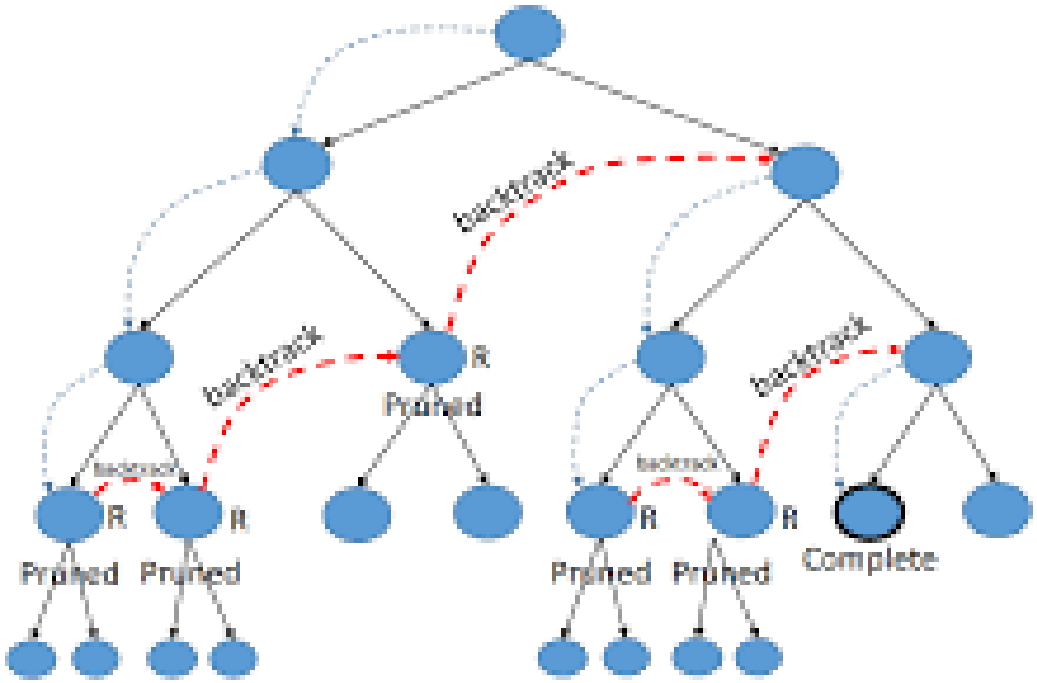
Backtracking algorithms


```
procedure EXPLORE(node n)
    if REJECT(n) then return
    if COMPLETE(n) then
        OUTPUT(n)
    for n_i : CHILDREN(n) do EXPLORE(n_i)
```



Backtracking

- Backtracking is a systematic way to go through all the possible configurations of a solutions space.

Backtracking

- Backtracking is a systematic way to go through all the possible configurations of a solutions space.
- Configurations include for example all possible arrangements of a set of objects (permutations) or all possible ways of building a collection of them (subsets).

Backtracking

- Backtracking is a systematic way to go through all the possible configurations of a solutions space.
- Configurations include for example all possible arrangements of a set of objects (permutations) or all possible ways of building a collection of them (subsets).
- Other applications may demand enumerating all spanning trees of a graph or all paths between two vertices, etc.

Backtracking

- Backtracking is a systematic way to go through all the possible configurations of a solutions space.
- Configurations include for example all possible arrangements of a set of objects (permutations) or all possible ways of building a collection of them (subsets).
- Other applications may demand enumerating all spanning trees of a graph or all paths between two vertices, etc.

- We must generate each one of the possible configurations exactly once.

Backtracking

- Backtracking is a systematic way to go through all the possible configurations of a solutions space.
- Configurations include for example all possible arrangements of a set of objects (permutations) or all possible ways of building a collection of them (subsets).
- Other applications may demand enumerating all spanning trees of a graph or all paths between two vertices, etc.

- We must generate each one of the possible configurations exactly once.
- To avoid repetitions and missing configurations we must define a systematic generation order among the possible configurations.

Exhaustive Search

- by exhaustive search we can solve small problems to optimality, althoug the time complexity may be enourmous

Exhaustive Search

- by exhaustive search we can solve small problems to optimality, althoug the time complexity may be enourmous
- backtracking is the basic technique for exhaustive search
- sometimes it is possible to speed up the search using pruning techniques like branch and bound or branch and cut.

(S. Skiena The algorithm design manual, Springer Verlag 1998)

Combinatorial Search

- In combinatorial search, we represent our configurations by a vector $A = (a_1, \ldots, a_n)$,
  where each element a_i is selected from an ordered set of possible candidates S_i for position i.

Combinatorial Search

- In combinatorial search, we represent our configurations by a vector $A = (a_1, \ldots, a_n)$,
  where each element a_i is selected from an ordered set of possible candidates S_i for position i.
- The search procedure works by growing solutions one element at a time.

Combinatorial Search

- In combinatorial search, we represent our configurations by a vector $A = (a_1, \ldots, a_n)$,
  where each element a_i is selected from an ordered set of possible candidates S_i for position i.
- The search procedure works by growing solutions one element at a time.
- At each step a partial solution $(a_1, \ldots, a_k)$ is constructed.
  - A candidate set S_{k+1} for position $(k+1)$ is defined,
  - try to extend the partial solution by adding the next element from S_{k+1}.
  - So long as the extension yields a longer partial solution, we continue to try to extend it.
  - At some point, $S_{k+1} = \emptyset$, if so, we must backtrack, and replace a_k, the last item in the solution value, with the next candidate in S_k.

Backtracking schema

procedure BACKTRACK(A)
  Compute S_1, the set of candidates for first position
  $k = 1$
  while $k > 0$ **do**
    while $S_k \neq \emptyset$ **do** ▷ (*advance*)
      a_k = the next element from S_k
      $S_k = S_k - \{a_k\}$
      if $A = (a_1, \ldots, a_k)$ is a solution **then**
        report it
      $k = k + 1$
      compute S_k, the set of candidate
      k-th elements of solution A.
    $k = k - 1$ ▷ (*backtrack*)

Recursive implementation

Backtracking performs a traversal of the tree of solutions.
We may use a recursive algorithm:

procedure BACKTRACKR(A, k)
  if $A = (a_1, \ldots, a_k)$ is a solution **then**
    report it
  else
    $k = k + 1$
    compute S_k
    while $S_k \neq \emptyset$ **do**
      $a_k =$ an element in S_k
      $S_k = S_k - \{a_k\}$
      BacktrackR(A, k)

Recursive implementation

Backtracking performs a traversal of the tree of solutions.
We may use a recursive algorithm:

procedure BACKTRACKR(A, k)
  if $A = (a_1, \ldots, a_k)$ is a solution **then**
    report it
  else
    $k = k + 1$
    compute S_k
    while $S_k \neq \emptyset$ **do**
      $a_k =$ an element in S_k
      $S_k = S_k - \{a_k\}$
      BacktrackR(A, k)

Each recursive call identifies a subproblem that is solved "recursively".

Generating subsets

- Let $[n] = \{1, \ldots, n\}$ be a set of n elements
- There are 2^n subsets in total and $\binom{n}{k}$ subsets with k elements

Generating subsets

- Let $[n] = \{1, \ldots, n\}$ be a set of n elements
- There are 2^n subsets in total and $\binom{n}{k}$ subsets with k elements
- We can use lexicographic order to enumerate all subsets

Generating subsets

- Let $[n] = \{1, \ldots, n\}$ be a set of n elements
- There are 2^n subsets in total and $\binom{n}{k}$ subsets with k elements
- We can use lexicographic order to enumerate all subsets
- For example when $n = 3$, and $*$ marks a configuration with no extensions, the backtracking algorithm follows the ordering

$() \rightarrow (1) \rightarrow (1,2) \rightarrow (1,2,3)* \rightarrow (1,2)* \rightarrow (1) \rightarrow (1,3)* \rightarrow (1)* \rightarrow$
$() \rightarrow (2) \rightarrow (2,3)* \rightarrow (2)* \rightarrow$
$() \rightarrow (3)* \rightarrow ()*$

Generating subsets

- Let $[n] = \{1, \ldots, n\}$ be a set of n elements
- There are 2^n subsets in total and $\binom{n}{k}$ subsets with k elements
- We can use lexicographic order to enumerate all subsets
- For example when $n = 3$, and $*$ marks a configuration with no extensions, the backtracking algorithm follows the ordering

$() \rightarrow (1) \rightarrow (1,2) \rightarrow (1,2,3)* \rightarrow (1,2)* \rightarrow (1) \rightarrow (1,3)* \rightarrow (1)* \rightarrow$
$() \rightarrow (2) \rightarrow (2,3)* \rightarrow (2)* \rightarrow$
$() \rightarrow (3)* \rightarrow ()*$

- The enumeration performs a depth-first traversal of the recursion tree

Generating subsets

- Let $[n] = \{1, \ldots, n\}$ be a set of n elements
- There are 2^n subsets in total and $\binom{n}{k}$ subsets with k elements
- We can use lexicographic order to enumerate all subsets
- For example when $n = 3$, and $*$ marks a configuration with no extensions, the backtracking algorithm follows the ordering

$$() \rightarrow (1) \rightarrow (1,2) \rightarrow (1,2,3)* \rightarrow (1,2)* \rightarrow (1) \rightarrow (1,3)* \rightarrow (1)* \rightarrow$$
$$() \rightarrow (2) \rightarrow (2,3)* \rightarrow (2)* \rightarrow$$
$$() \rightarrow (3)* \rightarrow ()*$$

- The enumeration performs a depth-first traversal of the recursion tree

$\binom{n}{k}$ can be upperbounded by n^k which is polynomial when k is a constant.

Generating all permutations

- There are $n!$ permutations of the elements of $[n]$.
- Than means, there are n choices for the first element and $n-1$ for the second and so on.

Generating all permutations

- There are $n!$ permutations of the elements of $[n]$.
- Than means, there are n choices for the first element and $n-1$ for the second and so on.
- The candidate set for the i-th position is the elements that are not in the previous position, using the notation of the backtrack algorithm $S_k = [n] - A$.

Generating all permutations

- There are $n!$ permutations of the elements of $[n]$.
- Than means, there are n choices for the first element and $n-1$ for the second and so on.
- The candidate set for the i-th position is the elements that are not in the previous position, using the notation of the backtrack algorithm $S_k = [n] - A$.

$n!$ can be bounded using Stirling's formula

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

- The algorithm performs a depth-first traversal of the configuration tree
- For example when $n = 3$, the configuration tree is

$$(1) \rightarrow (1,2) \rightarrow (1,2,3)* \rightarrow (1,2) \rightarrow (1) \rightarrow (1,3) \rightarrow (1,3,2)* \rightarrow$$
$$(1,3) \rightarrow (1) \rightarrow () \rightarrow (2) \rightarrow (2,1) \rightarrow (2,1,3)* \rightarrow (2,1) \rightarrow$$
$$(2) \rightarrow (2,3)* \rightarrow (2,3,1)* \rightarrow (2,3) \rightarrow (2) \rightarrow () \rightarrow (3) \rightarrow$$
$$(3,1) \rightarrow (3,1,2)* \rightarrow (3,1) \rightarrow (3) \rightarrow (3,2) \rightarrow (3,2,1)* \rightarrow$$
$$(3,2) \rightarrow (3) \rightarrow ()$$

Travelling Sales Person

Given n cities and the distances d_{ij} between any two of them, we wish to find the shortest tour going, only once, through all the cities.

Travelling Sales Person

Given n cities and the distances d_{ij} between any two of them, we wish to find the shortest tour going, only once, through all the cities.

First idea

Use the backtracking algorithm generating all permutations and modify it to compute the length of the associated permutation and take the minimum.

this algorithm will produce the optimum

in $O(n!d)$ where n is the number of cities and d is the length of the maximum distance.

Travelling Sales Person

Given n cities and the distances d_{ij} between any two of them, we wish to find the shortest tour going, only once, through all the cities.

Second idea: prun the tree

If we are lucky and found early a short tour, we can exclude all partial solutions with higher cost.

This may speed up the algorithm, however the worst case is still as hard as before, and we have exponential cost.

Travelling Sales Person

To implement the branch and cut prunning idea we have to

- Set a huge length as upper bound to start with.
  This can be n times the maximum length, no tour will have bigger cost
- For each partial solution keep the length of the initial part.
  This can be computed incrementally
  and continue extending the solution only when the computed length is below the upper bound.
- Each time we arrive a complete solution, update the global bound
  we have the guarantee that the bound corresponds to the best seen solution

0-1 Knapsack

We have a set I of n items, item i is of weight w_i and worth v_i. We can carry at most weight W in our knapsack. Considering that we can NOT take fractions of items, what items should we carry to maximize the profit?

First idea

Use the backtracking algorithm generating all subsets and modify it to compute the width and profit of the associated selection and take the minimum among those with weight not overpassing W.

this will produce the optimum

in time $O(2^n M)$ where n is the number of objects and M is the maximum length of the numbers in the input.

Second idea: do not consider infeasible assignments

Adapt the backtracking algorithm generating all subsets to generate only those subsets with weight not overpassing W.

It will be useful to sort items by weight, compute incrementally the weight of a partial solution, and redefine the set of candidates for next position.

This will produce the optimum, and may be faster than the first but the worst case has de same exponential cost.

Third idea: prun the tree

If we are lucky and found early a worth assignment, by excluding all partial solutions with smaller cost we can reduce de overall search.

The implementation uses the same idea as for the Min-TSP

However the worst case is still as hard as before, and we have exponential cost.

Branch and bound

Fourth idea: discard some branches by bounding improvement

Assume that objects are sorted in decreasing ratio of value/weigth, that is

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \cdots \geq \frac{v_{n-1}}{w_{n-1}} \geq \frac{v_n}{w_n}$$

Branch and bound

- When objects are sorted in decreasing ratio of value/weigth, for a partial solution $(i_1, \ldots, i_k)$ for which

$$\sum_{j=1}^{k} w_{i_j} \leq W$$

Branch and bound

- When objects are sorted in decreasing ratio of value/weigth, for a partial solution $(i_1, \ldots, i_k)$ for which

$$\sum_{j=1}^{k} w_{i_j} \leq W$$

- the maximum value that can be added to this selection is

$$\leq \sum_{j=1}^{k} v_{i_j} + \left(W - \sum_{j=1}^{k} w_{i_j} \right) \frac{v_{k+1}}{w_{k+1}}$$

Branch and bound

- When objects are sorted in decreasing ratio of value/weigth, for a partial solution $(i_1, \ldots, i_k)$ for which

$$\sum_{j=1}^{k} w_{i_j} \leq W$$

- the maximum value that can be added to this selection is

$$\leq \sum_{j=1}^{k} v_{i_j} + \left(W - \sum_{j=1}^{k} w_{i_j} \right) \frac{v_{k+1}}{w_{k+1}}$$

- We can discard the exploration of a branch for which the maximum possible plus the actual value is equal or less than the best seen assignment value.