select one of the graphs with uniform probability
2. Gilbert's model, or uniform probability $G(n,p)$:

   $$Pr[(i,j) \in E] = p \qquad (i \in V, j \in V \setminus \{i\})$$

   All graphs with the same number of edges m have the same probability $p^m (1-p)^{\frac{n(n-1)}{2}-m}$ (different for each m)

   If $p = 0.5$, it coincides with the equiprobability model
3. Erdős-Rényi model $G(n,m)$: extract m unordered vertex pairs out of $\frac{n(n-1)}{2}$ with uniform probability and create an edge for each one

   If $m = p\frac{n(n-1)}{2}$, it resembles the uniform probability model, but some instances cannot be generated

Probabilistic models for logic functions

Random CNF with a given number of variables n
and a given number of literals k for each logic clause

1. fixed-probability ensemble:
   list all $\binom{n}{k}2^k$ clauses of k distinct and consistent literals and add each one to the CNF with probability p
2. fixed-size ensemble:
   build m clauses, adding to each one k distinct and consistent literals, extracted with uniform probability

   If $m = p\binom{n}{k}2^k$, it resembles the fixed-probability model, but some instances cannot be generated

Computational cost of heuristic algorithms

The time complexity of a heuristic algorithm is usually

- strictly polynomial (with low exponents)
- fairly robust with respect to secondary parameters

Therefore, the worst-case estimation is also good on average

Metaheuristics use random steps or memory

- the complexity is well defined for single components of the algorithm
- the overall complexity is not clearly defined
  - in theory, it could extend indefinitely (but the pseudorandom number generator or the memory configurations would yield an infinite loop)
  - in practice, it is defined by a condition imposed by the user

(*more about this later*)

Their analysis usually focuses on the single components

Conclusions

So,why discussing the previous topics in a course on heuristics?

1. to guide the search for the correct algorithm: an exact algorithm can be efficient in the given case, even if inefficient in the worst case
2. to show that exact and heuristic algorithms can interact proficuously: heuristic algorithms provide information to improve exact algorithms (*they become more efficient*)
3. to show that kernelisation improves also heuristic algorithms (*they become more efficient and more effective*)
4. to identify *a priori* the harder instances (*of course, not all algorithms have the same hard instances*)