Dynamic programming

Peter Hajnal

Bolyai Institute, University of Szeged, Hungary

Fall Semester, 2026

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 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 uv 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. ℓ 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. 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

• 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

(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.

(Output) Announce the answer for (T_r, r) as the output.

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)

- 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 (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

(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. Using the above ideas we can compute not only the maximal size, but one of the optimal independent set too.

Break

The Basic Question

Finding the Largest Subset in Convex Position of a Finite Planar Point Set.

Given n points in the plane: $\mathcal{P} = \{P_i(x_i, y_i)\}_{i=1}^{n}$.

// We assume that the x-coordinates of the points are distinct.

Determine the largest subset $\mathcal{R}$ which is in convex position.

Definition: Convex Position

A finite set of points $\mathcal{R} \subset \mathbb{R}^2$ is in convex position if it forms the vertex set of a convex polygon with $|\mathcal{R}|$ vertices.

Example

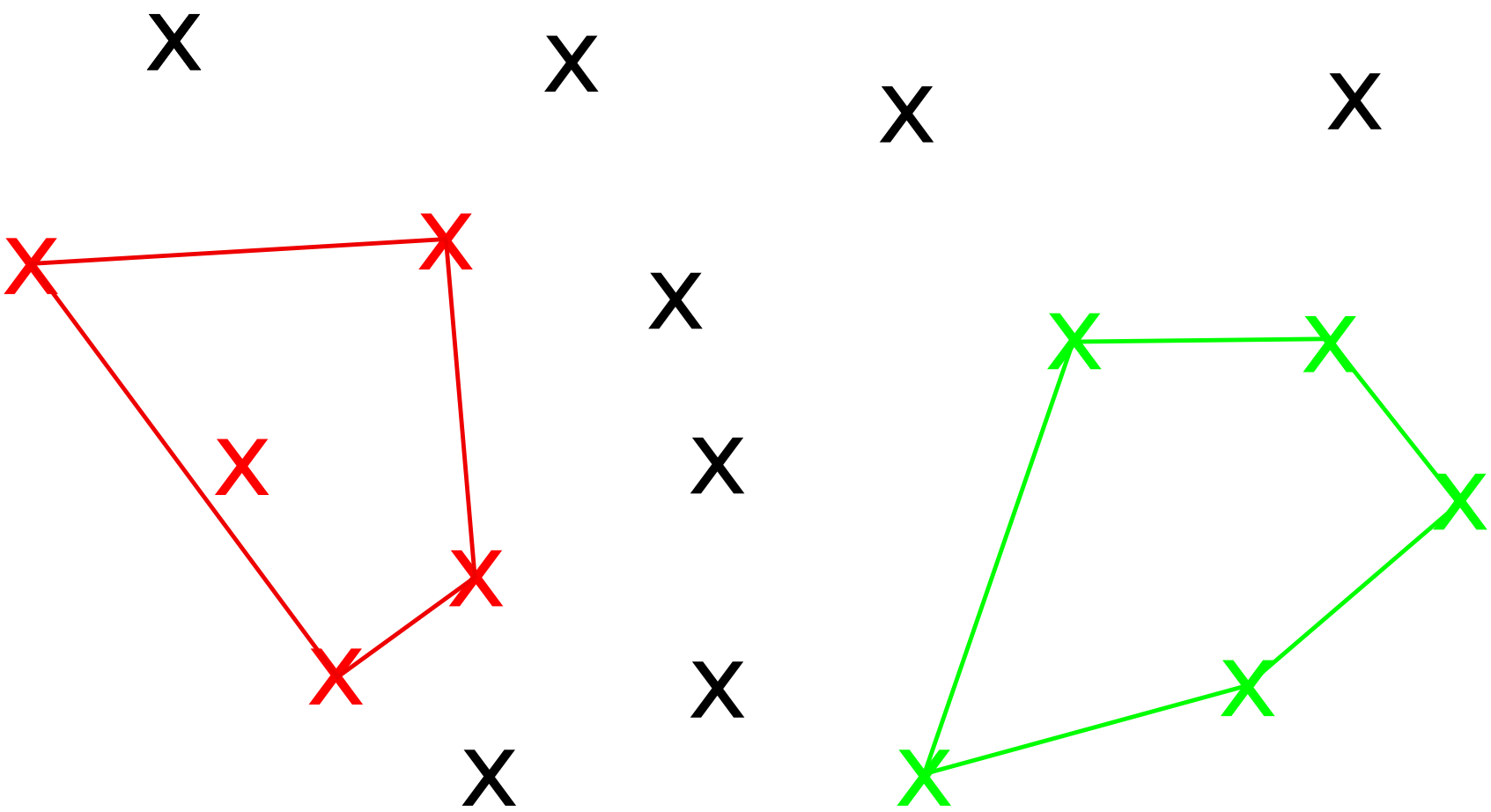

Figure: A finite set of points and a subset of five points (in red) which is not in convex position, along with another subset of five points (in green) which is in convex position.

Initial Steps: Sweeping

- First, we sweep through the point set from left to right. More precisely, we sort the points by their x-coordinates.

- // For simplicity, we assume that for $i < j$, $x_i < x_j$.

- This part of the algorithm can be solved with $\mathcal{O}(n \log n)$ comparisons.

Initial Steps: Decomposition into Cap/Cup Problem

• Second, the convex hull of a convex set of k points is a convex polygon with k vertices.

• We highlight two points on the perimeter: the vertices with the minimal and maximal x-coordinates (m and M).

• This perimeter, viewed as a circularly ordered set of vertices, is split into two *arcs* (both arcs include the two highlighted vertices): an upper arc and a lower arc.

• The vertices on the upper arc are also in convex position, meaning their convex hull includes all points as vertices. Moreover, mM will be one side of the convex hull, and the entire convex hull lies above this side. We refer to the upper arc as a *cap*.

• The lower arc can be treated similarly. We refer to it as a *cup*.

Example

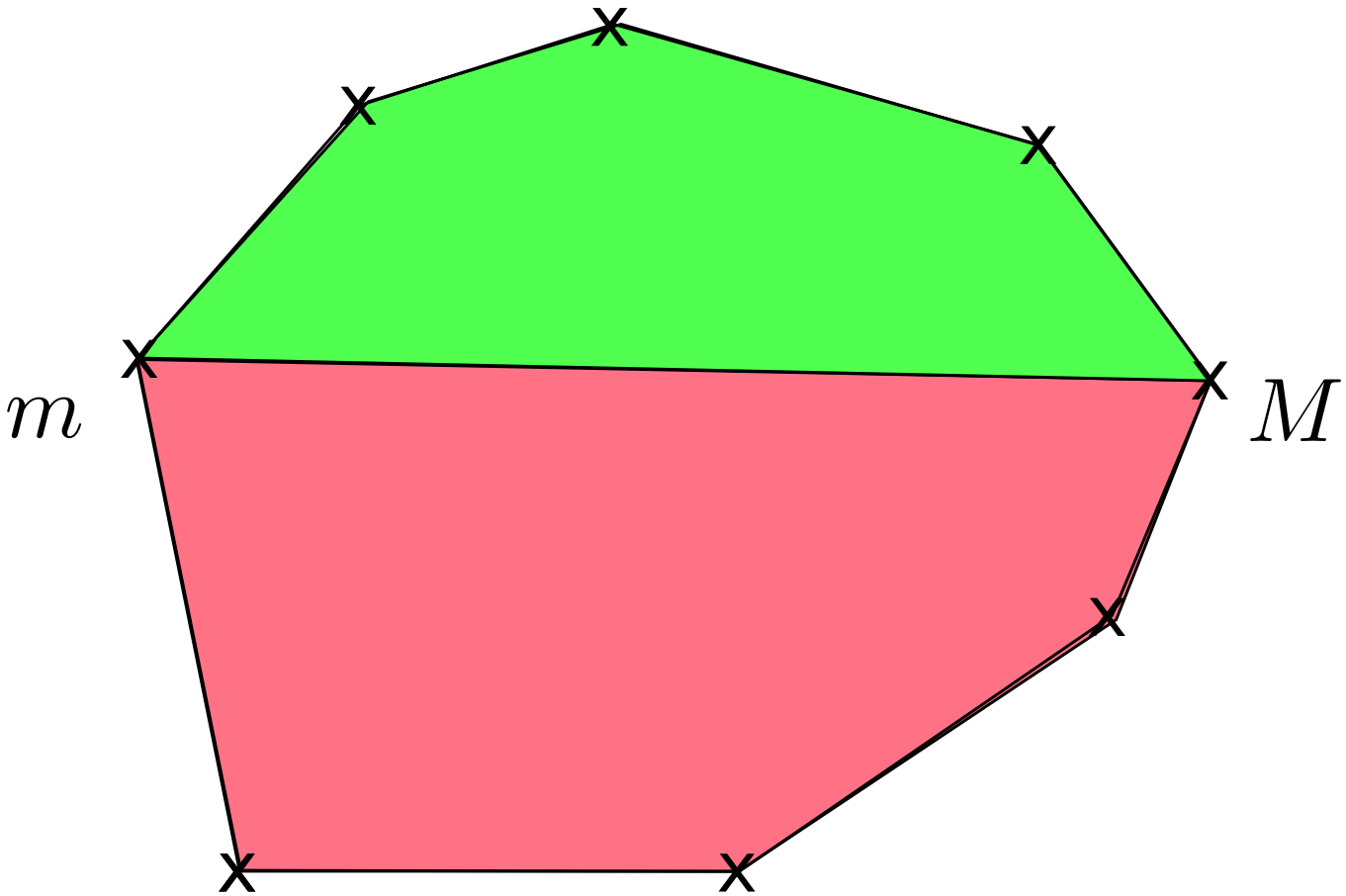

Figure: A point set in convex position with two highlighted vertices. The green part is the cap component of the the convex hull, and the red part is the cup component of the convex hull.

Initial Steps: It Suffices to Consider Caps

• If we know the size of the longest cap and cup starting at P_i and ending at P_j for every $i < j$, we can solve the basic question.

• This idea already represents a *dynamic programming* approach.

• We use this approach in the first step of the design, where we *split* the original problem, and focus only on determining the largest cap size between points P_i and P_j.

• Naturally, the following ideas can also be applied to determine the largest cup sizes.

• Thus, the algorithm solving the original problem can be derived from the following steps.

The Problem of Finding the Longest Cap from P_i to P_j Using Dynamic Programming

- For simplicity of notation, we assume that $i = 1$ and $j = n$.

The Task Set

Let $\mathcal{F}_{k,\ell,s}$ $(k \leq \ell < s)$ be the task that determines the longest cap starting at P_k, ending at P_s, and directly passing through P_ℓ.

- Thus, we define $\mathcal{O}(n^3)$ subproblems.

The Roll-up Order

We process the subproblems in increasing order of $\ell - k$.

- The simplest/base subproblems are the $\mathcal{F}_{k,k,s}$ problems. In this case, the two points P_k and P_s form a cap, which is the optimal cap.

The Roll-up Step

• Suppose $k < \ell$, and we are dealing with the $\mathcal{F}_{k,\ell,s}$ problem, but we already know the solution to all $\mathcal{F}_{k_0,\ell_0,s_0}$ problems where $\ell_0 - k_0 < k - \ell$.

• Only those subproblems are of interest where the triplet P_k, P_ℓ, P_s forms a cap (this was not an issue for the case $K = \ell < s$). We will assume this from now on.

• Let's guess the vertex that could be the third-to-last point of the cap (i.e., the point just before P_ℓ).

• This vertex is geometrically constrained within a triangle: It must lie to the left of the vertical line through P_k, above the line P_kP_ℓ, and below the line $P_\ell P_s$. The vertex cannot lie on the boundary of the triangle, except for P_k, which may appear as the point directly before P_ℓ.

The Roll-up Step is continued

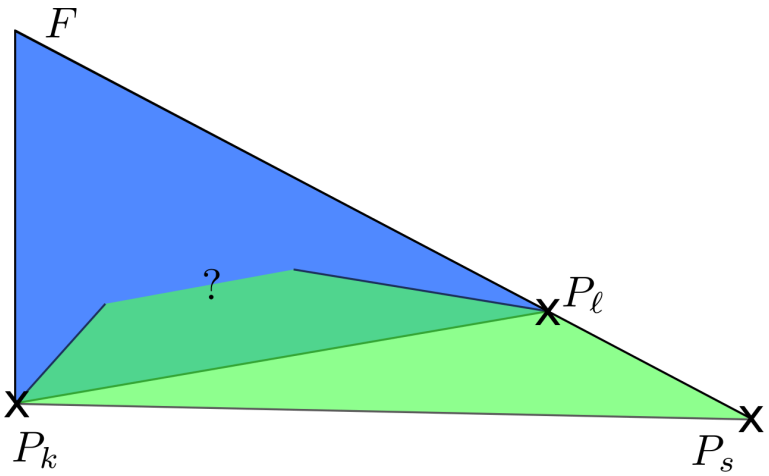

Figure: F is the intersection point of the vertical line passing through P_i and the line $P_\ell P_s$. The optimal cap must be found within the blue triangle $P_i P_\ell F$.

We take the points that lie within the triangle $P_k P_\ell F$ (these include P_k and potentially other interior points of the triangle). For each point P_{ℓ^-}, we retrieve the solution to $\mathcal{F}_{k,\ell^-,\ell}$ from our previous work. By taking the maximum of these values and adding 1, we obtain the solution to $\mathcal{F}_{k,\ell,s}$.

Final Remarks

• It is crucial that we store the solutions to the already solved $\mathcal{O}(n^3)$ subproblems and retrieve the necessary information from them as needed.

• It is easy to modify the algorithm so that it not only provides the maximum size, but also outputs the largest set of points in convex position.

This is the end!

Thank you for your attention!