Optimal Binary Search Trees

Tobias Nipkow and Dániel Somogyi
Technical University Munich

August 7, 2022

Abstract

This article formalizes recursive algorithms for the construction of optimal binary search trees given fixed access frequencies. We follow Knuth [1], Yao [4] and Mehlhorn [2].

The algorithms are memoized with the help of an AFP entry for memoization [3], thus yielding dynamic programming algorithms.

Contents

1 Introduction

These theories formalize algorithms for the construction of optimal binary search trees from fixed access frequencies for a fixed list of items. The work is based on the original article by Knuth [1] and the textbook by Mehlhorn [2, Part III, Chapter 4].

Initially the algorithms are expressed as naive recursive functions and have exponential complexity. Nevertheless we already refer to them as the cubic (Section 3) and the quadratic algorithm (Section 5), their running times of their fully memoized dynamic programming versions. In Section 7 the algorithms are memoized with the help of an existing framework [3].

1.1 Data Representation

Instead of labeling our BSTs with (ascending) keys $x_i < \ldots < x_j$ we label them with the indices of the actual keys, some interval of integers. Functions taking two integer arguments i and j construct or analyze trees such that $inorder\ t = [i..j]$.

The access frequencies are given by two tables (functions) a and b:

$a\,k$ $(i \leq k \leq j+1)$ is the frequency of (failing) searches with a key in the interval (x_{k-1}, x_k).

$b\,k$ $(i \leq k \leq j)$ is the frequency of (successful) searches with key x_k.

2 Weighted Path Length of BST

theory *Weighted_Path_Length*
imports *HOL−Library.Tree*
begin

This theory presents two definitions of the *weighted path length* of a BST, the objective function we want to minimize, and proves them equivalent. Function *Wpl* is the intuitive global definition that sums a over all leaves and b over all nodes, taking their depth (= number of comparisons to reach that point) into account. Function *wpl* is a recursive definition and thus suitable for the later dynamic programming approaches to building a BST with the minimal weighted path length.

lemma *inorder_upto_split*:
  assumes $inorder\ \langle l,k,r\rangle = [i..j]$
  shows $inorder\ l = [i..k-1]$ $inorder\ r = [k+1..j]$ $i \leq k$ $k \leq j$
$\langle proof \rangle$

fun $incr2 :: int \times nat \Rightarrow int \times nat$ **where**
$incr2\ (x,n) = (x,\ n + 1)$

fun $leaves :: int \Rightarrow int\ tree \Rightarrow (int * nat)\ set$ **where**
$leaves\ i\ Leaf = \{(i,0)\}$ |
$leaves\ i\ (Node\ l\ k\ r) = incr2\ `\ (leaves\ i\ l \cup leaves\ (k+1)\ r)$

fun $nodes :: int\ tree \Rightarrow (int * nat)\ set$ **where**

nodes Leaf = {} |
nodes (*Node l k r*) = {(*k*,1)} ∪ *incr2* ‘ (*nodes l* ∪ *nodes r*)

lemma *finite_nodes*: *finite* (*nodes t*)
⟨*proof*⟩

lemma *finite_leaves*: *finite* (*leaves i t*)
⟨*proof*⟩

lemma *notin_nodes*0: (*k*, 0) ∉ *nodes t*
⟨*proof*⟩

lemma *sum_incr2*: *sum f* (*incr2* ‘ *A*) = *sum* (λ*xy*. *f*(*fst xy*,*snd xy*+1)) *A*
⟨*proof*⟩

lemma *fst_nodes*: *fst* ‘ *nodes t* = *set_tree t*
⟨*proof*⟩

lemma *fst_leaves*: ⟦ *inorder t* = [*i*..*j*]; *i* ≤ *j*+1⟧ ⟹ *fst* ‘ *leaves i t* = {*i*..*j*+1}
⟨*proof*⟩

lemma *sum_leaves*: ⟦ *inorder t* = [*i*..*j*]; *i* ≤ *j*+1 ⟧ ⟹
(∑ *x*∈*leaves i t*. (*f*(*fst x*) :: *nat*)) = *sum f* {*i*..*j*+1}
⟨*proof*⟩

lemma *sum_nodes*: *inorder t* = [*i*..*j*] ⟹
(∑ *xy*∈*nodes t*. (*f*(*fst xy*) :: *nat*)) = *sum f* {*i*..*j*}
⟨*proof*⟩

locale *wpl* =
fixes *w* :: *int* ⇒ *int* ⇒ *nat*
begin

fun *wpl* :: *int* ⇒ *int* ⇒ *int tree* ⇒ *nat* **where**
wpl i j Leaf = 0 |
wpl i j (*Node l k r*) = *wpl i* (*k*−1) *l* + *wpl* (*k*+1) *j r* + *w i j*

end

locale *Wpl* =
fixes *a b* :: *int* ⇒ *nat*
begin

definition *Wpl* :: *int* ⇒ *int tree* ⇒ *nat* **where**
Wpl i t = *sum* (λ(*k*,*c*). *c* ∗ *b k*) (*nodes t*) + *sum* (λ(*k*,*c*). *c* ∗ *a k*) (*leaves i t*)

definition *w* :: *int* ⇒ *int* ⇒ *nat* **where**
w i j = *sum a* {*i*..*j*+1} + *sum b* {*i*..*j*}

sublocale *wpl* **where** $w = w$ ⟨*proof*⟩

lemma $inorder\ t = [i..j] \Longrightarrow wpl\ i\ j\ t = Wpl\ i\ t$
⟨*proof*⟩

end

end

3 Optimal BSTs: The 'Cubic' Algorithm

theory *Optimal_BST*
imports *Weighted_Path_Length Monad_Memo_DP.OptBST*
begin

3.1 Function *argmin*

Function *argmin* was moved to *Monad_Memo_DP.argmin*. It iterates over a list and returns the rightmost element that minimizes a given function:

```
argmin ?f (?x # ?xs) =
(if ?xs = [] then ?x
 else let m = argmin ?f ?xs in if ?f ?x < ?f m then ?x else m)
```

An optimized version that avoids repeated computation of $f\ x$:

```
fun argmin2 :: ('a ⇒ ('b::linorder)) ⇒ 'a list ⇒ 'a * 'b where
argmin2 f (x#xs) =
  (let fx = f x
   in if xs = [] then (x, fx)
      else let mfm = argmin2 f xs
           in if fx < snd mfm then (x,fx) else mfm)
```

lemma *argmin2_argmin*: $xs \neq [] \Longrightarrow argmin2\ f\ xs = (argmin\ f\ xs,\ f(argmin\ f\ xs))$
⟨*proof*⟩

lemma *argmin_argmin2*[*code*]: $argmin\ f\ xs = (if\ xs = []\ then\ undefined\ else\ fst(argmin2\ f\ xs))$
⟨*proof*⟩

lemma *argmin_in*: $xs \neq [] \Longrightarrow argmin\ f\ xs \in set\ xs$
⟨*proof*⟩

lemma *argmin_pairs*: $xs \neq [] \Longrightarrow$
$(argmin\ f\ xs, f\ (argmin\ f\ xs)) = argmin\ snd\ (map\ (\lambda x.\ (x, f\ x))\ xs)$
⟨*proof*⟩

lemma *argmin_map*: $xs \neq [] \Longrightarrow argmin\ c\ (map\ f\ xs) = f(argmin\ (c\ o\ f)\ xs)$
⟨*proof*⟩

3.2 The 'Cubic' Algorithm

We hide the details of the access frequencies a and b by working with an abstract version of function w definied above (summing a and b). Later we interpret w accordingly.

locale *Optimal_BST* =
fixes $w :: int \Rightarrow int \Rightarrow nat$
begin

3.2.1 Functions *wpl* and *min_wpl*

sublocale *wpl* **where** $w = w$ ⟨*proof*⟩

Function *min_wpl i j* computes the minimal weighted path length of any tree t where *inorder* $t = [i..j]$. It simply tries all possible indices between i and j as the root. Thus it implicitly constructs all possible trees.

declare *conj_cong* [*fundef_cong*]
function *min_wpl* :: $int \Rightarrow int \Rightarrow nat$ **where**
min_wpl i j =
  (*if* $i > j$ *then* 0
   else Min ((λ*k*. *min_wpl i* (*k*−1) + *min_wpl* (*k*+1) *j*) ‘ {*i*..*j*}) + *w i j*)
⟨*proof*⟩
termination ⟨*proof*⟩
declare *min_wpl.simps*[*simp del*]

Note that for efficiency reasons we have pulled + *w i j* out of *Min*. In the lemma below this is reversed because it simplifies the proofs. Similar optimizations are possible in other functions below.

lemma *min_wpl_simps*[*simp*]:
  $i > j \Longrightarrow$ *min_wpl i j* = 0
  $i \leq j \Longrightarrow$ *min_wpl i j* =
    Min ((λ*k*. *min_wpl i* (*k*−1) + *min_wpl* (*k*+1) *j* + *w i j*) ‘ {*i*..*j*})
⟨*proof*⟩

lemma *upto_split1*:
  $[\![i \leq j;\ j \leq k]\!] \Longrightarrow [i..k] = [i..j-1]\ @\ [j..k]$
⟨*proof*⟩

Function *local.min_wpl* returns a lower bound for all possible BSTs:

theorem *min_wpl_is_optimal*:
  inorder $t = [i..j] \Longrightarrow$ *min_wpl i j* ≤ *wpl i j t*
⟨*proof*⟩

Now we show that the lower bound computed by *local.min_wpl* is the wpl of an optimal tree that can be computed in the same manner.

3.2.2 Function *opt_bst*

This is the functional equivalent of the standard cubic imperative algorithm. Unless it is memoized, the complexity is again exponential. The pattern of recursion is the same as for *local.min_wpl* but instead of the minimal weight it computes a tree with the minimal weight:

function *opt_bst* :: *int* ⇒ *int* ⇒ *int tree* **where**
opt_bst i j =
  (*if i* > *j then Leaf*
   else argmin (*wpl i j*) [⟨*opt_bst i* (*k*−1), *k*, *opt_bst* (*k*+1) *j*⟩. *k* ← [*i*..*j*]])
⟨*proof*⟩
termination ⟨*proof*⟩
declare *opt_bst.simps*[*simp del*]

corollary *opt_bst_simps*[*simp*]:
  i > *j* ⟹ *opt_bst i j* = *Leaf*
  i ≤ *j* ⟹ *opt_bst i j* =
    (*argmin* (*wpl i j*) [⟨*opt_bst i* (*k*−1), *k*, *opt_bst* (*k*+1) *j*⟩. *k* ← [*i*..*j*]])
⟨*proof*⟩

As promised, *local.opt_bst* computes a tree with the minimal wpl:

theorem *wpl_opt_bst*: *wpl i j* (*opt_bst i j*) = *min_wpl i j*
⟨*proof*⟩

corollary *opt_bst_is_optimal*:
  inorder t = [*i*..*j*] ⟹ *wpl i j* (*opt_bst i j*) ≤ *wpl i j t*
⟨*proof*⟩

3.2.3 Function *opt_bst_wpl*

Function *local.opt_bst* is simplistic because it computes the wpl of each tree anew rather than returning it with the tree. That is what *opt_bst_wpl* does:

function *opt_bst_wpl* :: *int* ⇒ *int* ⇒ *int tree* × *nat* **where**
opt_bst_wpl i j =
  (*if i* > *j then* (*Leaf*, 0)
   else argmin snd [*let* (*t*1,*c*1) = *opt_bst_wpl i* (*k*−1);
                          (*t*2,*c*2) = *opt_bst_wpl* (*k*+1) *j*
                      in (⟨*t*1,*k*,*t*2⟩, *c*1 + *c*2 + *w i j*). *k* ← [*i*..*j*]])
⟨*proof*⟩
termination
  ⟨*proof*⟩
declare *opt_bst_wpl.simps*[*simp del*]

Function *opt_bst_wpl* returns an optimal tree and its wpl:

lemma *opt_bst_wpl_eq_pair*:
  opt_bst_wpl i j = (*opt_bst i j*, *wpl i j* (*opt_bst i j*))
⟨*proof*⟩

corollary *opt_bst_wpl_eq_pair′*: *opt_bst_wpl i j* = (*opt_bst i j*, *min_wpl i j*)
⟨*proof*⟩

end

end

4 Quadrangle Inequality

theory *Quadrilateral_Inequality*
imports *Main*
begin

definition *is_arg_min_on* :: (*′a* ⇒ (*′b*::*linorder*)) ⇒ *′a set* ⇒ *′a* ⇒ *bool* **where**
is_arg_min_on f S x = (*x* ∈ *S* ∧ (∀ *y* ∈ *S*. *f x* ≤ *f y*))

definition *Args_min_on* :: (*int* ⇒ (*′b*::*linorder*)) ⇒ *int set* ⇒ *int set* **where**
Args_min_on f I = {*k*. *is_arg_min_on f I k*}

lemmas *Args_min_simps* = *Args_min_on_def is_arg_min_on_def*

lemma *is_arg_min_on_antimono*: **fixes** *f* :: _ ⇒ _::*order*
shows ⟦ *is_arg_min_on f S x*; *f y* ≤ *f x*; *y* ∈ *S* ⟧ ⟹ *is_arg_min_on f S y*
⟨*proof*⟩

lemma *ex_is_arg_min_on_if_finite*: **fixes** *f* :: *′a* ⇒ *′b* :: *linorder*
shows ⟦ *finite S*; *S* ≠ {} ⟧ ⟹ ∃ *x*. *is_arg_min_on f S x*
⟨*proof*⟩

locale *QI* =
  fixes *c_k* :: *int* ⇒ *int* ⇒ *int* ⇒ *nat*
  fixes *c* :: *int* ⇒ *int* ⇒ *nat*
  and *w* :: *int* ⇒ *int* ⇒ *nat*
  assumes *QI_w*: ⟦*i* ≤ *i′*; *i′* < *j*; *j* ≤ *j′*⟧ ⟹
    w i j + *w i′ j′*≤ *w i′ j* + *w i j′*
  assumes *monotone_w*: ⟦*i* ≤ *i′*; *i′* < *j*; *j* ≤ *j′*⟧ ⟹ *w i′ j* ≤ *w i j′*
  assumes *c_def*: *i* < *j* ⟹ *c i j* = *Min* ((*c_k i j*) ‘ {*i*+1..*j*})
  assumes *c_k_def*: ⟦ *i* < *j*; *k* ∈ {*i*+1..*j*} ⟧ ⟹
    c_k i j k = *w i j* + *c i* (*k*−1) + *c k j*
begin

abbreviation *mins i j* ≡ *Args_min_on* (*c_k i j*) {*i*+1..*j*}

definition *K i j* ≡ (*if i* = *j then i else Max* (*mins i j*))

lemma *c_def_rec*:
  i < *j* ⟹ *c i j* = *Min* ((λ*k*. *c i* (*k*−1) + *c k j* + *w i j*) ‘ {*i*+1..*j*})

⟨*proof*⟩

lemma *mins_subset*: $mins\ i\ j \subseteq \{i{+}1..j\}$
⟨*proof*⟩

lemma *mins_nonempty*: $i < j \Longrightarrow mins\ i\ j \neq \{\}$
⟨*proof*⟩

lemma *finite_mins*: *finite*(*mins i j*)
⟨*proof*⟩

lemma *is_arg_min_on_Min*:
assumes *finite A is_arg_min_on f A a* **shows** $Min\ (f\ `\ A) = f\ a$
⟨*proof*⟩

lemma *c_k_with_K*: $i < j \Longrightarrow c\ i\ j = c\_k\ i\ j\ (K\ i\ j)$
⟨*proof*⟩

lemma *K_subset*: **assumes** $i \leq j$ **shows** $K\ i\ j \in \{i..j\}$ ⟨*proof*⟩

lemma *lemma_2*:
  $[\![\ l = nat\ (j' - i);\ \ i \leq i';\ \ i' \leq j;\ \ j \leq j'\ ]\!]$
  $\Longrightarrow c\ i\ j + c\ i'\ j' \leq c\ i\ j' + c\ i'\ j$
⟨*proof*⟩

corollary *QI'*: **assumes** $i < k\ \ k \leq k'\ \ k' \leq j\ \ \ c\_k\ i\ j\ k' \leq c\_k\ i\ j\ k$
shows $c\_k\ i\ (j{+}1)\ k' \leq c\_k\ i\ (j{+}1)\ k$
⟨*proof*⟩

corollary *QI''*: **assumes** $i{+}1 < k\ k \leq k'\ k' \leq j{+}1\ c\_k\ i\ (j{+}1)\ k' \leq c\_k\ i\ (j{+}1)$
k
shows $c\_k\ (i{+}1)\ (j{+}1)\ k' \leq c\_k\ (i{+}1)\ (j{+}1)\ k$
⟨*proof*⟩

lemma *lemma_3_1*: **assumes** $i \leq j$ **shows** $K\ i\ j \leq K\ i\ (j{+}1)$
⟨*proof*⟩

lemma *lemma_3_2*: **assumes** $i \leq j$ **shows** $K\ i\ (j{+}1) \leq K\ (i{+}1)\ (j{+}1)$
⟨*proof*⟩

lemma *lemma_3*: **assumes** $i \leq j$
  shows $K\ i\ j \leq K\ i\ (j{+}1)\ K\ i\ (j{+}1) \leq K\ (i{+}1)\ (j{+}1)$
⟨*proof*⟩

end

end

5 Optimal BSTs: The 'Quadratic' Algorithm

theory *Optimal_BST2*
imports
Optimal_BST
Quadrilateral_Inequality
begin

Knuth presented an optimization of the previously known cubic dynamic programming algorithm to a quadratic one. A simplified proof of this optimization was found by Yao [4]. Mehlhorn follows Yao closely. The core of the optimization argument is given abstractly in theory *Optimal_BST.Quadrilateral_Inequality*. In addition we first need to establish some more properties of *argmin*.

An index-based specification of *argmin* expressing that the last minimal list-element is picked:

lemma *argmin_takes_last*: $xs \neq [] \Longrightarrow$
$argmin\ f\ xs = xs\ !\ Max\ \{i.\ i < length\ xs \wedge (\forall x \in set\ xs.\ f(xs!i) \leq f\ x)\}$
(**is** $\_ \Longrightarrow \_ = \_\ !\ Max\ (?M\ xs)$)
⟨*proof*⟩

lemma *Min_ex*: $[\![\ finite\ F;\ F \neq \{\}\ ]\!] \Longrightarrow \exists m \in F.\ \forall n \in F.\ m \leq (n::\_::linorder)$
⟨*proof*⟩

A consequence of *argmin_takes_last*:

lemma *argmin_Max_Args_min_on*: **assumes** [*arith*]: $i \leq j$
shows $argmin\ f\ [i..j] = Max\ (Args\_min\_on\ f\ \{i..j\})$
⟨*proof*⟩

As a consequence of *argmin_Max_Args_min_on* the following lemma allows us to justify the restriction of the index range of *argmin* used below in the optimized (quadratic) algorithm.

lemma *argmin_red_ivl*:
assumes $i \leq i'\ argmin\ f\ [i..j] \in \{i'..j'\}\ j' \leq j$
shows $argmin\ f\ [i'..j'] = argmin\ f\ [i..j]$
⟨*proof*⟩

fun $root:: 'a\ tree \Rightarrow 'a$ **where**
$root\ \langle\_,r,\_\rangle = r$

Now we can formulate and verify the improved algorithm. This requires two assumptions on the weight function *w*.

locale $Optimal\_BST2 = Optimal\_BST\ +$
assumes *monotone_w*: $[\![i \leq i';\ i' \leq j;\ j \leq j']\!] \Longrightarrow w\ i'\ j \leq w\ i\ j'$
assumes *QI_w*: $[\![i \leq i';\ i' \leq j;\ j \leq j']\!] \Longrightarrow w\ i\ j + w\ i'\ j' \leq w\ i'\ j + w\ i\ j'$
begin

When finding an optimal tree for $[i..j]$ the optimization consists in reducing the search for the root from $[i..j]$ to $[root\ (opt\_bst2\ i\ (j - (1::'b)))..root\ (opt\_bst2\ (i + (1::'a))\ j)]$:

function *opt_bst2* :: *int* ⇒ *int* ⇒ *int tree* **where**
opt_bst2 i j =
 (*if i* > *j then Leaf else*
 if i = *j then Node Leaf i Leaf else*
 let left = *root* (*opt_bst2 i* (*j*−1)) *in*
 let right= *root* (*opt_bst2* (*i*+1) *j*) *in*
 argmin (*wpl i j*) [⟨*opt_bst2 i* (*k*−1), *k*, *opt_bst2* (*k*+1) *j*⟩. *k* ← [*left*..*right*]])
⟨*proof*⟩

The termination of *opt_bst2* is not completely obvious. We first need to establish some functional properties of the terminating computations. We start by showing that the root of the returned tree is always between *left* and *right*. This is essentially equivalent to proving that *left* ≤ *right* because otherwise *argmin* is applied to [], which is undefined.

lemma *left_le_right*:
opt_bst2_dom(*i*,*j*) ⟹
 (*i*=*j* ⟶ *root*(*opt_bst2 i j*) = *i*) ∧
 (*i*<*j* ⟶ *root*(*opt_bst2 i j*) ∈ {*root*(*opt_bst2 i* (*j*−1)) .. *root*(*opt_bst2* (*i*+1) *j*)})
⟨*proof*⟩

Now we can bound the result of *opt_bst2* easily:

lemma *root_opt_bst2_bound*:
 opt_bst2_dom (*i*,*j*) ⟹ *i* ≤ *j* ⟹ *root* (*opt_bst2 i j*) ∈ {*i*..*j*}
⟨*proof*⟩

Now termination follows easily:

lemma *opt_bst2_dom*: ∀ *args*. *opt_bst2_dom args*
⟨*proof*⟩

termination ⟨*proof*⟩

declare *opt_bst2.simps*[*simp del*]

abbreviation *min_wpl3 i j k* ≡ *min_wpl i* (*k*−1) + *min_wpl* (*k*+1) *j* + *w i j*

The correctness proof [**?**] is based on a general theory of 'quatrilateral inequalities' developed in locale QI that we now instantiate:

interpretation *QI*
 where
 c = λ*i j*. *min_wpl* (*i*+1) *j*
 and *c_k* = λ*i j*. *min_wpl3* (*i*+1) *j*
 and *w* = λ*i j*. *w* (*i*+1) *j*
⟨*proof*⟩

lemma *K_argmin*: *i* < *j* ⟹ *K i j* = *argmin* (*min_wpl3* (*i*+1) *j*) [*i*+1..*j*]

⟨*proof*⟩

theorem *opt_bst2_opt_bst*: *opt_bst2 i j* = *opt_bst i j*
⟨*proof*⟩

corollary *opt_bst2_is_optimal*: *wpl i j* (*opt_bst2 i j*) = *min_wpl i j*
⟨*proof*⟩

function *opt_bst_wpl2* :: *int* ⇒ *int* ⇒ *int tree* × *nat* **where**
opt_bst_wpl2 i j =
  (*if i* > *j then* (*Leaf*,0) *else*
   if i = *j then* (*Node Leaf i Leaf*, *w i i*) *else*
   let l = *root*(*fst*(*opt_bst_wpl2 i* (*j*−1)));
      r = *root*(*fst*(*opt_bst_wpl2* (*i*+1) *j*)) *in*
     argmin snd
       [*let* (*tl*,*wl*) = *opt_bst_wpl2 i* (*k*−1); (*tr*,*wr*) = *opt_bst_wpl2* (*k*+1) *j*
        in (⟨*tl*, *k*, *tr*⟩, *wl* + *wr* + *w i j*) . *k* ← [*l*..*r*]])
⟨*proof*⟩

lemma *left_le_right2*:
 opt_bst_wpl2_dom(*i*,*j*) ⟹
  (*i*=*j* ⟶ *root*(*fst*(*opt_bst_wpl2 i j*)) = *i*) ∧
  (*i*<*j* ⟶ *root*(*fst*(*opt_bst_wpl2 i j*)) ∈
    {*root*(*fst*(*opt_bst_wpl2 i* (*j*−1))) .. *root*(*fst*(*opt_bst_wpl2* (*i*+1) *j*))})
⟨*proof*⟩

Now we can bound the result of *opt_bst_wpl2*:

lemma *root_opt_bst_wpl2_bound*:
  opt_bst_wpl2_dom (*i*,*j*) ⟹ *i* ≤ *j* ⟹ *root* (*fst*(*opt_bst_wpl2 i j*)) ∈ {*i*..*j*}
⟨*proof*⟩

Now termination follows easily:

lemma *opt_bst_wpl2_dom*: ∀ *args*. *opt_bst_wpl2_dom args*
⟨*proof*⟩

termination ⟨*proof*⟩

declare *opt_bst_wpl2*.*simps*[*simp del*]

lemma *opt_bst_wpl2_eq_pair*:
  opt_bst_wpl2 i j = (*opt_bst2 i j*, *wpl i j* (*opt_bst2 i j*))
⟨*proof*⟩

corollary *opt_bst_wpl2_eq_pair′*: *opt_bst_wpl2 i j* = (*opt_bst i j*, *min_wpl i j*)
⟨*proof*⟩

end

end

theory *Optimal_BST_Examples*
imports *HOL−Library.Tree*
begin

Example by Mehlhorn:

definition *a_ex*1 :: *int* ⇒ *nat* **where**
*a_ex*1 *i* = [4,0,0,3,10] ! *nat i*

definition *b_ex*1 :: *int* ⇒ *nat* **where**
*b_ex*1 *i* = [1,3,3,0] ! *nat i*

definition *t_opt_ex*1 :: *int tree* **where**
*t_opt_ex*1 = ⟨⟨⟨⟩, 0, ⟨⟨⟩, 1, ⟨⟩⟩⟩, 2, ⟨⟨⟩, 3, ⟨⟩⟩⟩

Example by Knuth:

definition *a_ex*2 :: *int* ⇒ *nat* **where**
*a_ex*2 *i* = 0

definition *b_ex*2 :: *int* ⇒ *nat* **where**
*b_ex*2 *i* = [32,7,69,13,6,15,10,8,64,142,22,79,18,9] ! *nat i*

definition *t_opt_ex*2 :: *int tree* **where**

```
t_opt_ex2 =
 ⟨
  ⟨
   ⟨⟨⟩, 0, ⟨⟨⟩, 1, ⟨⟩⟩⟩,
   2,
   ⟨
    ⟨
     ⟨⟨⟩, 3, ⟨⟨⟩, 4, ⟨⟩⟩⟩,
     5,
     ⟨⟨⟩, 6, ⟨⟨⟩, 7, ⟨⟩⟩⟩
    ⟩,
    8,
    ⟨⟩
   ⟩
  ⟩,
  9,
  ⟨⟨⟨⟩, 10, ⟨⟩⟩,
   11,
   ⟨⟨⟩, 12, ⟨⟨⟩, 13, ⟨⟩⟩
   ⟩
  ⟩
 ⟩
```

end

6 Code Generation (unmemoized)

theory *Optimal_BST_Code*
imports
  Optimal_BST2
  Optimal_BST_Examples
begin

locale *Wpl_Optimal_BST* = *Wpl a b* + *Optimal_BST* **where** *w* = *Wpl.w a b*
for *a b*

locale *Wpl_Optimal_BST2* = *Wpl a b* + *Optimal_BST2* **where** *w* = *Wpl.w a b*
for *a b*

global_interpretation *Wpl_Optimal_BST* + *Wpl_Optimal_BST2*
defines *wpl_ab* = *wpl* **and** *opt_bst_ab* = *opt_bst* **and** *opt_bst2_ab* = *opt_bst2*
⟨*proof*⟩

Examples:

lemma *opt_bst_ab a_ex1 b_ex1* 0 3 = *t_opt_ex1*
⟨*proof*⟩

lemma *opt_bst2_ab a_ex2 b_ex2* 0 13 = *t_opt_ex2*
⟨*proof*⟩

end

7 Memoization

theory *Optimal_BST_Memo*
imports
  Optimal_BST
  Monad_Memo_DP.State_Main
  HOL−Library.Product_Lexorder
  HOL−Library.RBT_Mapping
  Optimal_BST_Examples
begin

This theory memoizes the recursive algorithms with the help of our generic memoization framework. Note that currently only the tree building (function *Optimal_BST.opt_bst*) is memoized but not the computation of *w*.

global_interpretation *Wpl*
where *a* = *a* **and** *b* = *b* **for** *a b*
defines *w_ab* = *w* **and** *wpl_ab* = *wpl.wpl w_ab* ⟨*proof*⟩

First we express *argmin* via *fold*. Primarily because we have a monadic version of *fold* already. At the same time we improve efficiency.

lemma *fold_argmin*: $fold\ (\lambda x\ (m,fm).\ let\ fx = f\ x\ in\ if\ fx \leq fm\ then\ (x,fx)\ else\ (m,fm))\ xs\ (x,f\ x)$
$= (argmin\ f\ (x\#xs),\ f(argmin\ f\ (x\#xs)))$
⟨*proof*⟩

lemma *argmin_fold*: $argmin\ f\ xs = (case\ xs\ of\ [] \Rightarrow undefined\ |$
$x\#xs \Rightarrow fst(fold\ (\lambda x\ (m,fm).\ let\ fx = f\ x\ in\ if\ fx \leq fm\ then\ (x,fx)\ else\ (m,fm))\ xs\ (x,f\ x)))$
⟨*proof*⟩

The actual memoization of the cubic algorithm:

context *Optimal_BST*
begin

memoize_fun opt_bst_m: *opt_bst* **with_memory** *dp_consistency_mapping*
monadifies (*state*) *opt_bst.simps*[*unfolded argmin_fold*]

thm $opt\_bst_m{}'.simps$

memoize_correct
⟨*proof*⟩

lemmas [*code*] = $opt\_bst_m.memoized\_correct$

end

Code generation:

global_interpretation *Optimal_BST*
where $w = w\_ab\ a\ b$
rewrites $wpl.wpl\ (w\_ab\ a\ b) = wpl\_ab\ a\ b$ **for** $a\ b$
defines $opt\_bst\_ab = opt\_bst$ **and** $opt\_bst\_ab' = opt\_bst_m{}'$
⟨*proof*⟩

Examples:

lemma $opt\_bst\_ab\ a\_ex1\ b\_ex1\ 0\ 3 = t\_opt\_ex1$
⟨*proof*⟩

lemma $opt\_bst\_ab\ a\_ex2\ b\_ex2\ 0\ 13 = t\_opt\_ex2$
⟨*proof*⟩

end

References

[1] D. E. Knuth. Optimum binary search trees. *Acta Inf.*, 1:14–25, 1971.

[2] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*, volume 1 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1984.

[3] S. Wimmer, S. Hu, and T. Nipkow. Monadification, memoization and dynamic programming. *Archive of Formal Proofs*, 2018. http://isa-afp.org/entries/Monad_Memo_DP.html, Formal proof development.

[4] F. F. Yao. Efficient dynamic programming using quadrangle inequalities. In *STOC*, pages 429–435. ACM, 1980.