Brute-Force Algorithms

Expression "Brute-force algorithms" is often used in a negative way. This is not correct. Brute-force algorithm may be an absolutely adequate solution in many cases:

- the size of a problem is small
- the problem is a one time deal
- there are no other solutions

Typically brute-force algorithm is in one of these forms

- find a element satisfying a given set of constraints
- find a element maximizing (or minimizing) a given function

Pseudo-code:

```
ALG1(...) {
    for each element E in all possible answers {
        if ( E satisfies constraints ) return E;
    }
    retrun NULL;
}
```

The above algorithm will traverse all possible answers and will return immediately when it finds one that satisfies all constraints. The worst case analysis of the run-time: the element that satisfies constraints is the last, or does not exist. The worst-case run-time is proportional to the number of iterations.

```
ALG2(...) {
    best = NULL;
    for each element E in all possible answers {
        if ( E is better then best ) { best = E; }
    }
    retrun E;
}
```

Typically this algorithm will traverse all elements (it cannot return early as the one before) keeping track of the best so far. If current element is better than the best so far, it will update the best so far with the current. Note that because algorithm always goes through all iterations the average-case and worst-case run-times are equal and usually proportional to the number of iterations (depends on the complexity and frequency of ``best = E;'')

Examples

1. Maximum and minimum of a collection (think array/sequence)

```
MAX(A, N) {
    index=NULL;
    for (i=0; i<N; ++i) {
        if ( A[i]>A[index] ) index = i;
    }
    return index;
}
```

2. Searching in a collection:

```
SEARCH(A, N, Val) {
    for (i=0; i<N; ++i) {
        if ( A[i] == Val ) return i;
    }
    return NULL;
}
```

Note that when element is not in the collection the returned index is NULL,

Run-time:

- best - element is at position 0, so 1 comparison O(1)
- worst - element is not in the collection, O(N)
- average - it depends on the probability of Val being in a collection. By default the probability should be assumed 0, and average case is same as the worst. If you know that element that you are looking is in the collection, the run-time is N/2 on average. Which is still O(N).

3. Sorting an array. Sorting an array requires finding a permutation that sort the input. Therefore brute-force algorithm should traverse all permutations. To check if current permutation is the answer we apply permutation to the original array and check if the result is sorted:

```
SORT(A, N) {
    for P in permutations {
        //apply P to A
        B = P( A );
        if ( B[0] < ... < B[N-1] ) return B;
    }
    return NULL; // should never be here
}
```

This is the slowest deterministic sorting algorithm – often called Snail-sort. Note that as we discussed in the beginning of the class, there is no always-fastest algorithm nor always-slowest. For example Snail-sort will beat QuickSort on an already sorted collection.

4. Closest-pair – given a collection of points in 2-D find two distance between which is the smallest. This is a minimization algorithms. The answer is a pair of points, thus:

```
CLOSESTPAIR(P,N) {
    ind1 = NULL;
    ind2 = NULL;
    min_dist = +infinity;
    for all pairs (i,j) {
        if ( dist( P[i], P[j] ) < min_dist ) {
            ind1 = i;
            ind2 = j;
            min_dist = dist( P[i], P[j] )
        }
    }
}
```

5. Convex Hull. Definition: given a set of point P on a plane, *convex hull* is a subset of points which forms a convex polygons and contain all other point of the set P inside.

   Using the above definition

```
CONVEXHULL1(P,N) { // with an error in function inside
    for all subsets S in P {
        if ( inside(P,S) ) {
            return S;
        }
    }
    return NULL; //should never be here
}

bool inside(P,S) {
    //assume S=(s_1,s_2,...,s_k)
    for all edges s_i,s_(i+1) {
        if ( NOT all points of P are on the same side of the edge ) return false;
    }
    return true;
}
```

   there is a small problem: subsets in mathematics (and the definiton of the convex hull) are unordered. While our code for `inside` function assumes that subset is ordered. To fix the problem we need to also consider all permutations of S:

```
bool inside(P,S) {
    for all permutation of S {
        //assume S=(s_1,s_2,...,s_k)
        for all edges s_i,s_(i+1) {
            if ( NOT all points of P are on the same side of the edge ) break;
        }
        return true;
    }
```

```
    return false;
}
```

While implemention `all points of P are on the same side of the edge` you will notice that edge P[i],P[j] belongs to convex hull iff all other points P[k] lie on the same side of the line P[i],P[j]. The previous statement may be considered as an alternative definition of convex hull. Using it we get another implementation:

```
CONVEXHULL2(P,N) { //N is the size of P
    S={}; //set of points forming convex hull
    for (i=0; i<N-1; ++i) {
        for (j=i+1; j<N; ++j) {
            same_side = true;
            for (k=0; k<N && same_side; ++k) {
                same_side = same_side && ( P[k] is on same side with other points)
            }
            if ( same_side ) add P[i] and P[j] to S
        }
    }
    return S
}
```

Run-time of `CONVEXHULL2` is N^3 – three nested loops.

There is an optimization – instead of considering all possible edges (i.e. pairs of points), always use the latest vertex that was added to the hull as one of the endpoints. To get the process started, note that vertex with the smallest (biggest) x (y) coordinate is guaranteed to belong to the hull. So

(a) add vertex v_1 with the smallest x coordinate to the hull

(b) consider all edges $v_1 - p_i$ and find v_2 so that $v_1 - v_2$ has all points of P on one side, add v_2 to convex hull

(c) consider all edges $v_2 - p_i$ and find v_3 so that $v_2 - v_3$ has all points of P on one side, add v_3 to convex hull

(d) .....

(e) if just added vertex is equal to v_1 stop.

Note that the number of steps above is exactly the number of vertices in the convex hull (denote it by K). Therefore run-time complexity with optimization is $K \times N^2$. In the worst-case all points of P belong to convex hull and there will be N^3 iterations.

Backtracking In a nutshell – backtracking is a recursive implementation of a brute-force algorithm. Usually the recursion is based on the recursive generation of all possible answers – that is instead of using `for each element E in all possible answers ...` the looping is done using recursion. Note that by default recursive implementation is leass efficient than iterative, but recursion has two possible betefits:

1. it may be easier to program
2. it may be optimizable (see branch-and-bound below)

Example – traversing all sets using recursive algorithm:

```
#include <iostream>
#include <vector>

//an example of backtracking. Recursion tree is a binary tree based on
//"add / not add the current element" current element is an element with index
//equal to the current depth of recursion
void subset_rec( std::vector<int> const& set, std::vector<int> & subset, int depth ) {
    if ( depth == set.size() ) { //base case -> subset complete, print it
        std::cout << "{ ";
        for (int i=0; i<subset.size(); ++i) { std::cout << subset[i] << " "; }
        std::cout << "}" << std::endl;
        return; //hit the bottom, start backtracking (go up)
    }

    subset.push_back( set[depth] );  //add
    subset_rec(set,subset,depth+1 ); //go down the recursion tree

    subset.pop_back(); //do not add - remove just added
    subset_rec(set,subset,depth+1 ); //go down the recursion tree

    return; //backtracking (go up)
}

//kick start recursion
void subsets( std::vector<int> const& set) {
    std::vector<int> subset;
    subset_rec(set,subset,0 );
}

int main () {
    std::vector<int> set;
    set.push_back(1); set.push_back(2); set.push_back(3); set.push_back(4);
    subsets(set);
}
```

The code above just prints all subsets, we can easily modify it to something useful – for example solve knapsack problem (add an array of items and perform calculation of the bag value to keep track of the largest).

Or we can solve subset sum problem: given a set of positive numbers, answer whether there exists subset whose elements add up to a given value.

```
#include <iostream>
#include <vector>
#include <numeric> //accumulate

bool subset_rec( std::vector<int> const& set, int sum, std::vector<int> & subset, int index ) {
    if ( index == set.size() ) {
        int curr_sum = std::accumulate( subset.begin(), subset.end(), 0);
        for (int i=0; i<subset.size(); ++i) { std::cout << subset[i] << " "; }
        std::cout << "  sum " << curr_sum << std::endl;
        if ( curr_sum == sum ) return true; //success, cancel recursion
        else                   return false; //hit the bottom, start backtracking
    }

    //add
    subset.push_back( set[index] );
    if ( subset_rec(set,sum,subset,index+1) ) {
        return true; //cancel recursion - we have a solution
    }

    subset.pop_back();
    //do not add
    if ( subset_rec(set,sum,subset,index+1) ) {
        return true; //cancel recursion - we have a solution
    }

    return false; // continue backtracking
}

std::vector<int> subset_sum( std::vector<int> const& set, int sum, bool& solution_found) {
    std::vector<int> subset;
    solution_found = subset_rec(set,sum,subset,0);
    return subset;
}

int main () {
    std::vector<int> set;
    set.push_back(2);  set.push_back(6);  set.push_back(9);
    set.push_back(11); set.push_back(14); set.push_back(23);
    bool solution_found = false;

    std::vector<int> subset = subset_sum(set,16,solution_found); // solution 2 14
    //std::vector<int> subset = subset_sum(set,17,solution_found); // no solution

    if ( solution_found ) {
        std::cout << "Answer:\n";
        for (int i=0; i<subset.size(); ++i) { std::cout << subset[i] << " "; }
```

```
        std::cout << std::endl;
    } else { std::cout << "No solution\n"; }
}
```

Assignment problem: given N workers and N jobs assign each worker a (single) job to minimize the total cost. Costs are given by an $N \times N$ matrix, element at position (i, j) is the cost of worker index i executing j's job. The answer is a sequence $j_0, j_1, \ldots, j_{N-1}$ where worker i executes job j_i. Since jobs are enumerated $0, 1, \ldots, N-1$ the sequence $j_0, j_1, \ldots, j_{N-1}$ is a permutation.

To solve the problem using backtracking we first need a way to generate permutations using recursion:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator> //ostream_iterator

////////////// BACKTRACKING ////////////////////////
template< typename T >
void generate_permutation_aux(
        std::vector<T> const & vec,    // what is permuted
        std::vector<T> & permutation ) //current permutation
{
    //termination check (if on the last level)
    //note that depth is permutation.size()
    if ( permutation.size() == vec.size() ) {
        std::copy( permutation.begin(), permutation.end(), std::ostream_iterator<T>(std::cout, "
        std::cout << std::endl;
        return;
    }

    //generate nodes
    typename std::vector<T>::const_iterator b = vec.begin(), e = vec.end();
    for ( ; b!=e; ++b ) {
        //skip j is already in permutation
        if ( std::find( permutation.begin(), permutation.end(), *b) != permutation.end() ) contin

        //otherwise add j and call recursively
        permutation.push_back( *b );
        generate_permutation_aux( vec, permutation );
        //since permutation is passed by reference we have to remove last job
        //to be able to assign a new instead:
        permutation.pop_back( );
    }
}

template< typename T >
void generate_permutation( std::vector<T> const& vec ) {
    std::vector<T> permutation;
    permutation.reserve( vec.size() ); //max size of permutation will be vec.size()
    generate_permutation_aux( vec, permutation );
}

int main() {
```

```
    typedef char TYPE;
    int const N = 4;
    std::vector<TYPE> vec;
    vec.reserve( N );
    //for (int i=0; i<N; ++i) { vec.push_back( i ); }
    for (TYPE i='a'; i<'e'; ++i) { vec.push_back( i ); }

    generate_permutation( vec );
    return 0;
}
```

Now add code that will keep track of the best assignment (assignment itself and the corresponding cost). Code is on course web-site - file `backtracking-variants.cpp` function `backtracking`

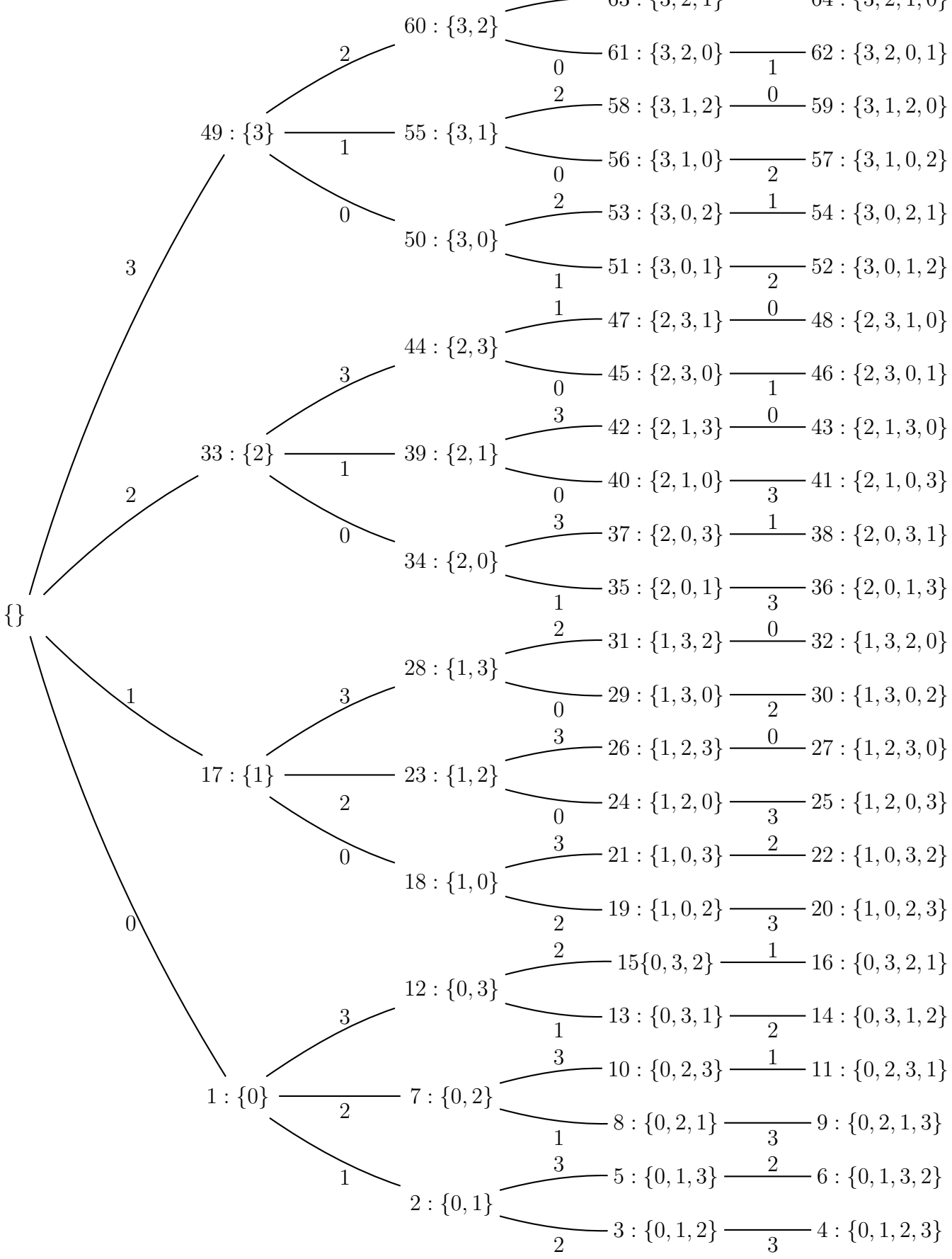
Here is a call tree for assignment problem with $N = 4$ and costs given by

	w1	w2	w3	w4
j1	6	2	4	8
j2	3	4	7	6
j3	2	7	8	5
j4	3	5	4	2

Node structure:

$$sn : \{j_i, j_k\}$$

where sn – recursive call number. $\{j_i, j_k\}$ – partial assignment (worker 1 is assigned job j_i, etc).


{}
3
2
1
0
49 : {3}
33 : {2}
17 : {1}
1 : {0}
2
1
0
60 : {3, 2}
55 : {3, 1}
50 : {3, 0}
44 : {2, 3}
39 : {2, 1}
34 : {2, 0}
28 : {1, 3}
23 : {1, 2}
18 : {1, 0}
12 : {0, 3}
7 : {0, 2}
2 : {0, 1}
61 : {3, 2, 0}
58 : {3, 1, 2}
56 : {3, 1, 0}
53 : {3, 0, 2}
51 : {3, 0, 1}
47 : {2, 3, 1}
45 : {2, 3, 0}
42 : {2, 1, 3}
40 : {2, 1, 0}
37 : {2, 0, 3}
35 : {2, 0, 1}
31 : {1, 3, 2}
29 : {1, 3, 0}
26 : {1, 2, 3}
24 : {1, 2, 0}
21 : {1, 0, 3}
19 : {1, 0, 2}
15{0, 3, 2}
13 : {0, 3, 1}
10 : {0, 2, 3}
8 : {0, 2, 1}
5 : {0, 1, 3}
3 : {0, 1, 2}
62 : {3, 2, 0, 1}
59 : {3, 1, 2, 0}
57 : {3, 1, 0, 2}
54 : {3, 0, 2, 1}
52 : {3, 0, 1, 2}
48 : {2, 3, 1, 0}
46 : {2, 3, 0, 1}
43 : {2, 1, 3, 0}
41 : {2, 1, 0, 3}
38 : {2, 0, 3, 1}
36 : {2, 0, 1, 3}
32 : {1, 3, 2, 0}
30 : {1, 3, 0, 2}
27 : {1, 2, 3, 0}
25 : {1, 2, 0, 3}
22 : {1, 0, 3, 2}
20 : {1, 0, 2, 3}
16 : {0, 3, 2, 1}
14 : {0, 3, 1, 2}
11 : {0, 2, 3, 1}
9 : {0, 2, 1, 3}
6 : {0, 1, 3, 2}
4 : {0, 1, 2, 3}

Branch and bound optimization First notice the following simple optimization (this is **NOT** branch and bound optimization) in the subset sum problem: once the sum of the current subset exceeds the goal sum, backtracking algorithm can cancel the branch – do not "go down" from the current vertex, since the sum can only increase, and since it is already greater than the goal sum, we will not find subset in a branch rooted at the current vertex. The check should be added right after the base case check (before any of the recursive calls):

```
//cancel ?
int curr_sum = std::accumulate( subset.begin(), subset.end(), 0);
if ( curr_sum > sum ) { std::cout << "cancel\n"; return false; }
```

As we go down the recursion tree we grow the object that represents a possible answer. The above optimization looks at the current (partial) answer and calculates it's current value. If the value already does not satisfy the condition then growing current answer (by going down the tree) does not make sense – adding more to the answer will not make it satisfy the condition. Note that not all problems have the property – even the subset-sum problem has the property only if all values in the set are positive.

The **branch and bound optimization** is an extension of the above idea applied to minimization/maximization problems. Assume for now it is minimization. Then branch and bound optimization will calculate

1. current value of the "partial" answer (denote by g) – exact value
2. estimate (lower bound) of the "future" value down the tree (denote by h, so that $h \leq$ actual cost).

if

$$g + h > \text{best seen answer so far}$$

then cancel the branch. Indeed – in the branch rooted at current vertex we will at best (remember it s a minimization problem) find an answer with value $g+h$, but we already have a better (smaller) value.

To apply branch and bound optimization to a maximization problem you have to switch lower bound to an upper and inequlaity that cancels branch to

$$g + h < \text{best seen answer so far}$$

The less obvious (and therefore more interesting) part of this optimization is figuring out the lower bound. Lower bound has to be a simple to calculate value – we will be calling the function in each node of the recursion tree (a lot).
suppose we have assigned job 1 to worker 2. Then we still have to assign jobs 2,3,4. Choose the cheapest worker (but not worker 2 who already has a job) for each job – simply choose minimum in rows j2, j3, and j4 (skip column w2):

$$\text{j2} = \min\{3, 7, 6\} = 3$$
$$\text{j3} = \min\{2, 8, 5\} = 2$$
$$\text{j4} = \min\{3, 4, 2\} = 2$$

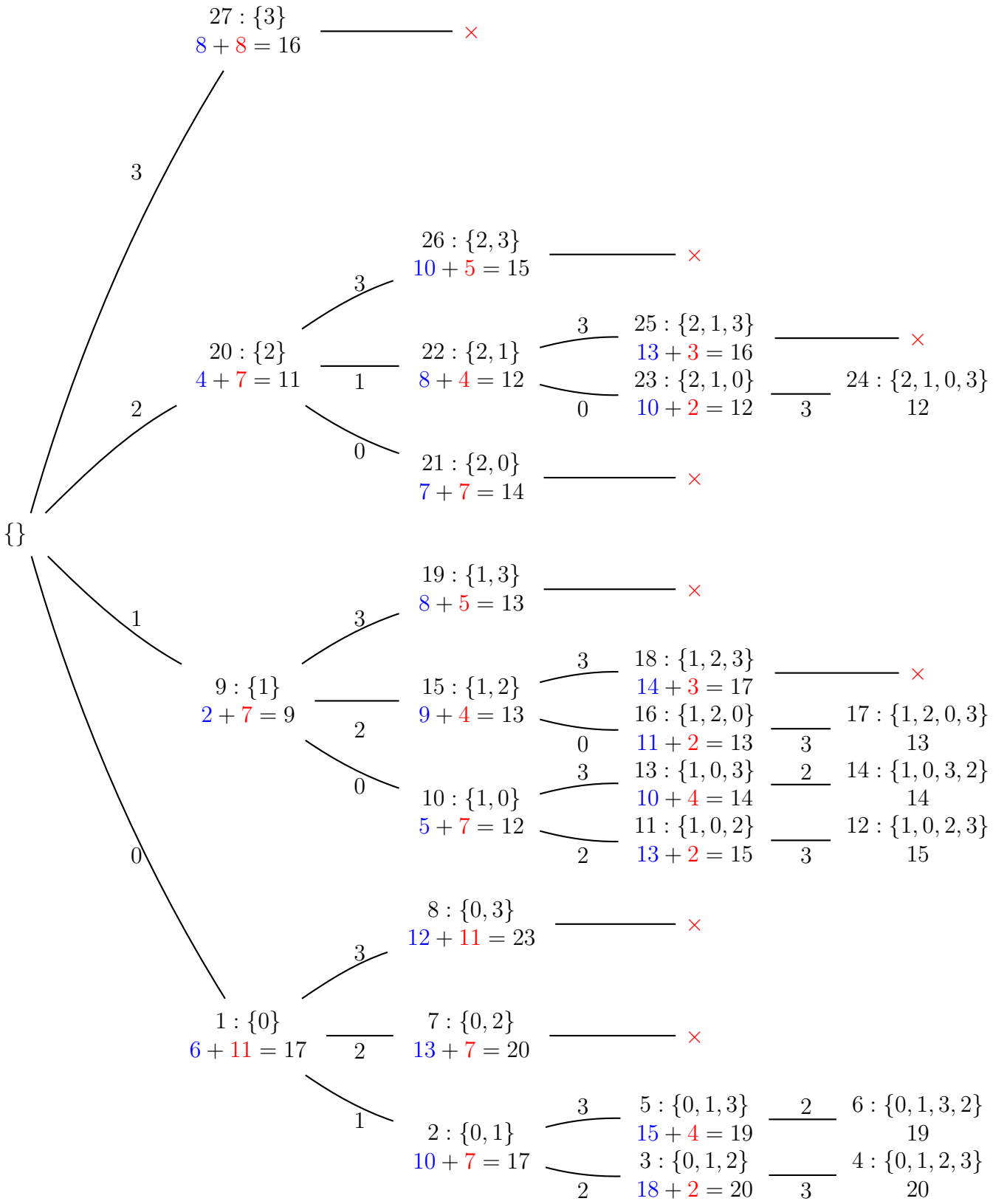
therefore the cheapest we can assign the 3 remaining jobs is $3+3+2=8$. Note that for both jobs 2 and 3 the cheapest worker is w1, and because of the requirement that one worker get exacty one job we cannot get costs 3 and 2 at the same time, thus value 8 is not achievable. But it is a **lower bound**.

Code is on course web-site - file `backtracking-variants.cpp` function `backtracking_branch_bound`.

Here is recursive calls tree for the assign problem with costs as above. Node structure:

$$sn : \{j_i, j_k\}$$
$$g+h = bound$$

where sn – recursive call number. $\{j_i, j_k\}$ – partial assignment (worker 1 is assigned job j_i, etc).


{}
3
27 : {3}
8 + 8 = 16
×
2
20 : {2}
4 + 7 = 11
3
26 : {2, 3}
10 + 5 = 15
×
1
22 : {2, 1}
8 + 4 = 12
3
25 : {2, 1, 3}
13 + 3 = 16
×
0
23 : {2, 1, 0}
10 + 2 = 12
3
24 : {2, 1, 0, 3}
12
0
21 : {2, 0}
7 + 7 = 14
×
1
9 : {1}
2 + 7 = 9
3
19 : {1, 3}
8 + 5 = 13
×
2
15 : {1, 2}
9 + 4 = 13
3
18 : {1, 2, 3}
14 + 3 = 17
×
0
16 : {1, 2, 0}
11 + 2 = 13
3
17 : {1, 2, 0, 3}
13
0
10 : {1, 0}
5 + 7 = 12
3
13 : {1, 0, 3}
10 + 4 = 14
2
14 : {1, 0, 3, 2}
14
2
11 : {1, 0, 2}
13 + 2 = 15
3
12 : {1, 0, 2, 3}
15
0
1 : {0}
6 + 11 = 17
3
8 : {0, 3}
12 + 11 = 23
×
2
7 : {0, 2}
13 + 7 = 20
×
1
2 : {0, 1}
10 + 7 = 17
3
5 : {0, 1, 3}
15 + 4 = 19
2
6 : {0, 1, 3, 2}
19
2
3 : {0, 1, 2}
18 + 2 = 20
3
4 : {0, 1, 2, 3}
20

Best-first optimizaion is an optimizaion on top of branch and bound: the idea is to visit children in the order of increaing $g + h$ (decreasing for maximization problem). Note that branch with the smallest lower bound does not necessarily contains the smallest (best) value.

Code is on course web-site - file `backtracking-variants.cpp` function `backtracking_branch_bound_best`