Veszprémi Egyetem

Complexity Theory

Programozási nyelvek elmélete II.

Working Notes

Sándor Dominich

Contents

Complexity Theory

The aim of the subject is to learn the basics of complexity theory in order to be able to assess the complexity of the different algorithms. (we want to decide wether they are feasible[1] or not, i.e., tractable[2] or not.)

In the following chapter we will learn about people of great importance, who have shaped the area of complexity theory.

KURT GÖDEL

/1906 Brünn (Austria) – 1978 Princeton (USA)/

1.

He made his studies in Brünn, and at the University of Vienna (1923). He made his PhD in logical systems (1929). Because of the Second World War he emigrated to the USA. In the USA he worked at Princeton University. His main result is called: Gödel's Incompleteness Theorem[3] wich states that:

In any logical system there must exist sentences that are neither provable nor refutable within the system (e.g., Euclidean Geometry).

This means that it is not possible to prove the *consistency* of an axiomatic system. Note: Gödel's theorem does not means that we should give up to use the axiomatic method.

2.

GÖDEL'S NUMBERING: a technique to encode characters.

Let assume we have a system with the following symbols:

symbols:

$$s_1,\ldots,s_i\ldots,s_n$$

these are drawn from a set of primitive symbols, e.g.:

[1] megvalósítható
[2] kezelhető

$$0,\ f,\ \vee,\ \Pi,\ (\ ),\ x_1,\ ...$$

$s_i \rightarrow g_i$ /unique number for every 1 to n, i=1… n

Example:
We have: 0, f, ~, ∨, Π, (,), x_1, x_n,
1, 3, 5, 7, 11, 13, 17, 19^1, 23^n ← *these are unique primes*,

any program $P = s_1,\ldots,s_i\ldots,s_n$ (it is a string)

$$2^{g_1}\ldots\ldots\ldots p_n^{g_n} \quad /p_n \text{ is the n-th prime, } g_1 \ldots g_n$$

$P = x_1 \vee \sim f \rightarrow 2^{19}*3^7*5^5*7^3 \in \mathbf{N}$ /Gödel number

This can translate any formula to a number. Note: The set **N** has a big importance in computer science.

STEPHEN COLE KLEENE

/1903 Hartford – 1994 Madison, USA/

1.

He studied at Amherst College 1930, and got his PhD in1934. His main result is:

A language can be defined only if it is recognised by a finite-state automaton.

ALONZO CHURCH

/1903 Washington – 1995 Hudson, USA/

1.

He made his studies at Princeton University 1924, and got his Phd in 1927. Church's Thesis:

Any function on natural numbers that can be computed by an algorithm can be defined by a formula of the lambda calculus.

2.

The lambda calculus is a model for computation developed by Church.

---

[3] nemteljességi tétel

ALAIN TURING

/1912 London – 1954 Wilmslow, UK/

1.

The Turing Machine is just another model of computation (other models: λ-calculus, von Neumann-automaton): methodical process to get precise results.

Recall the Turing Machine (TM)

$\Sigma=\{s_0,s_1,\ldots,s_p\}$ alphabet

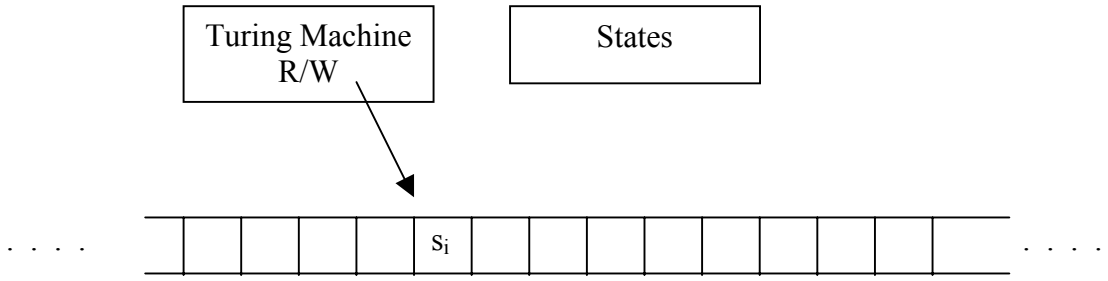
$S=\{q_0,q_1,\ldots,q_m,q_Y,q_N\}$ states

q_0 – start state

q_Y q_N – halt states (Yes, No state)

$P=\{I \mid I= q_i\, s_j\, s_k\, X\, q_j\}$ $X\in \{L,R,N\}$ (Left,Right,None)

The head is in state q_i, reads s_j, s_j will be replaced by s_k, and the machine will transit to state q_j.



Deterministic TM (DTM)

We must have the program, the variables, the initial state on the tape.

Example:

$$f: N^m \rightarrow N$$

$$f(x_0,x_1,\ldots,x_m)$$

we have on the tape:

- the program
- $(x_0,x_1,\ldots,x_m)$

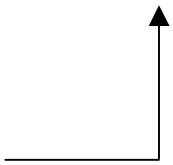
- q_0 initial state

Non-deterministic TM (NDTM)

NDTM is a DTM which is added a new module called a Guessing Module (GM). GM has a write head which writes symbols on the tape at random

- this may never end ⟹ trouble
- this ends ⟹ checking phase begins ($q_{,Y}$ or q_N)

If checking results in q_N then request from the beginning.



Recursion and complexity theory

Primitive recursion functions

Simple basic functions which are used to represent other more sophisticated functions.

Zero function

$O: \mathbf{N} \rightarrow \mathbf{N}$ $\quad O(x)=0 \forall x \in \mathbf{N}$

Successor function

$S: \mathbf{N} \rightarrow \mathbf{N}$ $\quad S(x)=x+1 \quad \forall x \in \mathbf{N}$

Projection function

${}^{n}\Pi_{m}: \mathbf{N}^{\mathbf{n}} \rightarrow \mathbf{N}$ $\quad n \geq m \geq 1$

${}^{n}\Pi_{m}(x_1,\ldots,x_m,\ldots,x_n)=x_m$

} base functions

Function-forming operators

Composition

given $f: \mathbf{N}^n \rightarrow \mathbf{N}$, $g_i: \mathbf{N}^m \rightarrow \mathbf{N}$ $\quad i=1,2,\ldots,n$

composition of f and g_i is h

$h: \mathbf{N}^n \rightarrow \mathbf{N}$ $\quad h(x)=f(g_1(x),\ldots..,g_n(x))$

Notation: $f \circ g_i$ (or $g_i \circ f$; a matter of convention)

Primitive recursion

given $f: \mathbf{N}^n \rightarrow \mathbf{N}$, $g_i: \mathbf{N}^{n+2} \rightarrow \mathbf{N}$

a new function h is defined, $h: \mathbf{N}^{m+1} \rightarrow \mathbf{N}$ by primitive recursion

$h(x,0)=f(x)$ x is the initial value

$h(x,y+1)=g(x,y,h(x,y))$ uses the previous values

Example:

Addition:

$f(x,0)=x$ $\quad f(x,y+1)=f(x,y)+1$

successor ↓

we increase by 1 the value x, while we get y

Multiplication:

$f(x,0)=0$ $f(x,y+1)=f(x,y)+x$

recursive definition for multiplication using addition

Note:

- in a computer multiplication is defined using binary addition
- multiplication in our brain: it might be a game of verbal patterns previously aquiered

Definition: primitive (partial) recursion function (PRF)

PRF are the set containing the zero, the successor, and projection functions, and this set is closed under composition and primitive recursion.

Definition: recursive functions (RF)

Let f: $\mathbf{N}^{n+1} \rightarrow \mathbf{N}$ be a PRF, and then a new function g is defined by minimising f: g(x)=min(y), g: $\mathbf{N}^{n} \rightarrow \mathbf{N}$ such that f(x,y)=0, then g is a recursive function. The RF are the set containing the zero, the successor, projection function, and closed under composition, primitive recursion and minimisation.

Turing Machine (TM)

The Turing machine is a mathematical abstraction of computation (procedure, algorithm) in general.

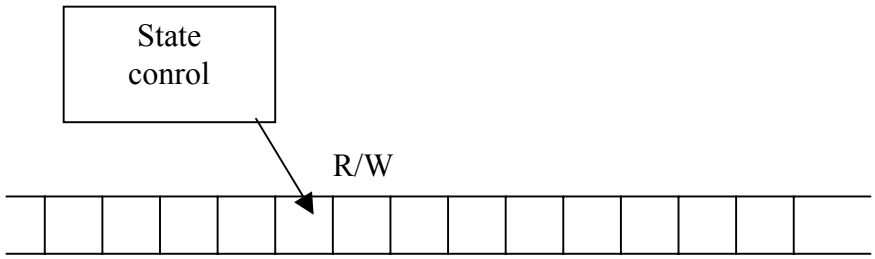
$\Sigma=\{s_0,s_1,\ldots,s_p\}$ alphabet

$S=\{q_0,q_1,\ldots,q_m,q_Y,q_N\}$ q_0 – start state q_Y q_N – halt states (Yes, No state)

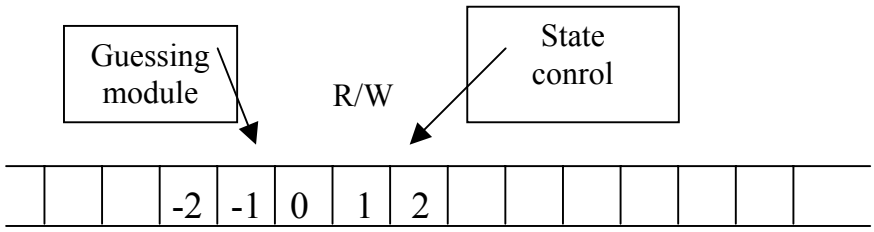
$M=\{I \mid I= q_i\, s_j\, s_k\, X\, q_j\}$ $X\in \{L,R,N\}$ (Left,Right,None)

TM operates in steps.

Deterministic Turing Machine (DTM)



Non-deterministic TM (NDTM)



The head of the guessing module can only move to the left. It scans and only writes the symbols of the alphabet randomly on the tape.

Turing Computability

The TM can be used to compute functions. Given f: $\mathbf{N}^{n+1}\rightarrow\mathbf{N}$, we say that f is Turing computable if the machine halts with a representation of f(x) on the tape, when it starts with a representation of x on the tape.

It's widely accepted: any function that can be computed by an algorithm (procedure) can also be computed by an appropiate TM. This is the *Church-Turing Thesis*, (this does not have a formal proof).

There exists a link between resucsive functions and TMs.

f is recursive $\Leftrightarrow$ f is Turing computable

If we have a problem which is solvable algorithmically $\Rightarrow$ then we can use recursive functions to solve that.

Taking into account: Theory + Thesis we can hypothesise

? algorithm
* Turing Machine
** recursive function

} These concepts may be regarded as being equivalent

Note:

? there isn't any mathematical definition
* defined mathematically
** defined formally

Example:

1. Halting problem

PROCEDURE HALT(P,x) /x is a value returned by HALT
P is input to HALT (can be any program)/

BODY

IF …. THEN x='HALTS'
ELSE x='DOESN'T HALT'

```
        END


PROCEDURE SELFHALT(P,x)
                HALT(P,x)
                IF x='HALT' THEN
                                        X='HALTS ON SELF'
                ELSE x='DOESN'T HALT ON SELF'
        END


PROCEDURE CONTRARY(P)
                SELFHALT(P,x)
                IF x='HALTS ON SELF' THEN WHILE TRUE DO x='x'
        END
```

Question: What happens if input P=CONTRARY ?
It is impossible to give an algorithm which can decide whether another algorithm will stop on a given input or not.

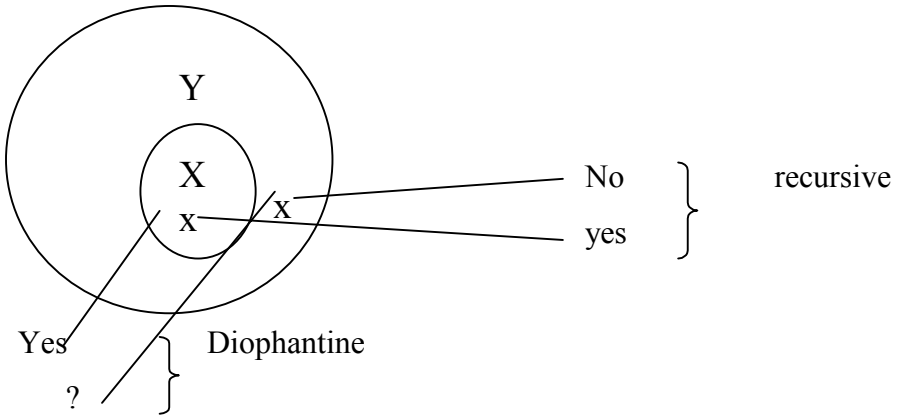
2. Equivalence problem

Is it possible to give an algorithm which can decide if two algorithms are equivalent or not?
The equivalence problem cannot be solved algorithmically.

Recursive sets and recursively enumerable sets

$X \subseteq Y$ is recursivaly enumerable (r.e.) or Diophantine if there exists an algorithm (TM, recursive function) which outputs Yes when presented with an element $x \in Y$ iff (if and only if) $x \in X$. If $x \notin X$ the algorithm is undecidable, will never terminate.

$X \subseteq Y$ is recursive (computable, decidable) if $\exists$ algorithm (TM, RF) which outputs Yes when presented with an element $x \in Y$ if $x \in X$ and outputs No if $x \notin X$.



We have two ways to define a set:

1. $X=\{x_1,\ldots,x_n\}$ by writing down the elements it contains
2. $X=\{x \mid P(x)\}$ P is property, usually a logical expression (or an algorithm)

Theory of P classes and NP classes

We have seen, that there are problems, that CANNOT be solved algorithmically.(e.g. Halting Problem).

The question is: Are there any similar problems or not. The answer is Yes, there are similar problems.

Let F be a function, and A be an algorithm (program). In computer science a function is an entity, that takes an input data, and produces an output, wich is a data too.

$$F(\text{ input }) = \text{output}$$

Let us assume, that we have the natural numbers in the following table:

N	F_1	F_2	F_3	…	F_i	...	F_e
1	1	1	2				1+1=2
2	2	4	4				4+1=5
3	3	9	6				6+1=7
…							
I	$F_1(i)$	$F_2(i)$	$F_3(i)$		$F_i(i)$		$F_i(i)$+1
…							

Let's consider a function wich takes on these numbers.

F_1 - is the unity function; F_2 - is the square function… in this way we can consider other functions to.

Assume: our table contains every function that can be considered on these numbers. The question is whether the function F_e is a member of the table or not. The answer is: NO. F_e is a function defined on the numbers of the table, but the funciton is not a member of the table.

There is a contradiciton with the assumption; hence the set of the function in the table is countable, but F_e does not belong to this set. The *set of all function is an uncountable set*.

Note: continuum= c = $|\mathbf{R}|$ xcardinality of the set **R** of real numbers

There exist functions, which CANNOT be computed algorithmically.

How many functions are there like this? → many functions like this can be contructed.

Intractable problems

(Towers of Hanoi, Chess, Decision Making,…) To solve these problems takes exponential time.

Our question is the following: *Are all intractable problems only games?*

The answer is NO, we have some other problems like that (e.g.: Traveling Salesman).
In the following we can see a real life example:

Example:
The Bureau of Fisheries asked the Bell Laboratories to solve the following problem. They made measurements in the Gulf of Mexico to find out the water temperature, the current strength, and the fish population. The measurements were made with buoys in the water at random places. The question was to find an efficient way to collect the buoys together. They wanted to choose the cheapest way, because of the crew, the fuel, and the time cost a lot of money. The Bell Labs were able to give an answer by running a program on their computer in 12 hours and for 24000 dollars, which made the ship 8 % efficient. Interesting was that a couple of clerks could achieve the same result for only 360 dollars.

There exists another examaple, with the same result. It is about the collection of garbage in a city. The garbages is collected twice a week following complementary routes.

$$\Pi : I \rightarrow S$$

where: - Π is the problem

- I are instances (an algorithm always solves an instance[4] of a problem)
- S are the solutions

instance: parameters take concrete (specific) values

π : is an instance of a problem. It is encoded binary and goes into an algorithm.

$\pi \in \{0,1\}^*$ a string containing 0 and 1

TM – decides (accepts) π.

If it stops with YES on it's tape then the problem is decided. →

π is a decision problem.

We can't get answer for the TSP and TH problems, because these are not decision problems. TSP and TH (Travelling Salesman Problem and Towers of Hanoi) should be transformed first into a decision problem. This is the method:

TSP: $\{ c_1,\ldots,c_i\ldots,c_n \}$ the cities

$D=(d_{ij})_{n*m}$ distances between city c_i and c_j

Obviously $d_{ii} = 0$ and D is symmetric.

Asked for the shortest tour:

$\min(\underbrace{\sum_{i=1}^{m} d_{\pi(i)\pi(i+1)} + d_{\pi(n)\pi(n+1)}}) \quad m=n-1$

$\leq B$ B is a limit

The decision problem is to say that if B equals min or not.

Given π instance, decision problem, encoded length=n

P-class = $\{\pi | T_{TM}(n) \leq p(n)\}$ solved in polynom time

(**P**olynomial-class)

NP-class = $\{\pi | T_{NDTM}(n) \leq p(n)\}$ guess if checked in polynom time

[4] eset

(**N**ondeterministic **P**olynomial-class)

THEOREM

If $\pi \in NP$ then there exists a polynomial p such that π can be solved by a DTM in $O(2^{p(n)})$

$\pi \in NP \Rightarrow \exists\ p(n)$: $T_{DTM}(n) = O(2^{p(n)})$

Proof:

$\pi \in NP \Rightarrow \exists$ algorithm A on NDTM which checks π in poly time.

$q(n)=c_1 n^{c_2}$ length(π)=n

Γ (is the alphabet), $\exists$ guess $\in \Gamma \leq q(n)$

$|\Gamma| = k$ no_of_guesses=$k^{q(n)}$

overall complexity $\leq q(n)k^{q(n)} = c_1 n^{c_2} * k^{c_1 n^{c_2}} \Rightarrow$

$c_1 n^{c_2}$

$\exists\ p(n)$: $c_1 n^{c_2} * k \leq p(n)$ because $c_1^2\ n^{2c_2} \log(k) \leq p(n)$

It can be solved in exponential time.

Polynomial transformation

Given two languages $L_1 \subseteq \Sigma_1^*$, $L_2 \subseteq \Sigma_2^*$

$f : \Sigma_1^* \rightarrow \Sigma_2^*$ is a polynomial transformation if:

a) $\exists$ DTM which computes f in poly time
b) $x \in L_1 \Leftrightarrow f(x) \in L_2$, $\forall x \in \Sigma_1$

} $L_1 \leq L_2$: $\leq$ - transforms ($\propto$)

THEOREM

$L_1 \leq L_2 \Rightarrow$ ($L_2 \in P \Rightarrow L_1 \in P$) or equivalently

($L_2 \notin P \Rightarrow L_1 \notin P$) These two mean the same thing.

Proof:

M_f Time DTM computes f M_2 Runningime of DTM

M_f - f(x) M_2 - $f(x) \in L_2$

Overall time is $M_f \circ M_2$, this is a polynomial time !

All problems that can be transformed into each other are equivalent.

In the NP-complete class all problems are equivalent with each other under a polynomial transformation.

SAT

Steven Cook proved that NP-complete problems did exist (1971)

SAT means Satisfiability

Given: - boolean expressions, Conjunctive Normal Form (CNF)

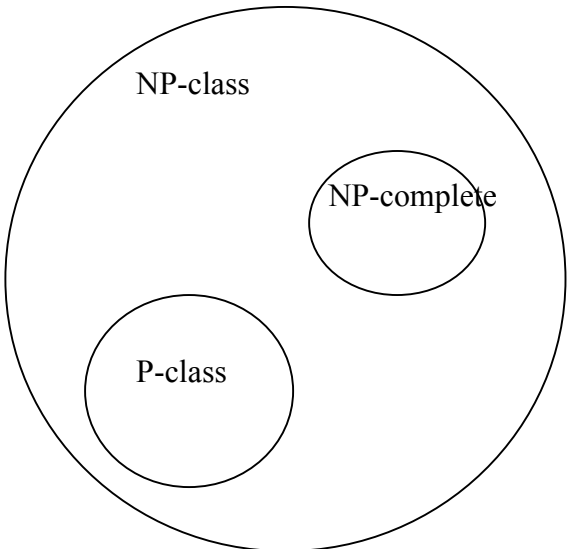
$$\wedge_i(\vee_j l_j) \qquad l_j \in \{ T,F \}$$

decide whether the given expression is true or not. Ex.: 2SAT at most two literals 3SAT .....
3SAT is NP-complete

TSP

HC (Hamiltonian Cycle)

Our current knowledge is that the NP-class is not the same as P-class, but it is not proved.



How to prove (tackle) that a problem π is NP-complete?

(Whether the problem is tractable or not?)

1. Does π belong to the P-class?
2. Suspect π might be NP
3. $\pi \in$ NP ! check in poly time if we have a solution or not?
4. we find a KNOWN NP-complete problem which is equivalent to π KNOWN NP-class $\leq_p \pi$
5. look for a polynomial transformation between the known problem and π

3. } $\Rightarrow$ $\pi \in$ NP-class
4. } proof of NP-completeness in practice

If we find an algorithm which solves one NP-complete problem in poly time, then all the others can be solved in poly time!

Phase transition

$\pi \in$ NP no polynomial time algorithm to solve all instances of π

On the other hand: some instances can be solved in " poly" time (tractable)

It seems: there exist instances of π that are tractable, and there exist instances of π that are not tractable. The wet of all instances of π transit from the state of being tractable to a state of being intractable

Phase transition: a point (place) where a system undergoes a sudden change
Random graph: given N nodes, number of all possible edges is $C_N^2, \binom{N}{2}, {}^2C_N$

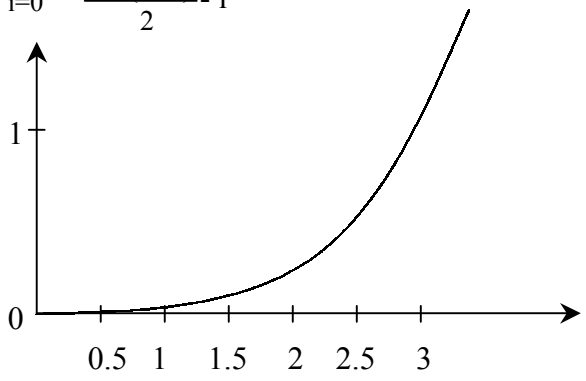
Select E edges without replacement from C_N^2 at random.
We get a graph G=(N, E), that is a random graph.

S denote the probability of a randomly given circuit having each edge in G.
Let us try to estimate it:
$e = |E|$
probability that the first edge of our circuit belongs to G: $\frac{e}{\frac{N(N-1)}{2}}$

probability that the second edge of our circuit belongs to G: $\frac{e}{\frac{N(N-1)}{2}-1}$

and so on ...

$S = \prod_{i=0}^{N-1} \frac{e}{\frac{N(N-1)}{2} - i}$ N, e = 0 ... C_N^2



The larger the number of edges larger the chance the graph will be connected

Erdős and Rényi

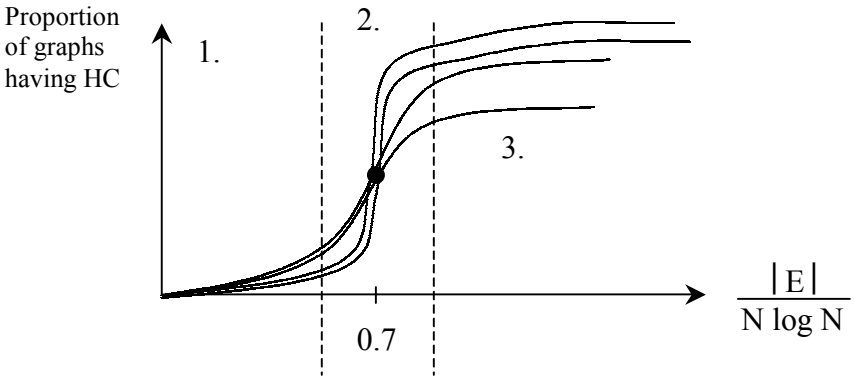
if $|E| = \frac{N \log N}{2} + cN$, $c \in \mathbf{R}$

Then P probability of G being connected = $e^{-e^{-2c}}$ when $N \to \infty$

if $|E| < \frac{N \log N}{2}$ then $P \to 0$

$|E| > \frac{N \log N}{2}$ then $P \to 1$

Frank and Martel



Part 1 practically no Hamiltonian Cycle

Part 3 most have Hamiltonian Cycle

Part 2 phase transition: sudden change between part 1 and part 2

crossover point: a point where the phase transition curves intersect each other.

constraint parameter: $\frac{|E|}{N \log N}$

Analysis of Program Execution Time

Theory of Programming Languages II

Definition of the task

Our aim is to investigate some behavioral aspects of a couple of simple algorithms, focusing particularly on the running time of different implementations in C. There are four exercises, exercise 1, 2 and 3 deals with multiple loops while exercise 4 is an example of an intractable computation, where we take a closer look at the problem of the Towers of Hanoi. The computer we used for the measurement was equipped with an Intel Celeron 333 Mhz processor and 64Mb RAM. It was running Microsoft Windows 98 as operating system and we used Borland C++ 3.0 as compiler.

$t = C0 + C \cdot n$

Our task is to measure the values of the constants and the running time at n=300 million.

The original source code:

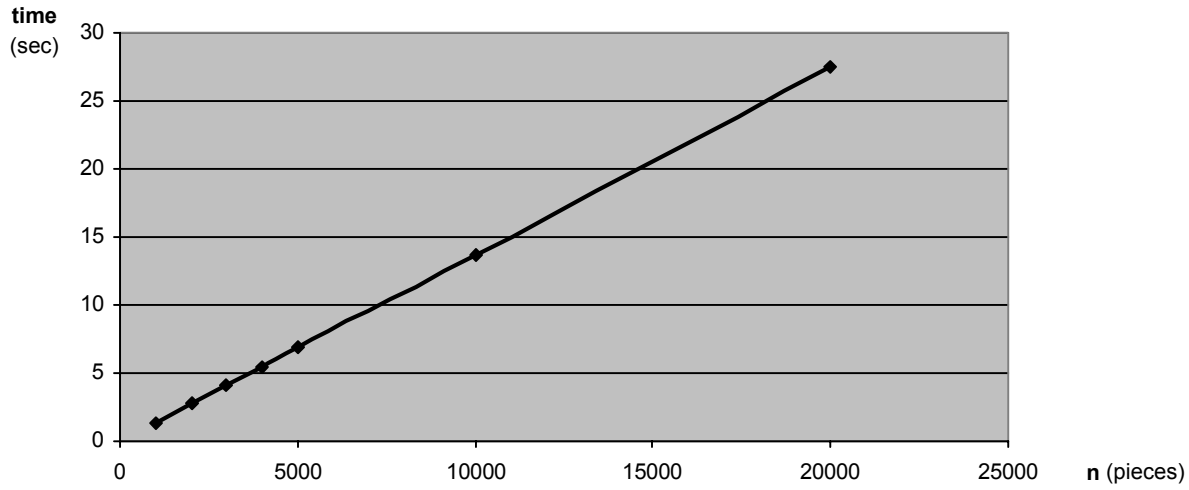
```
program E1;
var
i,n:integer;
begin
readln(n);
i := 1;
while i <= n do
   begin
     writeln(i);
   i := i + 1;
   end;
writeln('done');
readln;
end.
```

The C version:

```
void main()
{
     int i, n;
     printf("n = ");
     scanf("%d", &n);
     i = 1;
     while (i <= n)
     {
          printf("%d\n", i);
          i++;
     }
     printf("done\n");
     getch();
}
```

We have measured the following values:

n	**1000**	**2000**	**3000**	**4000**	**5000**	**10000**	**20000**
time(sec)	1,373	2,747	4,121	5,494	6,868	13,736	27,473



The constant values turned out to be:

$C_0 = 0$ (not measurable) $C_1 = 1{,}373626 \cdot 10^{-3}$

Considering these values for n=300 million the execution time would be 412087,8 seconds = 6868,13 minutes = 114,47 hours = 4,77 days.

Excersise 2

We add some modifications to the previous program making multiple loops. The running time function shall look something like:

$$t = C_0 + C_1 \cdot n^2$$

The modified source code follows:

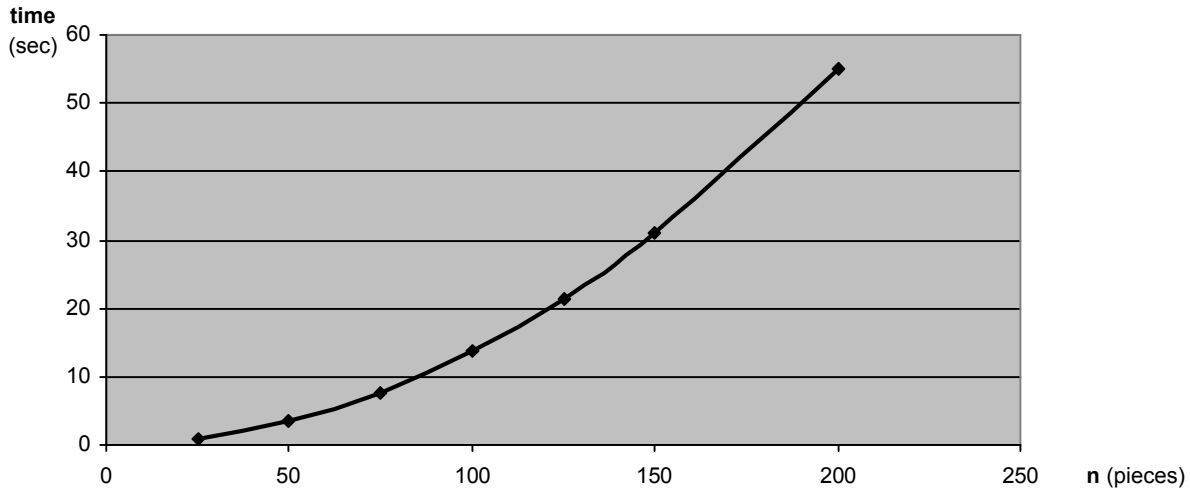
```
program E2;
var
   i,j,k,n:integer;
begin
readln(n);
i := 1; j := 1;
while j  <= n do
   begin
   k := 1;
   while k <= n do
        begin
             writeln(i);
             i := i + 1;
        k := k + 1;
        end;
   j := j + 1;
   end;
writeln('done'); readln;
end.
```

The C version:

```
void main()
{
  long int i, j, k, n;
  printf("n = ");
  scanf("%ld", &n);
  i = 1; j = 1;
  while (j <= n)
  {
       k = 1;
       while (k <= n)
       {
            printf("%ld\n", i);
            i++;
            k++;
       }
       j++;
  }
  printf("done\n"); getch();
}
```

We have measured the following values:

n	**25**	**50**	**75**	**100**	**125**	**150**	**200**
time(sec)	0,879	3,461	7,692	13,736	21,429	30,879	54,945



The constant values turned out to be:

$C_0 = 0{,}03663$ $C_1 = 1{,}369963 \cdot 10^{-3}$

Excersise 3

Now we modify the code to contain three multiple loops. We will measure the running time and guess the time function.

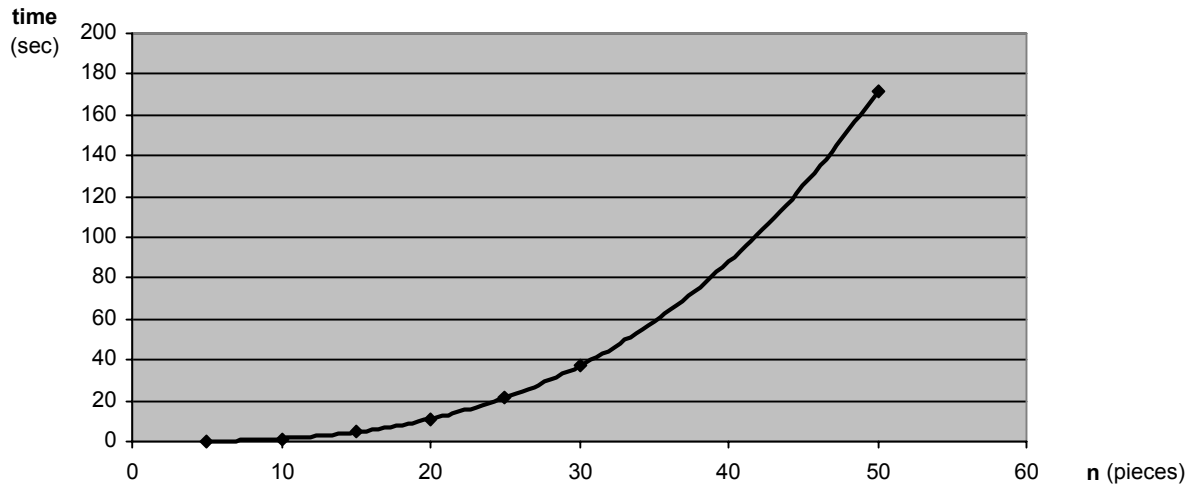
The C code:

```
void main()
{
     int i, j, k, l, n;
     printf("n = ");
     scanf("%d", &n);
     i = 1;
     j = 1;
     while (j <= n)
     {
          k = 1;
          while (k <= n)
          {
               l = 1;
               while (l <= n)
               {
                    printf("%d\n", i);
```

```
                    i++;
                    l++;
                }
                k++;
            }
            j++;
        }
        printf("done\n");
}
```

The following running times have been measured:

n	**5**	**10**	**15**	**20**	**25**	**30**	**50**
time(sec)	0,165	1,374	4,615	10,989	21,429	36,978	171,87



The running time function is approximately:

$t = C_0 + C_1 \cdot n^3$, where $C_0 = 0$ (unmeasurable) and $C_1 = 1{,}373626 \cdot 10^{-3}$.

Notice, that the C_0 and C_1 values are the same with the ones we measured in excercise 1.

Intractable Computations

Exercise 4 – Towers of Hanoi

In case of intractable computations, execution time increases with increasing n faster than any polynomial function. If we look at the measured data, we might get the concept of the word ’intractable’.

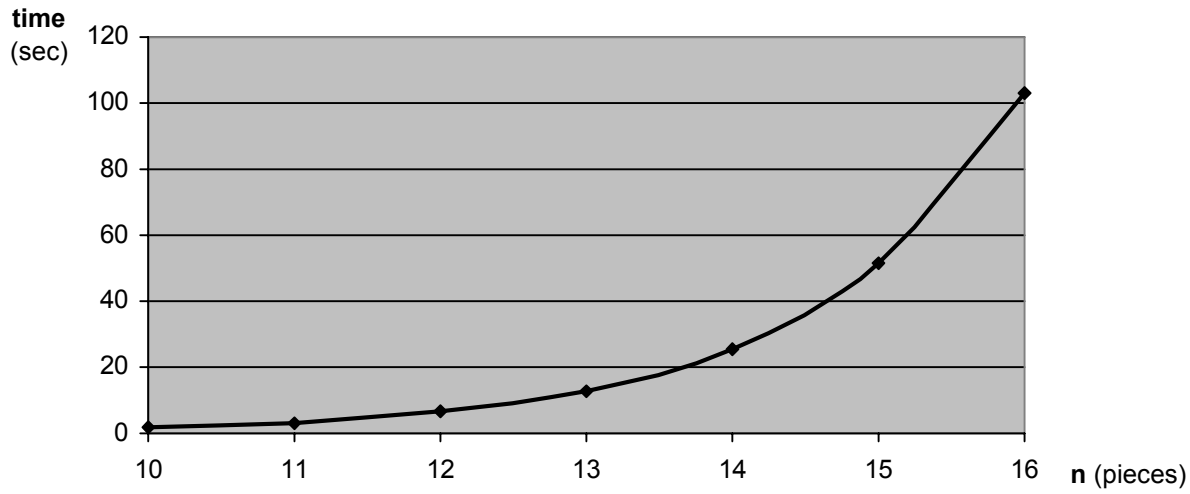
The famous problem of Towers of Hanoi turned out to be such a computation.The source code of the examined programme follows here:

#include <stdio.h>

```
#include <time.h>

void move_from_to(char from, char to);
void hanoi(int n, char from, char which, char to);

void main()
{
	int qdisks;
	printf("Quantity of disks: ");
	scanf("%d", &qdisks);
	printf("\n");
	hanoi(qdisks, 'A', 'B', 'C');
}

void hanoi(int n, char from, char which, char to)
{
	if (n==1)
	{
		move_from_to(from, to);
	}
	else
	{
		hanoi(n-1, from, to, which);
		move_from_to(from, to);
		hanoi(n-1, which, from, to);
	}
}

void move_from_to(char from, char to)
{
	printf("Put the disk from rod  %c", from);
	printf("  to rod  %c.\n", to);
}
```

We have measured the following running times:

n	**10**	**11**	**12**	**13**	**14**	**15**	**16**
time(sec)	1,593	3,241	6,483	12,857	25,714	51,538	103,29



The running time function is approximately:

$t = C_0 \cdot 2^n \qquad C_0 = 1{,}572814 \cdot 10^{-3}$

It contains an exponential function, which grows faster than any polinomial. For n=300 million it would be such a big number, that we even cannot write down.

Permutációt végző függvény és futásidejének vizsgálata

FELADATOK:

I.

Egy program elkészítése, amely n elemű karakterekből álló sztringek permutációját végzi.

A kimeneten meg kell jelennie a lehetséges permutációknak, a számolt futásidővel.

A permutáció során a véletlenszerűen előállított sztring elemeinek permutációját végezzük, azaz az elemek sorrendjének összes variációját írjuk le.

Pl.: A sztring : abc ; a kimenet : abc, acb, bac, bca, cab, cba.

Alkalmazott módszerek:

Magát a permutációt rekurzív módszerrel oldottuk meg. Meghatároztunk egy eredeti és egy cél sztringet. A permutáció során az eredeti sztringből a cél sztringbe mozgattuk a sztring elemeit egy másik, eddig még nem szerepelt kombinációval. Ezt rekurzívan ismételgetve az összes variációt megkaptuk.

LÁSD : A függelek III. pontjában.

II.

Utazó ügynök probléma megoldása a sztring permutáció segítségével, majd minimumkereséssel.

Alkalmazott módszerek:

Egy n elemű sztring permutációját végeztetjük el a fent említett módszerrel. A sztring i-ik eleme egy várost szimbolizál (i=1,...,n). Az egyes városok közötti utak hosszát véletlenszerűen meghatározzuk. Így egy mátrixban ábrázolva a városok közötti minden létező út hosszát feltüntetjük. Ezután minimum keresésével a létező utak közül kiválasszuk a legrövidebbet.

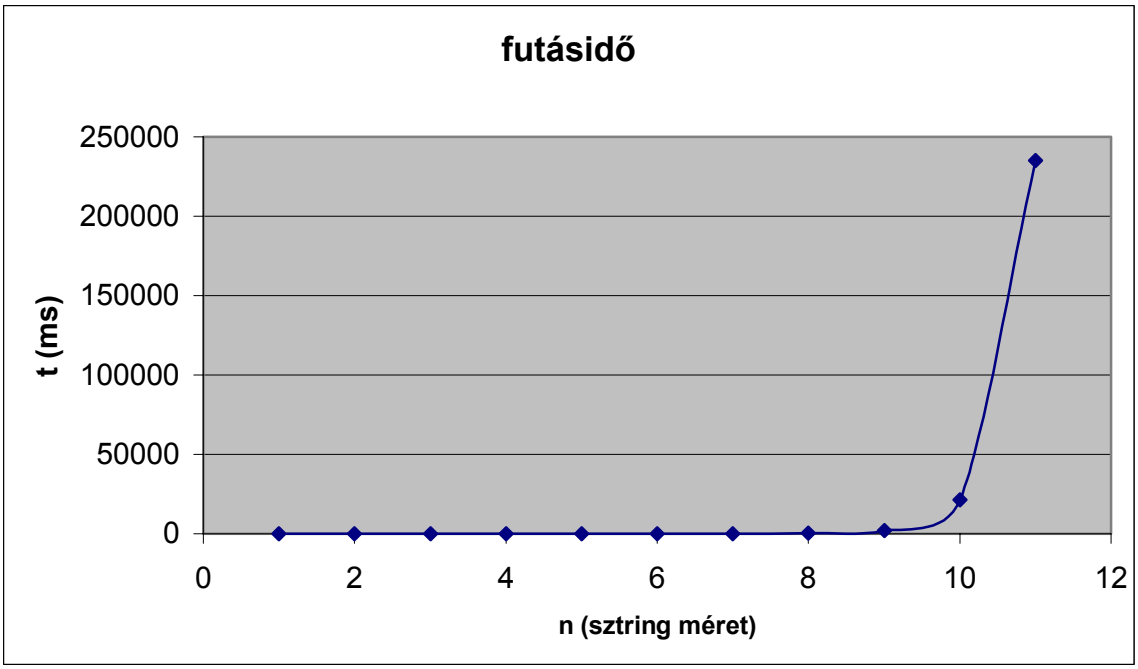
LÁSD : A függelek IV. pontjában

A feledatok komplexitása mindkét esetben n!, amit a Stirlig formulával $n^n \geq n! \geq \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ becsülve, exponenciális futásidőt kapunk, ezt a gyakorlatban is tapasztaltuk. A két feladat futásideje a Függelek I. II. pontjában látható.

Függelék:

I. Mérési eredmények (n) elemszámú, hosszú sztringekre:

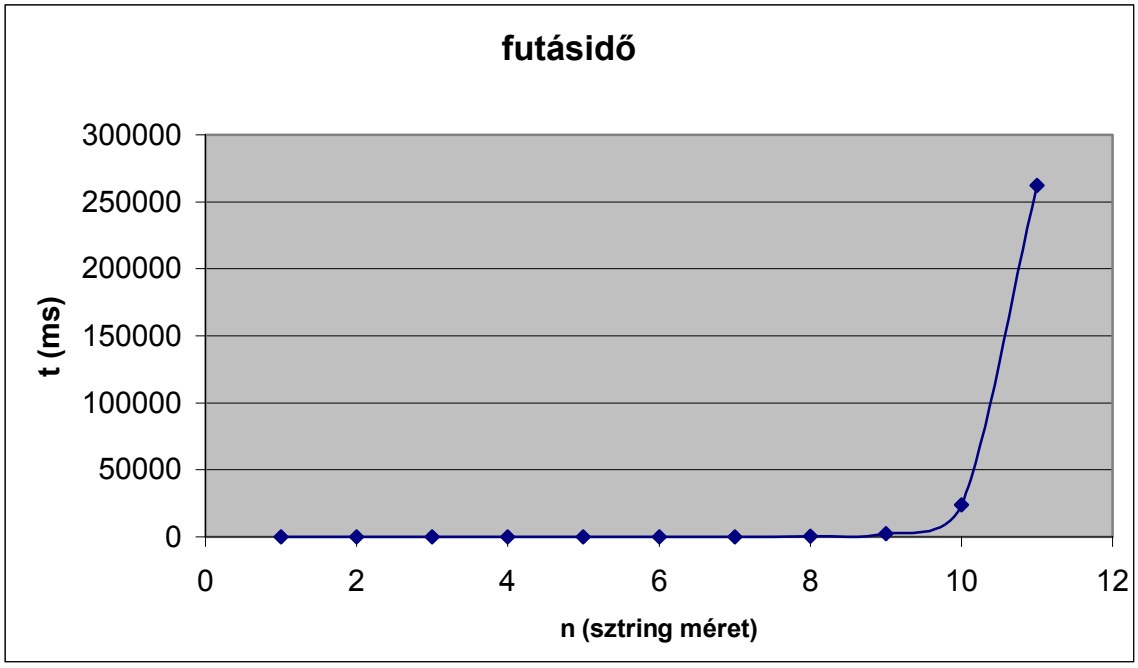
Sztring elemszáma (n)	Futásidő (ms)
1	0
2	0
3	0
4	0
5	0
6	4
7	33
8	240
9	2163
10	21451
11	234978



A futásidő ábrázolva sztringek permutációja esetén

II. A sztring permutáló program segítségével megoldott Utazó ügynök probléma futásidejei:

Sztring elemszáma (n)	Futásidő (ms)
1	0
2	0
3	0
4	0
5	0
6	7
7	33
8	271
9	2420
10	24021
11	262043



A futásidő ábrázolva az Utazó ügynök probléma esetén

III. A sztring permutációját végző program forráskódja:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>


char * strins(char* dest, char* source, unsigned int stay);
void Permutation( char *target, char *original);

int N;

void main(void)
{
        char *perm;
        char *blnk;
        clock_t ido1[2],ido2[2],ido;
```

```
        int k,elemszam;
        FILE* fp;

        fp=fopen("eredmeny.txt","wt");
        if (fp==NULL)
        {
                printf("Sikertelen file nyitas!\n");
                exit(-1);
        }
        fprintf(fp,"String merete:          futasido ezred mp-ben:\n\n");

        printf("Max sztring meret?\n");
        scanf("%d",&N);

        perm = (char*)calloc(N+1,sizeof(char));
        blnk = (char*)calloc(N+1,sizeof(char));
        if (perm == NULL || blnk == NULL) return;


        for(elemszam=0;elemszam<N;elemszam++)
        {


          ido1[0]=0;
          ido1[1]=0;
          ido2[0]=0;
          ido2[1]=0;


                perm[elemszam]='a'+elemszam;
                perm[elemszam+1]='\0';

                for (k=0;k<3;k++)          //a permutáció időmérése
                  {
                        ido1[0]=clock();
                        Permutation(blnk,perm);
                        ido2[0]=clock();

                        ido1[1]+=ido1[0];
                        ido2[1]+=ido2[0];
                }
                        ido=(ido2[1]/3)-(ido1[1]/3);

                fprintf (fp,"%10d %35d\n",elemszam+1,ido); // a mert idok fájlba tarolása
                printf ("%d meretu stringre a futasido ezred mpben: %d\n",elemszam+1,ido);
        }

        free(perm);
        free(blnk);

        fflush(fp);
        fclose(fp);

}


void Permutation( char *target, char *original) //a permutációt végző rekurzív fv
{
        int i;
        int len;
        char *target1,*original1;
```

```
        target1 = (char*)calloc(N+1,sizeof(char));
        original1 = (char*)calloc(N+1,sizeof(char));
        if ( target1 == NULL || original1 == NULL) return;

        i = 0;
        len = strlen(original);
        while ( i < len)    // egy lehetséges megoldás elkészitése
        {
                strcpy(target1,target);
                target1[strlen(target1)]=original[i];
                original1=strins(original1,original,i);
                Permutation(target1,original1);
                i++;
        }
        free(target1);
        free(original1);

}

char * strins(char* dest, char* source, unsigned int stay)
{
        unsigned int i,j=0;
        for(i=0;i<strlen(source);i++)
        {
                if(stay!=i)
                {
                        dest[j]=source[i];
                        j++;
                }

        }
        return dest;
}
```

IV. Az Utazó ügynök probléma megoldása a sztring permutációját végző programmal:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "math.h"



char * strins(char* dest, char* source, unsigned int stay);
void Permutation( char *target, char *original);



int N;
int min;
int **array;
char *temp;

void main(void)
{
        char *perm;
```

```
char *blnk;
clock_t ido1[2],ido2[2],ido;
int i,k,elemszam;
FILE* fp;


fp=fopen("eredmeny.txt","wt");
if (fp==NULL)
{
        printf("Sikertelen file nyitas!\n");
        exit(-1);
}
fprintf(fp,"String merete:          futasido ezred mp-ben:  minimalis: ut:\n\n");

printf("Max sztring meret?\n");
scanf("%d",&N);

perm = (char*)calloc(N+1,sizeof(char));
blnk = (char*)calloc(N+1,sizeof(char));
if (perm == NULL || blnk == NULL) return;

temp = (char*)calloc(N+1,sizeof(char));
if (temp == NULL) return;


srand( (unsigned)time( NULL ) );

array= (int**)calloc(N,sizeof(int*));
if(array==NULL)return;

for(k=0;k<N;k++)
{
        array[k]=(int*)calloc(N,sizeof(int));
        if(array[k]==NULL)return;
}

for(i=0;i<N;i++)                        //a súly mátrix feltöltése véletlen számmal
        for(k=i;k<N;k++)
        {
                if(i==k)array[i][k]=0;
                else
                {
                        array[k][i]=array[i][k]=(rand()*100)/RAND_MAX;

                }
        }


for(elemszam=0;elemszam<N;elemszam++)
{

  min=10000;

  ido1[0]=0;
  ido1[1]=0;
  ido2[0]=0;
  ido2[1]=0;


        perm[elemszam]='a'+elemszam;
        perm[elemszam+1]='\0';
```

```
                for (k=0;k<3;k++)                   //idő mérés
                  {

                        ido1[0]=clock();
                        Permutation(blnk,perm);
                        ido2[0]=clock();

                        ido1[1]+=ido1[0];
                        ido2[1]+=ido2[0];
                }
                        ido=(ido2[1]/3)-(ido1[1]/3);

        )



                fprintf (fp,"%10d %35d %15d %s\n",elemszam+1,ido,min,temp); // a mert idok fájlba tarolása
                printf ("%d meretu stringre a futasido ezred mpben: %d\n",elemszam+1,ido);
                printf ("A legrovidebb ut: %d es a kobinacio: %s\n",min,temp);
        }

        fprintf (fp,"  ------------------------------------------------\n");

        for(i=0;i<N;i++)
                for(k=0;k<N;k++)
                {
                        fprintf (fp,"%d,",array[i][k]);
                        if(k+1==N)fprintf (fp,"\n");
                }


        free(perm);
        free(blnk);

        fflush(fp);
        fclose(fp);

}


void Permutation( char *target, char *original)
{
        int i;
        int len;
        char *target1,*original1;


        target1 = (char*)calloc(N+1,sizeof(char));
        original1 = (char*)calloc(N+1,sizeof(char));
        if ( target1 == NULL || original1 == NULL) return;


        if (strlen(original) == 0)                  //a permutált sztringre a súlyok kiszámítása
                                                    //súlymátrix segítségével
        {
                int i;
                int osszeg=0;
                int hossz;
                hossz=(int)strlen(target);
```

```
                for(i=0;i<hossz;i++)
                {

                        if(i+1<hossz)
                        {
                                if((array[target[i]-'a'][target[i+1]-'a'] < (-1)) )
                                {
                                        osszeg=0;
                                        break;
                                }
                                osszeg+=array[target[i]-'a'][target[i+1]-'a'];
                        }
                        else
                        {
                                if((array[target[i]-'a'][target[0]-'a'] < (-1)) )
                                {
                                        osszeg=0;
                                        break;
                                }
                                osszeg+=array[target[i]-'a'][target[0]-'a'];
                        }
                }

                if(min>osszeg)
                {
                        min=osszeg;
                        strcpy(temp,target);

                }


        }
        else
        {
                i = 0;
                len = strlen(original);
                while ( i < len)    // egy lehetseges megoldas elkészítése
                {
                        strcpy(target1,target);
                        target1[strlen(target1)]=original[i];
                        original1=strins(original1,original,i);
                        Permutation(target1,original1);
                        i++;
                }
        }
        free(target1);
        free(original1);

}

char * strins(char* dest, char* source, unsigned int stay)
{
        unsigned int i,j=0;
        for(i=0;i<strlen(source);i++)
        {
                if(stay!=i)
                {
                        dest[j]=source[i];
                        j++;
                }
```

```
        }
        return dest;
}
```

Az utazó ügynök probléma

Legyen G=(V,E) egy súlyozott élű gráf a $w:E\rightarrow\mathbb{R}^+$ súlyfüggvénnyel. Az utazó ügynök probléma minimális összköltségű Hamilton kört keres G-ben, azaz egy olyan $H\subseteq E$ Hamilton kört, amelynek $w(H):=\sum w(e)$,$(e\in H)$ súlyánál kisebb súlyú Hamilton kör nincs G-ben. Vagyis az ügynök a legrövidebb/legolcsóbb úton kíván végigmenni.
⇒ ***Hamilton kör:*** Egy $C\subseteq G$ kör Hamilton kör, ha G gráf minden csúcsán pontosan egyszer halad át. A Hamilton körök mindig egyszerűek.

Az utazó ügynök problémájának megoldására két algoritmust alkalmazunk:

- ***Greedy***, azaz mohó algoritmus a kezdő csúcs (home city) szomszédai közül azt a csúcsot (várost) választja, amellyel a legkisebb súlyú él köti össze. Ezzel a módszerrel keresi a Hamilton kört a gráfban.

  Előnye: gyors

  Hátránya: nem mindig kapunk optimális megoldást

- ***Brute force*** algoritmus megvizsgálja az összes (n-1)!/2esetet, ezek közül választja ki az optimális Hamilton kört (n- a csúcsok száma a gráfban).

  Előnye: mindig az optimális megoldáshoz jutunk

  Hátránya: a városok számának növelésével egyre lassabb

Tetszőleges G gráfban Hamilton kör létezését eldöntő probléma *NP* teljes.
Egy probléma *NP* teljes, ha a problémára lenne gyors polinomiális algoritmus, akkor a világ összes *NP* teljes problémájára lenne gyors polinomiális algoritmus.□

```
program Travelling_Salesman;
   uses crt;
   const N = 6;
   type tablazat = array[1..N,1..N] of byte;
        vektor = array[1..N] of word;
   var t : tablazat;
       volt : vektor; {latogatas sorszamat tartalmazza}
       sorrend : array[1..N+1] of byte;
       minsum : word; { Brute force minimuma }
       s : string;
   procedure feltolt;
      var i,j : byte; { For ciklusvaltozok }
          r : byte;   { Veletlenszam }
    begin
     Randomize;
     for i:=1 to N do
      begin
       for j:=i+1 to N do
        begin
```

```
       r:=random(80)+20;      { Veletlen szam 20-100 kozott }
       t[i,j]:=r;
       t[j,i]:=r;
      end;
     t[i,i]:=0;
    end;
 end;
procedure kiir;
   var i,j : byte;
 begin
  writeln('------------------------------------------------------');
  for i:=1 to N do
   begin
     gotoxy(i*4,wherey);
     write(chr(ord('A')+i-1));
   end;
  writeln;
  writeln('------------------------------------------------------');
  for j:=1 to N do
   begin
    write(chr(ord('A')+j-1));
    for i:=1 to N do
     begin
      gotoxy(i*4,wherey);
      write(t[i,j]);
     end;
    writeln;
   end;
 end;

function kozelivaros(hely: byte):word; {varos sorszamat adja vissza}
   var legkozelebbivaros: word;
       i,j : byte;
 begin
   legkozelebbivaros:=100;
   j:=0;
   for i:=1 to N do
    begin
      if (t[i,hely]<legkozelebbivaros) and (volt[i]=0) then begin
         legkozelebbivaros:=t[i,hely];
         j:=i;
      end;
    end;
    kozelivaros:=j;
 end;
procedure moho(otthon : byte);
   var i : byte;
       most,kov,summa : word;
       lepesszam : word;
 begin
  summa:=0;
  for i:=1 to N do volt[i]:=0;
  lepesszam:=1;
  most:=otthon;
  kov:=most;
  s:='';
  for i:=1 to N do
   begin
    volt[most]:=lepesszam;
    kov:=kozelivaros(most);
    if i=N then kov:=otthon;
    summa:=summa+t[most,kov];
```

```
       s:=s+chr(ord('A')+most-1);
       Inc(lepesszam);
       most:=kov;
      end;
     s:=s+chr(ord('A')+otthon-1);
     writeln('"Greedy" algoritmus:');
     writeln('Az £tvonal: ',s);
     writeln('Az £t hossza: ',summa);
     writeln;
    end;
    procedure kiszamol;
       var i : byte;
           sum : word;
     begin
      sorrend[N+1]:=sorrend[1];
      sum:=0;
      for i:=1 to N do
       begin
        sum:=sum+t[sorrend[i],sorrend[i+1]];
       end;
      if sum<minsum then begin
       minsum:=sum;
       s:='';
       for i:=1 to N do s:=s+chr(ord('A')+sorrend[i]-1);
       s:=s+s[1];
      end;
     end;
    procedure brute_force(depth : byte);
       var i : byte;
     begin
      for i:=1 to N do
       begin
        if volt[i]=0 then begin
         volt[i]:=depth;
         sorrend[depth]:=i;
         if depth=N then kiszamol else brute_force(depth+1);
         volt[i]:=0;
        end;
       end;
     end;
    procedure nyersero(otthon : byte);
       var i : byte;   { For ciklus }
     begin
      for i:=1 to N do volt[i]:=0;
      minsum:=N*100;
      volt[otthon]:=1;
      sorrend[1]:=otthon;
      s:='';
      brute_force(2);
      writeln('"Brute force" algoritmus:');
      writeln('Az £tvonal: ',s);
      writeln('Az £t hossza: ',minsum);
      writeln;
     end;
begin
 ClrScr;
 feltolt;
 nyersero(1);
 writeln;
 moho(1);
 kiir;
end.
```

```
"Brute force" algotitmus:
Az útvonal: ACDBEA
Az út hossza: 268


"Greedy" algotitmus:
Az útvonal: AEDBCA
Az út hossza: 304

-------------------------
   A   B   C   D   E
-------------------------
A  0   97  85  98  70
B  97  0   90  35  34
C  85  90  0   44  46
D  98  35  44  0   24
E  70  34  46  24  0



"Brute force" algoritmus:
Az útvonal: ABCDEA
Az út hossza: 163


"Greedy" algoritmus:
Az útvonal: ABDCEA
Az út hossza: 170

-------------------------
   A   B   C   D   E
-------------------------
A  0   28  90  75  41
B  28  0   31  26  59
C  90  31  0   26  49
D  75  26  26  0   37
E  41  59  49  37  0


"Brute force" algoritmus:
Az útvonal: ADBCEA
Az út hossza: 251


"Greedy" algoritmus:
Az útvonal: AECBDA
Az út hossza: 251

-------------------------
   A   B   C   D   E
-------------------------
A  0   85  41  67  23
B  85  0   65  60  61
C  41  65  0   71  36
D  67  60  71  0   96
E  23  61  36  96  0



"Brute force" algoritmus:
Az útvonal: AEBCDFA
Az út hossza: 267
```

```
"Greedy" algoritmus:
Az útvonal: AFBDCEA
Az út hossza: 269

----------------------------
   A   B   C   D   E   F
----------------------------
A  0   97  86  56  51  38
B  97  0   44  36  46  20
C  86  44  0   45  79  58
D  56  36  45  0   90  43
E  51  46  79  90  0   81
F  38  20  58  43  81  0
```

```
"Brute force" algoritmus:
Az útvonal: ABEGFIKJDHCA
Az út hossza: 346
```

```
"Greedy" algoritmus:
Az útvonal: ACJKEGFIHDBA
Az út hossza: 414

-----------------------------------------------
   A   B   C   D   E   F   G   H   I   J   K
-----------------------------------------------
A  0   23  21  89  46  51  77  88  72  21  86
B  23  0   65  95  30  78  96  57  89  35  39
C  21  65  0   67  72  37  74  39  46  29  94
D  89  95  67  0   97  49  60  42  75  28  44
E  46  30  72  97  0   72  27  52  96  38  36
F  51  78  37  49  72  0   42  65  25  82  57
G  77  96  74  60  27  42  0   69  96  54  90
H  88  57  39  42  52  65  69  0   51  73  76
I  72  89  46  75  96  25  96  51  0   65  46
J  21  35  29  28  38  82  54  73  65  0   23
K  86  39  94  44  36  57  90  76  46  23  0
```

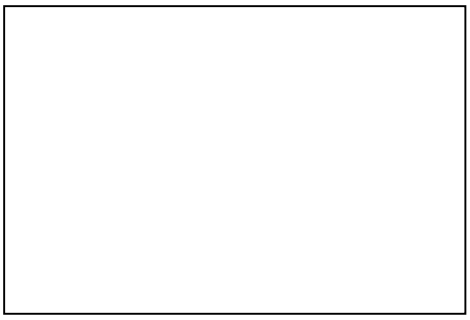
Bin-packing

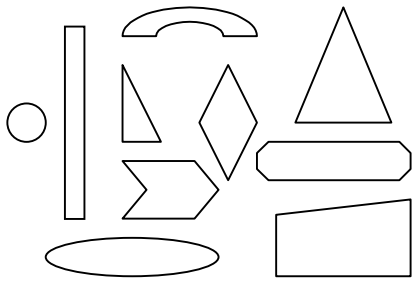
avagy egy adott felület legjobb lefedettsége

Probléma:

Adott:

- lefedendő felület
- különböző méretű és alakú síkidomok (elemek)

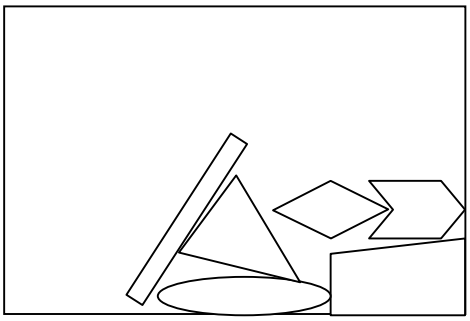


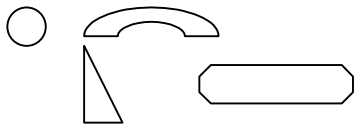


Lefedendő felületLefedésre felhasználható elemek

Feladat:

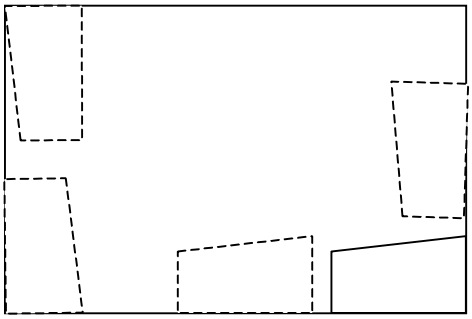
A megadott felületet az adott elemekkel úgy lefedni (az elemek nem fedhetik egymást), hogy a lefedetlenül maradt terület minimális legyen.



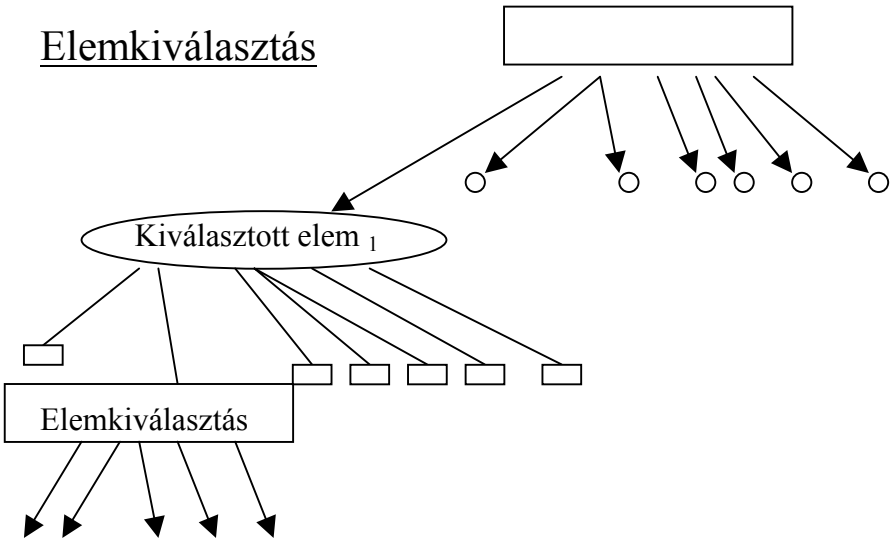


„Megoldás”:

Válasszunk ki egy elemet a megadottak közül, és ezt helyezzük rá a lefedendő felületre. Ezt az elemet sokféleképpen ráhelyezhetjük a felületre. Valamelyik ráhelyezést kiválasztjuk és rögzítjük. Majd veszünk egy másik elemet a megadott elemek közül és ezt is rátesszük valamilyen módon. Ezt is többféle módon helyezhetjük rá. Ezt folytatjuk mindaddig amíg van olyan elem, ami ráfér a még lefedetlen területre.



A Bin-Packing probléma NP-nehéz feladat, hiszen az első elem kiválasztása még egyszerűen megadható, de ezt az elemet sokféleképpen helyezhetjük rá a felületre. Minden egyes ráhelyezéshez egy-egy fa rajzolható.



A fa egyre ágasabb és ágasabb minél mélyebb szintre megyünk. Így az ábrából is látható, hogy nem tudjuk megadni az összes lehetőség számát, mert nem tudjuk megadni azt, hogy egy elemet hányféleképpen helyezhetünk rá az éppen még lefedetlen felületre. A megoldásokat a fa levelei jelentik. A feladat azért NP - nehéz, mert ha az optimális megoldást a levelek közül kellene kiválasztani, akkor arra nem létezik olyan gyors algoritmus, ami ezt polinomiális időben megtenné. Ha lenne ilyen algoritmus, akkor a világ összes hasonlóan bonyolult problémájára megadhatnánk a választ polinomiális időben.

A feladatra heurisztikus megoldások adhatók, de ezekből nem dönthető el, hogy az adott megoldásnál létezik-e jobb megoldás.

A feladat egyik módozata, mikor nem ismerjük előre az elhelyezendő elemeket. Az elemeket on-line módon kapjuk. A beérkezett elemet le kell tennünk valamilyen módon és helyzetén később már nem változtathatunk.

A feladat egyszerűbbnek tűnő változata:
2D-s téglalapok pakolása
A lefedendő felületek (kosarak) és az elemek különböző méretű téglalapok.
A feladat:

- a kosarakat (felületeket) egymás után kell lefedni
- minden felületen az elemek oldalai párhuzamosak a felület oldalaival
- a már egyszer elhelyezett elemek helyzetét már nem módosíthatjuk
- az elem szélessége igazodik a felület szélességéhez

Cél: minimalizálni kell a felhasznált felületek (kosarak) számát.

Gyakorlati alkalmazásai a Bin-packing -nak:

- tetris (on-line)
- szabászat
- üvegtáblából megfelelő méretű üvegdarabok kivágása
- raktárak minél jobb kihasználása
- térelrendezés, tér legjobb kitöltése
- raktérbe történő bepakolás

Legjobb választások keresése döntési környezetekben

A feladat egy olyan program írása volt, ami egy összetett döntési környezetben képes gyorsan, és bizonyos pontossággal dönteni. Az elkészült példaprogram dáma játékot játszik két játékos között, akiket a gép irányit.

A dáma játékot sakktáblán játsszák, a bábuk csak fehér mezőkön mozoghatnak, vagy egyet a szomszédos mezőre, de csak előre, vagy egy ellenséges bábut átugorva, és azt levéve kettőt. Amennyiben az egyik bábunk az ellenfél alapvonalára ér dáma lesz belőle, ami minden irányban léphet bármennyit, visszafelé is.

Bár a feladatot ilyen egyszerű szabályokkal meg lehet fogalmazni, a nehézséget az jelenti, hogy amennyiben jól akarunk lépni nagy mélységben előre kell gondolkoznunk. Mivel átlagosan 7 lépési lehetőség van minden alkalommal, igy egy olyan fát kapunk aminek mérete exponenciálisan növekszik. Ezért szükség van olyan eszközökre, amikkel a vizsgált részeket nagy mértékben csökkenteni tudjuk anélkül, hogy a lépések jóságából, jelentősen, veszítenénk.

Az ilyen feladatokra több módszert ismerünk (B&B, MINIMAX), mi a minimax megoldást valósítottuk meg. Ehhez szükségünk van egy kiértékelő függvényre, ami a játék bármely állásához tud egy értéket rendelni. Ekkor két játékosunk, akiket nevezünk min. és max. játékosoknak, egymás ellen játszik, egyikük a kiértékelő függvény minimumára, a másik a maximumára törekszik.

Az egyszerűség kedvéért a kiértékelő függvény a következő:
F = 1*FEHER_BABUK_SZAMA + 5*FEHER_DAMAK_SZAMA – 1*FEKETE_BABUK_SZAMA – 5*FEKETE_DAMAK_SZAMA
A dámák igen jelentős szereplői ennek a játéknak, ezért van ilyen nagy súlyúk.

A továbbiakban meg kell határoznunk egy döntési stratégiát, amit majd a játékosok követnek. Természetesen mindkét játékos a saját céljait akarja elérni ezen stratégián keresztül, tehát a minimumot vagy a maximumot. A program stratégiája:

A megadott mélységig minden szinten csak a első n. legjobb eredményt terjeszti, vizsgálja tovább. Így egy olyan fát kap ami szintén exponenciális ugyan, de mérete korlátozható a mélységgel, és azzal, hogy hány darab legjobbat vizsgálunk.

Ezen az egyszerű feladaton látható, hogy a döntési feladatok milyen kombinatorikus korlátokba ütköznek. Amennyiben a minimax módszert használjuk ezen korlátok leküzdésére, nagy figyelmet kell szentelni a kiértékelő függvény minél alkalmasabb meghatározására, valamint a feladathoz illő stratégia kiválasztására.

Problémák kezelhetőségének vizsgálata

A futási idők vizsgálata alapján a problémákat két osztályba sorolhatjuk a kezelhető és a kezelhetetlen problémák osztályába. Az előbbi osztály tartalmazza azokat a problémákat amelyeknek a futási ideje elfogadható a legtöbb esetben, még akkor is ha nagy mennyiségű adatot kell feldolgoznia. A második osztályba tartozó feladatok esetén a megoldáshoz általában mérhetetlenül sok idő kell, ha a feladat mérete nagy. Mindkét fajta számítás fontos a gyakorlatban. Némely kezelhetetlen feladat esetén közelítő megoldás adható elfogadható időn belül, és ez elég pontos, hogy a gyakorlatban is alkalmazni tudjuk. Sok ilyen probléma és a közelítő megoldások megtalálása a számítástudományi kutatások tárgya. Lehetséges, hogy a számításokat nem tudjuk elvégezni az idő mellet más erőforrás hiánya miatt sem. El lehet képzelni olyan számításokat amelyeknek mérhetetlenül sok memória kell, és ezek a memória korlátozott mérete miatt nem szolgáltatnak eredményt. A gyakorlatban ez ritkán fordul elő, fontosabb eset amikor a feladat a futási idő nagysága miatt kezelhetetlen.

A polinomiális rendben megoldható feladatot kezelhetőnek nevezzük, azaz az olyan feladatot aminek a megoldásához szükséges idő a feladat méretének polinomiális függvénye. Ha egy feladat megoldásához szükséges idő a feladat méretének exponenciális függvénye akkor a feladatot kezelhetetlennek nevezzük.

Három feladat kezelhetőségét vizsgáltuk.
A három feladat a következő:

1. egy n elemű listából kell eltávolítani azokat a listaelemeket amelyek többször előfordulnak a listában
2. n nevet tartalmazó telefonkönyvből kell kikeresni egy adott névhez tartozó telefonszámot
3. egy gazdasági modellben egy döntési sorozatot kell megadni egy n hónap hosszúságú intervallumra úgy, hogy a n. hónap végén a profit a lehető legnagyobb legyen

Mindegyik feladatról azt akarjuk eldönteni, hogy kezelhető vagy kezelhetetlen.

Az első feladat esetén egy n elmű listából kell eltávolítani a többszörös listaelemeket. Az n elemű lista minden eleméről el kell dönteni, hogy szerepel-e belőle másik példány a listában, amelyet törölni kell. Ezt legegyszerűbben úgy tehetjük meg, hogy a lista minden elemét összehasonlítjuk a többi elemmel, és ha ugyanolyan elemet találunk akkor azt töröljük. Ezt n elemű lista esetén legrosszabb esetben n*(n-1) lépésben tehetjük meg. Az feladatot általában ennél kevesebb lépésben elvégezhetjük, mert a már törölt elemeket nem kell a későbbiek során vizsgálni. Ha egy elemet csak a listában utána következő elemekkel hasonlítunk össze akkor $\sum_{i=1}^{n-1} n-i = \frac{n*(n-1)}{2}$ lépés szükséges a feladat elvégzéséhez. (Az i. elemet az utána következő n-i elemmel kell összehasonlítani.) Ebben az esetben is ez a legrosszabb eset, ennél általában kevesebb lépésre van szükség. Mindkét algoritmus futási ideje a méretének polinomiális függvénye tehát kezelhető a probléma. (Mindkét algoritmus $\theta(n^2)$ rendű.) Ha a listában levő elemek sorrendjét nem kell megőrizni, akkor egy másik algoritmussal ennél jobb eredmény is elérhető. Egy $\theta(n*\log n)$ rendű rendezéssel rendezzük a listát majd ezután minden elemet csak a szomszédjával kell összehasonlítani. Ekkor a lépések száma $\theta(n*\log n)+n$ tehát az algoritmus $\theta(n*\log n)$ rendű. A használt rendezés lehet például gyorsrendezés.

A második feladatban egy n bejegyzést tartalmazó telefonkönyvben kell megkeresni egy névhez tartozó telefonszámot. Legrosszabb esetben az összes bejegyzést meg kell vizsgálni. Mivel a nevek hossza nem függ a feladat méretétől ezért az algoritmus futási ideje a bejegyzések számával arányos. Hash tábla alkalmazásával a keresett telefonszám jóval kevesebb lépésben is megtalálható. Ha a hasítófüggvényt úgy képezzük, hogy minden névnek más kulcs feleljen meg akkor egy lépésben megtalálhatjuk a keresett telefonszámot. Ebben az esetben nagyon sok memóriára van szükség, ami miatt a feladat kezelhetetlenné válhat. Ha olyan hasítófüggvényt használunk ahol több névhez is tartozhat ugyanaz a kulcs, akkor a memóriaigény csökkenthető és a keresés sebessége is jelentősen növelhető a szekvenciális kereséshez képest. Mivel az algoritmus futási ideje a feladat méretével arányos ezért a probléma kezelhető.

A harmadik feladat, egy gazdasági modellben n hónapra döntési sorozatot kell készíteni úgy, hogy a profit maximális legyen. Minden hónapban véges számú választási lehetőség van amik közül ki kell választani azt amelyre az n hónap után a profit maximális. Valamilyen kiértékelő függvény segítségével a választott döntések alapján eldönthető az n hónap után várható profit. A döntések függetlenek egymástól, tehát ha az egyik hónapban hoztunk valamilyen döntést akkor ez nem határozza meg, hogy a következő hónapban milyen döntést kell hozni. A döntések csak olyan értelemben befolyásolják a későbbi döntéseket, hogy a választási lehetőségek változnak a korábban meghozott döntések függvényében. Ezért a döntések összes kombinációjára meg kell határozni a várható profitot. Ha minden hónapban m választási lehetőségünk van akkor a lehetséges kombinációk száma m^n. Ezek alapján a futási idő a feladat méretének exponenciális függvénye, tehát a feladat kezelhetetlen. A megvizsgálandó döntési sorozatok száma különböző módszerekkel csökkenthető, de a feladat exponenciális jellege akkor is megmarad.

4-el való oszthatóság

10-es számrendszerben:

Alapelv: 4-el való oszthatóság vizsgálatakor elég a 2 legkisebb helyiértékû számjegyet vizsgálni.

Σ:={#,0,1,2,3,4,5,6,7,8,9} X:={L, R, N}
S:={q_0,q_1,q_2,....,q_9,q_Y,q_N}
P$\subseteq$S$_x$$\Sigma_x$$\Sigma_xX_x$S

P={

q0	#	#	L	q1
q1	#	#	R	q6
q1	0	0	L	q2
q1	4	4	L	q2
q1	8	8	L	q2
q1	1	1	L	q3
q1	2	2	L	q4
q1	3	3	L	q3
q1	5	5	L	q3
q1	6	6	L	q4
q1	7	7	L	q3
q1	9	9	L	q3
q2	0	0	L	q5
q2	2	2	L	q5
q2	4	4	L	q5
q2	6	6	L	q5
q2	8	8	L	q5
q2	#	#	R	q6
q2	1	1	L	q3
q2	3	3	L	q3
q2	5	5	L	q3
q2	7	7	L	q3
q2	9	9	L	q3
q3	0	0	L	q3
....	3			
	5			
	7			
q4	9	9	L	q5
q5	0	0	L	q5
....				
q5	9	9	L	q5

```
        q5      #       #       R       q7
        q6      0       N       R       q8
        ....
        q6      9       N       R       q8
        q7      0       Y       R       q9
        ......
        q7      9       Y       R       q9
        q8      0       #       N       qN
        ....
        q8      9       #       N       qN
        q9      0       #       N       qY

        q9      9       #       N       qY
                                                }
```

q0 alap hibaellenõrzés
q1 1. Számjegy vizsgálata
q2 2. Számjegy vizsgálata
q3 Hiba # keresés balra
q4 2. Számjegy vizsgálata
q5 OK # keresés
q6 N kiírása
q7 Y kiírása
q8 végsõ # kiírása (N eset)
q9 végsõ # kiírása (Y eset)

4-el oszthatóság vizsgálata bináris számokon

Σ:={#,0,1,2,3,4,5,6,7,8,9} X:={L, R, N}
S:={q_0,q_1,q_4,q_6,q_7,q_9,q_{10},q_Y,q_N}
P$\subseteq S_x\Sigma_x\Sigma_x X_x S$

P={

q0	#	#	L	q1
q1	0	#	L	q4
q1	1	#	L	q9
q4	#	#	R	q9
q4	0	#	L	q9
q4	1	#	L	q6
q6	0	#	L	q6
q6	1	#	L	q6
q6	#	#	R	q7
q7	#	Y	N	qY
q9	0	#	L	q9
q9	1	#	L	q9
q9	#	#	R	q10
q10	#	N	N	qN

4-el oszthatóság feltétele bináris számok esetében az, hogy a két legkisebb helyiértékû számjegy 0 legyen

Programozási nyelvek elmélete,

A Clay Mathematics Institute – nonprofit oktatási alapítvány az USA-ban – 1 millió dolláros díjat ajánlott fel a 7 híres feladat megoldásáért. Ezek egyike a notorius P vagy NP kérdés. Annak ellenére, hogy ez egy ördögien bonyolult rejtvény, egy okos amatőr is képes lehet megoldani a majdnem minden PC-n futó Aknakereső játék segítségével.

Richard Kaye (Birminghami Egyetem) egy közelmúltban megjelent – Az Aknakereső NP-teljes - cikkében mutatott rá a a játék és a P vagy NP kérdés közötti kapcsolatra.

Először nézzük, hogyan kell az Aknakeresőt játszani. Kezdéskor a számítógép egy négyzethálót mutat, üres négyzetekkel, amelyek közül néhány robbanó aknákat tartalmaz. Feladatunk kitalálni, hol vannak ezek az aknák, annélkül, hogy felrobbannánk. Első lépésként bármelyik üres négyzetet választhatjuk. Ha ez alatt akna van, pech, az akna felrobbant minket, és elveszítjük a játékot. Ellenben ha első lépésünkre nem lépünk aknára, a számítógép egy számot ír abba a négyzetbe, ami megmondja, hány akna van a 8 szomszédos mezőben.

Ezt az információt felhasználva határozzuk meg, melyik legyen a következő felfedendő mező, és újra két lehetőség van: vagy felrobbanunk, vagy felfedünk még egy mezőt, benne a szomszédos aknákat mutató számmal. Ha az akna lehetséges helyét egy négyzetre korlátoztuk, akkor azt egy kis zászlóval megjelöljük. Néhány lépés utáni tipikus állás látható a bal felső ábrán. Akkor nyerünk, ha minden akna helyét meghatároztuk a négyzethálóban.

> Most nézzük a P vagy NP rejtvényt! Emlékeztetőül: az algoritmus egy lépésről-lépésre végrehajtható eljárás, amit számítógépen lehet futtatni feladatok megoldására. Központi kérdés a számítógépes matematikában, hogy: Milyen hatékonyan tud egy algoritmus egy adott problémát megoldani? Más szóval, hogyan nő a futási idő – a válasz produkálásához szükséges elvégzendő számítások száma – a kiinduló adatoktól függően. A feladatokat nehézség, bonyolultság szerint két csoportba sorolhatjuk, vannak a P típusuak, amik polinom idő alatt lefutnak, illetve amik nem. Egy feladat P típusú, ha megoldható olyan algoritmussal, aminek futási ideje nem nő gyorsabban mint néhány szimbólum – amiket a kezdeti adatok határoznak meg - fix hatványa. Ha a megoldás nem található meg ily módon, akkor a probléma nem-P. A P típusú feladatok hatékonyan kezelhetők számítógéppel, míg a nem-P feladatok nem oldhatok meg ilyen egyszerűen, mivel bármely megoldó algoritmus futási ideje túlságosan nagy lenne ahhoz, hogy választ kaphassunk.

Bebizonyíthatod, hogy egy probléma P típusú, ha találsz olyan algoritmust, ami polinom idő alatt oldja meg. Például egy számhalmazt növekvő vagy csökkenő sorba rendezni P típusú, ezért tudnak az adatbázis programok hatékonyan sorba rendezni nagy mennyiségű számot. Ezzel szemben az Utazó ügynök problémájáról – találjuk meg a legrövidebb utat, ami során az ügynök

minden várost meg tud látogatnni az utitervéből – széles körben hiszik, hogy nem-P, holott ezt még nem bizonyították be. Egy szám prímtényezőinek megtalálása szintén nem-P, de még ezt sem bizonyították.

Miért olyan nehéz bebizonyítani egy feladatról, hogy nem-P? Azért, mert nem elég csupán néhány algoritmust elemezni, végig kell nézni minden lehetséges algoritmust és megmutatni, hogy egyik sem oldja meg polinom idő alatt az adott feladatot. A legjobb módszer: bebizonyítani, hogy a nem-P feladatok egy széles csoportja azonos tulajdonságokkal rendelkezik, azonos osztályba tartozik. Ha ezek közül bármelyik polinom idő alatt megoldható, akkor a többi is. Ezekre a problémákra mondják, hogy nemdeterminisztikus polinom futási idejűek, ezért hívják ezeket NP típusúnak.

Az NP az nem ugyanaz, mint a nem P. Egy feladat NP, ha polinom idő alatt le lehet ellenőrizni, hogy egy adott megoldás jó-e. Ez jóval kevésbé szigorú feltétel – legalábbis annak látszik - , mint megtalálni a megoldást polinom idő alatt. Jó példa erre a kirakós puzzle. Egy ilyen puzzle megfejtése nagyon bonyolult lehet, de ha valaki kirakta, akkor általában néhány másodperc alatt le lehet ellenőrizni, hogy az jó-e vagy sem. Csak meg kell nézni minden darabot, hogy passzol-e a szomszédjával. Az ehhez szükséges idő kvázi független a kirakó darabszámától, az ellenőrzés polinom idő alatt fut le. De ilyen puzzle-okat nem lehet polinom idő alatt kirakni. Ha úgy próbáljuk megoldani a feladatot, hogy kipróbálunk minden lehetséges megoldást, akkor a futási idő hatalmassá válik, mivel a potenciális megoldások száma sokkal gyorsabban nő, mint a darabszám bármely rögzített hatványa.

Kiderült, hogy sok NP problémának azonos nagyságrendű a futási ideje. Speciálisan, egy NP problémát NP-teljesnek mondanak, ha az adott problémán talált polinom idejű megoldás maga után vonzaná minden NP probléma polinom idejű megoldhatóságát. Ezért ha meg tudunk oldani egy NP-teljes feladatot polinom idő alatt, akkor mindet meg tudjuk oldani.

Itt jön a P vagy NP kérdés, vajon ugyanaz-e a P és az NP, minden megjelenésbeli különbség ellenére? Azt várjuk. hogy nem. De ha bármelyik NP-teljes feladatról kiderül, hogy van polinom idejű megoldása, akkor az NP-nek azonosnak kell lennie P-vel.

Sok NP-teljesnek hitt probléma létezik. A legegyszerűbbek egyike a SAT-probléma, benne a logikai áramkörök, ÉS, VAGY és a NEM kapuk. A bevitt adat vagy T(true), vagy F(false), és azt nézzük, mi jön ki válaszként. Például egy NEM kapu T inputot F outputra változtat, és fordítva. A SAT-probléma arra keres választ, van-e adott logikai áramkörre olyan bemenet, amire a válasz T lesz. A feladat gyerekjáték egy egyszerű rendszer esetén, de nagyon bonyolulttá válik, ha a hálózat sok kaput és beviteli adatot tartalmaz.

Itt jön a képbe az Aknakereső Konzisztencia Problémát (AKP). Nem az aknák megtalálni, hanem azt kell megvizsgálni, hogy egy adott állás logikailag konzisztens-e. Ellenpélda: a jobb felső ábrán látható.

Cikkében Kaye bebizonyítja, hogy a SAT-probléma bármilyen adott logikai áramkörre átalakítható AKP-re, speciális állásokkal. Sőt, megmutatja, hogy ez a konvertálás polinomiális idő alatt fut. Ebben a változatban a logikai kapuk in- és output adatai az

aknák elhelyezkedése a négyzethálóban: az aknát tartalmazó négyzet a T helyett van, míg az üres az F-et helyettesíti. A vezeték, ami összekapcsolja a hálózat kapuit, a középső ábrán van illusztrálva. Ebben az Aknakerseő „vezetékben" az y-nal jelölt négyzetek értéke ellentétes az x-szel jelöltének – ha az x mező tartalmaz aknát (T), akkor az y nem (F), illetve fordítva. Láthatjuk, hogy a többi négyzetben látható számok helyesek mindkét x értékre. A vezeték hatása : továbbítja a jelet (T v. F) hosszanti irányban.

Hasonló eljárással írható le maga a logikai kapu is. Az Aknakereső állás az alsó ábrán egy NEM kaput ábrázol. A középen álló számok okozzák x és y pozícióinak cseréjét: x y után jön a bemeneten, de megelőzi azt a kimenő vezetéken. Pontosan ezt várjuk el egy NEM kaputól.

Hajlítás és ketté ágaztatás is probléma, de Kaye-nek mindezeket sikerül megoldania cikkében. A nagy „fegyvertény" az lenne, ha valaki egy adott állásra meg tudná oldani AKP-t polinom idő alatt. Ekkor a SAT-probléma is megoldott lenne az ekvivalens áramkörre polinom idő alatt. Vagyis az Aknakereső NP-teljes. Tehát ha valaki talál megoldást a AKP-re, akkor minden NP problémának lenne polinom idejű megoldása, vagyis P megegyezne NP-vel. Másik lehetőség, ha valaki bebizonyítja, hogy nem létezik megoldás az Aknakereső problémára, akkor P nem egyenlő NP-vel.

Mielőtt nagyon felvillanyozódnánk, megjegyezzük, hogy az Aknakereső Konzisztencia Probléma nehéz feladat. Annak meghatározása, hogy egy adott állás konzisztens-e, nagyon bonyolulttá válik ha hatalmas tábláról van szó, és a legtöbb matematikus és tudós úgy gondolja, hogy nincs általános polinom idejű megoldás. Sőt, a Clay Institute szigorú feltételeket szabott a résztvevőknek: mielőtt egy megoldást jónak fogadnak el, publikálni kell egy elismert folyóiratban és a matematikai társadalomnak általánosan el kell fogadnia a publikációt követő két éven belül.

Zárszó: Talán sosme tudjuk bebizonyítani, de egyre több módon járjuk körül a P vagy NP problémát. Hiába vannak egyre jobb gépek, az emberi intuíciót és tudást egyelőre nem pótolhatja semmi, még a csúcsszámítógépek sem.

Irodalom: Scientific American, October 2000, Mathematical Recreations, Computation_Algorithms

Önálló projekt feladat

A 3SAT probléma NP-teljességének bizonyítása

1. A 3SAT probléma:

Definíció:

Adott egy logikai kifejezés konjunktív normál formában, amelynek minden tagja pontosan 3 változót tartalmaz.

Döntsük el, hogy a kifejezés kielégíthető-e, azaz van-e olyan értéke a változóinak, amelyre a kifejezés igaz.

2. Az NP-teljesség bizonyítása:

Gyakorlati módszer:

1. A feladat a P-osztályba tartozik? Ha nem…
2. A feladat közelebbi vizsgálata
3. A feladat az NP-osztályba tartozik?
4. Egy ismert NP-teljes probléma polinomiálisan transzformálható-e az adott feladatba?

Az utolsó két feltétel teljesülése esetén az adott feladat NP-teljes.

A konkrét megoldás:

A 3SAT probléma NP-teljességének bizonyításához megmutatjuk, hogy a 3SAT probléma egy megoldása felhasználható a SAT probléma megoldásához.

1. A 3SAT probléma egyértelműen NP-osztályú.
   Ellenőrzés (checking): A kifejezés egy tetszőleges megoldásának helyességét polinom időben ellenőrizni tudjuk.

2. Legyen **E egy tetszőleges SAT feladat**. E minden tagját helyettesíthetjük olyan tagokkal, amelyek pontosan 3 változót tartalmaznak:

Legyen $\mathbf{C = (x_1 + x_2 + .. + x_k)}$ **E egy tetszőleges tagja, amelyre k >= 4.**

A jelölés egyszerűsítésének érdekében tegyük fel, hogy minden változó ponált (nem negált).

C helyettesíthető 3 változót tartalmazó tagokkal. Az ötlet **új változók $y_1, y_2, .. , y_{k-3}$** bevezetése, amelyek segítségével C 3SAT formára alakítható, anélkül, hogy C „kielégíthetősége” megváltozna.

Az új változók segítségével C transzformálható C’- be, ahol

$$\mathbf{C' = (x_1 + x_2 + \overline{y_1}) (x_3 + y_1 + \overline{y_2}) (x_4 + y_2 + y_3) .. (x_{k-1} + x_k + y_{k-3})}$$

Állítjuk, hogy **C’ pontosan akkor elégíthető ki, ha C kielégíthető:**

Ha C igaz/kielégíthető, akkor legalább egy $x_i = 1$.
Ekkor az y_i értékeket megadhatjuk úgy, hogy C’ minden tagja igaz legyen:
Pl.:

$x_3 = 1$

ekkor legyen: $y_1 = 1$ (így C' első tagja rendben)
$y_2 = 0$ (ezzel a második tag is rendben)
és a többi $y_i = 0$.

Általánosan: **Ha $x_i = 1$, akkor $y_1, y_2, .. y_{i-2} = 1$, a többi pedig 0.**

Ekkor C' igaz/kielégített.

Fordítva: Ha C' igaz, akkor legalább egy $x_i = 1$.
Ha minden $x_i = 0$, akkor a kifejezés a következőképpen alakul:

$\mathbf{C' = (y_1) (\overline{y}_1 + y_2) (\overline{y}_2 + y_3) .. (y_{k-3})}$, amely kifejezés egyértelműen kielégíthetetlen/hamis.

Ezzel a redukcióval minden olyan tagot, amely háromnál több változót tartalmaz, átalakíthatunk pontosan 3 változót tartalmazó tagokból álló kifejezéssé.

Már csak az **egy és két változós tagok**at kell transzformálnunk.

Ha $\mathbf{C = (x_1 + x_2)}$, z új változó bevezetésével:

$$\overline{\mathbf{C'}} = \mathbf{(x_1 + x_2 + z) (x_1 + x_2 + z)}$$

Ha $\mathbf{C = x_1}$, akkor y és z új változók bevezetésével:

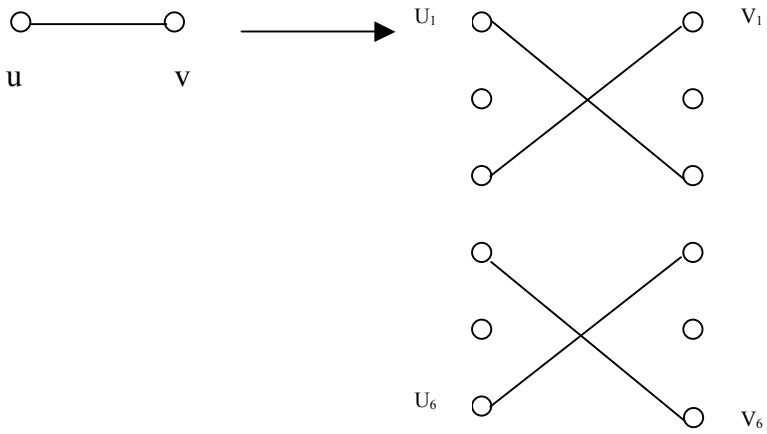
$$\mathbf{C' = (x_1 + y + z) (\overline{x}_1 + y + z) (x_1 + \overline{y} + \overline{z}) (\overline{x}_1 + y + z)}$$

Így redukáltunk egy SAT feladatot, egy 3SAT feladatba, úgy, hogy az egyik feladat pontosan akkor elégíthető ki, ha a másik kielégíthető; és a redukció polinom időben elvégezhető.
Tehát bebizonyítottuk, hogy a 3SAT feladat NP-teljes.

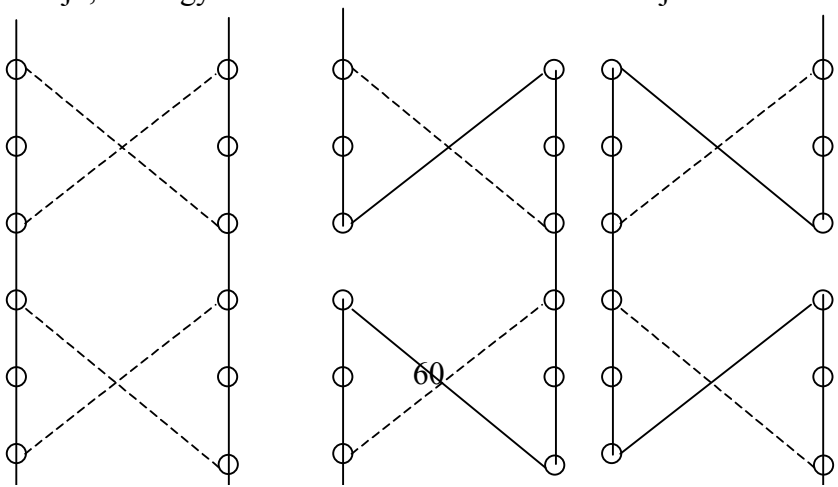
Az Utazó Ügynök probléma visszavezethető a hamilton kör keresésre, ha a gráf éleinek a súlyát egységesen 1-nek választjuk, a nem gráfbeli éleket pedig 2 súlyúnak. A kérdés az, hogy a súlyozott éleket bejárva (minden csúcsot csak egyszer érintve) a súlyok összege kisebb-e mint egy alkalmasan megválasztott k érték . Ha belátjuk, hogy a Hamilton kör keresés NP-teljes, akkor az utazó ügynök probléma is az.

Tétel: A Hamilton kör keresés NP-teljes

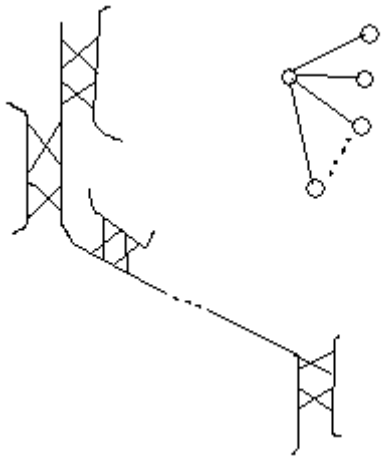
Bizonyítás: Egy gráf lefedését használjuk a bizonyításhoz, ami egy gráfot és egy k konstanst jelent, k a gráf legkisebb elfogadott lefedésének mértéke. Létre kell hoznunk egy új gráfot is, ahol az eredeti gráf egy élét a következő gráf reprezentál:



A kapcsolatteremtés ezzel a gráffal az u_1-u_6 és v_1-v_6 csúcsokon keresztül történik. Ez a megjelenítés azért jó, mert így csak három út létezik a csúcsok bejárására:



Vegyük észre, hogy mindegyik esetben ugyanazon az oldalon lépünk ki, ahol beléptünk. Ennek a gráfnak az oldalait rendeljük hozzá az él végpontjaihoz. Néhány tetszőleges rendezést végrehajtva a csúcshoz kapcsolódó éleken megfelelően össze tudjuk kötni a fenti gráfokat. Az eredeti gráf minden csúcsára végrehajtva n db. összefont láncot kapunk, ami n belépési és n kilépési ponttal rendelkezik.



Így "lekódoltuk" a kiindulási gráfunkat. Ezután felveszünk k darab csúcsot és mindegyiket összekötjük az n db. kiindulási és végponttal. Az új gráf mérete:
GE+K csúcs és 12E+2kN+2E él, így a generálása polinomiális időben és méretben.

„kiválasztó élek"=k-beli élek

Ebben a gráfban létezik Hamilton kör akkor és csak akkor, ha létezik k méretű lefedése.

1. Tfh. $\{v_1,\ldots v_n\}$ egy Hamilton kör, ami a k csúcs egyikéből indul. Ennek keresztül kell mennie a kis gráfok egy láncolatán úgy, hogy egy másik k-beli csúcsba érkezik. Mivel ez egy Hamilton kör, ezért minden kis gráfon átmentünk, ilyenkor a „k"-k láncolata megfelel a lefedés beli csúcsoknak. Vegyük észre, hogy ha minden éllel összekapcsolt csúcs lefedésre kerül, akkor a kis gráfok két részre választhatók, különben egy lánc is elég.
   Ahhoz hogy elkerüljük egy csúcs többszöri érintését, mindegy láncszem egy k-beli csúcshoz kötődik.
2. Tfh. a lefedés mérete <=k.
   Ilyenkor mindig tudunk csúcsokat adni a lefedéshez úgy, hogy mérete k legyen. Minden lefedés beli csúcshoz kezdjük el a lánc bejárását. Minden olyan ponton, ahol egy kis gráfba lépünk be ellenőrizzük, hogy a másik csúcsa benn van-e a lefedésben és eszerint haladjunk át rajta.
   Válassznuk egy k-beli élet a kör befejezéséhez.

->NP Teljes.

Hasonlóan a Hamilton út keresés is NP teljes, amit beláthatunk úgy, hogy egy kezdő és vég csúcsot helyezünk az út kezdő és végpontjához, ezek lesznek az első és utolsó „kiválasztó” élek.

Az NP-teljesség bizonyításának művészete

Az NP-teljesség bizonyítása begyakorolható. Ha egyszer valaki rájön a titkára, meglepően könnyű és kellemes elvégezni. Tulajdonképpen az NP-teljesség bizonyításának titka az; hogy általában könnyebb újraalkotni, mint elmagyarázni, és könnyebb a régi bizonyítást átdolgozni, mint megérteni.

Egy megadott probléma NP-teljességének bizonyításához a következő tanácsok adhatók:

1) *Amennyire csak lehet, egyszerűsítsük le a forrásproblémát.*

Soha ne használjuk például az általános utazó ügynök problémát bizonyítandó célként. Ehelyett például a következő egyszerűsítéseket tehetjük:

- Használjuk a probléma háromszög egyenlőtlenségekre korlátozott változatát.
- Alkalmazzunk Hamilton köröket, ahol az élek súlya 1 vagy ∞.
- A körök helyett használjunk Hamilton utakat.
- Használjunk Hamilton utakat irányított gráfokon, ahol minden csúcs fokszáma 3.

Az említett problémák egymással ekvivalensek, és minél jobban leegyszerűsítjük a problémát, annál könnyebb lesz bizonyítani.

2) *Amennyire csak lehet, tegyük nehézzé a célproblémát.*

Ne féljünk extra megkötésekkel vagy általánosításokkal bővíteni a problémánkat (legalábbis átmenetileg).

3) *A megfelelő forrásproblémát válasszuk ki.*

A megfelelő forrásprobléma nagyban befolyásolja a bizonyítás bonyolultságát. Ezen a helyen nagyon könnyű hibázni.

Válasszunk magunknak „kedvenc” forrásproblémákat, és ezeket tanulmányozzuk alaposan. Melyik változatuk NP-teljes, melyik nem. Néhány javasolt forrásprobléma:

- Egész számmal való osztás. Az egyetlen lehetőség akkor, ha a probléma nagy számok használatával kapcsolatos.
- Csúcspontok lefedettsége. Olyan gráfproblémák esetén, amelyek csúcsok vagy élek megfelelő kijelölésén alapulnak.
- Hamilton út. Olyan gráfproblémák esetén, amelyek rendezésen alapulnak. Útvonalválasztási problémáknál és ütemezésnél jól használható.

- 3SAT probléma. Ha az eddigiek egyike sem tűnik megfelelőnek, hasznos lehet visszatérni a gyökerekhez.

4) *Sarkítsuk a problémát.*

A probléma visszavezetésénél a cél, hogy az eredeti probléma a lehető legkevésbé változzon meg. A legjobb út erre, hogy világosan jelöljük a nemkívánatos lehetőségeket. Minél jobban láthatók a nem kívánt következmények, annál könnyebb az „akkor és csak akkor” feltételt bizonyítani.

5) *Gondolkodjunk stratégiailag általános szinteken, aztán ügyeskedésekkel támasszuk alá a taktikánkat.*

Tegyük fel magunknak a következő kérdéseket:

- „Hogy tudom elérni, hogy A lehetőség és B lehetőség együtt ne legyen kijelölve?”
- „Hogy tudom elérni, hogy A lehetőség a B lehetőség előtt legyen kijelölve?”
- „Mit kezdjek azokkal a lehetőségekkel, amik nincsenek kijelölve?”

6) *Ha megakadunk, próbáljunk algoritmust keresni a problémára.*

Néha azért nem tudjuk bizonyítani az NP-teljességet, mert létezik hatékony algoritmus a problémánk megoldására. Ha végképp nem sikerül a bizonyítás, érdemes megvizsgálni ezt a lehetőséget is.