

Algorithmics Spring 2011

- Backtracking and combinatorial search
- Generating
 - ★ Sets
 - ★ Permutations
 - ★ s - t paths
- Variations
 - ★ Branch and cut
 - ★ Branch and bound

Backtracking

- By **exhaustive search** we can solve small problems to optimality, although the time complexity may be enormous
- **Backtracking** is a systematic way to go through all the possible **configurations** of a solutions space.
- Configurations may be all possible arrangements of objects (**permutations**) or all possible ways of building a collection of them (**subsets**).
- **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]

Other applications of backtracking may demand **enumerating** the set of solutions (i.e. **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.

Often it is interesting to make use of the incremental structure of the solutions to do incremental computations instead of lengthy final evaluations.

Combinatorial Search

In **combinatorial search**, we represent our configurations by a vector

$$A = (a_1, \dots, a_n),$$

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, \dots, 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 .

It is this backtracking step that gives the procedure its name!

Recursive Backtracking schema

Backtracking performs a traversal of the tree of solutions

```
function backtrack( $A, k$ )  
  if  $A = (a_1, \dots, a_k)$  is a solution  
    then report it  
  else  
     $k := k + 1$   
    compute  $S_k$   
    while  $S_k \neq \emptyset$  do  
       $a_k :=$  the next element from  $S_k$   
       $S_k := S_k - \{a_k\}$   
      backtrack( $A, k$ )  
    end while  
  end if  
end
```

Iterative Backtracking schema

The recursive algorithm can be converted into iterative

function backtrack(A)

 Compute S_1 , the set of candidate 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 $A = (a_1, \dots, a_k)$ is a solution, report it

$k := k + 1$

 compute S_k

end while

$k := k - 1$ (*backtrack*)

end while

end

Generating all subsets

Let $[n] = \{1, \dots, n\}$ be a set of n elements

it has 2^n subsets in total

We can use lexicographic order to enumerate all subsets without repetitions

A subset (a_1, \dots, a_k) , $k \leq n$, is a solution but also a partial solution.

We have to fix the set of candidates to extend a partial solution, assuming lexicographic order,

$$S_k(a_1, \dots, a_k) = \{i \mid i > a_k\}.$$

A solution cannot be extended when $S_k = \emptyset$.

The total cost is $O(2^n)$ as we do not generate more than once any configuration.

Generating subsets of a given size

Let $[n] = \{1, \dots, n\}$ be a set of n elements,

there are $\binom{n}{k}$ subsets with k elements.

The backtracking is similar to the one for all subsets with some modifications.

For partial solution (a_1, \dots, a_i) , $i < k$,

$$S_k(a_1, \dots, a_i) = \{j \mid j > a_i\}.$$

All solutions have k elements.

$\binom{n}{k}$ can be upper bounded by n^k which is polynomial when k is a constant or by 2^n when k is a function of n .

Generating all permutations

There are $n!$ permutations of the elements of $[n]$.

That means, there are n choices for the first element and $n - 1$ for the second and so on.

For partial solution (a_1, \dots, a_k) , $k < n$,

$$S_k(a_1, \dots, a_k) = \{1, \dots, n\} \setminus \{a_1, \dots, a_k\}.$$

All solutions have n elements.

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

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

Generating all simple paths between two vertices in a graph

Let $G = (V, E)$ be a graph and s and t two of its vertices.

Enumerate all **simple paths** from s to t is more complicated than generating all permutations, because not all vertices can be used as candidates.

A simple path has no repeated vertices.

There is no closed formula counting the number of paths, because it depends on the structure of G .

- The starting point must be s , therefore $S_1 = \{s\}$.
- Candidates for the second position, are the neighbors of s .
- In general S_{k+1} contains the vertices u , such that $(u, a_k) \in E$ that are not in the current partial solution.
- We can report a successful path whenever $a_k = t$, in such a case the candidates set will be the empty set.

Cost?

Cost considerations

Observe that any s - t path have a limit of $n - 2$ vertices, and that the degree of any vertex is at most n therefore a global upper bound is n^n

When vertices have constant degree d it is better to bound the size of the tree by d^n .

Branch and cut

Avoid the exploration of a branch that will not provide an optimal solution.

For a minimization problem, if we are lucky and found early a low cost solution, we can exclude all partial solutions with higher cost.

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

Traveling 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.

We will design some algorithms for the problem using backtracking.

We assume that d_{ij} are numbers that require k bits. We will analyze the algorithms as if the sum and comparisons of k -bits numbers is a constant.

Backtracking

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

this will produce the optimum

We can estimate the cost of a tour once we have obtained the permutation with $O(n)$ operations per leaf ($n!$)

or keep together with the partial solution its partial cost, this will require one extra operation of addition/subtraction when going down on the recursion.

Total number of operations is $O(n!)$ computing incrementally the cost of a solution.

Branch and cut

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

This is due to the fact that the distances are non negative numbers and therefore a the cost of a solution is always greater than the cost of a part of the tour.

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

To implement the **branch and cut** pruning 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 to a complete solution, update the global bound

we have the guarantee that the bound corresponds to the best seen solution

Branch and bound

Avoid the exploration of a branch that will not provide an optimal solution.

For a maximization problem, after finding a solution. If we are able to bound the maximum profit that can be achieved in the solutions in a subtree and if this maximum does not exceed the actual bound on the optimum, we can exclude all partial solutions in this subtree.

Again, this may speed up the algorithm, however **the worst case cost is still as expensive as before, and we might have exponential cost.**

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?

We assume that w_i and W are numbers that require k bits. We will analyze the algorithm as if the sum and comparisons of k -bits numbers is a constant.

Backtracking-1

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

Again we will compute weight and profit incrementally.

this will produce the optimum in time $O(2^n)$.

Branch and cut

Do not consider infeasible assignments

Adapt the backtracking algorithm to cut a branch every time that the solution overpasses W .

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

Branch and bound

Idea: discard some branches by bounding improvement

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

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

In this case for a partial solution (i_1, \dots, i_k) for which

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

the maximum value that can be attained in this branch is at most

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

therefore 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.

Exercise

How much will change the behaviour of the backtracking algorithms for TSP and Knapsack when the cost of the arithmetic operations is not a constant?

Conclusions

- Combinatorial search, augmented with tree pruning techniques, can be used to find the optimal solution of small optimization problems. How small depends upon the specific problem, but the size limit is likely to be somewhere between items.
- Clever pruning techniques can speed up combinatorial search to an amazing extent. Proper pruning will have a greater impact on search time than any other factor.