```
::::::::::::::
run.sh
::::::::::::::
#!/bin/bash
#-------------------------------------------------------------------------------
#  File Name:
#     run.sh
#
#  Purpose:
#
#     bash run file
#
#  Parameters:
#
#
#  Discussion:
#
#
#  Licensing:
#
#    This code is distributed under the GNU LGPL license.
#
#  Modified:
#
#    2018.12.05 Wed
#
#  Author:
#
#    Young Won Lim
#
#-----------------------------------------------------------------------------

# bash -x run.sh

fname=binary_search
dname=~/Work/CORDIC/1.binary_tree_search

echo on

cd $dname

make binary_search N=10 DISP=1

cd ~/

./$fname 3  |tee $fname.log

enscript -o - $fname.log | ps2pdf - $fname.log.pdf

pdfunite binary_tree_*.pdf $fname.log.pdf $fname.out.pdf

cp $fname.out.pdf $dname/output


# for i in $(seq 1 5 ); do
#   ./binary_search 1  |tee binary_search_i_$i.out
# done

::::::::::::::
library.sh
::::::::::::::
#!/bin/bash
#-------------------------------------------------------------------------------
#  File Name:
#     library.sh
#
#  Purpose:
#
```

```
#     bash build a library file
#
#  Parameters:
#
#
#  Discussion:
#
#
#  Licensing:
#
#    This code is distributed under the GNU LGPL license.
#
#  Modified:
#
#    2018.12.19 Wed
#
#  Author:
#
#    Young Won Lim
#
#-----------------------------------------------------------------------------

# bash -x run.sh

fname=binary_search
dname=~/Work/CORDIC/1.binary_tree_search

echo on

cd $dname

make binary_library N=10 DISP=0



# for i in $(seq 1 5 ); do
#   ./binary_search 1  |tee binary_search_i_$i.out
# done

::::::::::::::
Makefile
::::::::::::::
#------------------------------------------------------------------------------
#  File Name:
#     Makefile
#
#  Purpose:
#
#     makefile for binary_search
#
#  Parameters:
#
#
#  Discussion:
#
#
#  Licensing:
#
#    This code is distributed under the GNU LGPL license.
#
#  Modified:
#
#    2018.12.05 Wed
#
#  Author:
#
#    Young Won Lim
#
```

```
#-----------------------------------------------------------------------------
#  make binary_search N=10 DISP=1
#  make binary_library N=10 DISP=0

CC=gcc
CFLAGS=-Wall
MACROS=-DN=$(N) -DDISP=$(DISP)
LIBS=-lm

DEPS = binary1_search_defs.h
SRC0 = binary2_search_defs.c \
       binary3_traverse.c \
       binary4_level.c \
       binary5_path.c \
       binary6_cordic.c \
       binary7_subtree.c \
       binary8_plot.c

SRCS = $(SRC0) binary9_main.c

OBJ0 = $(SRC0:.c=.o)
OBJS = $(SRCS:.c=.o)

PRNS = run.sh library.sh Makefile $(DEPS) $(SRCS)

# FNAME = ./print/binary_search.$(shell date +%Y%m%d).c
FNAME = ./print/binary_search.c


.SUFFIXES : .o .c .cpp

.c.o : $(DEPS)
        $(CC) -c $(CFLAGS) $(MACROS) -o $@ $<

binary_search: $(OBJS)
        $(CC) $(CFLAGS) -o ~/binary_search $^  $(LIBS)
        rm -f *.o *~ core

binary_library: $(OBJ0)
        ls libbinary.a && rm libbinary.a
        ar rcs libbinary.a $(OBJ0)
        cp libbinary.a ../5.testbench
        rm -f *.o *~ core


print: run.sh Makefile $(DEPS) $(SRCS)
        /bin/more $(PRNS) > $(FNAME)
        enscript -o - --highlight=c $(FNAME) | ps2pdf - $(FNAME).pdf


clean:
        rm -f *.o *~ core

::::::::::::::
binary1_search_defs.h
::::::::::::::
//------------------------------------------------------------------------------
//  File Name:
//     binary1_search_defs.h
//
//  Purpose:
//
//     Definitions and macros
//
//  Parameters:
//
//
//  Discussion:
```

```
//
//
//  Licensing:
//
//    This code is distributed under the GNU LGPL license.
//
//  Modified:
//
//    2018.12.05 Wed
//
//  Author:
//
//    Young Won Lim
//
//------------------------------------------------------------------------------
// #define N 8     // the depth of a binary tree
#define R 2      // the number of expanding choices = R=2
#define PRE  "/home/young/Data/"
#define TREE "binary_tree"
//----------------------------------------------------------
// (R)-ary tree node
// 1st R choices -a(i) at the step i  // 0
// 2nd R choices +a(i) at the step i  // 1
//----------------------------------------------------------
// for the file IO in an R script, arrange members
// that leaves no hole in memory
//----------------------------------------------------------
typedef struct node {
  double theta;                 // input angle to the i-th step
  int    branch;                // denotes which child of the parent
  int    depth;                 // denotes the i-th step computation
  int    id;                    // serial number for expand nodes

  int    child[R];              // pointers to the 2 children
  int    parent;                // pointers to the parent
} nodetype;



//----------------------------------------------------------
// queue node type
// used for breadth first search traversal
//----------------------------------------------------------
typedef struct qnode {
  struct  node * node;          // angle tree node
  struct qnode * next;          // queue node
} qnodetype;


//--- binary2.search_defs.c --------------------------------
nodetype * create_node();
qnodetype * create_qnode();

//--- binary3.traverse.c -----------------------------------
void pr_node(nodetype *p);
void copy_node(nodetype *p, nodetype *q);
void expand_node(nodetype *p, int rid);
void tree_traverse(nodetype *p);

//--- binary4.level.c --------------------------------------
void print_level_nodes(int depth);
nodetype find_level_min_node(int depth, int flag);
nodetype find_global_min_node();

//--- binary5.path.c ---------------------------------------
qnodetype* find_path(nodetype *p);
void print_path(qnodetype *q, char *str);
void delete_path(qnodetype* q, char *str);
```

```
//--- binary6.cordic.c -----------------------------------
nodetype* cordic_expand(nodetype *p, int rid);
qnodetype* cordic_traverse(nodetype *p);
qnodetype *find_cordic_path(nodetype *p);
nodetype find_cordic_node(nodetype *p);

//--- binary7.subtree.c -----------------------------------
void write_subtree_leaves(int depth_leaf, int depth_root);
void read_subtree_leaves(int depth_leaf, int depth_root);
void write_subtree_nodes(int depth_root, int class, int depth_leaf);
void read_subtree_nodes(int depth_root, int class, int depth_leaf);

//--- binary8.plot.c --------------------------------------
void plot_path(qnodetype *q, char *str);


//-----------------------------------------------------------
// Global Variables
//-----------------------------------------------------------
typedef struct param {
  int NN;  // the depth/height of a binary tree
  int RR;  // R=2 : binary tree
  double theta;

  char tstring[256];

} paramtype;

paramtype Param;

double a[2*N];  // because of quaternary search tree



::::::::::::::
binary2_search_defs.c
::::::::::::::
//-------------------------------------------------------------------------------
//  File Name:
//     binary2_search_defs.c
//
//  Purpose:
//
//     create node and qnode
//
//  Parameters:
//
//
//  Discussion:
//
//
//  Licensing:
//
//    This code is distributed under the GNU LGPL license.
//
//  Modified:
//
//    2018.12.05 Wed
//
//  Author:
//
//    Young Won Lim
//
//-----------------------------------------------------------------------------
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
```

```
#include "binary1_search_defs.h"


//---------------------------------------------------------------
// create a node for an angle tree
//---------------------------------------------------------------
nodetype * create_node() {
  nodetype * p = (nodetype *) malloc (sizeof(nodetype));

  if (p == NULL) {
    perror("node creation error \n");
    exit(1);
  }
  else {
    return p;
  }
}

//---------------------------------------------------------------
// create a node for a queue
//---------------------------------------------------------------
qnodetype * create_qnode() {

  qnodetype * q = (qnodetype *) malloc (sizeof(qnodetype));

  if (q == NULL) {
    perror("qnode creation error \n");
    exit(1);
  }
  else {
    return q;
  }
}





::::::::::::::
binary3_traverse.c
::::::::::::::
//-----------------------------------------------------------------------------
//  File Name:
//     binary3_traverse.c
//
//  Purpose:
//
//     tree traverse and expanding a node
//
//  Parameters:
//
//
//  Discussion:
//
//
//  Licensing:
//
//    This code is distributed under the GNU LGPL license.
//
//  Modified:
//
//    2018.12.05 Wed
//
//  Author:
//
//    Young Won Lim
//
```

```
//---------------------------------------------------------------------------
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

#include "binary1_search_defs.h"


FILE *fp_r;  // read file pointer
FILE *fp_w;  // write file pointer

//---------------------------------------------------------------
// print node record information
//---------------------------------------------------------------
void pr_node(nodetype *p) {
  int i;

  printf("id=%d pa=%d ch=[", p->id, p->parent);
  for (i=0; i<R; ++i) printf("%d ", p->child[i]);
  printf("] th=%f br=%d dp=%d ", p->theta, p->branch, p->depth);
  printf("\n");
}

//---------------------------------------------------------------
// copy a node structure (p <- q)
//---------------------------------------------------------------
void copy_node(nodetype *p, nodetype *q) {
   int i;

   p->theta  = q->theta;
   p->branch = q->branch;
   p->depth  = q->depth;
   p->id     = q->id;
   p->parent = q->parent;
   for (i=0; i<R; ++i)
     p->child[i] = q->child[i];
}


//---------------------------------------------------------------
// expand R children nodes of a current node p
//---------------------------------------------------------------
void expand_node(nodetype *p, int rid) {
  nodetype c;            // child node
  int i, j, depth;
  double ntheta, theta;  // new theta is computed from theta
  static int id = 1;     // id counter

  // printf("* expanding a node... \n");

  if (rid) id = 1;       // reset id counter to 1

  theta = p->theta;
  depth = p->depth;

  for (i=0; i<R; ++i) {
    if      (i < (R-1))   ntheta = theta - 1 * a[depth];
    else if (i == (R-1))  ntheta = theta + 1 * a[depth];

    // printf("%d %f =(%f %f) \n", i, ntheta, theta, a[i]);

    c.parent     = p->id;
    c.theta      = ntheta;
    c.depth      = p->depth +1;
    c.branch     = i;
    c.id         = id++;
    for (j=0; j<R; ++j) c.child[j] = 0;
    p->child[i]  = c.id;

    fwrite(&c, sizeof(c), 1, fp_w);    // write a child node
  }

  fseek(fp_r, -sizeof(*p), SEEK_CUR);   // move the file pointer backward
  fwrite(p, sizeof(*p), 1, fp_r);       // overwwrite the parent node

  // printf("* end of expand\n");
}


//---------------------------------------------------------------
// BFS Tree Traversal - level by level
//---------------------------------------------------------------
void tree_traverse(nodetype *r) {
  nodetype p;
  int depth, rid;

  char fname_r[64];
  char fname_w[64];

  // printf("* tree traversing ... \n");

  sprintf(fname_w, "%s%s_L%02d.dat", PRE, TREE, 0);
  fp_w = fopen(fname_w, "w");
  fwrite(r, sizeof(*r), 1, fp_w);    // write root node r
  fclose(fp_w);


  for (depth=0; depth<N; ++depth) {

    // printf("* depth= %d \n", depth);

    sprintf(fname_r, "%s%s_L%02d.dat", PRE, TREE, depth);
    sprintf(fname_w, "%s%s_L%02d.dat", PRE, TREE, depth+1);

    fp_r = fopen(fname_r, "r+");
    fp_w = fopen(fname_w, "w");

    while (fread(&p, sizeof(p), 1, fp_r) != 0) {
      rid = !depth;
      expand_node(&p, rid);
    }

    fclose(fp_r);
    fclose(fp_w);
  }

  // printf("* end of tree traversing ... \n");
}

::::::::::::::
binary4_level.c
::::::::::::::
//-----------------------------------------------------------------------------
//  File Name:
//     binary4_level.c
//
//  Purpose:
//
//     find the minimum cost leaf node
//
//  Parameters:
//
//
//  Discussion:
//
```

```
//
//  Licensing:
//
//    This code is distributed under the GNU LGPL license.
//
//  Modified:
//
//    2018.12.05 Wed
//
//  Author:
//
//    Young Won Lim
//
//-----------------------------------------------------------------------------
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

#include "binary1_search_defs.h"


//---------------------------------------------------------------
// print all the nodes of the given level
//---------------------------------------------------------------
void print_level_nodes(int depth) {
  FILE *fp;
  char fname[64];
  nodetype p;
  int i;

  printf("* print %d level node \n", depth);

  sprintf(fname, "%s%s_L%02d.dat", PRE, TREE, depth);

  fp = fopen(fname, "rb");

  while (fread(&p, sizeof(p), 1, fp) != 0) {
    printf(" %-5d %+f (Level %2d) ", p.id, p.theta, depth);
    printf("child: ");
    for (i=0; i<R; ++i) printf("%2d ", p.child[i]);
    printf("parent: %2d ", p.parent);
    printf("\n");
  }
  printf("------------------------------\n");

  fclose(fp);

}

//---------------------------------------------------------------
// find a node having the min residue at the given level
//---------------------------------------------------------------
nodetype find_level_min_node(int depth, int flag) {
  nodetype p, p_min;
  double minval = 1e100;
  double residue;

  FILE *fp;
  char fname[64];

  sprintf(fname, "%s%s_L%02d.dat", PRE, TREE, depth);

  fp = fopen(fname, "rb");

  while (fread(&p, sizeof(p), 1, fp) != 0) {
    residue = fabs(p.theta);
    if (minval > residue) {
      minval = residue;
      p_min = p;
    }
  }

  if (flag) printf("* leaf min node : ");
  else printf("level min node : ");
  printf("depth=%3d ", depth);
  printf("theta=%+14.6e ", p_min.theta);
  // printf("minval=%+14.6e ", minval);
  printf("id=%d \n", p_min.id);

  fclose(fp);

  return(p_min);
}


//-----------------------------------------------------------------
// find the node with the globally min residue angle
//-----------------------------------------------------------------
nodetype find_global_min_node() {
  nodetype p, p_min;
  double minval = 1e100;
  double residue;
  int i, i_min;

  for (i=0; i<N; ++i) {      // over all depths
     p = find_level_min_node(i, 0);
     residue = fabs(p.theta);
     if (minval > residue) {
       minval = residue;
       p_min = p;
       i_min = i;
     }
  }

  printf("\n* global min node : ");
  printf("depth=%3d ", i_min);
  printf("theta=%+14.6e ", p_min.theta);
  // printf("minval=%+14.6e ", minval);
  printf("id=%d \n", p_min.id);

  return(p_min);
}

//-----------------------------------------------------------------
// sorting residue angles at the given level
//-----------------------------------------------------------------
// void sort_level_nodes(int depth) { T.B.D.

::::::::::::::
binary5_path.c
::::::::::::::
//--------------------------------------------------------------------------------
//  File Name:
//     binary5_path.c
//
//  Purpose:
//
//     find and print the optimal path
//
//  Parameters:
//
//
//  Discussion:
//
//
//  Licensing:
```

```
//
//    This code is distributed under the GNU LGPL license.
//
//  Modified:
//
//    2018.12.05 Wed
//
//  Author:
//
//    Young Won Lim
//
//---------------------------------------------------------------------------
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

#include "binary1_search_defs.h"


//---------------------------------------------------------------
// find a path from the root to a node p
//---------------------------------------------------------------
qnodetype *find_path(nodetype *p) {
  qnodetype *q, *path;
  int depth, pid;
  FILE *fp;
  char fname[64];

  // printf("* find a path from the root to the given node \n");

  path = NULL;

  depth = p->depth;  // depth of a given node

  while (depth >= 0) {

    q = create_qnode();
    q->next = path;
    q->node = p;
    path = q;

    pid = p->parent;

    p = create_node();

    depth--;

    if (depth < 0) break;

    sprintf(fname, "%s%s_L%02d.dat", PRE, TREE, depth);

    fp = fopen(fname, "rb");

    fread(p, sizeof(*p), 1, fp);

    if (p->id != pid) {
      fseek(fp, (pid - p->id -1)*sizeof(*p), SEEK_CUR);
      fread(p, sizeof(*p), 1, fp);
    }

    fclose(fp);
  }

  // printf("* end of find optimal path \n");
  return(path);

}
```

```
//-----------------------------------------------------------------
// print nodes in a path from root to node
//-----------------------------------------------------------------
void print_path(qnodetype* q, char *str) {
  int u, d;

  // printf("* print the found path\n");

  printf("\npath type : %s \n", str);

  while (q) {
    printf("dp=%2d ", (q->node)->depth);
    printf("th=%-+12.6g ", (q->node)->theta);
    printf("%+16.10e ", (q->node)->theta);

    d = (q->node)->depth;

    q = q->next;

    if (q == NULL) {
      printf("\n");
      break;
    }
    printf("br=%2d ", (q->node)->branch);

    if      ((q->node)->branch <  (R-1))    u = +1;  // ==0
    else if ((q->node)->branch == (R-1))    u = -1;  // ==1

    if (0) {
      printf("-u=%+2d ", -u);
      printf("a[%2d]=%10.6f ", d, a[d]);
      printf("\n");
    } else {
      printf(" :");
      printf(" %10.6f", -a[d]);
      printf(" %10.6f", +a[d]);
      printf("\n");

    }

  }

}


//-----------------------------------------------------------------
// deallocate node in a given path
//-----------------------------------------------------------------
void delete_path(qnodetype* q, char* str) {
  qnodetype* t;

  // printf("* deallocate nodes in the %s path \n", str);

  while (q) {
    t = q->next;

    if (t == NULL) break;

    free(q->node);
    free(q);

    q = t;
  }
}
```

```
::::::::::::::
binary6_cordic.c
::::::::::::::
//------------------------------------------------------------------------------
//  File Name:
//     binary6_cordic.c
//
//  Purpose:
//
//     finding the cordic path
//
//  Parameters:
//
//
//  Discussion:
//
//
//  Licensing:
//
//    This code is distributed under the GNU LGPL license.
//
//  Modified:
//
//    2018.12.05 Wed
//
//  Author:
//
//    Young Won Lim
//
//-----------------------------------------------------------------------------
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

#include "binary1_search_defs.h"

//------------------------------------------------------------------
// node id is newly created
//------------------------------------------------------------------
// nodetype* cordic_expand(nodetype *p, int rid);
// qnodetype* cordic_traverse(nodetype *p);
//------------------------------------------------------------------
// node id in the search tree is reused
//------------------------------------------------------------------
// qnodetype *find_cordic_path(nodetype *p);
// nodetype find_cordic_node(nodetype *p);
//------------------------------------------------------------------


//-----------------------------------------------------------------
// create (R) children node to the current node pointed by p
//-----------------------------------------------------------------
nodetype* cordic_expand(nodetype *p, int rid) {
  nodetype *np;
  int i, depth, mindex=0;
  double ntheta[R], theta, minval=1E+10;
  static int id = 1;

  // printf("* cordic node... \n");

  if (rid) id = 1;          // reset the id counter

  theta = p->theta;
  depth = p->depth;

  for (i=0; i<R; ++i) {
    if      (i < (R-1))   ntheta[i] = theta - 1 * a[depth];
    else if (i == (R-1))  ntheta[i] = theta + 1 * a[depth];
```

```c
  }

  for (i=0; i<R; ++i) {
    if (minval > fabs(ntheta[i])) {
      minval = fabs(ntheta[i]);
      mindex = i;
    }
  }

  // printf("%d %f =(%f %f) \n", mindex, ntheta[mindex], theta, a[depth]);


  np = create_node ();
   p->child[mindex] = id;
  np->parent        = p->id;
  np->theta         = ntheta[mindex];
  np->depth         = p->depth +1;
  np->branch        = mindex;
  np->id            = id++;


  //-- if (ntheta > theta) np->branch = -1;

   return np;
}




//---------------------------------------------------------------
// CORDIC Traversal
//---------------------------------------------------------------
qnodetype* cordic_traverse(nodetype *p) {
  qnodetype *cordic_path=NULL;             // CORDIC Queue Head
  qnodetype *cordic_tail=NULL;             // CORDIC Queue Tail
  qnodetype *q, *nq;
  nodetype *np;
  int k =0, rid;

  // printf("* cordic traversing ... \n");

  q = create_qnode();
  q->node = p;

  cordic_path = q;
  cordic_tail = q;

  while (cordic_tail != NULL) {
    // printf("* node %d to be expanded \n", k);

    rid = k ? 0 : 1;        // reset id

    k++;

    if ((q->node)->depth >= (N-1) ) {
      cordic_tail->next = NULL;

      printf("* find level %d cordic node : ", np->depth);
      printf("theta=%10.6f ", np->theta);
      printf("id=%d \n", np->id);

      break;
    }

    if (q != NULL) np = cordic_expand(q->node, rid);

    nq = create_qnode();
    nq->node = np;
    cordic_tail->next = nq;
    cordic_tail = nq;

    q = nq;
  }

  return (cordic_path);
}


//------------------------------------------------------------------
// find a cordic path from any node p to a cordic leaf node
//------------------------------------------------------------------
qnodetype *find_cordic_path(nodetype *p) {
  nodetype c[R];
  qnodetype *q;
  qnodetype *path=NULL;             // CORDIC Queue Head
  qnodetype *tail=NULL;             // CORDIC Queue Tail
  int depth, cid, i, mindex;
  double minval=1E+10;
  FILE *fp;
  char fname[64];

  // printf("* find a cordic node at the given depth \n");


  depth = p->depth;
  // pr_node(p);

  q = create_qnode();
  q->node = p;

  path = q;
  tail = q;

  while (depth < N-1) {

    cid = p->child[0];

    //.......................................................
    sprintf(fname, "%s%s_L%02d.dat", PRE, TREE, depth+1);

    fp = fopen(fname, "rb");

    fread(c, sizeof(*p), 1, fp);

    if (c[0].id != cid) {
      fseek(fp, (cid - c[0].id -1)*sizeof(*p), SEEK_CUR);
      fread(c, sizeof(*p), 1, fp);
    }

    for (i=1; i<R; ++i) {
      fread(c+i, sizeof(*p), 1, fp);
    }

    fclose(fp);
    //.......................................................

    minval = 1E+10;
    for (i=0; i<R; ++i) {
      if (minval > fabs(c[i].theta)) {
        minval = fabs(c[i].theta);
        mindex = i;
      }
    }

    p = create_node();
    copy_node(p, &c[mindex]);

    depth = p->depth;


    q = create_qnode();
    q->node = p;
    q->next = NULL;

    tail->next = q;
    tail = q;

  }

  printf("cordic min node : depth=%3d ", depth);
  printf("theta=%10.6f ", p->theta);
  printf("minval=%10.6f ", minval);
  printf("id=%d \n", cid);


  //  printf("* end of find a cordic path \n");
  return(path);

}


//-----------------------------------------------------------------
// find a cordic leaf node only
//-----------------------------------------------------------------
nodetype find_cordic_node(nodetype *p) {
  nodetype c[R], np;
  int depth, cid, i, mindex;
  double minval=1E+10;
  FILE *fp;
  char fname[64];

  // printf("* find a cordic node at the given depth \n");

  copy_node(&np , p);

  depth = np.depth;


  while (depth < N-1) {

    cid = np.child[0];


    //.....................................................
    sprintf(fname, "%s%s_L%02d.dat", PRE, TREE, depth+1);

    fp = fopen(fname, "rb");

    fread(c, sizeof(np), 1, fp);

    if (c[0].id != cid) {
      fseek(fp, (cid - c[0].id -1)*sizeof(np), SEEK_CUR);
      fread(c, sizeof(np), 1, fp);
    }

    for (i=1; i<R; ++i) {
      fread(c+i, sizeof(np), 1, fp);
    }

    fclose(fp);
    //.....................................................
    minval = 1E+10;
    for (i=0; i<R; ++i) {
      if (minval > fabs(c[i].theta)) {
        minval = fabs(c[i].theta);
        mindex = i;
      }
    }

    copy_node(&np, &c[mindex]);

    depth = np.depth;

  }

  printf("* cordic min node : depth=%3d ", depth);
  printf("theta=%+14.6e ", c[mindex].theta);
  // printf("minval=%+14.6e ", minval);
  printf("id=%d \n", cid);


  //  printf("* end of find a cordic path \n");

  return(np);

}




::::::::::::::
binary7_subtree.c
::::::::::::::
//-------------------------------------------------------------------------------
//  File Name:
//     binary7_subtee.c
//
//  Purpose:
//
//     read / write subtrees and their leaf nodes
//
//  Parameters:
//
//
//  Discussion:
//
//
//  Licensing:
//
//    This code is distributed under the GNU LGPL license.
//
//  Modified:
//
//    2018.12.05 Wed
//
//  Author:
//
//    Young Won Lim
//
//-----------------------------------------------------------------------------
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

#include "binary1_search_defs.h"



//-----------------------------------------------------------------
```

```
// write all classified leaf nodes
//---------------------------------------------------------------
void write_subtree_leaves(int depth_root, int depth_leaf) {
  nodetype p;
  int cnum;   // the number of classes at depth_root
  int lnum;   // the number of leaves per each class at depth_leaf
  int i, j, cnt;

  FILE *fp1;  // read file pointer
  FILE *fp2;  // write file pointer

  char fname1[64];
  char fname2[64];


  cnum = (int) pow(R, depth_root);          // no of classes
  lnum = (int) pow(R, depth_leaf) / cnum;   // no of leaves per class


  sprintf(fname1, "%s%s_L%02d.dat", PRE, TREE, depth_leaf);
  fp1 = fopen(fname1, "r");

  for (i=0; i<cnum; i++) {
    sprintf(fname2, "%s%s_L%02d.G%02d.dat", PRE, TREE, depth_leaf, i);
    fp2 = fopen(fname2, "w");

    for (j=0; j<lnum; j++) {
       cnt = fread(&p, sizeof(p), 1, fp1);
       if (cnt == 0) {
         perror("* error in reading file ...\n");
         exit(1);
       }
       fwrite(&p, sizeof(p), 1, fp2);
    }

    fclose(fp2);
  }

  fclose(fp1);

}

//---------------------------------------------------------------
// read all classified leaf nodes
//---------------------------------------------------------------
void read_subtree_leaves(int depth_root, int depth_leaf) {
  nodetype p;
  int cnum;   // the number of classes at depth_root
  int lnum;   // the number of leaves per each class at depth_leaf
  int i, j;

  FILE *fp2;  // write file pointer

  char fname2[64];


  cnum = (int) pow(R, depth_root);          // no of classes
  lnum = (int) pow(R, depth_leaf) / cnum;   // no of leaves per class


  for (i=0; i<cnum; i++) {
    sprintf(fname2, "%s%s_L%02d.G%02d.dat", PRE, TREE, depth_leaf, i);
    fp2 = fopen(fname2, "r");

    for (j=0; j<lnum; j++) {
       fread(&p, sizeof(p), 1, fp2);
       // printf(" %d", p.id);
    }
    // printf(" * Group %02d\n", i);

    fclose(fp2);
  }

}

//-----------------------------------------------------------------
// write subtree nodes
//-----------------------------------------------------------------
void write_subtree_nodes(int depth_root, int class, int depth_leaf) {
  nodetype p;
  int cnum;   // the number of clases
  int lnum;   // the number of leaves per each class at depth_leaf
  int i, j, cnt;

  FILE *fp1;  // read file pointer
  FILE *fp2;  // write file pointer

  char fname1[64];
  char fname2[64];


  for (i=depth_root; i<=depth_leaf; i++) {
    cnum = (int) pow(R, depth_root);     // no of classes
    lnum = (int) pow(R, i) / cnum ;      // no of leaves per class

    sprintf(fname1, "%s%s_L%02d.dat", PRE, TREE, i);
    fp1 = fopen(fname1, "r");

    sprintf(fname2, "%s%s_L%02d.G%02d", PRE, TREE, i, class);
    sprintf(fname2, "%s.L%02d.dat", fname2, i - depth_root);
    fp2 = fopen(fname2, "w");

    fseek(fp1, class*lnum*sizeof(p), SEEK_CUR);
    for (j=0; j<lnum; j++) {
       cnt = fread(&p, sizeof(p), 1, fp1);
       if (cnt == 0) {
         perror("* error in reading file ...\n");
         exit(1);
       }
       fwrite(&p, sizeof(p), 1, fp2);
    }

    fclose(fp2);
    fclose(fp1);
  }

}

//-----------------------------------------------------------------
// read subtree nodes
//-----------------------------------------------------------------
void read_subtree_nodes(int depth_root, int class, int depth_leaf) {
  nodetype p;
  int cnum;   // the number of clases
  int lnum;   // the number of leaves per each class at depth_leaf
  int i, j;

  FILE *fp2;  // write file pointer

  char fname2[64];


  for (i=depth_root; i<=depth_leaf; i++) {
    cnum = (int) pow(R, depth_root);     // no of classes
    lnum = (int) pow(R, i) / cnum ;      // no of leaves per class
    sprintf(fname2, "%s%s_L%02d.G%02d", PRE, TREE, i, class);
    sprintf(fname2, "%s.L%02d.dat", fname2, i - depth_root);
    fp2 = fopen(fname2, "r");

    for (j=0; j<lnum; j++) {
       fread(&p, sizeof(p), 1, fp2);
       printf(" %d", p.id);
    }
    printf(" * Level %02d (%02d)\n", i, i-depth_root);

    fclose(fp2);
  }

}




::::::::::::::
binary8_plot.c
::::::::::::::
//-----------------------------------------------------------------------------
//  File Name:
//     binary8_plot.c
//
//  Purpose:
//
//     find and print the optimal path
//
//  Parameters:
//
//
//  Discussion:
//
//
//  Licensing:
//
//    This code is distributed under the GNU LGPL license.
//
//  Modified:
//
//    2018.12.05 Wed
//
//  Author:
//
//    Young Won Lim
//
//---------------------------------------------------------------------------
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>

#include "binary1_search_defs.h"

#if DISP==0
  #define NO_DISP
#else
  #undef NO_DISP
#endif

//---------------------------------------------------------------
// latex plot a path from root to node
//---------------------------------------------------------------
char *tree_construct(char * path);
void tree_string(FILE *fp, char * path);
void table_string(FILE *fp, char * path);
void create_tex_file(char *path, char *str);
void plot_path(qnodetype* q, char *str);


//----------------------------------------------------------------
char *tree_construct(char * path) {
  char *t, *s, u[256], v[256];
  int br;
  static int i = 0;

  // printf("path=%s \n", path);

  s = malloc(256);  // s must be a heap memory
  t = strtok(path, " ");

  if (t == NULL) {
    sprintf(s, "%d", i++);
    i = 0;
    // printf("s=%s \n", s);
    return(s);
  } else {
    br = atoi(t);
    sprintf(u, "%d", i++);
    sprintf(v, "%s", tree_construct(path+2));
    switch (br) {
      case R-2 : sprintf(s, "[.%s %s x ] ", u, v); break;
      case R-1 : sprintf(s, "[.%s x %s ] ", u, v); break;
    }
    // printf("s=%s \n", s);
    return(s);
  }
}

//----------------------------------------------------------------
void tree_string(FILE *fp, char * path) {
  char p1[256]="";
  char p2[256]="";

  strcpy(p1, path);     // strtok corrupts the input string
  strcpy(p2, tree_construct(p1));
  printf("tree=%s\n", p2);

  fprintf(fp,"\\Tree %s\n", p2);
}

//----------------------------------------------------------------
void table_string(FILE *fp, char * path) {
  char *t, p[256];
  int br, ui;
  double theta = Param.theta;
  int i = 0;

  strcpy(p, path);  // strtok corrupts the input string
  // printf("path=%s \n", p);

  fprintf(fp, "\\hline\n");
  fprintf(fp, "$i$ & $br$ & $theta$ & $-u(i)$ & $a(i)$ & $theta’$");
  fprintf(fp, " \\\\ \\hline\n");
  fprintf(fp, "\\hline\n");

  t = strtok(p, " ");

  // printf("t=%c \n", *t);

  while (t != NULL) {
    // printf("t=%c \n", *t);

    br = atoi(t);
    switch (br) {
      case R-2 : ui = +1; break;
      case R-1 : ui = -1; break;
    }

    fprintf(fp, "%d & %d & %f & %d & %f & ", i, br, theta, -ui, a[i]);
    theta = theta - ui * a[i++];
    fprintf(fp, "%f \\\\ \\hline\n", theta);

    t = strtok(NULL, " ");
  }

}


//-----------------------------------------------------------------
void create_tex_file(char *path, char* str) {
  FILE *fp;
  char fname[256]="", bname[256]="", cmd[256]="";
  int cnt;

  if (!strcmp(str, "leafmin")) cnt = 0;
  else if (!strcmp(str, "globalmin")) cnt = 1;
  else if (!strcmp(str, "cordic")) cnt = 2;
  else cnt = 0;

  sprintf(bname, "%s_%d_%s", TREE, cnt+1, str);
  sprintf(fname, "%s_%d_%s.tex", TREE, cnt+1, str);

  fp = fopen(fname, "w");

  fprintf(fp,"\\documentclass{article}\n");
  fprintf(fp,"\\usepackage[margin=1in]{geometry}\n");
  fprintf(fp,"\\usepackage{graphicx}\n");
  fprintf(fp,"\\usepackage{tikz-qtree}\n");
  fprintf(fp,"\\begin{document}\n");

  fprintf(fp,"\\setcounter{section}{%d}\n", cnt);
  fprintf(fp,"\\section{%s (%s) (N=%d R=%d theta=%f)}\n",
             Param.tstring, str, Param.NN, Param.RR, Param.theta);

  fprintf(fp,"\\begin{tikzpicture}[scale=1]\n");
  //.........................................
  tree_string(fp, path);
  //.........................................
  fprintf(fp,"\\end{tikzpicture}\n");


  fprintf(fp, "\\begin{center}\n");
  fprintf(fp, "\\begin{tabular}{ |r|r|r|r|r|r|r| }\n");
  //.........................................
  table_string(fp, path);
  //.........................................
  fprintf(fp, "\\end{tabular}\n");
  fprintf(fp, "\\end{center}\n");

  fprintf(fp,"\\end{document}\n");

  fclose(fp);

  if (0) {
    sprintf(cmd, "latex %s.tex", bname);      printf("%s\n", cmd); system(cmd);
    sprintf(cmd, "dvipdf %s.dvi", bname);     printf("%s\n", cmd); system(cmd);
#ifndef NO_DISP
    sprintf(cmd, "xreader -w %s.pdf", bname); printf("%s\n", cmd); system(cmd);
#endif
  } else {
    sprintf(cmd, "latex %s.tex > /dev/null", bname);
    printf("%s\n", cmd); system(cmd);
    sprintf(cmd, "dvipdf %s.dvi > /dev/null", bname);
    printf("%s\n", cmd); system(cmd);
#ifndef NO_DISP
    sprintf(cmd, "xreader -w %s.pdf > /dev/null", bname);
    printf("%s\n", cmd); system(cmd);
#endif
  }

}


//----------------------------------------------------------------
void plot_path(qnodetype* q, char *str) {
  char path[256]="", p[256]="";

  while (q) {
    q = q->next;

    if (q == NULL) {
      printf("\n");
      break;
    }
    sprintf(p, "%d ", (q->node)->branch);
    strcat(path, p);
  }

  printf("%s path=%s\n", str, path);


  create_tex_file(path, str);


}


::::::::::::::
binary9_main.c
::::::::::::::
//------------------------------------------------------------------------------
//  File Name:
//     binary9_main.c
//
//  Purpose:
//
//     binary angle tree search main
//
//  Parameters:
//
//
//  Discussion:
//
//
//  Licensing:
//
//    This code is distributed under the GNU LGPL license.
//
//  Modified:
//
//    2018.12.05 Wed
//
//  Author:
//
//    Young Won Lim
//
//----------------------------------------------------------------------------
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
```

```
#include <string.h>

#include "binary1_search_defs.h"




qnodetype *leafmin_path;
qnodetype *globalmin_path;
qnodetype *cordic_path;

//---------------------------------------------------------------
// main - Ternary Angle Tree Search
//---------------------------------------------------------------
int main(int argc, char *argv[]) {
  double theta; // = 4*atan(pow(2,-5));
  int i;

  nodetype p;
  nodetype min_leaf;
  nodetype min_global;
  nodetype cordic_node;


  if (argc != 2) {
    printf("binary_search i (theta=2^(-i)) \n");
    return 0;
  }

  i = atoi(argv[1]);
  theta = atan(pow(2, -1*i));

  printf("binary angle tree search (N=%d) \n", N);
  printf("theta= atan(pow(2,%d) = %10g \n", -1*i, theta);


  for (i=0; i<2*N; ++i) {
    a[i] = atan(1./pow(2, i));
  }


  Param.NN    = N;
  Param.RR    = R;
  Param.theta = theta;
  strcpy(Param.tstring, "binary angle tree");

  p.theta = theta;
  p.depth = 0;
  p.id = 0;
  p.branch = 0;
  for (i=0; i<R; ++i) p.child[i]= i+1;

  tree_traverse(&p);


  printf("\n======================================\n");
  printf("* the leaf optimal path \n");
  printf("======================================\n");
  min_leaf = find_level_min_node(N-1, 1);
  leafmin_path = find_path(&min_leaf);
  print_path(leafmin_path, "leafmin");
  plot_path(leafmin_path, "leafmin");


  printf("\n======================================\n");
  printf("* the global optimal path \n");
  printf("======================================\n");
  min_global = find_global_min_node();
  globalmin_path = find_path(&min_global);
  print_path(globalmin_path, "globalmin");
  plot_path(globalmin_path, "globalmin");


  printf("\n======================================\n");
  printf("* the cordic path \n");
  printf("======================================\n");
  // cordic_path = cordic_traverse(&p);  // method 1
  // cordic_path = find_cordic_path(&p); // method 2
  cordic_node = find_cordic_node(&p); // method 3
  cordic_path = find_path(&cordic_node);
  print_path(cordic_path, "cordic");
  plot_path(cordic_path, "cordic");


/*
  printf("* classify leaf nodes \n");
  write_subtree_leaves(2, N-1);
  read_subtree_leaves(2, N-1);

  printf("* subtree nodes \n");
  write_subtree_nodes(2, 3, 5);
  read_subtree_nodes(2, 3, 5);

  printf("* print level nodes \n");
  for (i=0; i<N; ++i) {
     print_level_nodes(i);
  }
*/


  delete_path(leafmin_path, "leafmin");
  delete_path(globalmin_path, "globalmin");
  delete_path(cordic_path, "cordic");

}
```