# Dynamic programming 

Peter Hajnal

Bolyai Institute, University of Szeged, Hungary
2023 Fall

## The basic idea

- We are given a problem $P$. We introduce a multitude of problems: $\mathcal{P}$.
- We assume that $P \in \mathcal{P}$. First, it seems that we are making our life harder. We introduces extra problems in addition to the initial one.
- We will have very easy problems in $\mathcal{P}$.
- We can order the elements of $\mathcal{P}$ a way, that solving the actual problem (following the order) will be always easy, based on the answers given so far.


## The mathematical content

- We need some intuition to choose a good $\mathcal{P}$. With a good choice of a suitable (often very natural) ordering of our problems.
- Solving problems of $\mathcal{P}$ is very similar then proving a multitude of claims by induction.


## Example: Fibonacci numbers

## Definition: Fibonacci numbers

$$
F_{1}=F_{2}=1,
$$

If $n>2$, then

$$
F_{n}=F_{n-1}+F_{n-2}
$$

## Example: the few first Fibonacci numbers

$1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584, \ldots$

## Fibonacci numbers and dynamic programming

- The following " program" computes $\mathrm{F}(18)$ :

```
var F(1..18): natural;
for i=1 to 18 do
    if n = 1 or 2 then F(i) = 1
    if i > 2 then F(i) = F(i-1) + F(i-2);
```

print F (18)

- It follows the logic of dynamic programming. It introduce an array of length 18 . At the end of the run the memory will contain 18 numbers, although only the last one, $F_{18}$ is important for us.
- The speed strongly depends on how many subproblems we have.
- The dynamical programming solution of a problem very often looks like just filling a sequence, or an array with numbers, and announcing the last number as the output.


## The Fibonacci numbers and recursion

- The following "program" computes the Fibonacci numbers too:

Fibonacci(n):
if $n>2$
return Fibonacci(n-1)+ Fibonacci(n-2)
else
return 1;
print Fibonacci(18).

- It is a procedure, that for large parameter refers to itself. What does a machine do, when running this code? That is a hard question.
- This program is much slower than the one, based on dynamic programming.

Break


## The basic problem

## Maximum independent sets in trees

Given a tree $T$. Find the size of its largest independent vertex set.

## Tree

A tree graph is a connected graph without cycle.

## Independent vertex set

$F \subset V(G)$ is an independent set iff there is no $u v$ edge, with
$u, v \in F$.

## Rooted trees

- $(T, r)$ is a rooted tree, it $T$ is a tree, an $r$ is special vertex, called root. This innocent notion enrich our language.
- $V$, the set of vertices can be classified into "generations" depending the distance from the root. Let $G_{i}$ be the set of vertices, of distance $i$ from $r . G_{0}=\{r\}, G_{1}$ is the vertex set containing exactly the neighbors of the root.

For each edge, $e$ there is an index/generation $i \in \mathbb{N}$, that $e$ connects $x \in G_{i}$ and $y \in G_{i+1}$. In this case we say that $x$ is the parent vertex, and $y$ is the child vertex (according to $e$ ).

## Rooted trees (cont'd)

- If $x$ is not the root then there is unique step towards the root. This step leads to the only parent of $x$ (often this parent is called father).
- The root is the only vertex without a father.
- A vertex of a tree is called leaf if it has no children.
- The ancestors of a vertex (other than the root) are the vertices in the path from the root to this vertex, excluding the vertex itself and including the root. Descendants The descendants of a vertex $v$ are those vertices that have $v$ as an ancestor.
- The root is an ancestor of any non-leaf vertex. Any non-root vertex is a descendant of the root. $\ell$ is leaf iff it has no descendant.


## Rooted trees (cont'd)

- Any vertex $x$ determines/generates a rooted subtree, $T_{x}$ : Its vertices are the vertex $x$ and its descendants, its root is $x$.
- A rooted tree has $|V|$ subtrees. $T_{r}$ is the original tree. $\ell$ is a leaf iff $T_{\ell}$ has only one vertex $(\ell)$.
- The depth of a rooted tree is the length of the longest root-leaf path. The depth is 0 if and only if the tree has only one vertex, that vertex is a root and a leaf at the same time.
- Many of the graph theoretical slang, introduced above, are originated in the language of family trees.


## Dynamics of the problem

- We are given a tree $T$. Take any vertex as a root. Note that distinguishing a vertex doesn't change the problem.


## The multitude of problems

$$
\left\{\mathcal{F}_{x}\right\}_{x \in V(T)},
$$

where $\mathcal{F}_{x}$ is the largest independent set problem for $\left(T_{x}, x\right)$.

- We have $|V(T)|$ problems in our collection of problems. One of them is the original problem (when $x$ is the original root, $r$ ).


## Roll up the problems

We order the problems $\mathcal{F}_{X}$ by the depth of $\left(T_{x}, x\right)$.

## The scheme

## Algorithm

(0) We start with the problems $\left(T_{x}, x\right)$, where the depth is 0 , i.e. $x$ is leaf.
// Then we have 1 vertex in the tree, and 1 is the size of the largest independent set.
(Ordering) We order our problems ( $T_{i}, r_{i}$ ). In our order depth $\left(T_{i}, r_{i}\right)$ is increasing.
(Loop) For $i=1,2, \ldots,|V|$ do the same. Assume that $\left(T_{i}, r_{i}\right)$ is the actual problem.
(Computation) We solve the the actual problem assuming that the previous problems are already solved.
(Output) Announce the answer for ( $T_{r}, r$ ) as the output.

## The idea for (Computation)

- We classify the independent sets of the actual $\left(T_{i}, r_{i}\right)$ :
(a) The independent sets not containing $r_{i}$, the root.
(b) The independent sets containing $r_{i}$, the root.

We determine the largest size among the two types.

## Independent sets of type (a)

- Let $s_{1}, \ldots, s_{d}$ be the children of $r_{i}$.


## Observation

To obtain an independent set of type (a) in $T_{r_{i}}$ take an arbitrary independent set for each $T_{s_{1}}, T_{s_{2}} \ldots, T_{s_{d}}$ and take their union.

- We can maximize the size of the independent set by taking a maximum size independent set from each of the $T_{s_{i}}$ 's. Let $M_{i}$ be the maximum size of independent sets in $T_{s_{i}}$. The value of $M_{i}$ is known when solving the actual problem.
- The answer for the actual problem is $\sum_{i} M_{i}$.


## Independent sets of type (b)

- Let $g_{1}, \ldots, g_{D}$ be the grandchildren of $r_{i}$, the root of the actual tree.


## Observation

To obtain an independent set of type (b) in $T_{r_{i}}$ take an arbitrary independent set for each $\left(T_{g_{i}}, g_{i}\right)$, take their union, and add to the union the vertex $r_{i}$.

- We can maximize the size of the independent set by taking a maximum size independent set from each of the $T_{g_{i}}$ 's. Let $\mu_{i}$ be the maximum size of independent sets in $T_{s_{i}}$. The value of $\mu_{i}$ is known when solving the actual problem.
- The answer for the actual problem is $1+\sum_{i} \mu_{i}$.


## The algorithm

## Algorithm for finding the maximal size of independent sets

(0) Fix a root $r$, introduce the rooted subtrees. We start with the problems $\left(T_{x}, x\right)$, where the depth is 0 , i.e. $x$ is leaf. Output 1.
(Ordering) We order our problems ( $T_{i}, r_{i}$ ). In our order depth $\left(T_{i}, r_{i}\right)$ is increasing.

Solve the problems following the order. Assume that $\left(T_{i}, r_{i}\right)$ is the actual "problem".
(Computation) From the solutions of the previous problems compute $\max \left\{\sum_{i} M_{i}, 1+\sum_{i} \mu_{i}\right\}$.
(Output) Print the solution for the rooted subtree ( $T_{r}, r$ ) ( $T_{r}=T$ ).

## Final remark

For the depth 0 initial case we know an optimal independent set too. Using the above ideas we can compute not only the maximal size, but one of the optimal independent set too.

## Thank you for your attention!

