Exhaustive Search Techniques

There are many practical problems that no one has efficient (= poly. time) algorithms for.

e.g. 0-1 knapsack, TSP, ind. set in graph, shortest path in a graph with negative weights (no repeated nodes)

Options:

- heuristics – run quickly but no guarantee on quality of solution
- approximation algorithms – guarantee quality of solution e.g. length of TSP ≤ 2 · min length of TSP
- exact solutions – take exponential time

Note that to experiment with heuristics and approx. algs, we need exact algs.

Backtracking – a systematic way to try all possible solutions
- like searching in an <u>implicit graph</u> of partial solns

Used for <u>decision</u> problems (we'll deal with optimization later.)

<u>Example:</u> <u>Subset Sum</u> (decision version of knapsack with value = weight)

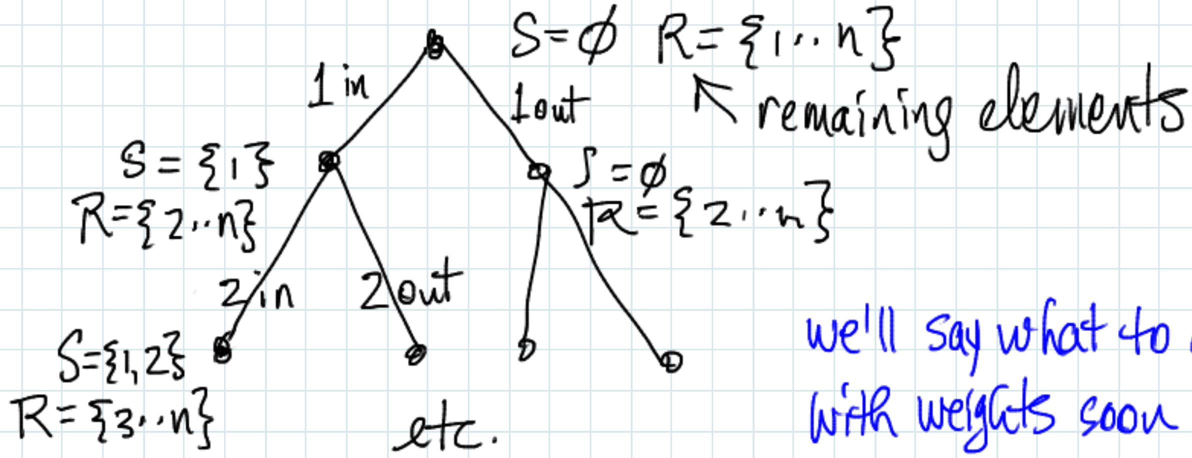
Given elements $1 \cdots n$ with weights $w_1 \cdots w_n$ and target weight W

Is there a subset $S \subseteq \{1 \cdots n\}$ s.t. $\sum_{i \in S} w_i = S$?

Fact: this problem is NP-complete (pf. later)

No one knows a poly. time alg.

Best we can do is explore all subsets



we'll say what to do with weights soon

General configuration $C = S, R$

$S \subseteq \{1 \ldots i-1\}$ $R = \{i \ldots n\}$

Two children — put i in or out

General Backtracking Algorithm

$\mathcal{A}$ — set of active configurations

initially — original config. (e.g. $S=\emptyset$ $R=\{1..n\}$)

while $\mathcal{A} \neq \emptyset$

$C \leftarrow$ remove config. from $\mathcal{A}$.

//explore C

if C solves problem — DONE SUCCESS

if C is a dead end — discard it

else expand C to $C_1 \ldots C_t$ by making additional choices and add each C_i to $\mathcal{A}$.

end

Store A as stack — DFS of config. space
size of A = height of tree

Store A as queue — BFS of config. space
size of A = width of tree

To reduce space, use DFS
e.g. for Subset Sum, width is 2^n, height is n.

Note: might also explore "most promising" config. first — use priority queue.

Back to Subset Sum

How to explore a config. S, R ?

Keep $w = \sum_{i \in S} w_i$
$r = \sum_{i \in R} w_i$
just update these as we go.

Then

If $w = W$ — SUCCESS (solved problem)

if $w > W$ — dead end (don't expand this config.)

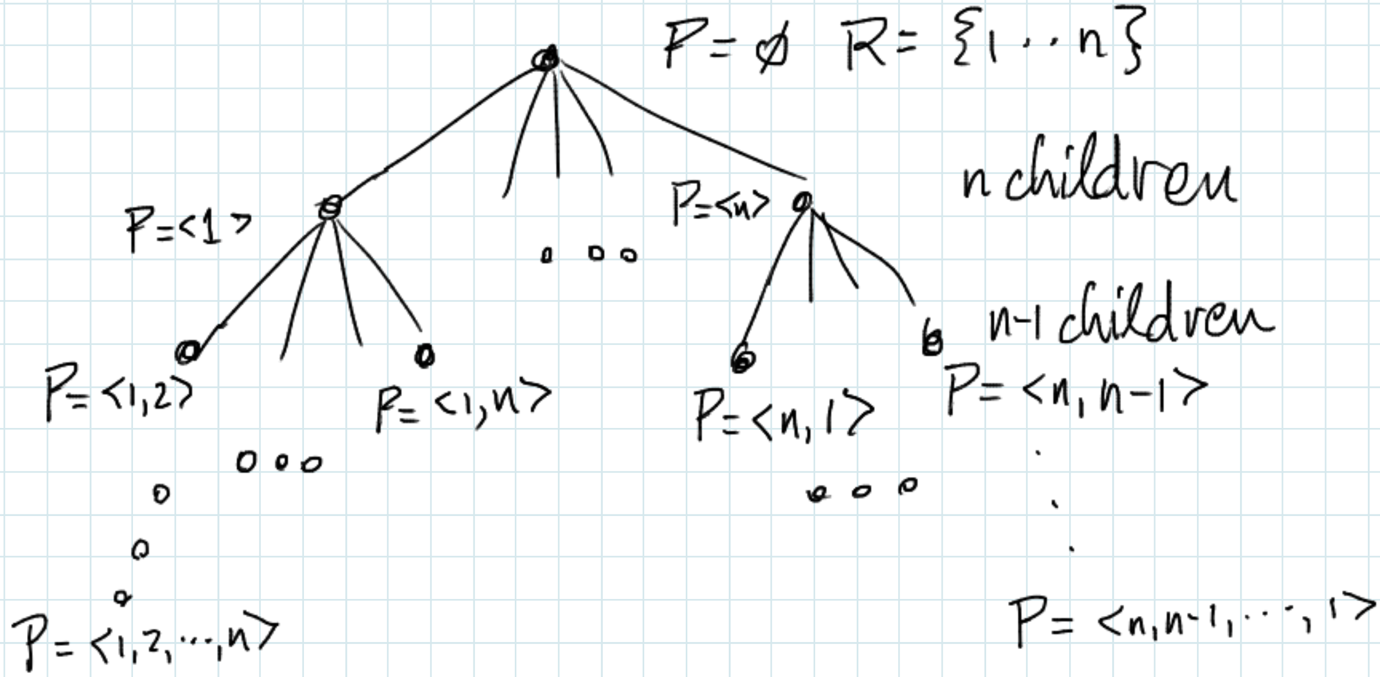
if $r + w < W$ — dead end "

Running time $O(2^n)$

like for knapsack

This is also a dynamic prog alg. with run time $O(n \cdot W)$.
Which is better? Depends! If W has n bits then backtracking is better.

Above we used backtracking to explore all subsets.
Can also explore all permutations of $1 \cdots n$



There are $n!$ leaves

Config. $C = \begin{cases} P \text{ – permutation of length } i \\ R \text{ – remaining elements.} \end{cases}$

Branch and Bound

- for optimization problems (say <u>minimize</u>)
  (backtracking was for decision problems)
- not DFS, but explore most promising config. first
- keep min so far
- "branch" – generate children
- "bound" – compute lower bound l_C on obj. fn and <u>prune</u> a config. C if l_C > current min.

General Branch and Bound Algorithm

$\mathcal{A}$ – set of active configurations
  initially the original config.
best-soln, best-cost – best so far.
while $\mathcal{A} \neq \emptyset$
  $C \leftarrow$ remove "most promising" config. from $\mathcal{A}$
  expand C to $C_1 \cdots C_t$ by making additional choices   BRANCH
  for $i = 1 \cdots t$
    if C_i solves whole problem
      then if cost(C_i) < best-cost then
        update best
    else if C_i is dead end then discard it
    else if lower bound(C_i) < best-cost   BOUND
      then add C_i to $\mathcal{A}$
end

Note that we test C_i when it is generated (rather than when it is removed from $\mathcal{A}$)
could have done this for backtracking too)

Example: <u>Travelling Salesman Problem</u>.

Given graph $G = (V, E)$ (undirected)
and non-neg. weights $w: E \to \mathbb{R}^{\geq 0}$
Find a cycle C that goes through every vertex exactly once, and has min. weight $\sum_{e \in C} w(e)$
called a TSP tour

This is a famous NP-complete problem. There is a book about it. An expert is Prof. Cook, C&O. http://www.math.uwaterloo.ca/tsp/index.html

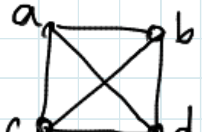
There are contests to solve big instances.

2004 24,978 cities in Sweden
2006 85,900 VLSI input

Branch and bound algorithm

- based on enumerating all subsets of edges
- Configuration C: $I \subseteq E$ – edges included in tour
  $X \subseteq E$ – edges excluded from tour
  with $I \cap X = \emptyset$

e.g.



if $X = \{(a,b)\}$ then only TSP tour is a c b d
if $X = \{(a,b)\}$ and $I = \{(c,d)\}$ there is no soln

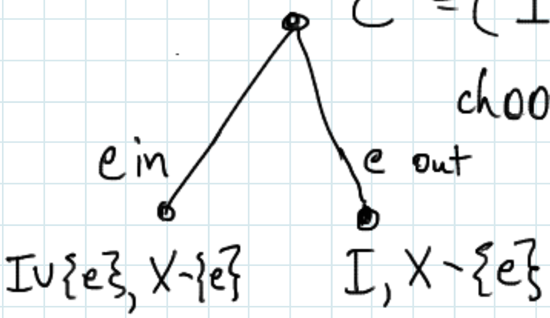
Necessary conditions – used to detect dead ends

- $E - X$ must be connected, actually biconnected
- I must have ≤ 2 edges incident to each vertex
- I contains no cycle (except on all vertices)

How to branch

$C = (I, X)$

choose $e \in E \setminus (I \cup X)$

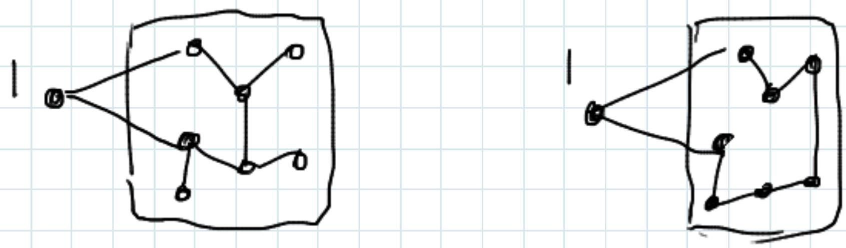


How to bound – want to compute lower bound efficiently

One idea based on MST:

A relaxed problem:

<u>1-tree</u> = spanning tree on vertices $2 \cdots n$ plus 2 edges from vertex 1



<u>Claim</u> Any TSP tour is a 1-tree

Thus $w(\text{min TSP}) \geq \underbrace{w(\text{min 1-tree})}$

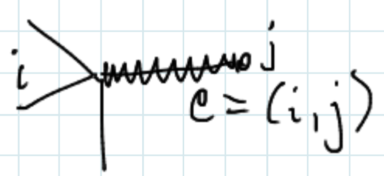
so this provides a lower bound

We can find a min 1-tree efficiently
even given X (excluded edges) — throw them out
I (included edges) — give them weight 0 (temporarily)
— just find MST on $\{2, \ldots, n\}$
and add 2 min weight edges incident to 1.

Can now use general branch and bound algorithm
lower bound for config. C = weight of min 1-tree for C

Enhancements

- choose "most promising" config. — the one with the min. weight 1-tree
- branch wisely
  e.g. find vertex i in min 1-tree of deg. ≥ 2
  and let e = max weight edge (i, j) in 1-tree but not in I ∪ X.



This plus further enhancements lead to competitive TSP algorithms.