```
function optimaltree(network, optdata::Dict; verbose::Bool = false)
    numtensors = length(network)
    allindices = unique(vcat(network...))
    numindices = length(allindices)
    costtype = valtype(optdata)
    allcosts = [get(optdata, i, one(costtype)) for i in allindices]
    maxcost = addcost(mulcost(reduce(mulcost, allcosts; init=one(costtype)),
        maximum(allcosts)), zero(costtype)) # add zero for type stability: Power ->
        Poly
    tensorcosts = Vector{costtype}(undef, numtensors)
    for k = 1:numtensors
        tensorcosts[k] = mapreduce(i->get(optdata, i, one(costtype)), mulcost,
network[k], init=one(costtype))
    end
    initialcost = min(maxcost,
        addcost(mulcost(maximum(tensorcosts),minimum(tensorcosts)),zero(costtype)))
        # just some arbitrary guess

    if numindices <= 32
        return _optimaltree(UInt32, network, allindices, allcosts, initialcost,
maxcost; verbose = verbose)
    elseif numindices <= 64
        return _optimaltree(UInt64, network, allindices, allcosts, initialcost,
maxcost; verbose = verbose)
    elseif numindices <= 128 && !(@static Int == Int32 && Sys.iswindows() ? true :
false)
        return _optimaltree(UInt128, network, allindices, allcosts, initialcost,
maxcost; verbose = verbose)
    else
        return _optimaltree(BitVector, network, allindices, allcosts, initialcost,
maxcost; verbose = verbose)
    end
end

storeset(::Type{BitSet}, ints, maxint) = sizehint!(BitSet(ints), maxint)
function storeset(::Type{BitVector}, ints, maxint)
    set = falses(maxint)
    for i in ints
        set[i] = true
    end
    return set
end
function storeset(::Type{T}, ints, maxint) where {T<:Unsigned}
    set = zero(T)
    u = one(T)
    for i in ints
        set |= (u<<(i-1))
    end
    return set
end
_intersect(s1::T, s2::T) where {T<:Unsigned} = s1 & s2
_intersect(s1::BitVector, s2::BitVector) = s1 .& s2
_intersect(s1::BitSet, s2::BitSet) = intersect(s1, s2)
_union(s1::T, s2::T) where {T<:Unsigned} = s1 | s2
```

```
_union(s1::BitVector, s2::BitVector) = s1 .| s2
_union(s1::BitSet, s2::BitSet) = union(s1, s2)
_setdiff(s1::T, s2::T) where {T<:Unsigned} = s1 & (~s2)
_setdiff(s1::BitVector, s2::BitVector) = s1 .& (.~s2)
_setdiff(s1::BitSet, s2::BitSet) = setdiff(s1, s2)
_isemptyset(s::Unsigned) = iszero(s)
_isemptyset(s::BitVector) = !any(s)
_isemptyset(s::BitSet) = isempty(s)

addcost(cost1::Integer, cost2::Integer, costs::Vararg{<:Integer}) =
Base.Checked.checked_add(cost1, cost2, costs...)
mulcost(cost1::Integer, cost2::Integer) = Base.Checked.checked_mul(cost1, cost2)
addcost(cost1, cost2, costs...) = +(cost1, cost2, costs...)
mulcost(cost1, cost2) = cost1*cost2
function computecost(allcosts, ind1::T, ind2::T) where {T<:Unsigned}
    cost = one(eltype(allcosts))
    ind = _union(ind1, ind2)
    n = 1
    @inbounds while !iszero(ind)
        if isodd(ind)
            cost = mulcost(cost, allcosts[n])
        end
        ind = ind>>1
        n += 1
    end
    return cost
end
function computecost(allcosts, ind1::BitVector, ind2::BitVector)
    cost = one(eltype(allcosts))
    ind = _union(ind1, ind2)
    @inbounds for n in findall(ind)
        cost = mulcost(cost, allcosts[n])
    end
    return cost
end
function computecost(allcosts, ind1::BitSet, ind2::BitSet)
    cost = one(eltype(allcosts))
    ind = _union(ind1, ind2)
    @inbounds for n in ind
        cost = mulcost(cost, allcosts[n])
    end
    return cost
end

function _optimaltree(::Type{T}, network, allindices, allcosts::Vector{S},
initialcost::C, maxcost::C; verbose::Bool = false) where {T,S,C}
    numindices = length(allindices)
    numtensors = length(network)
    indexsets = Array{T}(undef, numtensors)

    tabletensor = zeros(Int, (numindices,2))
    tableindex = zeros(Int, (numindices,2))

    adjacencymatrix = falses(numtensors,numtensors)
    costfac = maximum(allcosts)
```

```
    @inbounds for n = 1:numtensors
        indn = findall(in(network[n]), allindices)
        indexsets[n] = storeset(T, indn, numindices)
        for i in indn
            if tabletensor[i,1] == 0
                tabletensor[i,1] = n
                tableindex[i,1] = _findfirst(isequal(allindices[i]), network[n])
            elseif tabletensor[i,2] == 0
                tabletensor[i,2] = n
                tableindex[i,2] = _findfirst(isequal(allindices[i]), network[n])
                n1 = tabletensor[i,1]
                adjacencymatrix[n1,n] = true
                adjacencymatrix[n,n1] = true
            else
                error("no index should appear more than two times: $i")
            end
        end
    end
    componentlist = connectedcomponents(adjacencymatrix)
    numcomponent = length(componentlist)

    # generate output structures
    costlist = Vector{C}(undef, numcomponent)
    treelist = Vector{Any}(undef, numcomponent)
    indexlist = Vector{T}(undef, numcomponent)

    # run over components
    @inbounds for c=1:numcomponent
        # find optimal contraction for every component
        component = componentlist[c]
        componentsize = length(component)
        costdict = Vector{Dict{T, C}}(undef, componentsize)
        treedict = Vector{Dict{T, Any}}(undef, componentsize)
        indexdict = Vector{Dict{T, T}}(undef, componentsize)

        for k=1:componentsize
            costdict[k] = Dict{T, C}()
            treedict[k] = Dict{T, Any}()
            indexdict[k] = Dict{T, T}()
        end

        for i in component
            s = storeset(T, [i], numtensors)
            costdict[1][s] = zero(C)
            treedict[1][s] = i
            indexdict[1][s] = indexsets[i]
        end

        # run over currentcost
        currentcost = initialcost
        previouscost = zero(initialcost)
        while currentcost <= maxcost
            nextcost = maxcost
            # construct all subsets of n tensors that can be constructed with cost
```

```
            <= currentcost
        for n=2:componentsize
            verbose &&
                println("Component $c: Constructing subsets of size $n with
cost $currentcost")
            # construct subsets by combining two smaller subsets
            for k = 1:div(n-1,2)
                for s1 in keys(costdict[k]), s2 in keys(costdict[n-k])
                    if _isemptyset(_intersect(s1, s2)) && get(costdict[n],
_union(s1, s2), currentcost) > previouscost
                        ind1 = indexdict[k][s1]
                        ind2 = indexdict[n-k][s2]
                        cind = _intersect(ind1, ind2)
                        if !_isemptyset(cind)
                            s = _union(s1, s2)
                            cost = addcost(costdict[k][s1], costdict[n-k][s2],
computecost(allcosts, ind1, ind2))
                            if cost <= get(costdict[n], s, currentcost)
                                costdict[n][s] = cost
                                indexdict[n][s] = _setdiff(_union(ind1,ind2),
cind)
                                treedict[n][s] = Any[treedict[k][s1],
treedict[n-k][s2]]
                            elseif currentcost < cost < nextcost
                                nextcost = cost
                            end
                        end
                    end
                end
            end
            if iseven(n) # treat the case k = n/2 special
                k = div(n,2)
                it = keys(costdict[k])
                elstate1 = iterate(it)
                while elstate1 !== nothing
                    s1, nextstate1 = elstate1
                    elstate2 = iterate(it, nextstate1)
                    while elstate2 !== nothing
                        s2, nextstate2 = elstate2
                        if _isemptyset(_intersect(s1, s2)) && get(costdict[n],
_union(s1, s2), currentcost) > previouscost
                            ind1 = indexdict[k][s1]
                            ind2 = indexdict[k][s2]
                            cind = _intersect(ind1, ind2)
                            if !_isemptyset(cind)
                                s = _union(s1, s2)
                                cost = addcost(costdict[k][s1],
costdict[k][s2], computecost(allcosts, ind1, ind2))
                                if cost <= get(costdict[n], s, currentcost)
                                    costdict[n][s] = cost
                                    indexdict[n][s] =
_setdiff(_union(ind1,ind2), cind)
                                    treedict[n][s] = Any[treedict[k][s1],
treedict[k][s2]]
                                elseif currentcost < cost < nextcost
```

```
                                        nextcost = cost
                                    end
                                end
                            end
                            elstate2 = iterate(it, nextstate2)
                        end
                        elstate1 = iterate(it, nextstate1)
                    end
                end
            end
            currentbiggestset = _findlast(x->!isempty(x), costdict)
            verbose &&
                println("Component $c: finished at cost $currentcost: maximum
subset has size $currentbiggestset")
            if !isempty(costdict[componentsize])
                break
            end
            previouscost = currentcost
            currentcost = min(maxcost, nextcost*costfac)
        end
        if isempty(costdict[componentsize])
            error("Component $c: maxcost $maxcost reached without finding
                solution") # should be impossible
        end
        s = storeset(T, component, numtensors)
        costlist[c] = costdict[componentsize][s]
        treelist[c] = treedict[componentsize][s]
        indexlist[c] = indexdict[componentsize][s]
        verbose &&
            println("Component $c: solution found with cost $(costlist[c]) and tree
$(treelist[c])")
    end
    p  = sortperm(costlist)

    tree = treelist[p[1]]
    cost = costlist[p[1]]
    ind = indexlist[p[1]]
    for c = 2:numcomponent
        tree = Any[tree, treelist[p[c]]]
        cost = addcost(cost, costlist[p[c]], computecost(allcosts, ind,
indexlist[p[c]]))
        ind = _union(ind, indexlist[p[c]])
    end
    verbose &&
        println("Solution found with cost $cost and tree $tree")

    return tree, cost
end

# For a given adjacency matrix of size n x n, connectedcomponents returns
# a list componentlist that contains integer vectors, where every integer
# vector groups the indices of the vertices of a connected component of the
# graph encoded by A. The number of connected components is given by
# length(componentlist).
#
```

```julia
# Used as auxiliary function to analyze contraction graph in contract.
function connectedcomponents(A::AbstractMatrix{Bool})
    n = size(A,1)
    @assert size(A,2) == n

    componentlist = Vector{Vector{Int}}(undef, 0)
    assignedlist = falses((n,))

    for i = 1:n
        if !assignedlist[i]
            assignedlist[i] = true
            checklist = [i]
            currentcomponent = [i]
            while !isempty(checklist)
                j = pop!(checklist)
                for k = findall(A[j,:])
                    if !assignedlist[k]
                        push!(currentcomponent, k)
                        push!(checklist,k)
                        assignedlist[k] = true;
                    end
                end
            end
            push!(componentlist, currentcomponent)
        end
    end
    return componentlist
end
```