Dynamic programming

Peter Hajnal

Bolyai Institute, University of Szeged, Hungary

2023 Fall

The basic idea

The basic idea

- We are given a problem P. We introduce a multitude of problems: $\mathcal{P}$.

The basic idea

- We are given a problem P. We introduce a multitude of problems: $\mathcal{P}$.
- We assume that $P \in \mathcal{P}$.

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.

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 $\mathcal{P}$.

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

The mathematical content

- We need some intuition to choose a good $\mathcal{P}$.

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.

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: 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,

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

$1, 1,$

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

$1, 1, 2,$

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

$1, 1, 2, 3,$

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

$1, 1, 2, 3, 5,$

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

$1, 1, 2, 3, 5, 8,$

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

$1, 1, 2, 3, 5, 8, 13,$

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

$1, 1, 2, 3, 5, 8, 13, 21,$

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

$1, 1, 2, 3, 5, 8, 13, 21, 34,$

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

$1, 1, 2, 3, 5, 8, 13, 21, 34, 55,$

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

$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89,$

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

$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,$

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

$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233,$

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

$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377,$

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

$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610,$

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

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

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

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

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

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

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

Fibonacci numbers and dynamic programming

- The following "program" computes F(18):

Fibonacci numbers and dynamic programming

- The following "program" computes 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)
```

Fibonacci numbers and dynamic programming

- The following "program" computes 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.

Fibonacci numbers and dynamic programming

- The following "program" computes 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.

Fibonacci numbers and dynamic programming

- The following "program" computes 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,

Fibonacci numbers and dynamic programming

- The following "program" computes 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.

Fibonacci numbers and dynamic programming

- The following "program" computes 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.

Fibonacci numbers and dynamic programming

• The following "program" computes 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 Fibonacci numbers and recursion

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

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).
```

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.

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?

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.

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

The basic problem

Maximum independent sets in trees

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

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.

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 uv edge, with $u, v \in F$.

Rooted trees

Rooted trees

- (T, r) is a rooted tree, it T is a tree, an r is special vertex, called root.

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.

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.

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.

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. $G_0 = \{r\}$,

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.

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 $y \in G_{i+1}$.

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)

Rooted trees (cont'd)

- If x is not the root then there is unique step towards the root.

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).

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.

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.

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.

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.

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.

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.

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. ℓ is leaf iff it has no descendant.

Rooted trees (cont'd)

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.

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.

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.

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. ℓ is a leaf iff T_ℓ has only one vertex (ℓ).

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. ℓ is a leaf iff T_ℓ has only one vertex (ℓ).

• The depth of a rooted tree is the length of the longest root-leaf path.

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. ℓ is a leaf iff T_ℓ has only one vertex (ℓ).

• 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.

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. ℓ is a leaf iff T_ℓ has only one vertex (ℓ).

• 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

Dynamics of the problem

- We are given a tree T.

Dynamics of the problem

- We are given a tree T. Take any vertex as a root.

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.

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

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

where $\mathcal{F}_x$ is the largest independent set problem for (T_x, x).

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

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

where $\mathcal{F}_x$ is the largest independent set problem for (T_x, x).

- We have $|V(T)|$ problems in our collection of problems.

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

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

where $\mathcal{F}_x$ is the largest independent set problem for (T_x, x).

• 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 (T_x, x).

The scheme

Algorithm

The scheme

Algorithm

(0) We start with the problems (T_x, x), where the depth is 0, i.e. x is leaf.

The scheme

Algorithm

(0) We start with the problems (T_x, x), 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.

The scheme

Algorithm

(0) We start with the problems (T_x, x), 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(T_i, r_i) is increasing.

The scheme

Algorithm

(0) We start with the problems (T_x, x), 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(T_i, r_i) is increasing.

(Loop) For $i = 1, 2, \dots, |V|$ do the same.

The scheme

Algorithm

(0) We start with the problems (T_x, x), 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 $\text{depth}(T_i, r_i)$ is increasing.

(Loop) For $i = 1, 2, \ldots, |V|$ do the same. Assume that (T_i, r_i) is the actual problem.

The scheme

Algorithm

(0) We start with the problems (T_x, x), 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(T_i, r_i) is increasing.

(Loop) For $i = 1, 2, \ldots, |V|$ do the same. Assume that (T_i, r_i) is the actual problem.

(Computation) We solve the the actual problem assuming that the previous problems are already solved.

The scheme

Algorithm

(0) We start with the problems (T_x, x), 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 $\text{depth}(T_i, r_i)$ is increasing.

(Loop) For $i = 1, 2, \ldots, |V|$ do the same. Assume that (T_i, r_i) 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)

The idea for (Computation)

- We classify the independent sets of the actual (T_i, r_i):

The idea for (Computation)

- We classify the independent sets of the actual (T_i, r_i):

(a) The independent sets not containing r_i, the root.

(b) The independent sets containing r_i, the root.

The idea for (Computation)

- We classify the independent sets of the actual (T_i, r_i):

(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)

Independent sets of type (a)

- Let $s_1, \ldots, s_d$ be the children of r_i.

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} ..., T_{s_d} and take their union.

Independent sets of type (a)

- Let $s_1, \dots, s_d$ 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} ..., 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.

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} ..., 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.

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)

Independent sets of type (b)

- Let $g_1, \ldots, g_D$ be the grandchildren of r_i, the root of the actual tree.

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 (T_{g_i}, g_i), take their union, and add to the union the vertex r_i.

Independent sets of type (b)

• Let $g_1, \dots, g_D$ 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 (T_{g_i}, g_i), 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.

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 (T_{g_i}, g_i), 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 μ_i be the maximum size of independent sets in T_{s_i}. The value of μ_i is known when solving the actual problem.

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 (T_{g_i}, g_i), 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 μ_i be the maximum size of independent sets in T_{s_i}. The value of μ_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

The algorithm

Algorithm for finding the maximal size of independent sets

(0) Fix a root r, introduce the rooted subtrees.

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 (T_x, x), where the depth is 0, i.e. x is leaf.

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 (T_x, x), where the depth is 0, i.e. x is leaf. Output 1.

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 (T_x, x), 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(T_i, r_i) is increasing.

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 (T_x, x), 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(T_i, r_i) is increasing.

Solve the problems following the order. Assume that (T_i, r_i) is the actual "problem".

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 (T_x, x), 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(T_i, r_i) is increasing.

Solve the problems following the order. Assume that (T_i, r_i) is the actual "problem".

(Computation) From the solutions of the previous problems compute $\max\{\sum_i M_i, 1 + \sum_i \mu_i\}$.

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 (T_x, x), 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(T_i, r_i) is increasing.

Solve the problems following the order. Assume that (T_i, r_i) is the actual "problem".

(Computation) From the solutions of the previous problems compute $\max\{\sum_i M_i, 1 + \sum_i \mu_i\}$.

(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.

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!