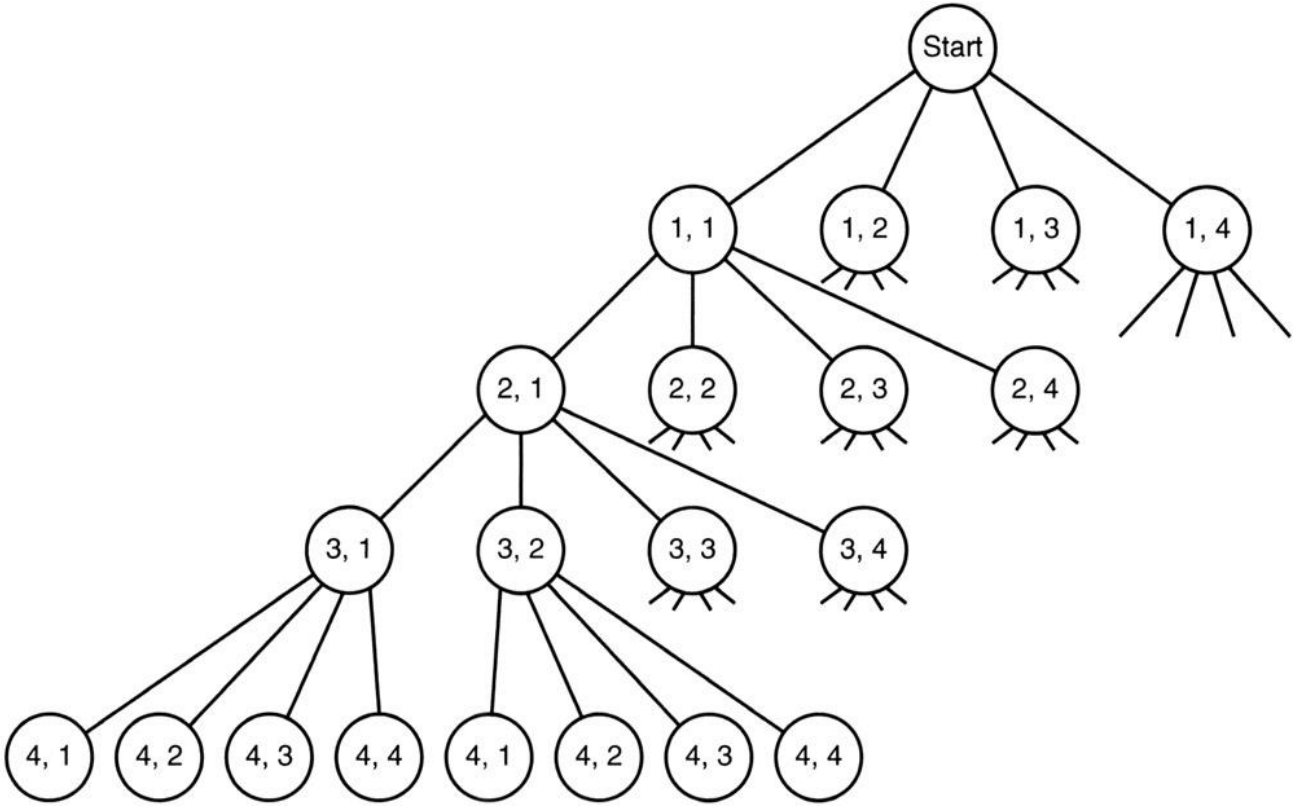
Randomized algorithms

Hamed Vahdat-Nejad

N-Queens

- Problem: position n queens on an $n \times n$ chessboard so that no two queens threaten each other.
- Let n=4
- There are $4 \times 4 \times 4 \times 4 = 256$ candidate solutions.
- We can create the candidate solutions by constructing a tree in which the column choices for the first queen (the queen in row 1) are stored in level-1 nodes in the tree, the column choices for the second queen (the queen in row 2) are stored in level-2 nodes, and so on.

- The tree is called a ***state space tree*.**
- The entire tree has 256 leaves.
- A path from the root to a leaf is a candidate solution .

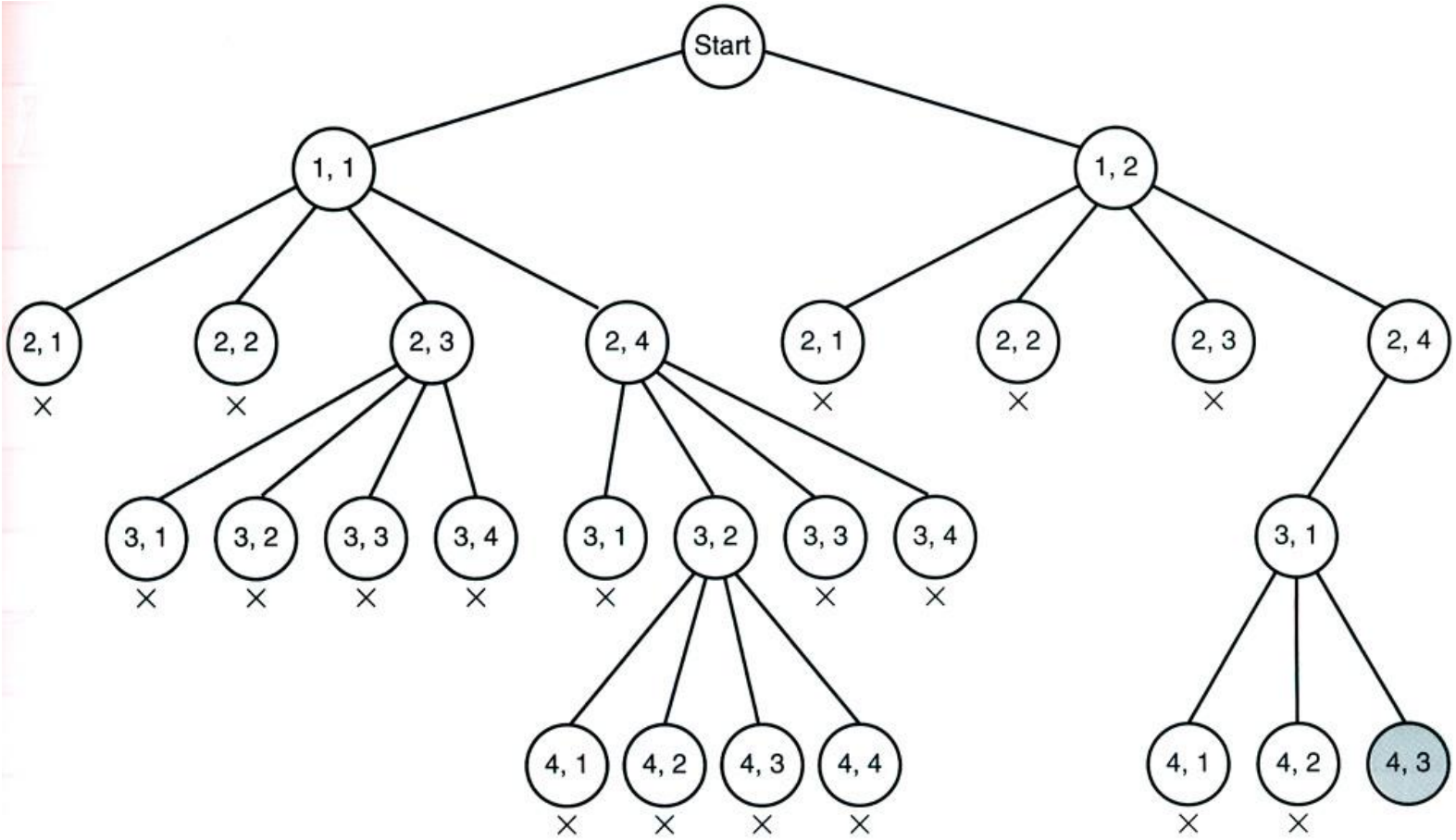

Start
1, 1
1, 2
1, 3
1, 4
2, 1
2, 2
2, 3
2, 4
3, 1
3, 2
3, 3
3, 4
4, 1
4, 2
4, 3
4, 4
4, 1
4, 2
4, 3
4, 4

BACKTRACKING

- ***Backtracking*** is the procedure whereby, after determining that a node can lead to nothing but dead ends, we go back ("backtrack") to the node's parent and proceed with the search on the next child.
- We call a node ***nonpromising*** if when visiting the node we determine that it cannot possibly lead to a solution.
- Otherwise, we call it ***promising***.
- This is called ***pruning*** the state space tree

The general algorithm of backtracking

```
void checknode (node v)
{
   node u;
   if (promising(v))
         if (there is a solution at v) write the solution;
   else
         for (each child u of v)
               checknode(u);
}
```


Start
1, 1
1, 2
2, 1
2, 2
2, 3
2, 4
2, 1
2, 2
2, 3
2, 4
3, 1
3, 2
3, 3
3, 4
3, 1
3, 2
3, 3
3, 4
3, 1
4, 1
4, 2
4, 3
4, 4
4, 1
4, 2
4, 3

- The backtracking algorithm checks 27 nodes before finding a solution.

- A depth-first search of the state space tree checks 155 nodes before finding that same solution.

ANALYSIS

- Given two instances with the same value of n, one algorithm may require that very few nodes be checked, whereas the other algorithm requires that the entire state space tree be checked.
- If we had an estimate of how efficiently a given backtracking algorithm would process a particular instance, we could decide whether using the algorithm on that instance was reasonable.
- We can obtain such an estimate using a Monte Carlo algorithm.

- Monte Carlo algorithms are probabilistic algorithms.
- By a ***probabilistic algorithm***, we mean that the next instruction executed is sometimes determined at random according to some probability distribution.
- Unless otherwise stated, we assume that probability distribution is the uniform distribution.
- By a ***deterministic algorithm***, we mean one in which this cannot happen.

- A ***Monte Carlo algorithm*** estimates the expected value of a random variable, defined on a sample space.
- There is no guarantee that the estimate is close to the true expected value.
- We can use a Monte Carlo algorithm to estimate the efficiency of a backtracking algorithm for a particular instance.
- We estimate the total number of nodes that would be checked to find all solutions.
- It is an estimate of the number of nodes in the pruned state space tree.

- The following conditions must be satisfied by the algorithm in order for the technique to apply:
  1. The same promising function must be used on all nodes at the same level in the state space tree.
  2. Nodes at the same level in the state space tree must have the same number of children.
- The Monte Carlo technique requires that we randomly generate a promising child of a node according to the uniform distribution.
- We mean that a random process is used to generate the promising child.

Monte Carlo technique

- Let m_0 be the number of promising children of the root.
- Randomly generate a promising node at level 1. Let m_1 be the number of promising children of this node.
- Randomly generate a promising child of the node obtained in the previous step. Let m_2 be the number of promising children of this node.
- . . .
- Randomly generate a promising child of the node obtained in the previous step. Let m_i be the number of promising children of this node.
- . . .
- This process continues until no promising children are found.

- m_i is an estimate of the average number of promising children of nodes at level i.
- t_i = total number of children of a node at level i.
- an estimate of the total number of nodes checked by the backtracking algorithm to find all solutions is given by

$$1 + t_0 + m_0 t_1 + m_0 m_1 t_2 + \cdots + m_0 m_1 \cdots m_{i-1} t_i + \cdots .$$

- In the algorithm, a variable *mprod* is used to represent the product $m_0 m_1 \ldots m_{i-1}$ at each level.

Monte Carlo estimate

- Problem: Estimate the efficiency of a backtracking algorithm using a Monte Carlo algorithm.
- Inputs: an instance of the problem that the backtracking algorithm solves.
- Outputs: an estimate of the number of nodes in the pruned state space tree produced by the algorithm, which is the number of the nodes the algorithm will check to find all solutions to the instance.

THE ALGORITHM

```
int estimate ()
{   node v;
    int m, mprod, t, numnodes;
    v = root of state space tree;
    numnodes = 1; m = 1; mprod = 1;
    while (m != 0)
    {       t = number of children of v;
            mprod = mprod * m;
            numnodes = numnodes + mprod * t;
            m = number of promising children of v;
            if (m != 0)
                    v = randomly selected promising child of v;
    }
     return numnodes;
}
```

The Monte Carlo for n-queen problem

```
int estimate_n_queens (int n)
{ index i, j, col [1 . . n]; int m, mprod, numnodes;
set_of_index  prom_children;
i = 0; numnodes = 1; m = 1; mprod = 1;
while (m != 0 && i != n)
{     mprod = mprod * m;
      numnodes = numnodes + mprod * n;
      i++;
      m = 0;
      prom_children = ∅ ;  // Initialize set of promising children
      for (j = 1; j <= n; j++)
      {           col[i] = j;
                  if (promising (i))
                  {                 m++;
                                    prom_children = prom_children ∪ {j};
                  }
      }
      if (m != 0)
      {           j = random selection from prom_children;
                  col [i] = j;
      }
}return numnodes;
}
```

- When a Monte Carlo algorithm is used, the estimate should be run more than once.
- The average of the results should be used as the actual estimate.
- Around 20 trials are ordinarily sufficient.
- Although the probability of obtaining a good estimate is high when the Monte Carlo algorithm is run many times, there is never a guarantee that it is a good estimate.
- The estimate produced by any one application of the Monte Carlo technique is for one particular instance.
- Given two instances with the same value of n, one may require that very few nodes be checked whereas the other requires that the entire state space tree be checked.

KNAPSACK PROBLEM

- We have a set of items, each of which has a weight and a profit.
- A thief plans to carry off stolen items in a knapsack, and the knapsack will break if the total weight of the items placed in it exceeds some positive integer *W*.
- The thief's objective is to determine a set of items that maximizes the total profit.

- We can solve this problem using a state space tree.
- We go to the left from the root to include the first item, and we go to the right to exclude it. Similarly, we go to the left from a node at level 1 to include the second item, and we go to the right to exclude it, and so on.
- Each path from the root to a leaf is a candidate solution.

- if ***weight*** is the sum of the weights of the items that have been included up to some node, the node is nonpromising if

$$weight \geq W.$$

- A non obvious non-promising measure
  - We first order the items in non-increasing order according to the values of p_i/w_i, where w_i and p_i are the weight and profit, respectively, of the ith item.
  - Let ***profit*** be the sum of the profits of the items included up to the node.
  - *Weight* is the sum of the weights of those items.
  - We initialize variables ***bound*** and ***totweight*** to *profit* and *weight*, respectively.

- Next we greedily grab items, adding their profits to *bound* and their weights to *totweight*, until we get to an item that if grabbedwould bring *totweight* above *W*.
- We grab the fraction of that item allowed by the remaining weight, and we add the value of that fraction to *bound*.
- *bound* is still an upper bound on the profit we could achieve by expanding beyond the node.

- Suppose the node is at level i, and the node at level k is the one that would bring the sum of the weights above W.

$$totweight = weight + \sum_{j=i+1}^{k-1} w_j, \quad \text{and}$$

$$bound = \underbrace{\left(profit + \sum_{j=i+1}^{k-1} p_j \right)}_{\text{Profit from first } k-1 \text{ items taken}} + \underbrace{(W - totweight)}_{\text{Capacity available for } k\text{th item}} \times \underbrace{\frac{p_k}{w_k}}_{\text{Profit per unit weight for } k\text{th item}}.$$

- If *maxprofit* is the value of the profit in the best solution found so far, then a node at level *i* is nonpromising if

$$bound \leq maxprofit.$$

```
void knapsack (index i, int profit, int weight)
{   if (weight <= W&& profit > maxprofit)
    {       maxprofit = profit;
            numbest = i;
            bestset = include;
    }
    if (promising(i))
    {
            include [i + 1] = "yes";
            knapsack(i + 1, profit + p[i + 1], weight + w[i + 1]);
            include [i + 1] = "no"; // Do not include
            knapsack (i + 1, profit, weight);
    }
}
bool promising (index i)
{   index j, k;          int totweight;          float bound;
    if (weight >= W)     return false;
    else
    {       j = i + 1;
            bound = profit;
            totweight = weight;
            while (j <= n && totweight + w[j] < = W)
            {           totweight = totweight + w[j];
                        bound = bound + p[j];
                        j++;
            } k = j;
            if (k <=n)  bound = bound + (W - totweight) * p[k]/w[k];
            return bound > maxprofit;
    }
}
```

- Monte Carlo technique applies in this problem, it can be used to estimate the efficiency of the algorithm for a particular instance.
- Worst-case number of entries that is computed by the dynamic programming algorithm for the Knapsack problem is in $O\ (minimum\ (2^n, nW))$.
- In the worst case, the backtracking algorithm checks $\Theta\ (2^n)$ nodes.
- It may appear that the dynamic programming algorithm is superior.

- It is difficult to analyze theoretically the relative efficiencies of the two algorithms.
- The algorithms can be compared by running them on many sample instances and seeing which algorithm usually performs better.
- Horowitz and Sahni (1978) did this and found that the backtracking algorithm is usually more efficient than the dynamic programming algorithm.

The Probabilistic Method

- The main proponent of the probabilistic method, was Paul Erd˝os.
- The basic technique is based on two basic observations:
  1. If $E[X] = \mu$, then there exists a value x of X, such that $x \geq E[X]$.
  2. If the probability of event E is larger than zero, then E exists and it is not empty.
- The surprising thing is that despite the elementary nature of those two observations, they lead to a powerful technique that leads to numerous nice and strong results.

Example

- Theorem: For any undirected graph G(V,E) with n vertices and m edges, there is a partition of the vertex set V into two sets A and B such that

$$\left|\left\{uv \in E \;\middle|\; u \in A \text{ and } v \in B\right\}\right| \geq \frac{m}{2}.$$

- Proof: Consider the following experiment: randomly assign each vertex to A or B, independently and equal probability.

- For an edge e = uv, the probability that one endpoint is in A, and the other in B is ½.
- let X_e be the indicator variable with value 1 if this happens.
- Clearly,

$$\mathbf{E}\left[\left|\left\{uv \in E \;\middle|\; u \in A \text{ and } v \in B\right\}\right|\right] = \sum_{e \in E(G)} \mathbf{E}[X_e] = \sum_{e \in E(G)} \frac{1}{2} = \frac{m}{2}$$

- If E[X] = μ, then there exists a value x of X, such that x≥E[X].
- Thus, there must be a partition of V that satisfies the theorem.

Example

- Definition: For a vector v = (v1, . . . , vn) $\in \mathbb{R}^n$,

$$\|v\|_\infty = \max_i |v_i|$$

- Theorem: Let A be an n × n binary matrix (i.e., each entry is either 0 or 1), then there always exists a vector b $\in \{-1, +1\}^n$ such that

$$\|Ab\|_\infty \leq 4\sqrt{n \log n}$$

Proof

- Let v = (v1, . . . , vn) be a row of A.
- Choose a random b = (b1, . . . , bn) $\in \{-1, +1\}^n$.
- Let $i_1, \ldots, i_m$ be the indices such that $v_{ij} = 1$.
- Clearly,

$$\mathbf{E}[v \cdot b] = \sum_i \mathbf{E}[v_i b_i] = \sum_j v_i \, \mathbf{E}[b_{i_j}] = 0.$$

- Let $X_j = 1$ if $b_{ij} = +1$, for j = 1, . . . ,m.
- We have

$$\mathbf{E}\left[\sum_j X_j\right] = n/2$$

- Then

$$\mathbf{Pr}\left[|v\cdot b|\geq 4\sqrt{n\ln n}\right]=2\,\mathbf{Pr}\left[v\cdot b\leq -4\sqrt{n\ln n}\right]=2\,\mathbf{Pr}\left[\sum_j X_j-\frac{n}{2}\leq -2\sqrt{n\ln n}\right]$$

$$\begin{aligned}
&= 2\,\mathbf{Pr}\left[\sum_j X_j < \left(1-4\sqrt{\frac{\ln n}{n}}\frac{n}{m}\right)\frac{m}{2}\right]\\
&\leq 2\exp\left(-\frac{m}{2}\left(4\sqrt{\frac{\ln n}{n}}\frac{n}{m}\right)^2\right)=2\exp\left(-\frac{m}{2}\left(16\frac{n\ln n}{m^2}\right)\right)\\
&= 2\exp\left(-\frac{8n\ln n}{m}\right)\\
&\leq 2\exp(-8\ln n)=\frac{2}{n^8}
\end{aligned}$$

Chernoff Inequality

- Let $X_1, \ldots, X_n$ be n independent random variables, such that

  $$\Pr[Xi = 1] = \Pr[Xi = -1] = 0.5$$

  , for i = 1, . . . , n.

- Let $Y = \sum_{i=1}^{n} X_i$.

- Then, for any $\Delta > 0$, we have

$$\mathbf{Pr}[Y \geq \Delta] \leq e^{-\Delta^2/2n}.$$

- the probability that any entry in Ab exceeds $4\sqrt{n \ln n}$ is smaller than $2/n^7$.
- Thus, with probability at least $1 - 2/n^7$, all the entries of Ab have value smaller than $4\sqrt{n \ln n}$

In particular, there exists a vector $b \in \{-1, +1\}^n$ such that $\|Ab\|_\infty \leq 4\sqrt{n \ln n}$.

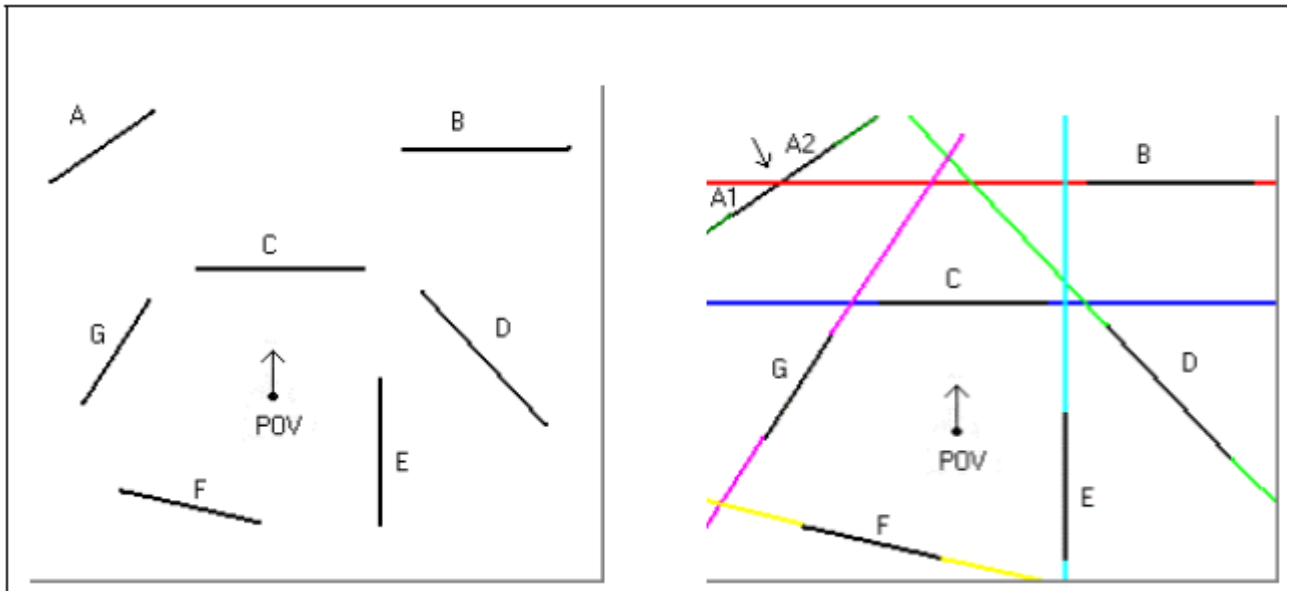
Binary Planar Partitions

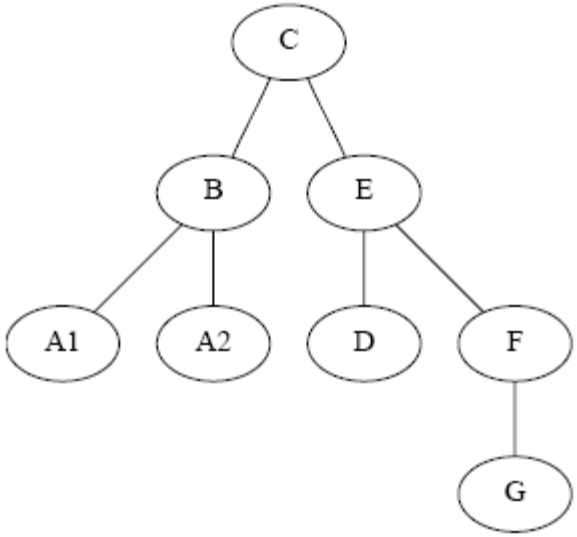
Problem

- Input: A set $S = \{s_1, s_2, \ldots, s_n\}$ of non-intersecting line segments in the plane.

- Output: A binary planar partition such that every region in the partition contains at most one line segment (or a portion of one line segment).

- A binary planar partition consists of a binary tree. Every internal node of the tree has two children.
- Associated with each node v of the tree is a region r(v) of the plane.
- Associated with each internal node v of the tree is a line l(v) that intersects r(v).
- The region corresponding to the root is the entire plane.
- The region r(v) is partitioned by l(v) into two regions $r_1(v)$ and $r_2(v)$, which are the regions associated with the two children of v.


A
B
C
G
D
POV
E
F
A2
A1
B
C
G
D
POV
E
F


C
B
E
A1
A2
D
F
G

- Any region r of the partition is bounded by the partition lines on the path from the root to the node corresponding to r in the tree.
- The storage requirement of a particular binary planar partition is the number of nodes in the associated tree.
- We want this tree to have as few nodes as possible.

- The best case is the one in which we are able to partition the n line segments without generating other new line segments (which are created when the original line segments are broken).
- In this case, we will need (n − 1) lines to partition the n line segments.
- Therefore, the total number of nodes in the binary planar partition tree (of n leaves and (n−1) interior nodes) is (2n−1).
- However, there is no guarantee that there is always a partition of size O(n).

- We will prove using a randomized algorithm that there always exists a binary planar partition of size O(n log n).
- For a line segment s, let l(s) denote the line obtained by extending s both sides to infinity.
- For the set S = $\{s_1, s_2, \ldots, s_n\}$ of line segments, a simple and natural class of partitions is the set of autopartitions, which are formed by only using lines from the set $\{l(s_1), l(s_2), \ldots, l(s_n)\}$ in constructing the partition.

The randomized algorithm

- Input: A set S = $\{s_1, s_2, \ldots, s_n\}$ of non-intersecting line segments.
- Output: A binary autopartition P of S.

1. Pick a permutation of of {1, 2, ..., n} uniformly at random from n! possible permutations.
2. While a region contains more than one segment, cut it with $l(s_i)$ where i is the first in the ordering such that s_i cuts that region.

- Theorem: The expected size of the autopartition produced by RandAuto is $O(n \log n)$.

Any question?