Backtracking / Branch-and-Bound

Optimisation problems are problems that have several *valid* solutions; the challenge is to find an *optimal* solution. How *optimal* is defined, depends on the particular problem. Examples of optimisation problems are:

Traveling Salesman Problem (TSP). We are given a set of n cities, with the distances between all cities. A traveling salesman, who is currently staying in one of the cities, wants to visit all other cities and then return to his starting point, and he is wondering how to do this. Any tour of all cities would be a *valid* solution to his problem, but our traveling salesman does not want to waste time: he wants to find a tour that visits all cities and has the smallest possible length of all such tours. So in this case, *optimal* means: having the smallest possible length.

1-Dimensional Clustering. We are given a sorted list $x_1, \ldots, x_n$ of n numbers, and an integer k between 1 and n. The problem is to divide the numbers into k subsets of consecutive numbers (clusters) in the best possible way. A valid solution is now a division into k clusters, and an optimal solution is one that has the nicest clusters. We will define this problem more precisely later.

Set Partition. We are given a set V of n objects, each having a certain cost, and we want to divide these objects among two people in the fairest possible way. In other words, we are looking for a subdivision of V into two subsets V_1 and V_2 such that

$$\left| \sum_{v \in V_1} cost(v) - \sum_{v \in V_2} cost(v) \right|$$

is as small as possible.

Activity selection. We are given n requests for activities $a_1, \ldots, a_n$, which all have a starting time and an end time. The problem is to find a largest possible subset of the activities such that no two activities in this subset overlap.

Linear programming (LP). We are given a linear function f of m variables $x_1, \ldots, x_m$, and a number of linear inequalities of the form:

$$a_{i,1}x_1 + \cdots + a_{i,m}x_m \leqslant b_i.$$

The problem is to give all variables a value such that all inequalities are satisfied, and $f(x_1, \ldots, x_m)$ is minimum.

For all these problems we have to find an optimal solution in a (possibly very large) set of valid solutions. Note that for the first four problems, it is actually quite easy to find *a* solution. For example, the traveling salesman could just visit all cities in the order in which they appear in the input. In these cases the problem is really that we have to find an *optimal* solution. In the case of linear programming, it is different: here it is already difficult to find just *any* solution that satisfies all inequalities.

In the first weeks of the algorithms course we will discuss three general techniques to find optimal solutions for optimization problems:

- backtracking / branch-and-bound (this hand-out)
- dynamic programming (chapter 15 of Cormen et al.)
- greedy algorithms (chapter 16 of Cormen et al.)

Later we will discuss *approximation algorithms*, which do not always find an optimal solution but which come with a guarantee how far from optimal the computed solution can be.

1 Backtracking

1.1 The Traveling Salesman Problem (TSP).

We will first illustrate backtracking using TSP. Assume that all cities are numbered from 1 to n, and that we have a distance table $distance[1..n, 1..n]$. The distance from city i to city j can thus be found in $distance[i, j]$.

We will first give a simple *brute-force* algorithm to solve this problem. Such an algorithm generates all valid solutions (tours in this case) one by one, and remembers the best solution found. The easiest way to generate the solutions is by means of a recursive algorithm.

Because we have to find a *tour*, that is a path that returns to its starting point, it does not matter where we start, so let's start in city 1. The question is what would be the second city on an optimal tour. Our brute-force algorithm will simply try every city as a second city. For every choice of the second city, we will try all possible choices for a third city, and so forth.

The algorithm *TSP_BruteForce1*, given below, follows this idea. It is a recursive algorithm with two parameters, R and S. Here, R is the sequence of cities already visited (in the order in which they were visited), and S contains the remaining, still unvisited cities. The call *TSP_BruteForce1*(R, S) computes the shortest tour out of all possible tours through the cities in $R \cup S$ that start by visiting the cities in R in order. This is done by trying all cities $i \in S$ as the next city, and recursively generate all possible tours where the initial part of the tour is "R and then i", and $S - \{i\}$ are the remaining cities to be visited. In the initial call, R contains only city 1, and S contains all other cities.

The lists R and S can be implemented as singly- or doubly-linked lists; this does not matter for the correct operation of the algorithm. An implementation with arrays would also be possible. We omit the details of the representation of R and S for now, and simply write things like: "**for** each city $i \in S$ **do** ...". A programmer can later fill in the details. The details may also be relevant for a full analysis of the running time, but we will worry about this later.

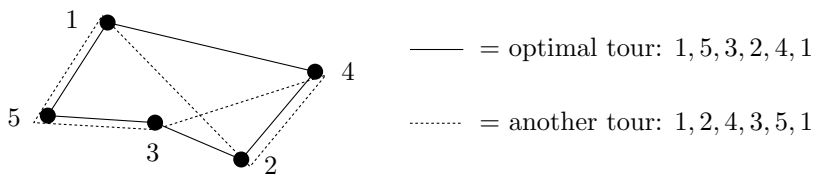

Figure 1: An example of TSP with five cities. If the distances in the *distance* table correspond to the Euclidean distances (the straight-line distances), then the tour $1, 5, 3, 2, 4, 1$ is optimal.

To keep the code clear, we also write: "$minCost \leftarrow \min(minCost, length)$". In a real implementation one would need to write an **if**-statement here, or write a function $\min(.,.)$ that does the job. When describing a recursive algorithm like the one below, it is important to clearly state the problem that the algorithm is solving and the precise definitions of its parameters.

> Let R be a sequence (that is, an *ordered* set) of cities and S be a set of cities. We want to compute a shortest tour visiting all cities in $R \cup S$, under the condition that the tour starts by visiting the cities from R in the given order.

From this problem statement, it is clear that solving the problem with R containing the city 1 and $S = \{\text{city } 2, \ldots, \text{city n}\}$ gives us an optimal tour for $\{\text{city } 1, \ldots, \text{city } n\}$, because we can without loss of generality start in city 1. Algorithm *TSP_BruteForce1* solves this problem, except that the algorithm computes the *length* of an optimal tour, not the tour itself. As an exercise, you may figure out yourself how to adapt the algorithm so that it returns the tour itself.

Algorithm *TSP_BruteForce1*(R, S)
1. **if** S is empty
2. **then** $minCost \leftarrow$ length of the tour represented by R
3. **else** $minCost \leftarrow \infty$
4. **for** each city $i \in S$
5. **do** Remove i from S, and append i to R.
6. $minCost \leftarrow \min(minCost,$ *TSP_BruteForce1*$(R, S))$
7. Reinsert i in S, and remove i from R.
8. **return** $minCost$

Note that using the distance table *distance* we can easily compute the length of a tour (line 2) in $O(n_R)$ time, where n_R is the number of cities in the tour.

How much time does *TSP_BruteForce1* take? To estimate this, we use a recurrence. Let $T(n_R, n_S)$ be the time taken by the algorithm when the set R consists of n_R cities and the set S consists of n_S cities. Then, assuming that lines 5 and 7 can be done in $O(1)$ time, we have:

$$T(n_R, n_S) = \begin{cases} O(n_R) & \text{if } n_S = 0 \\ n_S \cdot (O(1) + T(n_R + 1, n_S - 1)) & \text{if } n_S > 0 \end{cases}$$

We can use this recurrence to prove that $T(n_R, n_S) = O((n_R + n_S)n_S!)$. In the initial call we have $n_R = 1$ and $n_S = n - 1$, so the total time for the algorithm is $T(1, n-1) = O(n!)$. You may have seen that already without the recurrence: the algorithm always chooses city 1 as the first city, and then generates all permutations of the remaining cities $2, \ldots, n$ for the rest of the tour. Because there are $(n-1)!$ permutations and for each of them, we use $O(n)$ time to compute the length of the corresponding tour, the total algorithm takes $O(n!)$ time. However, this ignores the overhead for generating the permutations (such as the operations on the sets R and S), so the analysis is not complete yet.

We will now make our algorithm more precise. Let's represent R and S by a single array A and an index ℓ. The array A contains all cities; more precisely it contains a permutation of the numbers 1 to n. The index ℓ indicates which parts of A represents R and which part of A represents S. More precisely, $A[1..\ell]$ represents R (so it contains the cities in the beginning of the tour, in order), and $A[\ell + 1..n]$ represents S (so it contains the cities that still need to

be visited). While we are working on the details anyway, let's speed up the algorithm a bit too: to avoid spending $O(n)$ time on line 2 to compute the length of a tour, we will use a variable *lengthSoFar* in which we maintain the length of the beginning of the tour (the part represented by R). Thus we get the following algorithm.

Algorithm *TSP_BruteForce2*$(A, \ell, \mathit{lengthSoFar})$
1. $n \leftarrow \mathit{length}[A]$ // number of elements in the array A
2. **if** $\ell = n$
3. **then** $\mathit{minCost} \leftarrow \mathit{lengthSoFar} + \mathit{distance}[A[n], A[1]]$ // finish by returning to city 1
4. **else** $\mathit{minCost} \leftarrow \infty$
5. **for** $i \leftarrow \ell + 1$ **to** n
6. **do** Swap $A[\ell + 1]$ and $A[i]$ // select $A[i]$ as the next city
7. $\mathit{newLength} \leftarrow \mathit{lengthSoFar} + \mathit{distance}[A[\ell], A[\ell + 1]]$
8. $\mathit{minCost} \leftarrow \min(\mathit{minCost}, \mathit{TSP\_BruteForce2}(A, \ell + 1, \mathit{newLength}))$
9. Swap $A[\ell + 1]$ en $A[i]$ // undo the selection
10. **return** *minCost*

In the initial call we have $A[i] = i$ for each $1 \leqslant i \leqslant n$, $\ell = 1$ (because we always start in city 1), and $\mathit{lengthSoFar} = 0$. The running time of the algorithm satisfies the same recurrence as the running time of *TSP_BruteForce1*, except that now we have $T(n_R, n_S) = O(1)$ when $n_S = 0$. This reduces the running time to $O((n-1)!)$, but unfortunately this is still very slow. (For example, 20! is already more than $2.4 \cdot 10^{18}$.) Of course the algorithm is so slow because it looks at *all* possible tours. We can make the algorithm more efficient by *pruning*: as soon as we can see that, with the given selection of cities in the beginning of the tour, we will never find an optimal solution, we skip the recursive calls that would try to complete such a tour. To be able to do this, we add *minCost*, the length of the best tour found so far, as a parameter to the recursive calls. In the initial call, we fill in ∞ for *minCost*. Again, we first specify the exact problem the algorithm is solving.

> Let A be an array of n cities and let $1 \leqslant \ell \leqslant n$ be a parameter. Let *lengthSoFar* be the length of the path $A[1], A[2], \ldots, A[\ell]$ and let $\mathit{minCost} > 0$ be another parameter. We want to compute $\min(\mathit{minCost}, L)$, where L is the minimum length of a tour visiting all cities in $A[1..n]$ under the condition that the tour starts by visiting the cities from $A[1..\ell]$ in the given order.

Algorithm *TSP_Backtrack*$(A, \ell, \mathit{lengthSoFar}, \mathit{minCost})$
1. $n \leftarrow \mathit{length}[A]$ // number of elements in the array A
2. **if** $\ell = n$
3. **then** $\mathit{minCost} \leftarrow \min(\mathit{minCost}, \mathit{lengthSoFar} + \mathit{distance}[A[n], A[1]])$
4. **else** **for** $i \leftarrow \ell + 1$ **to** n
5. **do** Swap $A[\ell + 1]$ and $A[i]$ // select $A[i]$ as the next city
6. $\mathit{newLength} \leftarrow \mathit{lengthSoFar} + \mathit{distance}[A[\ell], A[\ell + 1]]$
7. **if** $\mathit{newLength} \geqslant \mathit{minCost}$ // this will never be a better solution
8. **then skip** // prune
9. **else** $\mathit{minCost} \leftarrow$
10. $\min(\mathit{minCost}, \mathit{TSP\_Backtrack}(A, \ell + 1, \mathit{newLength}, \mathit{minCost}))$
11. Swap $A[\ell + 1]$ and $A[i]$ // undo the selection
12. **return** *minCost*

In line 7, we check if the beginning of the tour is already longer than the best complete tour found so far. If this is the case, we will not look for tours that begin in this way, and undo the selection just made. Another way to describe it, is that we maintain that *minCost* is an upper bound on the length of the best tour. This bound is improved every time we find a better tour. Before we enter any branch of recursion, we first compute a lower bound on the length of any solutions that could be found in that branch. This lower bound is *newLength*. When the lower bound is higher than our current upper bound, we skip the branch, otherwise we explore the branch to look for a better upper bound. Backtracking algorithms with pruning are also called *branch-and-bound* algorithms.

The above branch-and-bound algorithm is probably much faster than the previous algorithm. However, it is very difficult to prove anything about it, because the time savings depend a lot on the order in which the tours are generated.

We can still try to make the pruning step a bit smarter. Consider, for example, *Euclidean TSP*, where the cities are represented by points in the plane, and the distance between any pair of cities is the length of the line segment between those cities. It is not hard to prove (you may try this as an exercise!) that an optimal tour can never intersect itself. Therefore we can add a test to step 7, to check if the newly added tour segment, from $A[\ell]$ to $A[\ell+1]$, intersects one of the segments in the partial tour $A[1..\ell]$. If this is the case, we do not have to bother exploring this choice for $A[\ell+1]$ recursively. This pruning technique may eliminate lots of recursive calls, and thus make the algorithm a lot faster.

However, the test in the pruning step in line 7 has now become a lot more expensive. Previously we could do the test in $O(1)$ time, but now we need $O(\ell)$ time to check for intersecting line segments. Therefore it is not so clear if this pruning technique is really a good idea. In general, when designing a backtracking algorithm, you have to find the right balance between elaborate pruning tests (which are often successful, but time-consuming) and simple pruning tests (which may be less successful, but faster).

1.2 A general framework for backtracking.

We will now describe a general framework for backtracking algorithms. To keep it simple, we assume that we want to solve an optimization problem where we have to find a solution with *minimum* cost among all valid solutions for the problem. Furthermore, we assume that the solution can be constructed by making a number of choices—in the TSP example: what is the first city on the tour, what is the second city, etc. In the algorithm given below, A is a data structure that somehow represents the choices already made (in other words, the partial solution constructed so far) and *minCost* is the value of the best solution found so far.

```
Algorithm Opt_Backtrack(A, minCost, ...)
1.   if A is a complete solution
2.     then minCost ← min(minCost, the cost of the solution represented by A)
3.     else  for each possible option for the next choice to be made
4.                do try that option, that is, change A to incorporate this choice
5.                   if A cannot become better than minCost
6.                     then skip    // prune
7.                     else  minCost ← min(minCost, Opt_Backtrack(A, ...))
8.                   change A back to undo the choice made in line 4
9.   return minCost
```

Note again that this algorithm does not return an optimal solution, but only the cost of an optimal solution. You may adapt the framework yourself so that the algorithm does return an actual solution.

To apply the above framework to a specific problem, you have to fill in a number of things. One of these things is how to represent a partial solution (what exactly is A), and in general, what parameters the procedure has. Furthermore you have to fill in steps 3 and 5.

In step 3 you have to generate all possible choices that can extend the partial solution represented by A. How this is done, is of course closely related to what structure is used to implement A. Sometimes it is useful to store some more information. In our TSP algorithm, for example, A does not only store a representation of the partial solution constructed so far, but also a list of cities that still need to be visited.

In step 5 we test if choice $\mathcal{C}$ is a potentially useful extension of A. To be able to do this test, it may also be necessary to pass on some additional parameters (for example, the best solution found so far). If choice $\mathcal{C}$ cannot lead to an optimal solution, we prune. As mentioned above, you will have to find a balance here between an elaborate and a simple pruning test. It is practically impossible to give general guidelines for this.

Warning. TSP has the convenient property that each possible choice brings you closer to a complete tour. So in a way, you always make progress, and the recursion cannot go arbitrarily deep.

However, it is not always so easy. Consider, for example, the well-known fifteen puzzle. Here we are given a field of 4×4 squares. On 15 of the 16 squares there is a square tile with a part of a picture, or a letter. The remaining square is empty. The goal of the puzzle is to shift the tiles such that together they show the correct picture or text. This problem can be solved with a backtracking algorithm. Generating all possible choices from a particular situation is easy: the only tiles that can shift are those that border on the empty square. However, shifting a tile does not necessarily bring us closer to a solution. It is even possible that shifting a tile takes us back to a situation where have been before. As a result, the algorithm may enter an infinite loop, for example if we keep shifting a tile back and forth. Therefore, in a backtracking algorithm for such a problem, we must (also) check in step 3 or 5 if the choice does not take us back to a situation where we have been before. Thus we avoid entering an infinite loop.

1.3 1-Dimensional Clustering.

Let's look at one more example of backtracking: the clustering problem. We will first define this problem precisely. The input of this problem is a sequence $X = x_1, \ldots, x_n$ of n numbers in ascending order, and an integer k (with $1 \leqslant k \leqslant n$). For $1 \leqslant i \leqslant j \leqslant n$ we define the cluster X_{ij} as $X_{ij} := \{x_i, \ldots, x_j\}$. The cost $cost(X_{ij})$ of cluster X_{ij} is defined as the sum of the distances of the points in the cluster to the average value in the cluster. In other words,

$$cost(X_{ij}) = \sum_{l=i}^{j} |x_l - \overline{x_{ij}}|,$$

where $\overline{x_{ij}} := \frac{1}{j-i+1} \sum_{l=i}^{j} x_l$. The smaller the cost $cost(X_{ij})$ of a cluster, the closer the numbers are together, and thus the more "clustery" X_{ij} is. The goal is to find a subdivision of X into

k clusters such that the sum of the cluster costs is as small as possible. (A cluster can also be empty; such a cluster has zero cost. However, an optimal solution would never include an empty cluster.)

Our backtracking algorithm for this problem will generate all possible subdivisions into k clusters, and prune a recursive call if it cannot lead to an optimal solution anymore. The choices we have to make to arrive at a subdivision are the leftmost points of each of the k clusters. A solution can therefore be represented by an array $A[1..k]$, where $A[i]$ is the index of the leftmost point in the i-th cluster. Thus the i-th cluster contains the points $\{x_{A[i]}, \ldots, x_{A[i+1]-1}\}$ if $i < k$, and the k-th cluster contains the points $\{x_{A[k]}, \ldots, x_{A[n]}\}$. Note that the first cluster always starts at x_1, so we can already fill in $A[1] = 1$. We assume that the set of points is given as an array $X[1..n]$. The problem that our backtracking algorithm must solve is the following.

> Let $X[1..n]$ be a sorted array of numbers and let k be a parameter with $1 < k \leqslant n$. Let $A[1..k]$ be an integer array; the value $A[i]$ will represent the index of the point where the i-th cluster starts. Let ℓ be a parameter with $1 \leqslant \ell \leqslant k$. The values of $A[1..\ell]$ are already filled in and satisfy $A[1] < A[2] < \ldots < A[\ell] \leqslant n$; they are the choices we already made. Let *costSoFar* be the cost of the partial solution represented by $A[1..\ell]$ (that is, the total cost of the clusters defined by $A[1..\ell]$). Let *minCost* > 0 be a parameter.
>
> We want to compute $\min(\mathit{minCost}, M)$, where M is the minimum cost of a partitioning of the points in $X[1..n]$ into k clusters under the condition that the first clusters are as specified by $A[1..\ell]$.

The algorithm is as follows. (For the initial call we have $\ell = 1$ and $A[1] = 1$, and *minCost* $= \infty$ and *costSoFar* $= 0$.)

```
Algorithm Cluster_Backtrack(X, k, A, ℓ, minCost, costSoFar)
1.  n ← length[X]
2.  if ℓ = k
3.     then minCost ← min(minCost, costSoFar + cost({x_A[k], ..., x_n}))
4.     else for i ← A[ℓ] + 1 to n − (k − m − 1)    // leave room for remaining clusters
5.             do A[ℓ + 1] ← i    // select i as starting point of the next cluster
6.                newCost ← costSoFar + cost({x_A[ℓ], ..., x_A[ℓ+1]−1})
7.                if newCost ⩾ minCost
8.                   then skip    // prune
9.                   else minCost ←
10.                          min(minCost, Cluster_Backtrack(X, k, A, ℓ+1, minCost, newCost))
11. return minCost
```

How fast is this algorithm? Again it is difficult to prove anything about the effect of pruning, so for a worst-case analysis, we will simply assume that recursive calls are never pruned. Instead of deriving a recurrence we will now give an intuitive explanation of the running time. The algorithm generates all possible subdivisions into k clusters, or, in other words, all possible ways to fill in $A[1..k]$. For each $A[i]$, with $2 \leqslant i \leqslant k$, we have at most n options; for $A[1]$ we have only one option, namely $A[1] = 1$. The total number of possible clusterings is therefore $O(n^{k-1})$.

Whenever we make a choice for a next cluster, we compute the cost of the previous cluster. Computing the cost of a cluster of m numbers takes $O(m)$ time. Because in any solution, the

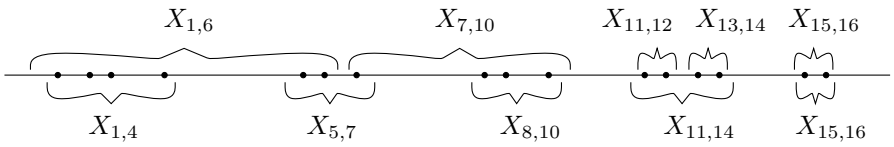

Figure 2: An example of the clustering problem with $n = 16$ and $k = 5$. The numbers are shown as points on the number line. Two possible subdivisions in five clusters are shown; the lower one is optimal.

total size of all clusters is n, the cost calculations for any particular clustering take $O(n)$ time in total. Thus the total running time of the above algorithm is $O(n^{k-1}) \times O(n) = O(n^k)$. Again, this is not very fast. Fortunately it is possible to devise a much faster solution for this problem. One way to do this is using dynamic programming, as we will see later in the course.

1.4 Exercises.

Below are several exercises to prepare for the exam. The homework assignment will be posted on the course website separately.

1. Prove by induction that the recurrence on page 3 solves to $T(n_R, n_S) = O((n_R+n_S)n_S!)$.

2. Adapt algorithm *TSP_Backtrack* such that it reports an optimal solution, instead of only the value of an optimal solution. Explain how the change affects the running time of the algorithm.

3. Suppose that the distances for a given TSP instance satisfy the triangle inequality, meaning that for any three cities i, j, k we have $\mathit{distance}[i, k] \leqslant \mathit{distance}[i, j] + \mathit{distance}[j, k]$. Show how to make the pruning test in *TSP_Backtrack* stronger (so that it will prune more) by taking into account that the tour must eventually go back to city 1. Explain why you need the triangle inequality.

4. Give pseudocode for an algorithm that prints, for a given positive integer n, all permutations of $1, \ldots, n$. Carefully explain the meaning of the various parameters of your algorithm and the exact problem that your algorithm solves (in terms of these parameters). Also specify the value of the parameters in the initial call.

5. Prove that an optimal solution for the 1-dimensional clustering problem will never use an empty cluster.

6. Consider the Set-Partition problem as defined on page 1 of this Handout. Assume that the input is an array $C[1..n]$ such that $C[i]$ is the cost of the i-th object in V.

   (i) Give pseudocode for a recursive backtracking algorithm for this problem that computes the value of an optimal solution. Carefully explain the meaning of the various parameters of your algorithm and the exact problem that your algorithm solves (in terms of these parameters). Also specify the value of the parameters in the initial call. (You don't have to use pruning and you don't have to find the most efficient backtracking algorithm: just describe the simplest version.)

(ii) Give a recurrence for the running time of your algorithm and solve the recurrence using induction. Also give an intuitive explanation of the running time.

(iii) Adapt your algorithm such that it reports an optimal solution, instead of only the value of an optimal solution.

(iv) Explain how pruning can be used to reduce the number of different partitions generated by the algorithm. (You don't have to describe the resulting algorithm in detail; just describe in a few lines which extra information you have to maintain and how to perform the pruning.)

7. The *N-Queens Problem* is the problem to place N queens on an $N \times N$ chess board such that no two queens attack each other. (A queen attacks another queen if the two queens are placed in the same row, then same column, or the same diagonal of the chess board. Note that by diagonal we not only mean the main diagonal (consisting of N squares) but all diagonals.) The N-Queens Problem is solvable for all $N > 3$.

   In more abstract terms the problem can be stated as follows.

   The input is an array $B[1..N, 1..N]$ of booleans, for some $N > 3$, modeling the chess board. Initially all entries are FALSE. The problem is to set N of the entries to TRUEin such a way that in each column, row, and diagonal of the array at most one entry is TRUE. Give pseudocode for a backtracking algorithm for this problem, and analyze the running time of your algorithm. Carefully explain the meaning of the various parameters of your algorithm and the exact problem that your algorithm solves (in terms of these parameters).