Solving the Minimum-Cost Matching Problem for Quasi-Convex Tours: An Efficient ANSI C Implementation[1]

Samuel R. Buss, University of California, San Diego
Kirk G. Kanzelberger, NEC Research Institute
David Robinson, University of California, San Diego
Peter N. Yianilos, NEC Research Institute
April 13, 1994

Abstract

We report an efficient and highly portable ANSI C implementation of the Buss-Yianilos minimum-cost matching algorithm for quasi-convex tours. A generic $O(\log n)$ time implementation of the required Ω predicate is included, resulting in worst-case $O(n \log n)$ runtime for arbitrary cost functions.

A constant-time Ω is provided for two special cases: the circle under Euclidean distance, and the real line where cost is defined as the square root of interpoint distance. In both of these settings, runtime is then $O(n)$.

Code is also provided for circular tours with cost defined as the square root of angular distance (arclength). In this case, the generic Ω computation is employed.

The test programs generate pseudorandom node patterns. To ensure correctness, a straightforward $O(n^3)$ dynamic programming solution may be optionally enabled, and the test programs will compare its result with that of the Buss-Yianilos algorithm.

As an additional option, the test programs produce PostScript-form graphical representations of the minimum-cost matchings found.

The performance of the package is reported for several modern RISC processors.

[1]This document is typeset in LaTeX using the NoWeb literate programming language (Norman Ramsey, norman@bellcore.com).

1 Introduction

This technical memorandum describes an ANSI C implementation of the Buss-Yianilos minimum-cost matching algorithm for quasi-convex tours [1].

The algorithm is rather intricate but is free of large hidden constants. We report our implementation in the hope that this algorithm might sooner find practical application. We begin by reviewing the general problem.

Consider the complete weighted graph G formed by a collection of N nodes together with a symmetric cost function $c(x, y)$ giving the cost of an edge between nodes x and y. In addition, assume that the nodes of G are given as a tour $x_1, \ldots, x_N$, thought of as cyclic ordering, such that for any x_i, x_j, x_k, x_ℓ in tour order but not necessarily adjacent, the inequality $c(x_i, x_\ell) + c(x_j, x_k) \leq c(x_i, x_k) + c(x_j, x_\ell)$ holds. A tour satisfying this condition is said to be *quasi-convex.* For example, if the nodes of G are points on the unit circle forming either a clockwise or counterclockwise tour, the quasi-convex condition is satisfied.

Our objective is to find, among all maximum size matchings for G, one with minimal total weight (cost). If the graph is bipartite and consists of n red nodes and n' blue nodes, we are interested in solutions for which edges exist only between nodes of different color. In the bipartite case, a tour is said to be *balanced* if $n = n'$. Finally, a tour $x_1, \ldots, x_N$ is said to be *line-like* if starting from any x_i, costs are non-decreasing as one moves towards the beginning or end of the tour.

The Buss-Yianilos algorithm provides solutions for several combinations of these characteristics, but in all cases requires the quasi-convexity condition. The conditions may be expressed as:

Bipartite <u>and</u> (Tour Balanced <u>or</u> Tour Line-like)

<u>or</u>

Non-Bipartite <u>and</u> (N even <u>or</u> Tour Line-Like)

In all of these cases $O(N \log N)$ time execution results, but in particular settings this may be improved to linear time. This improvement hinges on constant-time implementation of a particular predicate Ω employed by the algorithm. We include such code for two settings:

G1 The unit circle under Euclidean distance

G2 The real line with costs given by $\sqrt{\text{distance}}$

Our code also includes a generic implementation of Ω that may be used in any problem setting. We demonstrate its use in settings **G1** and **G2**, as well as in a third setting for which no constant-time Ω was implemented:

G3 The unit circle with costs given by $\sqrt{\text{angular distance}}$

The main code consists of an include file `qcmatch.h` and a single source file `qcmatch.c`. To use the package, one need only supply a cost function and a graph to obtain a solution in $O(N \log N)$ time. Improving to linear time requires a constant time Ω function suitable for the problem domain. This may involve considerable additional application-specific work, and may pay off only for large problem sizes.

Three test programs are provided, corresponding to the three geometric settings **G1-G3** above. They are built by combining a test program driver `qctest.c` with one of three customization files: `qcsqrtarc.c`, `qcarc.c`, or `qcsqrtline.c`. All three programs optionally compare the Buss-Yianilos solution with that obtained by a straightforward $O(n^3)$ dynamic programming algorithm. PostScript plots of solutions can also be generated for the circular or linear cases; the code for this is in files `qcgrapharc.c` and `qcgraphline.c`.

Finally, we report performance measurements on several RISC processors: Sun/Weitek SPARC, DEC Alpha, and SGI/MIPS R4400.

2 Making and Testing the Package

Here is the `Makefile` for compiling the library `qcmatch.a`, and the three test programs `testsqrtarc`, `testarc`, and `testsqrtline`.

⟨*Makefile* 2⟩≡

```
CC      = gcc -ansi -pedantic -Wall

CFLAGS  = -O4
LFLAGS  =
LLIBS   = -lm

RANLIB  = ranlib

ARCHIVE = qcmatch.a

QCMATCH_OBJS      = qcmatch.o
QCMATCH_SRC       = qcmatch.c

TESTSQRTARC_OBJS  = qctest.o qcsqrtarc.o qcgrapharc.o
TESTSQRTARC_SRC   = qctest.c qcsqrtarc.c qcgrapharc.c

TESTARC_OBJS      = qctest.o qcarc.o qcgrapharc.o
TESTARC_SRC       = qctest.c qcarc.c qcgrapharc.c

TESTSQRTLINE_OBJS = qctest.o qcsqrtline.o qcgraphline.o
TESTSQRTLINE_SRC  = qctest.c qcsqrtline.c qcgraphline.c
```

```
TARGETS = ${ARCHIVE} testsqrtarc testarc testsqrtline

all:    ${TARGETS}

${ARCHIVE}:     ${QCMATCH_OBJS}
        ar rc $@ $?
        ${RANLIB} $@

testsqrtarc:   ${TESTSQRTARC_OBJS} ${ARCHIVE}
        ${CC} ${CFLAGS} -o $@ ${TESTSQRTARC_OBJS} ${ARCHIVE} ${LLIBS}

testarc:        ${TESTARC_OBJS} ${ARCHIVE}
        ${CC} ${CFLAGS} -o $@ ${TESTARC_OBJS} ${ARCHIVE} ${LLIBS}

testsqrtline:   ${TESTSQRTLINE_OBJS} ${ARCHIVE}
        ${CC} ${CFLAGS} -o $@ ${TESTSQRTLINE_OBJS} ${ARCHIVE} ${LLIBS}

%.o:    %.c qcmatch.h
        ${CC} ${CFLAGS} -c $<

clean:
        @rm -f ${TARGETS} *.o
```

This code is written to file `Makefile`.

When the `-g` command-line option is specified, the three test programs output PostScript commands which are to be appended to `qcheader.ps`. Use the following commands to compile and run the programs, and print the PostScript solution plots:

⟨*Compiling and Running the Programs* 3a⟩≡

```
make
(cat qcheader.ps; testsqrtline -g) | lpr
(cat qcheader.ps; testsqrtarc  -g) | lpr
(cat qcheader.ps; testarc      -g) | lpr
```

Root chunk (not used in this document).

To run a quick test of all three programs, use the following commands:

⟨*Quick Test Program* 3b⟩≡

```
testsqrtline -d -n 100
testsqrtarc  -d -n 100
testarc      -d -n 100
```

Root chunk (not used in this document).

Here the `-d` option activates the dynamic programming algorithm to verify the main algorithm's solution. `-n 100` indicates that 100 randomly-drawn problem instances will be generated.

At the conclusion of any test involving the `-d` option, the test program displays a two-line summary containing three numeric values. The two values on the first line represent (1) the number of problem instances for which there was agreement between the Buss-Yianilos algorithm and the dynamic programming algorithm, and (2) the number of problem instances for which the total cost agreed, but the particular minimum-cost matching differed. (Of course, these are not errors, since there sometimes exists more than one minimal-cost solution.) The value on the second line reports actual failures and should *always* be zero.

3 Calling the Matching Function

Access to our implementation of the Buss-Yianilos algorithm is provided by a single entry point `qc_match` which accepts seven arguments. Its ANSI function prototype is given in the `qcmatch.h` file described later. The return value from `qc_match` is a real number that is the total cost of the minimum-cost matching. Here is a detailed description of each argument:

`n` The number of input nodes.

`input` An array of pointers to input nodes. The nodes themselves may be of arbitrary structure, since their internals will be referenced only by the user-supplied `cost` slave function. The array of pointers must be of length `n`.

`colors` An integer array of length `n` whose entries specify the color of the corresponding node in the `input` array. The two color values are +1 and -1 (also defined, respectively, as the manifest constants `BLUE` and `RED` in the header file `qcmatch.h`).

Unbalanced tours (i.e. tours with unequal numbers of blue and red nodes) are only permitted for the "line-like" case. (See the `linelike` parameter below.)

For non-bipartite tours, the user should initialize the `colors` array with alternating color values, i.e., `BLUE`, `RED`, `BLUE`, `RED`, ... The resulting matching is guaranteed to be a valid minimum-cost matching for the non-bipartite tour (see [1]). Note that the condition of the previous paragraph must still be fulfilled: If the tour is not line-like, there must be equal numbers of `BLUE`'s and `RED`'s – i.e. `n` must be *even*.

`matching` An integer array of length `n`, into which `qc_match` will write the minimum-cost matching found by the Buss-Yianilos algorithm. Specifically, `matching[i]` will be the index in the `input` array of the node that the algorithm matched with `input[i]`.

For line-like tours that are either unbalanced, or non-bipartite and of odd length, there will be at least one `i` such that node `input[i]` is *unmatched.* For each such `i`, `matching[i]` will contain the value `-1`.

`linelike` An boolean value (0=`FALSE`,1=`TRUE`) which, if `TRUE`, indicates that the tour is line-like (see the definition on page 1).

`cost` The user must always supply a `cost` function. It should return a real number representing the cost of putting an edge between a pair of nodes. The node pair is passed as two parameters of type `void *`, which are just the node pointers from the `input` array. See the example `cost` functions given in Appendices 2-4 for circular and linear tours.

`omega` This is the function `qc_match` will use as the Ω function. The purpose of the Boolean Ω predicate is rather technical and is described in [1]. A generic Ω function is supplied below that will work for any cost function. To use this generic Ω, simply pass `NULL` as the `omega` argument to `qc_match`. Users can often improve performance for particular problem domains by writing their own constant-time Ω functions. Custom Ω functions take the same parameters as the generic Ω function (see page 28 below).

4 Working Storage

The working storage required is allocated by the package-internal procedure `qc_match_alloc`. Given 32-bit words, the space required comes to $14 \cdot n$ bytes, where n is tour length. This storage is *not freed* upon return from `qc_match`. Instead, a separate entry point `qc_match_free` is provided for this purpose. Routine `qc_match_alloc` is called by `qc_match` only when the working storage already allocated is too small for the current problem. In this event, `qc_match` frees its storage, and then allocates just enough for the new problem. In summary, then, the user in most cases can ignore the issue of dynamic storage allocation, since `qc_match` and associated internal routines will manage it.

5 The Test Programs

We supply three test programs: `testsqrtline`, `testsqrtarc`, and `testarc`. The first considers graphs on the real line, with cost given by the square root of interpoint distance. The second implements a graphs on a circle, with cost given by the square root of interpoint arc length, i.e. angular distance. The third again considers a circular setting, but with cost given by Euclidean distance (i.e.

chord length). These three programs are all built from a common test driver module `qctest.c`, which is linked with problem-specific modules `qcsqrtline.c`, `qcsqrtarc.c`, and `qcarc.c`, respectively.

The result of the Buss-Yianilos algorithm may be verified by comparison with the dynamic programming result for the same tour (`-d` option). If there is a discrepancy, error messages will be generated. Since Buss-Yianilos algorithm is correctly implemented, the only effect should be the time penalty for running the dynamic programming algorithm.

The test programs can produce graphical representations of the minimum-cost matchings for the tours tested, in the form of PostScript commands written to standard output (`-g` option). These commands should be appended to the PostScript header `qcheader.ps` to generate a printable file. A maximum tour length may be specified (`-s` option), as well as the number of pseudorandom problem instances generated (`-n` option).

The test programs also accept a powerful -t option that specifies the "tour type." This option enables stress-testing and benchmarking of specific tour node distributions, and requires an argument consisting of three letters:

The first letter ...must be either `B` (Balanced) or `U` (Unbalanced). Balanced tours always contain equal numbers of red and blue nodes.

The second letter ...must be either `R` (Random) or `A` (Alternating). Alternating tours constitute the strongest test of the central algorithm.

The third letter ...must be either `C` (Cyclic) or `L` (Linear/line-like). In the two circular settings, requesting line-like tours causes the nodes to be distributed on the half-open semicircle $[0, \pi)$, rather than on the full circle.

The following six tour types are legal: `BRC` (the default), `BAC`, `BRL`, `BAL`, `URL`, and `UAL`. The following two tour types are illegal: `URC` (Unbalanced Random Cyclic) and `UAC` (Unbalanced Alternating Cyclic), since for unbalanced tours the Buss-Yianilos algorithm only works in the line-like case.

Note that the minimum-cost matchings for tour types `BAC` and `BAL` are equivalent to those for even-length non-bipartite graphs, and those for tour type `UAL` are equivalent to those for odd-length non-bipartite graphs.

The original implementation of this software package handled balanced tours only, but was then generalized to handle unbalanced line-like tours. The two constant-time Ω implementations, however, have not been updated with respect to this generalization. Consequently, when unbalanced tours are requested, the test programs will force the use of the generic Ω function (equivalent to the `-O` command-line option).

A "benchmarking mode" is available with the `-B` option. This option forces all tours to be of the maximum size (either the default of 128 nodes, or the value specified with `-s`), and also turns on code to time the execution of the Buss-Yianilos algorithm. The average time per tour is accumulated, and printed out along with the tour size at the end of the program. The plots in the performance studies below were generated by accumulating such pairs of values in a single file, and then submitting them to the `gnuplot` plotting utility.

Here is a summary of the command-line flags accepted by each test program:

`-B` turns on benchmarking mode.

`-O` forces use of the generic Ω implementation.

`-d` turns on dynamic programming verification of the Buss-Yianilos algorithm.

`-g` turns on PostScript output.

`-t` *xyz* requests only tours of type *xyz* (default `BRC`).

`-s` *size* specifies a maximum tour length of *size* nodes (default 128).

`-n` *n* requests testing *n* random tours (default 1).

`-r` *seed* specifies the *seed* for the random number generator (default 102).

Here are some graphical output samples, generated with `-g`. Figure 1 shows the results for three randomly-generated problem instances solved using `testsqrtarc`, and figure 2 shows three solutions from `testsqrtline`. In these examples, edges of the matching are depicted with circular arcs the length of which does not necessarily correspond to edge costs.

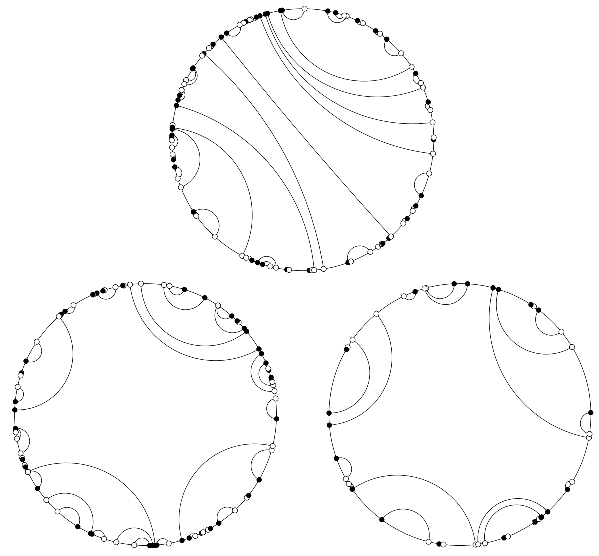

Figure 1: Graphical representations of minimum-cost matchings for three circular tours with $\sqrt{\text{arclength}}$ costs.

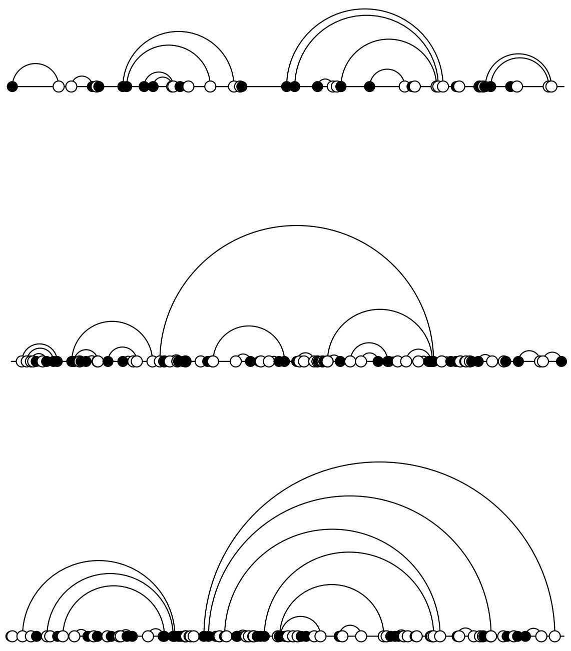

Figure 2: Graphical representations of minimum-cost matchings for three tours on the real line with $\sqrt{\text{length}}$ costs.

6 Performance Study

We conducted performance studies on several different RISC machine architectures. In all cases, we used the `gcc` ANSI C compiler with `-O4` level optimization. In addition to confirming the linear-time performance of the Buss-Yianilos algorithm, our results demonstrate the high degree of comparability between different RISC machines when execution times are normalized with respect to clock rate. We therefore report all results in terms of normalized RISC cycles.

We first present a complete series of measurements obtained on an 80 MHz Sun/Weitek SPARCstation. The graphs in figures 3, 4, and 5 show the performance of the Buss-Yianilos quasi-convex matching algorithm for balanced alternating tours with three different cost functions. Each data point in these graphs represents the average runtime in RISC cycles for 200 balanced alternating tours of a particular tour length, with the nodes randomly distributed over the tour. (Balanced alternating tours represent the algorithm's worst-case performance.)

Figure 3 plots only the "generic Ω" case, while figures 4 and 5 both allow a comparison of generic Ω to a constant-time Ω implementation. It may be seen from the graphs that in these problem settings, the worst-case $\log n$ behavior of generic Ω does not dominate performance – the generic Ω graphs appear very linear. *Please note, however, that this may not hold true for other settings where node distribution is not uniform and random.*

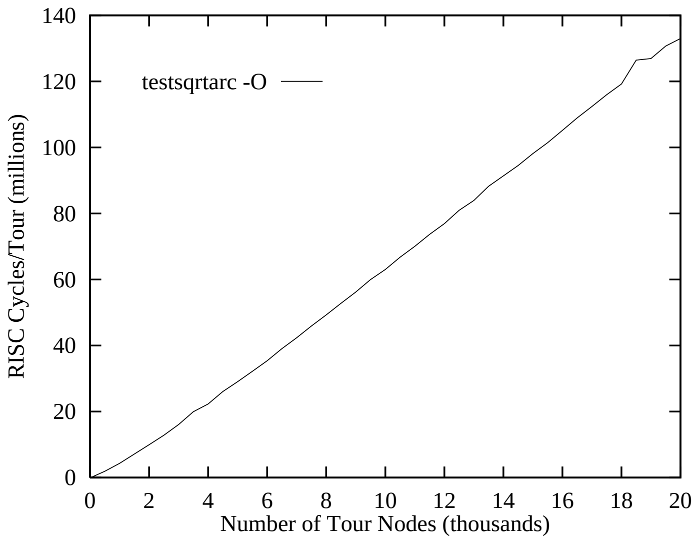

Figure 3: Average runtime in RISC cycles for the Buss-Yianilos algorithm as a function of tour length, for the circular square root case.

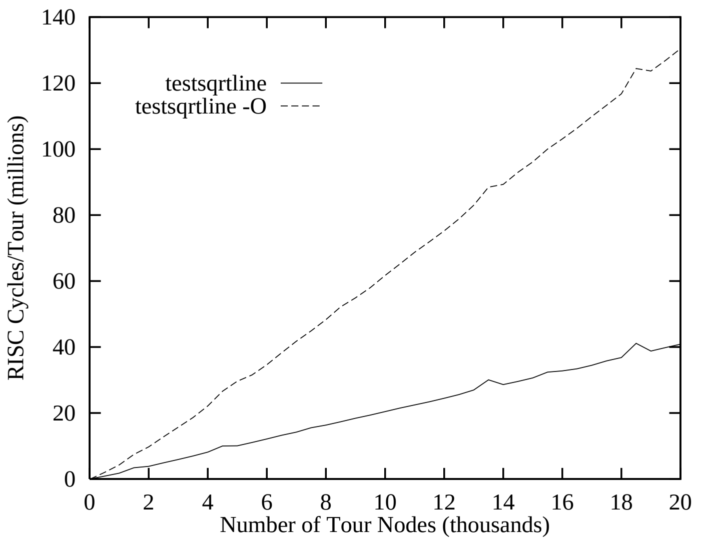

Figure 4: Average runtime in RISC cycles for the Buss-Yianilos algorithm as a function of tour length, for the linear square root case.

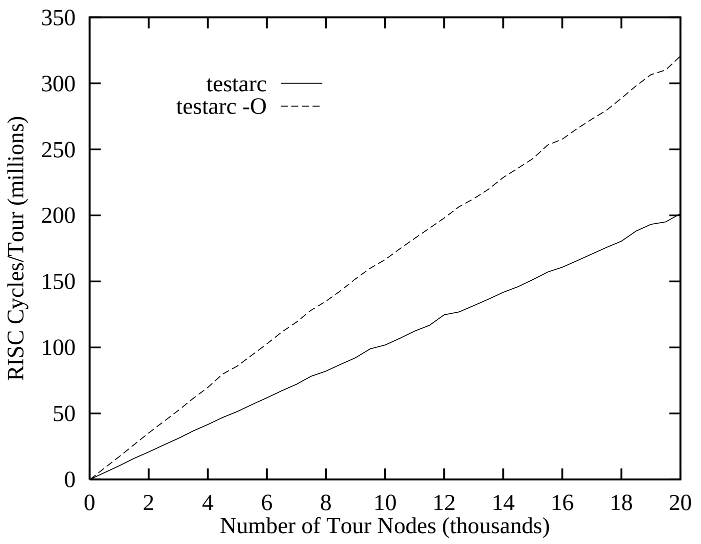

Figure 5: Average runtime in RISC cycles for the Buss-Yianilos algorithm as a function of tour length, for the circular Euclidean case.

We were especially interested in computing values for the leading constant of the $O(n)$ Buss-Yianilos algorithm. This constant is equal to the slope of the lines corresponding to the constant-time Ω implementations in figures 4 and 5. For the linear case with square root cost, the leading constant is almost exactly 2000 RISC cycles per tour node, and for the circular case with Euclidean cost it is approximately 10,000 RISC cycles per tour node on the Sun/Weitek SPARC-station. The five-fold difference may be attributed to the expensive Euclidean circular cost function implementation, which is forced to do trigonometry to convert from angular to rectangular coordinates (see next section for further discussion).

The constant-time Ω benchmarks were then run on two other RISC machines – a 150 MHz SGI/MIPS R4400 and a 150 MHz DEC Alpha. Code was compiled, again, using `gcc` and `-O4` level optimization. The results are plotted together with the Sun/Weitek results in figures 6 and 7. It is interesting that in both problem settings, the Sun/Weitek machine performs somewhat more efficiently than its faster counterparts, with the greatest divergence being that between Sun/Weitek and SGI/MIPS in the linear problem setting ($> 25\%$). Factors contributing to this disparity might include differences between the particular RISC architectures and/or the corresponding `-O4` optimizers, and the higher relative penalty (in terms of clock cycles) for cache misses in the two 150 MHz machines.

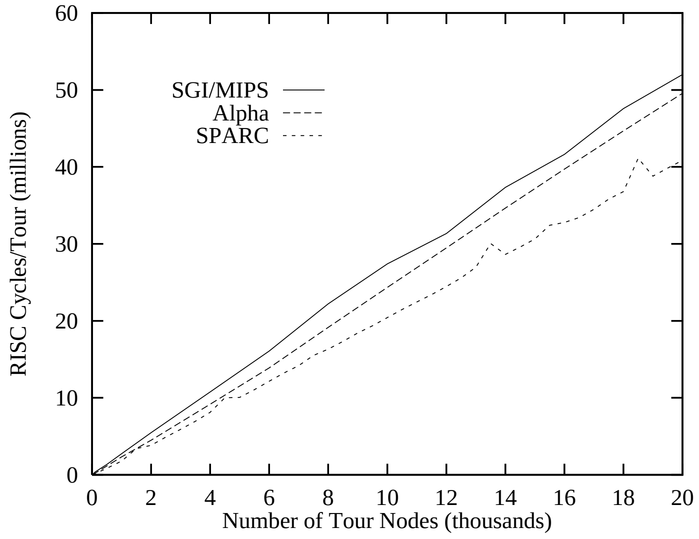

Figure 6: Average runtime in RISC cycles for the linear square root case on three different RISC architectures.

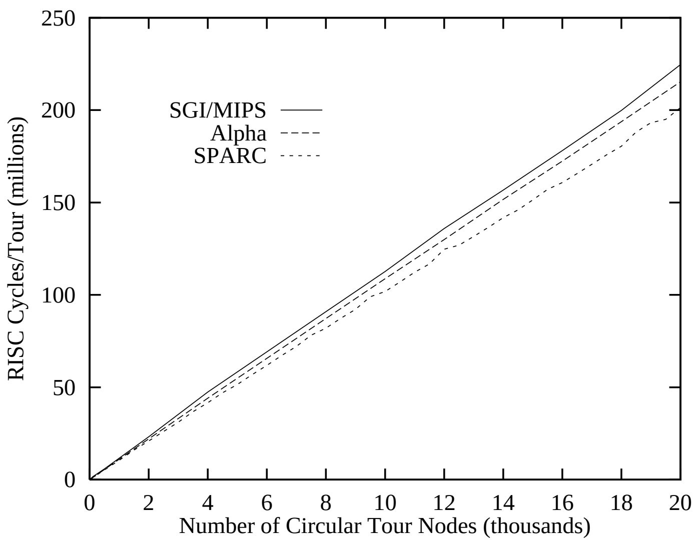

Figure 7: Average runtime in RISC cycles for the circular Euclidean case on three different RISC architectures.

7 Large-Scale Tests

The following large-scale tests of the Buss-Yianilos algorithm ran to completion on the Sun/Weitek SPARCstation:

⟨*Large-Scale Tests* 12⟩≡
```
  testsqrtline    -t BAL -s 100 -n 1000000 -d
          Successes:      1000000 (257 with different matchings)
          Failures:       0
  testsqrtline -O -t URL -s 100 -n 1000000 -d
          Successes:      1000000 (2693 with different matchings)
          Failures:       0

  testsqrtarc  -O -t BAC -s 100 -n 1000000 -d
          Successes:      1000000 (157 with different matchings)
          Failures:       0
  testsqrtarc  -O -t URL -s 100 -n 1000000 -d
          Successes:      1000000 (2693 with different matchings)
          Failures:       0
```
Root chunk (not used in this document).

8 Tips For Maximum Efficiency

The inefficiencies that remain in our test problems chiefly have to do with calculating the cost function. The main code of the Buss-Yianilos algorithm calls the user's cost function an average of $6 \cdot n$ times for balanced alternating tours, where n is the length of the tour. The percentage of runtime devoted to calculating the cost function is about 30% for our square root linear case (`testsqrtline`), and nearly 50% for our Euclidean circle case (`testarc`), both running with constant-time Ω functions.

When generic Ω is used, the number of cost function invocations is observed to approach $k \cdot n$, where k is a constant that is problem-dependent. This constant is about 40 for the linear and circular square root cases `testsqrtline` and `testsqrtarc`), and about 16 for the circular Euclidean case (`testarc`). For tour sizes of 20,000 nodes, the percentage of runtime devoted to calculating the cost function is about 60% for `testsqrtline` and `testsqrtarc`, and over 90% for `testarc`.

Clearly, optimizing the cost calculation is paramount in efficient implementations for a particular problem domain.

If costs were to be precomputed, then the cost function would reduce to a 2-D table lookup. Note that in certain problem domains, such as the string matching applications described in [1], the costs (being a simple function of string displacement) are invariant over all problem instances, and can be stored in a 1-D array indexed by the (integral) displacement value.

Subroutine linkage overhead can be dealt with by making the cost function a macro. For an ultra-efficient implementation, the deque maintenance routines (`push_M_right`, etc.) could also be moved in-line.

If you do not precompute costs, remember to choose node coordinate systems such that the cost computation will be fast. The difference is illustrated by the two test programs `testsqrtarc` and `testarc`. Nodes on the unit circle are specified with angular coordinates, which results in a fast computation for square root of angular distance, but an exceedingly slow one for Euclidean distance (chord length), since the cost function is forced to do trigonometry to convert to rectangular coordinates.

9 The Header file `qcmatch.h`

The header file `qcmatch.h` contains definitions used by the matching package. The user should `#include` this file. In addition to basic `#define`'s and `typedef`'s, this file defines the structure of an element of the main deque $\mathcal{M}$ of the Buss-Yianilos paper. Applications which define customized Ω functions require knowledge of this deque.

⟨*qcmatch.h* 13⟩≡
```
#include <math.h>

#define BLUE    1
#define RED     (-1)

typedef double  REAL;

#define FALSE   0
#define TRUE    1

typedef int     BOOLEAN;

typedef struct deque_node {
        int     x;      /* Input node index */
        REAL    i;      /* The value I[x] from the Buss-Yianilos paper */
} MNODE;

typedef REAL    (*COSTFUNC)(void *, void *);

typedef BOOLEAN (*OMEGAFUNC)( MNODE *, MNODE *, MNODE *, void **,
                             COSTFUNC, int );

                                /* The quasi-convex matching function */
REAL    qc_match( int n, void *input[], int colors[], int matching[],
                 BOOLEAN linelike, COSTFUNC cost, OMEGAFUNC omega);

void    qc_match_free( void );
```
This code is written to file `qcmatch.h`.

10 The Central Algorithm

We now turn to our implementation of the central algorithm. Familiarity with the Buss-Yianilos paper is required to understand the portions of our description.

The color toggle ψ of the paper corresponds to `static int Psi` and alternates between -1 and +1. The main deque $\mathcal{M}$ corresponds to `static MDEQUE M`. The leftmost and rightmost elements $\mathcal{M}_L$ and $\mathcal{M}_R$ are written `M.l` and `M.r` in our implementation. The two left deques $\mathcal{L}^1$ and $\mathcal{L}^{-1}$ are implemented via `static LDEQUE *L`. So for example, $\mathcal{L}^{\psi}_{R}$ is written `L[Psi].r`. The node within that element is `L[Psi].r->x` and the improvement value is `L[Psi].r->i`. So the program structures are seen to correspond in a close and natural way to the pseudocode syntax of the paper.

⟨*qcmatch.c* 15⟩≡
```
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <limits.h>
#include <assert.h>
#include "qcmatch.h"

static int      Max_node;       /* Memory is allocated for tours of up to this
                                   size */

static BOOLEAN   LineLike;      /* Whether the tour is line-like */
static OMEGAFUNC Omega;         /* The Omega predicate */
static COSTFUNC  Cost;          /* The user-supplied cost function */

#define COST(nodex, nodey)      ((nodex == NULL || nodey == NULL) ?     \
                                 (REAL) 0.0 : (*Cost)(nodex, nodey))

static int      Psi;            /* The color toggle, 1=BLUE, -1=RED */
static int      Xn;             /* The X of Algorithm 2 */
static REAL     TotalCost;      /* The cost of an optimal matching  */

static void **  Nodes;          /* Working copy of the input nodes array */

/* Linked lists for reducing tour to alternating subtours */

static int     *LevelHeads, *LevelTails, *LevelLinks;

/* The M deque is implemented via the following structure.  Its size
   is 3*Max_node */

typedef struct {
        MNODE *r;       /* Points to element M_{R} */
        MNODE *l;       /* Points to element M_{L} */
        MNODE *d ;      /* Vector of MNODE elements */
        int rx;         /* Index of the deque's rightmost element */
        int lx;         /* Index of the deque's leftmost element */
} MDEQUE;

static MDEQUE M;

/* The two left deques L^{1} and L^{-1} are implemented via the
   structure below.  Each contains 3*Max_node entries.  */

typedef struct {
        MNODE *r;       /* Points to element L_{R} */
        MNODE *rm1;     /* Points to element L_{R-1} */
        MNODE *l;       /* Points to element L_{L} */
        MNODE **d;      /* Vector of pointers into the M deque,
                           comprising each L deque */
        int rx;         /* Index of the deque's rightmost element */
        int lx;         /* Index of the deque's leftmost element */
} LDEQUE;

static LDEQUE *L;

                                /* Local procedures */
static void     qc_match_alt( int level, int matching[] );
static void     process_node( int matching[] );
static void     qc_match_alloc( int toursize );
static void    *qc_match_malloc( int bytes );
static void     push_L_right( int c, MNODE *de);
static void     push_M_right( int xx );
static void     pop_L_left( int c );
static int      pop_M_left( void );
static void     pop_L_right( int c );
static int      pop_M_right( void );
static void     match_pair( int matching[] );
static BOOLEAN  generic_omega( MNODE *iLrm, MNODE *iLr, MNODE *iMr,
                              void *input[], COSTFUNC cost, int psi);
```

This definition is continued in chunks 17, 20, 22, 26, and 28.
This code is written to file qcmatch.c.

We now present the source code for the matching function `qc_match`. It works by splitting the nodes into $\sim$-equivalence classes where each class represents a balanced, alternating-color tour. Each node is assigned to a unique class based on its `level` value, and then added to the tail of the linked list for that class. The heads of these lists are stored in `LevelHeads`, and a list is traversed by looking up each node's successor in `LevelLinks`. Since each node occurs in exactly one list, each node has exactly one successor, so a single array of "links" suffices. This "levelling process" is described in [1].

During the levelling process for line-like tours (specified with the `linelike` parameter to `qc_match`), any odd-size levels are forced to have even size through the creation of an "extra" node, as described in [1]. This allows `qc_match` to handle both unbalanced line-like tours, and non-bipartite line-like tours of odd length.

Once we have reduced the matching problem to a set of alternating subtours, the solution for each such subtour is found by `qc_match_alt`.

$\langle$*qcmatch.c* 15$\rangle$+$\equiv$

```
REAL
qc_match( int n, void *input[], int colors[], int matching[], BOOLEAN linelike,
                        COSTFUNC cost, OMEGAFUNC omega )
{
        int     x, d, level;
        int     min_level, max_level;


        /*  Step 1:  Initialization.  */

        LineLike = linelike;
        Cost     = cost;
        Omega    = omega ? omega : generic_omega;

        if (n > Max_node) {
                qc_match_free();
                qc_match_alloc(n);
        }

        memcpy(Nodes, input, n*sizeof(void *));
        Nodes[n] = NULL;        /* "Extra" node for any odd-length subtours */


        /*  Step 2:  Read input colors and break into levels. */

        d = 0;                  /* d(0) */
        min_level = INT_MAX;
        max_level = INT_MIN;
        memset((char *) LevelLinks,           -1, (  Max_node+1)*sizeof(int));
memset((char *)(LevelHeads-Max_node), -1, (2*Max_node+1)*sizeof(int));

for (x=0 ; x < n ; x++) {

    /* Calculate level(x), and then d(x+1) for next time. */

        if (colors[x] == BLUE) {
                level = d;
                d--;
        } else {        /* RED */
                level = d + 1;
                d++;
        }

    /* Keep high and low watermarks for level */

        if (level < min_level) {
                min_level = level;
        }
        if (level > max_level) {
                max_level = level;
        }

    /* Add node x to the tail of the linked list for its level */

        if (LevelHeads[level] == -1){
                LevelHeads[level] = x;
        } else {
                LevelLinks[LevelTails[level]] = x;
        }

        LevelTails[level] = x;
}

for (level = min_level; level <= max_level; level++) {

    /* If subtour is odd-length, add the "extra" node of index n
       to the level.  LevelLinks[n] will always be -1.
     */
        if (colors[LevelHeads[level]] == colors[LevelTails[level]]) {
                assert(LineLike);
                LevelLinks[LevelTails[level]] = n;
        }
}

/*  Step 3:  Do matching for each level.  */
```

```
        TotalCost = 0.0;

        for (level = min_level; level <= max_level; level++) {
                qc_match_alt(level, matching);
        }

        return TotalCost;
}
```

Here is the `qc_match_alt` function, which solves the matching problem for the alternating color subtour associated with `level`. It corresponds to Algorithm 2 of [1].

The subtour is retrieved by traversing the linked list corresponding to the `level` value.

Note that for line-like tours, the Buss-Yianilos algorithm's "second scan" through the tour is unnecessary and is therefore skipped altogether, thus improving performance. (The possibility of skipping the second scan for the line-like case is mentioned in [1], but not reflected in the pseudocode.)

⟨*qcmatch.c* 15⟩+≡

```
static void
qc_match_alt( int level, int matching[] )
{
                                /* "Initialization" */
        Psi = -1;

        M.lx = 0;
        M.rx = -1;

        L[RED].lx = 0;
        L[RED].rx = -1;

        L[BLUE].lx = 0;
        L[BLUE].rx = -1;

                                /* "Read Input into the M Deque" */
        Xn = LevelHeads[level];
        do {
                push_M_right(Xn);
                Psi = -Psi;
        }
        while ((Xn = LevelLinks[Xn]) != -1);

                                /* "The First Scan" */
        while ((L[Psi].rx < L[Psi].lx) || (M.l->x != L[Psi].l->x)) {
                Xn = pop_M_left();
                process_node(matching);
                push_L_right(Psi, M.r);
                Psi = -Psi;
        }

                                /* "The Second Scan" */
        if (!LineLike) {
                while ((L[RED].rx >= L[RED].lx) || (L[BLUE].rx >= L[BLUE].lx)) {
                        Xn = pop_M_left();
                        if (Xn == L[Psi].l->x) {
                                pop_L_left(Psi);
                        }

                        process_node(matching);
                        Psi = -Psi;
                }
        }

                                /* "Windup Processing" */
        while (M.lx <= M.rx) {
                match_pair(matching);
        }
}
```

The internal function process_node below corresponds directly to the pseudocode procedure Process_Node() in [1], with two exceptions.

First, [1] doesn't bother mentioning that one should check for at least two elements on the $\mathcal{L}$deques before executing the while loops that reference those elements.

Secondly, in the line-like case where we've added a node to a odd-length subtour to make the length even, we encounter the "extra" node only on the $\mathcal{M}$deque at the very end of the first scan. Since there is no second scan for line-like tours, we care about neither the value of the Ω predicate, nor the future integrity of the $\mathcal{L}$deques, so we just **return**. The main reason for the early **return** is that then the Ω function doesn't need to handle the case of the "special" extra node.

⟨*qcmatch.c* 15⟩+≡
```
  static void
  process_node( int matching[] )
  {
          push_M_right(Xn);

          if (L[-Psi].rx > L[-Psi].lx) {

                  while ((L[-Psi].rx > L[-Psi].lx) &&
                         ((COST(Nodes[L[-Psi].rm1->x], Nodes[M.r->x]) -
                           COST(Nodes[L[-Psi].r  ->x], Nodes[M.r->x])) <
                          (Psi*(L[-Psi].r->i - L[-Psi].rm1->i)))) {

                                  pop_L_right(-Psi);
                  }
          }

          if ((L[-Psi].rx >= L[-Psi].lx) &&
              (COST(Nodes[M.r->x], Nodes[L[-Psi].r->x]) <
               Psi*(M.r->i - L[-Psi].r->i))) {

                  Xn = pop_M_right();

                  while (M.r->x != L[-Psi].r->x) {
                          match_pair(matching);
                  }

                  push_M_right(Xn);
          }

          if (LineLike && Nodes[M.r->x] == NULL) {
                  return;
          }
```

```
                                                /* >= 2 elements there */
        while ((L[Psi].rx > L[Psi].lx) &&
               ((*Omega)(L[Psi].rm1, L[Psi].r, M.r, Nodes, Cost, Psi) == TRUE)) {

                pop_L_right(Psi);
        }
}

static void
push_L_right( int c, MNODE *de)
{
        L[c].d[++L[c].rx] = de;

        L[c].rm1 = L[c].r;
        L[c].r   = de;

        /* Arrange for "l" to be set after first push */

        if (L[c].rx  == L[c].lx) {
                L[c].l = de;
        }
}

static int
pop_M_left( void )
{
        ++M.lx;
        return (M.l++)->x;
}

static void
pop_L_left( int c )
{
        ++L[c].lx;

        L[c].l = L[c].d[L[c].lx];
}

static int
pop_M_right( void )
{
        --M.rx;
        return (M.r--)->x;
}

static void
pop_L_right( int c )
```

```
{
        --L[c].rx;

        L[c].r   = L[c].rm1;
        L[c].rm1 = L[c].d[L[c].rx-1];
}

/* This is the push_M_right() procedure from the paper */

static void
push_M_right( int x )
{
        REAL   i;

    /* Compute the "Improvement" function */

        if (M.rx >= M.lx) {
                i = M.r->i + Psi * COST(Nodes[M.r->x], Nodes[x]);
        } else {
                i = 0.0;
        }

    /* Push right the node and improvement value onto the MAIN deque */

        M.r = &M.d[++M.rx];
        M.r->x = x;
        M.r->i = i;

        if (M.rx == M.lx) {
                M.l = M.r;
        }
}

static void
match_pair( int matching[] )
{
        int     a, b;

        a = (M.r - 1)->x;
        b = pop_M_right();

        if (Nodes[a]) {
                matching[a] = Nodes[b] ? b : -1;
        }
        if (Nodes[b]) {
                matching[b] = Nodes[a] ? a : -1;
        }
```

```
                                /* The next line is not in the paper. It
                                 * accumulates the total cost.
                                 */
        TotalCost += COST(Nodes[a], Nodes[b]);

        if ((L[Psi].rx >= L[Psi].lx) && (M.r->x == L[Psi].r->x)) {
                pop_L_right(Psi);
        }

        pop_M_right();
}
```

The following three routines take care of allocating and releasing all variable-size working structures needed to process problems of up to `toursize` nodes.

⟨*qcmatch.c* 15⟩+≡
```
static void
qc_match_alloc( int toursize )
{
        Max_node = toursize;

        Nodes       = (void **) qc_match_malloc((Max_node+1) * sizeof(void *));

        LevelLinks  = (int *)   qc_match_malloc((  Max_node+1) * sizeof(int));
        LevelTails  = (int *)   qc_match_malloc((2*Max_node+1) * sizeof(int));
        LevelHeads  = (int *)   qc_match_malloc((2*Max_node+1) * sizeof(int));
        M.d        = (MNODE *)  qc_match_malloc(3 * Max_node * sizeof(MNODE));
        L          = (LDEQUE *) qc_match_malloc(3 * sizeof(LDEQUE));
        L[0].d     = (MNODE **) qc_match_malloc(3 * Max_node * sizeof(MNODE *));
        L[2].d     = (MNODE **) qc_match_malloc(3 * Max_node * sizeof(MNODE *));

        LevelTails += Max_node;
        LevelHeads += Max_node;
        L += 1;
}

void
qc_match_free( void )
{
        if (Max_node <= 0)
                return;

        L -= 1;
        LevelTails -= Max_node;
        LevelHeads -= Max_node;

        free(L[2].d);
        free(L[0].d);
        free(L);
        free(M.d);
        free(LevelHeads);
        free(LevelTails);
        free(LevelLinks);

        Max_node = 0;
}

static void *
qc_match_malloc( int bytes )
{
        void *p;

        p = malloc(bytes);

        if (p == NULL) {
                fprintf(stderr, "Cannot allocate enough memory!\n");
                exit(-1);
        }

        return(p);
}
```

The `generic_omega` function defined below uses a binary search to compare the relative positions of crossovers, yielding a worst-case $O(n \log n)$ time algorithm for quasi-convex matching. For special cases, it is often possible to substitute a constant-time Ω function to obtain a linear time algorithm for quasi-convex matching. Note that the test programs will ignore a customized Ω for unbalanced tour types (see description above of the `-t` test program option).

If you supply a customized Ω function, it should follow the same calling sequence as the generic function.

The first three parameters are of type `DEQNODE`. A `DEQNODE` is a structure with two fields: an integer `x`, which is a particular node's index in the `input` array, and a real number `i`, which is that node's "I" value, as described in [1].

`iLrm` The deque node $\mathcal{L}^{\psi}_{R-1}$.

`iLr` The deque node $\mathcal{L}^{\psi}_{R}$.

`iMr` The deque node $\mathcal{M}_R$.

`input` The array of input nodes.

`cost` The cost function originally passed to `qc_match`.

`psi` The value ψ of [1].

The return value of your `omega` function should be the (boolean) value of the Ω predicate as defined by [1].

$\langle$*qcmatch.c* 15$\rangle$+$\equiv$

```
static BOOLEAN
generic_omega( MNODE *iLrm, MNODE *iLr, MNODE *iMr, void *input[],
               COSTFUNC cost, int psi)
{
        int     ilow, ihigh, k;
        BOOLEAN pastxoverA, pastxoverB;
        void    *np_Lrm, *np_Lr, *np_Mr, *np_Mk;

        ilow = M.lx;
        ihigh = (iLrm - &M.d[0]) + 1;  /* Index of the node after iLrm */

                                /* Get the node pointers for the three nodes */
        np_Lrm = input[iLrm->x];
        np_Lr  = input[iLr->x];
        np_Mr  = input[iMr->x];

                                /* Search M-deque entries M[ilow] through
                                 * M[ihigh-2] to find which crossover comes
                                 * first.
                                 */
        while (ihigh > ilow + 1) {
                k = ilow + 2 * ((ihigh - ilow) >> 2);
                np_Mk = input[M.d[k].x];        /* Precompute M[k] */

                                /* The following boolean variable is TRUE
                                 * if M[k] is past the L(sub R-1)(sup psi),
                                 * L(sub R)(sup psi) crossover point.
                                 */
                pastxoverA = (COST(np_Lrm, np_Mk) - COST(np_Lr, np_Mk) <
                              psi * (iLrm->i - iLr->i));

                                /* The following boolean variable is TRUE
                                 * if M[k] is past the L(sub R)(sup psi),
                                 * M(sub R) crossover point.
                                 */
                pastxoverB = (COST(np_Lr, np_Mk) - COST(np_Mr, np_Mk) <
                              psi * (iLr->i - M.r->i));

                if (pastxoverA) {
                        if (pastxoverB) {
                                ihigh = k;
                        } else {
                                return TRUE;
                        }
                } else {
                        if (pastxoverB) {
                                return FALSE;
                        } else {
                                ilow = k + 2;
                        }
                }
        }
                                /* Returning either TRUE or FALSE at this
                                 * point will find a minimum-cost matching.
                                 */
        return TRUE;
}
```

11 APPENDIX 1: The Test Driver Module

⟨*qctest.c* 30⟩≡

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <assert.h>
#include <sys/time.h>
#include <sys/resource.h>
#include "qcmatch.h"

extern void     srand48(long);
extern long     lrand48(void);
extern double   drand48(void);
extern int      getrusage( int who, struct rusage *rusage );

#ifndef M_PI
#define M_PI    3.14159265358979323846
#endif

#define ROUNDOFF        .000005

#define MAX(A,B) ( ((A)>(B)) ? (A) : (B) )
#define MIN(A,B) ( ((A)<(B)) ? (A) : (B) )

                                /* Macros for returning a pseudo-random
                                 * integer between 0 and max-1.
                                 */
#define SRAND(s)        (srand48(s))
#define RAND(max)       (lrand48() % (max))

typedef struct {
        REAL    coord;
} NODE;

int             MaxNodes   = 128;
int             NumTours   = 1;
unsigned        RandomSeed = 102;

REAL    MaxCoord = 2.0*M_PI;

REAL    BYCost;                 /* Total cost according to B-Y algorithm */
REAL    DPCost;                 /* Total cost according to dynamic programming */

int     TourSize, TourExcess;
int     *Colors, *BYMatching, *Excess;
```

```
NODE    **Nodes;

REAL    **DPArray;              /* Dynamic programming array */
int     *DPMatching, *Verify;

int     Successes               = 0;
int     DifferentMatching       = 0;
int     Failures                = 0;
BOOLEAN MoreThanOneSolution;

char    TourType[4]        = "BRC";
BOOLEAN BalancedTours      = TRUE;
BOOLEAN AlternatingTours   = FALSE;
BOOLEAN LineLike           = FALSE;

BOOLEAN BenchmarkMode      = FALSE;
BOOLEAN DynamicProgramming = FALSE;
BOOLEAN GraphicsOutput     = FALSE;
BOOLEAN ForceGenericOmega  = FALSE;

char    *ProgName;

extern COSTFUNC         CostFunc;
extern OMEGAFUNC        OmegaFunc;

                                /* Local procedures */
void    process_options( int argc, char *argv[] );
void    random_test( void );
REAL    dynamic_program( void );
void    dynamic_output( int b, int e );
void    compare_results( void );
REAL    verify_matching( int matching[] );
void    print_matchings( void );
int     coordcmp( const void *n1, const void *n2 );
double  cpu_interval( void );
void *  get_mem( int bytes );

extern void     graphics_output( int, NODE **, int *, int * );

int
main( int argc, char *argv[] )
{
        setbuf(stdout, NULL);

        ProgName = argv[0];

        process_options(argc, argv);
        random_test();

        return 0;
}

void
process_options( int argc, char *argv[] )
{
        extern int      getopt(int argc, char *argv[], char *optstring);
        extern char     *optarg;

        int     opt, err = FALSE;

        while ((opt = getopt(argc, argv, "-BOdgs:t:n:r:")) != -1) {
                switch (opt) {
                case 'B':
                        BenchmarkMode = TRUE;
                        break;
                case 'O':
                        ForceGenericOmega = TRUE;
                        break;
                case 'd':
                        DynamicProgramming = TRUE;
                        break;
                case 'g':
                        GraphicsOutput = TRUE;
                        break;
                case 's':
                        MaxNodes = atoi(optarg);
                        if (MaxNodes <= 0) {
                                fprintf(stderr, "Bad -n value!\n");
                                err = TRUE;
                        }
                        break;
                case 't':
                        strcpy(TourType, optarg);
                        if (strlen(TourType) != 3) {
                                fprintf(stderr,
                                        "Tour type must have 3 letters\n");
                                err = TRUE;
                        }
                        break;
                case 'n':
                        NumTours = atoi(optarg);
                        if (NumTours <= 0) {
                                fprintf(stderr, "Bad -n value!\n");
                        err = TRUE;
                }
                break;
        case 'r':
                RandomSeed = (unsigned) atoi(optarg);
                break;
        case '?':
                err = TRUE;
                break;
        }
}

if (err) {
        fprintf(stderr, "Usage: %s [-BOdg] [-t {B|U}{R|A}{C|L}] ",
                ProgName);
        fprintf(stderr, "[-s #] [-n #] [-r #]\n");
        fprintf(stderr,
                "\t-B\t\tBenchmark/timing mode (overrides -d,-g)\n");
        fprintf(stderr,
                "\t-O\t\tForce use of generic omega function\n");
        fprintf(stderr,
                "\t-d\t\tVerify with dynamic programming\n");
        fprintf(stderr,
                "\t-g\t\tProduce graphical representation\n");
        fprintf(stderr,
                "\t-t xyz\t\tSpecify 3-letter tour type [BRC]\n");
        fprintf(stderr,
                "\t-s #\t\tSpecify maximum tour size [128]\n");
        fprintf(stderr,
                "\t-n #\t\tSpecify number of problem instances [1]\n");
        fprintf(stderr,
                "\t-r #\t\tSpecify seed for random generator [102]\n");
        exit(1);
}

                        /* Process problem type key */
switch (TourType[0]) {
case 'B':       BalancedTours = TRUE;           break;
case 'U':       BalancedTours = FALSE;          break;
default:        err = TRUE;                     break;
}

switch (TourType[1]) {
case 'R':       AlternatingTours = FALSE;       break;
case 'A':       AlternatingTours = TRUE;        break;
default:        err = TRUE;                     break;
}
```

```
        switch (TourType[2]) {
        case 'C':       LineLike = FALSE;               break;
        case 'L':       LineLike = TRUE;                break;
        default:        err = TRUE;                     break;
        }

        if (!BalancedTours) {
                if (!LineLike) {
                        fprintf(stderr, "Unbalanced tours must be line-like!\n");
                        err = TRUE;
                }

                if (OmegaFunc && !ForceGenericOmega) {
                        fprintf(stderr, "WARNING: Unbalanced tours must ");
                        fprintf(stderr, "use generic Omega!\n");
                        fprintf(stderr, "Forcing -O ...\n");
                        ForceGenericOmega = TRUE;
                }
        }

        if (err) {
                fprintf(stderr, "Illegal tour type\n");
                exit(1);
        }

                                /* Cancel -d and -g in benchmark mode */
        if (BenchmarkMode) {
                DynamicProgramming = FALSE;
                GraphicsOutput     = FALSE;
        }
}

void
random_test( void )
{
        int     node, tour, nblues;
        double  bias, avgtime;


        SRAND(RandomSeed);


    /* ALLOCATE MEMORY NEEDED FOR MAXIMUM TOUR SIZE */

        Colors     = get_mem(MaxNodes * sizeof(int));
        BYMatching = get_mem(MaxNodes * sizeof(int));
    Nodes = get_mem(MaxNodes*sizeof(NODE *));
    for (node=0 ; node < MaxNodes ; node++) {
            Nodes[node] = get_mem(sizeof(NODE));
    }

    if (DynamicProgramming) {
            DPMatching = get_mem( MaxNodes    * sizeof(int));
            Verify     = get_mem( MaxNodes    * sizeof(int));
            Excess     = get_mem((MaxNodes+1) * sizeof(int));

            DPArray = get_mem((MaxNodes+3)*sizeof(REAL *));
            for (node=0 ; node < MaxNodes+3 ; node++) {
                    DPArray[node] = get_mem((MaxNodes+3)*sizeof(REAL));
            }
    }


/* SOLVE MATCHING FOR NumTours PROBLEM INSTANCES */

    avgtime = 0.0;

    for (tour=0 ; tour < NumTours ; tour++) {

        /* PICK PROBLEM SIZE */

            if (BenchmarkMode) {
                            /* Force single tour size */
                    TourSize = MaxNodes;

            } else {
                    if (BalancedTours) {
                            /* Any even number from 0 to 2*(MaxNodes/2) */
                            TourSize = 2 * RAND(MaxNodes/2 + 1);

                    } else {
                            /* Any number from 0 to MaxNodes */
                            TourSize = RAND(MaxNodes+1);
                    }
            }

        /* DISTRIBUTE COLORS */

            if (BalancedTours) {

                            /* First make all nodes red */
                    for (node=0 ; node < TourSize ; node++) {
```

```
                    Colors[node] = RED;
            }

                    /* Alternating: Make nodes 1,3,... blue */
            if (AlternatingTours) {
                    for (node=1 ; node < TourSize ; node+=2) {
                            Colors[node] = BLUE;
                    }

                    /* Non-alternating: Make a random half blue */
            } else {
                    nblues = 0;
                    while (nblues < TourSize / 2) {
                            node = RAND(TourSize);

                            if (Colors[node] == RED) {
                                    Colors[node] = BLUE;
                                    nblues++;
                            }
                    }
            }

    } else {
                    /* Pick colors by flipping a biased coin */
            bias = drand48();

            for (node=0 ; node < TourSize ; node++) {
                    Colors[node] = (drand48() <= bias) ? RED : BLUE;
            }
    }

/* DISTRIBUTE NODES */

                    /* If line-like, restrict circular tours
                     * to the semicircle [0, pi).
                     */
    if (LineLike) {
            MaxCoord = M_PI;
    }
                    /* Distribute nodes randomly in the half-open
                     * interval [0,MaxCoord), with resolution 32000.
                     */
    for (node=0 ; node < TourSize ; node++) {
            Nodes[node]->coord = RAND(32000)*(MaxCoord/32000.0);
    }

    qsort(Nodes, TourSize, sizeof(NODE *), coordcmp);
```

```
      /* RUN BUSS-YIANILOS ALGORITHM */

        if (BenchmarkMode) {
            (void) cpu_interval();
            BYCost = qc_match(TourSize, (void **)Nodes, Colors,
                     BYMatching, LineLike, CostFunc,
                     ForceGenericOmega ? NULL : OmegaFunc);
            avgtime += (cpu_interval() / NumTours);

        } else {
            BYCost = qc_match(TourSize, (void **)Nodes, Colors,
                     BYMatching, LineLike, CostFunc,
                     ForceGenericOmega ? NULL : OmegaFunc);
        }

      /* CHECK RESULTS WITH DYNAMIC PROGRAM */

        if (BalancedTours) {
            assert(TourExcess == 0);
        }

        if (DynamicProgramming) {
            DPCost = dynamic_program();
            compare_results();
        }

      /* DO GRAPHIC OUTPUT */

        if (GraphicsOutput) {
            graphics_output(TourSize, Nodes, Colors, BYMatching);
        }
    }

    if (BenchmarkMode) {
        printf("%6d %10.6f\n", TourSize, avgtime);
    }

    if (DynamicProgramming) {
        fprintf(stderr,
            "\tSuccesses:\t%u (%u with different matchings)\n",
            Successes, DifferentMatching);
        fprintf(stderr, "\tFailures:\t%u\n", Failures);
    }
}
```
This definition is continued in chunks 38 and 41.
This code is written to file `qctest.c`.

We now describe briefly our dynamic programming algorithm for bipartite matching.

We wish to build up a minimum-cost solution to a particular matching problem with `TourSize` nodes. We do this by finding solutions for all "contiguous" subtours, beginning with the trivial case of 1-node subtours. The number of such trivial subtours is `TourSize`. Next we find solutions for all 2-node subtours (of which there are `Toursize-1`), then for all 3-node subtours, etc. At each stage, a solution for each subproblem is computed in terms of solutions for already-solved "lesser" subproblems.

The algorithm consists of three nested loops. The outer and middle loops determine a particular subproblem $[x, y]$ in terms of size and position within the overall tour. The subproblem sizes increase from `1` to `TourSize` nodes, and all subproblem positions within the overall tour are considered. Obviously, only one position is possible for the largest size, and the minimum-cost matching for this problem is our final solution.

The index x' of the innermost loop ranges over all nodes in $[x, y]$ that could possibly be matched with the first node x (including x itself, if x is possibly *unmatched* within an unbalanced subproblem $[x, y]$). Clearly, for at least one such x', the edge $x \leftrightarrow x'$ must be part of a minimum-cost solution for $[x, y]$. In addition, we know by Lemma 4 of [1] that there exists a minimum-cost matching for $[x, y]$ with no crossed edges. Let $\mathrm{edgecost}(x \leftrightarrow x')$ be the cost of the edge $x \leftrightarrow x'$. Let $\mathrm{mincost}(I)$ be the minimum cost for the subsubproblem $I = [x+1, x'-1]$ "interior" to this edge, and let $\mathrm{mincost}(E)$ be the minimum cost for the subsubproblem $E = [x'+1, y]$ "exterior" to this edge. Being smaller problems, I and E have already been solved. Since we can ignore all matchings with edges that would cross $x \leftrightarrow x'$, the minimum cost for $[x, y]$ must be equal to:

$$\min_{x'} [\mathrm{edgecost}(x \leftrightarrow x') + \mathrm{mincost}(I) + \mathrm{mincost}(E)]$$

Not every x' is a valid candidate for matching with x. It is obvious that $x' = x$ (corresponding to unmatched x) is valid only if $[x, y]$ is unbalanced in favor of the color of x, and that for all other x', x and x' must be of opposite colors. But in order to ensure a *maximal* matching for $[x, y]$, we must also disallow any x' for which a maximal matching would *require* crossover jumpers from I to E. This corresponds to the case where I and E are unbalanced in favor of *opposite* colors.

Minimum-cost solutions to subproblems are stored in the 2-D array `DPArray`, which is always accessed through the `MATCHCOST()` macro. `MATCHCOST(x,y)` represents the minimum-cost maximal matching for the subproblem $[x, y]$.

Other macros of interest are `SUBPROB(x,y)`, which tells whether the subproblem $[x, y]$ exists (i.e. has length greater than zero), and `EXCESSCOL(x,y)`, which returns the color in excess over the interval $[x, y]$ (zero if $[x, y]$ is balanced).

⟨*qctest.c* 30⟩+≡

```
#define SGN(x)                  ((x) ? (((x) > 0) ? 1 : -1) : 0)
#define SUBPROB(x,y)            ((x) <= (y))
#define EXCESS(x,y)             (Excess[(y)+1] - Excess[(x)])
#define EXCESSCOL(x,y)          (SGN(EXCESS((x),(y))))
#define BALANCED(x,y)           (EXCESS((x),(y)) == 0)

#define EDGECOST(x,y)           ((*CostFunc)(Nodes[(x)],Nodes[(y)]))
#define MATCHCOST(x,y)          (DPArray[(x)+1][(y)+1])

REAL
dynamic_program( void )
{
        int     i, isize, x, y, xp;
        BOOLEAN interior, exterior;
        REAL    mincost, cost;

                                /* Calculate the balance of the tour */
        Excess[0]  = 0;
        for (i=0 ; i < TourSize ; i++) {
                Excess[i+1] = Excess[i] + Colors[i];
        }
        TourExcess = Excess[TourSize];

                                /* "Null intervals" occurring in the algorithm
                                 * have the form [x,x-1].  MATCHCOST() should
                                 * return zero cost in such cases.
                                 */
        for (i=0 ; i <= TourSize ; i++){
                MATCHCOST(i,i-1) = 0.0;
        }


    /* CONSIDER ALL SUBPROBLEM SIZES (1, 2, 3, 4, ..., TourSize) */

        for (isize=1 ; isize <= TourSize ; isize++) {

            /* CONSIDER SUBPROBLEM INTERVAL AT ALL POSSIBLE POSITIONS [x,y] */

                for (x=0, y=x+isize-1 ; y < TourSize ; x++, y++) {

                    /* CONSIDER ALL MAXIMAL MATCHINGS FOR SUBPROBLEM [x,y]. */

                        mincost = HUGE_VAL;

                                /* If [x,y] is unbalanced in favor of x's color,
                                 * we have to consider the case of unmatched x.
                                 */
                        if (Colors[x] == EXCESSCOL(x,y)) {
                                mincost = MATCHCOST(x+1,y);
                        }

                                /* Loop over all possible matchings for x */
                        for (xp=x+1 ; xp <= y ; xp++) {

                                if (Colors[x] == Colors[xp]) {
                                        continue;
                                }

                                interior = SUBPROB(x+1,xp-1);
                                exterior = SUBPROB(xp+1,y);

                                if (interior && exterior &&
                                    !BALANCED(x+1,xp-1) &&
                                    EXCESSCOL(x+1,xp-1) == -EXCESSCOL(xp+1,y)) {
                                        continue;
                                }

                                cost = EDGECOST(x, xp);
                                if (interior) {
                                        cost += MATCHCOST(x+1, xp-1);
                                }
                                if (exterior) {
                                        cost += MATCHCOST(xp+1, y);
                                }

                                mincost = MIN(mincost, cost);
                        }

                        MATCHCOST(x,y) = mincost;
                }
        }

        MoreThanOneSolution = FALSE;

        dynamic_output(0,TourSize-1);

                                /* Return cost for complete tour interval */
        return MATCHCOST(0,TourSize-1);
}
```

Rather than maintaining explicit backpointers in the dynamic program, we construct a solution by executing the innermost loop again, finding the jumpers by comparing the costs stored in the dynamic programming array.

⟨*qctest.c* 30⟩+≡
```
void
dynamic_output( int b, int e )          /* Begin, end */
{
        BOOLEAN jumper;
        int     c;


    /* CASE 1: Null problem */
        if (b > e) {
                return;
        }

    /* CASE 2: Single unmatchable point */

        if (b == e) {
                DPMatching[b] = -1;
                return;
        }

    /* CASE 3: Adjacent pair (with possibly identical coordinates) */

        if (e == b+1) {
                if (Colors[b] == Colors[e]) {
                        DPMatching[b] = -1;
                        DPMatching[e] = -1;
                } else {
                        DPMatching[e] = b;
                        DPMatching[b] = e;
                }
                return;
        }

    /* CASE 4: Arbitrary subproblem of length > 2 */

        jumper = TRUE;

                                /* Either b and e are matched with a jumper, or
                                 * there is some c such that the cost of
                                 * subproblem [b,c] plus the cost of subproblem
                                 * [c+1,e] equals the cost of problem [b,e].
                                 * Try to find that c.
                                 */
        for (c=b ; c < e ; c++) {
```

```
                                /* Make sure that the division of the problem
                                 * at c does not imply a non-maximal matching.
                                 */
                if (!BALANCED(b,c) && EXCESSCOL(b,c) == -EXCESSCOL(c+1,e)) {
                        continue;
                }

                if (fabs(MATCHCOST(b,e) - (MATCHCOST(b,c) + MATCHCOST(c+1,e)))
                    <= ROUNDOFF) {

                                /* First valid c: Recurse for the subproblems */
                        if (jumper) {
                                dynamic_output(b, c);
                                dynamic_output(c+1, e);
                                jumper = FALSE;

                                /* Succeeding valid c's: Just set flag to
                                 * indicate more than one solution.
                                 */
                        } else {
                                MoreThanOneSolution = TRUE;
                        }
                }
        }

        if (jumper) {
                                /* If no c was found for this b and e, then
                                 * b and e must be matched.  Record this
                                 * matching in the matching array...
                                 */
                DPMatching[e] = b;
                DPMatching[b] = e;

                                /* We know there are intervening nodes, so
                                 * recurse on that subproblem.
                                 */
                dynamic_output(b+1, e-1);
        }
}

void
compare_results( void )
{
        int     kk;
        BOOLEAN costs_same, outputs_same;
        REAL    byver, dpver;
        byver = verify_matching(BYMatching);
        dpver = verify_matching(DPMatching);

        outputs_same = TRUE;
        for (kk = 0; kk < TourSize; kk++) {
                if (BYMatching[kk] != DPMatching[kk]) {
                        outputs_same = FALSE;
                }
        }

        costs_same = (fabs(BYCost - DPCost) < 20*ROUNDOFF);

        if (costs_same) {
                Successes++;
                if (!outputs_same) {
                        DifferentMatching++;
                }

        } else {
                Failures++;
                if (outputs_same) {
                        printf("COST MISMATCH: Same matchings ???\n");
                } else {
                        printf("COST MISMATCH: Different matchings\n");
                }

                print_matchings();
                printf("B-Y Cost: %10f (verify=%f)  DP Cost: %10f (verify=%f)\n",
                       BYCost, byver, DPCost, dpver);

                if (MoreThanOneSolution == TRUE) {
                        printf("More than one best solution\n");
                }

                printf("\n");
        }
}

REAL
verify_matching( int matching[] )
{
        int     i, npairs, excess;
        REAL    total_cost;

                                /* Copy matching into another buffer */
        for (i=0 ; i < TourSize ; i++) {
                Verify[i] = matching[i];
        }

        total_cost = 0.0;
        npairs     = 0;
        excess     = 0;

        for (i=0 ; i < TourSize ; i++) {

                switch (Verify[i]) {

                case -2:                /* Already-paired node */
                        continue;

                case -1:                /* Unmatched node */
                        assert(SGN(Colors[i]) == SGN(TourExcess));
                        excess += Colors[i];
                        continue;

                default:                /* First node of a matched pair */
                        assert(Verify[Verify[i]] == i);

                        npairs++;
                        total_cost += (*CostFunc)(Nodes[i], Nodes[Verify[i]]);

                        Verify[Verify[i]] = -2;
                        Verify[i]         = -2;

                        break;
                }
        }

        assert(excess == TourExcess);
        assert(npairs*2 + SGN(excess)*excess == TourSize);

        return total_cost;
}

void
print_matchings( void )
{
        int     jjj, jkj;


        printf("Num nodes = %d\n", TourSize);
        jjj = 0;
        jkj = 0;

        while (1) {
                while (jjj < TourSize && jjj > BYMatching[jjj]){
                        jjj++;
                }

                while (jkj < TourSize && jkj > DPMatching[jkj]) {
                        jkj++;
                }

                if (jjj >= TourSize || jkj >= TourSize) {
                        break;
                }

                printf("%10f  %10f    %10f  %10f\n", Nodes[jjj]->coord,
                       Nodes[BYMatching[jjj]]->coord, Nodes[jkj]->coord,
                       Nodes[DPMatching[jkj]]->coord);

                jjj++;
                jkj++;
        }
}

int
coordcmp( const void *n1, const void *n2 )
{
        REAL    diff;

        diff = (*(NODE **)n1)->coord - (*(NODE **)n2)->coord;

        return (diff > 0.0) ? 1 : ((diff < 0.0) ? -1 : 0);
}

double
cpu_interval( void )
{
        static double cputime;
        double thetime, theinterval;
        struct rusage rusage;

    /* Return CPU time in seconds since last call */

        getrusage(RUSAGE_SELF, &rusage);

        thetime = rusage.ru_utime.tv_sec + rusage.ru_utime.tv_usec/1E6;
        theinterval = thetime - cputime;
        cputime = thetime;

        return(theinterval);
}

void *
get_mem( int bytes )
{
        void    *p;

        if (bytes > 0 && (p = malloc(bytes))) {
                return p;
        } else {
                fprintf(stderr, "%s: malloc failed\n", ProgName);
                exit(1);
        }
}
```

12 APPENDIX 2: Testing Circular Tours

In order to test circular tours (or tours on the line-like semicircular arc $[0, \pi)$), the test driver module `qctest.c` must be linked with either `qcsqrtarc.c` or `qcarc.c`, and also with `qcgrapharc.c`. The source listings follow for these three files.

The legal tour types (`-t` command-line option) for testing full circular tours are `BRC` and `BAC`. Unbalanced tours cannot be solved on the full circle, since the full circle is not line-like. If the test program is run with tour types `BRL`, `BAL`, `URL`, or `UAL`, the test driver will automatically restrict the nodes to lie on the semicircular arc $[0, \pi)$.

The file `qcsqrtarc.c` contains a cost function for circular tours that returns the square root of the angular distance between two nodes.

$\langle$*qcsqrtarc.c* 47$\rangle \equiv$

```
#include <stdio.h>
#include <math.h>
#include "qcmatch.h"

#ifndef M_PI
#define M_PI    3.14159265358979323846
#endif

typedef struct {
        REAL    coord;
} NODE;

REAL    sqrtcost( void *nx, void *ny );

COSTFUNC        CostFunc  = sqrtcost;
OMEGAFUNC       OmegaFunc = NULL;

REAL
sqrtcost( void *nx, void *ny )
{
        REAL   c;

                              /* Inputs are real numbers between 0 and 2*PI */
        c = fabs(((NODE *)nx)->coord - ((NODE *)ny)->coord);

        if (c > M_PI) {
                c = 2 * M_PI - c;
        }

        return sqrt(c);
}
```

This code is written to file `qcsqrtarc.c`.

The file `qcarc.c` contains a cost function for circular tours that returns the Euclidean distance (chord length) between two nodes. It also contains a constant-time Ω function implementation.

Both of these functions perform their computations in terms of ordinary x-y coordinates. Since the test driver generates random problem instances in terms of angular node coordinates (in radians), both functions must convert these to rectangular coordinates. The trigonometry involved imposes a significant performance penalty, which could be eliminated by defining the nodes initially in terms of rectangular coordinates.

⟨*qcarc.c* 49⟩≡

```
#include <stdio.h>
#include <math.h>
#include <assert.h>
#include "qcmatch.h"

#ifndef M_PI
#define M_PI    3.14159265358979323846
#endif

typedef struct {
        REAL    coord;
} NODE;

REAL    euclid_cost( void *nx, void *ny );

BOOLEAN
Circle_Omega ( MNODE *x1, MNODE *x2, MNODE *x3, void **input, COSTFUNC cost,
        int psi );

static BOOLEAN Circle_Get_Xover( REAL x1, REAL y1, REAL x2, REAL y2,
                REAL delta, REAL *x3, REAL *y3 );

COSTFUNC        CostFunc  = euclid_cost;
OMEGAFUNC       OmegaFunc = Circle_Omega;


REAL
euclid_cost( void *nx, void *ny )
{
        REAL   x1, x2, y1, y2;

        x1 = cos(((NODE *)nx)->coord);
        y1 = sin(((NODE *)nx)->coord);

        x2 = cos(((NODE *)ny)->coord);
```

```
          y2 = sin(((NODE *)ny)->coord);

          return(sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2)));
  }
```

This definition is continued in chunks 51 and 53.
This code is written to file `qcarc.c`.

The two functions that follow implement a constant-time Ω for points on the unit circle under Euclidean distance. After converting to x-y coordinates, **Circle_Omega** calls **Circle_Get_Xover** twice to compute the two theoretical crossovers y and z of the paper. This routine returns **FALSE** if none exist. **Circle_Omega** handles this case specially in accordance with the paper. Once the crossovers have been computed, their counterclockwise ordering is compared – where the middle point **x2** defines the tour's start. This is accomplished by coordinate rotation, after which four cases suffice to complete the determination.

$\langle$*qcarc.c* 49$\rangle$+$\equiv$

```
BOOLEAN
Circle_Omega ( MNODE *x1, MNODE *x2, MNODE *x3, void **input, COSTFUNC cost,
        int psi )
{
        REAL a1,b1;     /* x-y coordinates of MNODE x1 */
        REAL a2,b2;     /* x-y coordinates of MNODE x2 */
        REAL a3,b3;     /* x-y coordinates of MNODE x3 */
        REAL a4,b4;     /* x-y coordinates of crossover of x1 and x2 */
        REAL a5,b5;     /* x-y coordinates of crossover of x1 and x2 */

        REAL c4,d4;     /* x-y coordinates for rotation of (a4,b4) */
        REAL c5,d5;     /* x-y coordinates for rotation of (a5,b5) */

        REAL del, eps;  /* Delta and Epsilon values for crossovers */

        BOOLEAN yexist; /* Does the first crossover exist? */
        BOOLEAN zexist; /* Does the second crossover exist? */


        a1 = cos(((NODE **)input)[x1->x]->coord);
        b1 = sin(((NODE **)input)[x1->x]->coord);
        a2 = cos(((NODE **)input)[x2->x]->coord);
        b2 = sin(((NODE **)input)[x2->x]->coord);
        a3 = cos(((NODE **)input)[x3->x]->coord);
        b3 = sin(((NODE **)input)[x3->x]->coord);

        del = psi * (x1->i - x2->i);
        eps = psi * (x2->i - x3->i);

        yexist = Circle_Get_Xover(a1, b1, a2, b2, del, &a4, &b4);
        zexist = Circle_Get_Xover(a2, b2, a3, b3, eps, &a5, &b5);

        if ( yexist && !zexist)
                return TRUE;

        if (!yexist &&  zexist)
                return FALSE;
        if (!yexist || !zexist)
                return FALSE;

        c4 = a4*a2 + b4*b2;
        d4 = b4*a2 - a4*b2;
        c5 = a5*a2 + b5*b2;
        d5 = b5*a2 - a5*b2;


        if (d4 >= 0) {
                return (d5 < 0) ? TRUE : (c5 < c4);

        } else {
                return (d5 >= 0) ? FALSE : (c4 < c5);
        }
}
```

The function `Circle_Get_Xover` finds the δ-crossover point of two points $(x1, y1)$ and $(x2, y2)$ on the unit circle. The cost between points is Euclidean distance (chord length).

This amounts to solving for the intersection of a hyperbola with the unit circle. The solution is straightforward, but several numerical details must be dealt with to produce a robust implementation. First, `hip` may slightly exceed its theoretical range of $[-0.5, 0.5]$, due to round-off error. Our code therefore clamps it to this range. Later, a degeneracy involving `asqr` and `csqr` must be dealt with to avoid a different problem.

Assertions have been left in the code (commented out) to verify that computed solutions lie both on the circle and the hyperbola.

$\langle$*qcarc.c* 49$\rangle$+$\equiv$

```
#define norm(a,b) (sqrt((a)*(a)+(b)*(b)))

BOOLEAN
Circle_Get_Xover( REAL x1, REAL y1, REAL x2, REAL y2, REAL delta,
                  REAL *x3, REAL *y3 )
{
        REAL a;                 /* Half of delta */
        REAL c;                 /* Half of distance from (x1,y1) to (x2,y2) */
        REAL d;                 /* Distance from origin to line through
                                   (x1,y1) and (x2,y2) */
        REAL u,v;               /* (u,v) coordinates of the crossover point */
        REAL alpha,beta;        /* Midpt between (x1,y1) and (x2,y2) on circle */


        REAL hip;               /* Half of inner product */
        REAL asqr, csqr;        /* Intermediate values, for clarity */


        a = 0.5 * delta;

        hip = (x1*x2 + y1*y2)/2;
        if (hip > 0.5) {
                hip = 0.5;
        } else if (hip < -0.5) {
                hip = -0.5;
        }

        c = sqrt(0.5 - hip);
        d = sqrt(0.5 + hip);

        if (-x2*y1 + x1*y2 < 0) {
                d = -d;
        }
        asqr = a*a;
        csqr = c*c;

        if (asqr >= csqr) {
                if ((-a) >= 0) {        /* if (-a >= c) */
                        *x3 = x1;
                        *y3 = y1;
                        return FALSE;
                } else {                /* if ( a >= c) */
                        *x3 = x2;
                        *y3 = y2;
                        return TRUE;
                }
        }

        u = -(1+d) * (1-(asqr/csqr));
        v = a * sqrt(1+(u*u)/(csqr-asqr));

        if (hip > 0) {
                alpha = (x1 + x2)/(2*d);
                beta  = (y1 + y2)/(2*d);
        } else {
                alpha = (y2 - y1)/(2*c);
                beta  = (x1 - x2)/(2*c);
        }

        *x3 = (u+d)*alpha -     v*beta;
        *y3 =     v*alpha + (u+d)*beta;

        /*
        assert(fabs(norm(*x3,*y3) - 1.0) < 1E-6);
        assert(fabs(-norm(*x3-x1,*y3-y1) + norm(*x3-x2,*y3-y2) + delta) < 1E-6);
        */

        return TRUE;
}
```

The file `qcgrapharc.c` contains the function `graphics_output` for circular tours that writes a PostScript figure for each tour to standard output. It assumes that each node is specified by an angular coordinate in radians (in the range $[0, 2\pi]$).

⟨*qcgrapharc.c* 55⟩≡

```
#include <stdio.h>
#include <math.h>
#include "qcmatch.h"

#ifndef M_PI
#define M_PI    3.14159265358979323846
#endif

#define NODESIZE         .01    /* Size of the circles used to represent nodes */
#define CENTX           0.5     /* The x value of the center of the circle */
#define CENTY           0.5     /* The y value of the center of the circle */
#define CIRRAD          0.5     /* The radius of the coordinate circle */

#define ROUNDOFF         .000005
#define BIGSLOPE  1000000.0;

typedef struct {
        REAL    coord;
} NODE;

void
graphics_output( int n, NODE *nodes[], int colors[], int matching[] )
{
        int     kj;
        double  x1, x2, y1, y2; /* The coordinates of the nodes */
        double  x3, y3;         /* The coordinates of the intersection of the
                                 * tangents. */
        double  radius;         /* The radius of an edge arc. */
        double  th1, th2;       /* The angles of the 2 nodes. */
        double  m1, m2;         /* The slopes of the tangents. */
        double  a1, a2;         /* Limits of the angle swept out by an arc. */


        printf("\nQCbegin\n");
        printf("PROOF\n");
                                /* Draw the coordinate circle */
        printf("%6f  %6f  %6f  CIRCLE\n", CENTX, CENTY, CIRRAD);

                                /* Draw the edge arcs */
        for (kj = 0; kj < n; kj++) {
                /* Unmatched or already-matched nodes */
if (matching[kj] < 0 || kj >= matching[kj]) {
        continue;
}

                /* Get the two angles in more convenient form. */
th1 = nodes[kj]->coord;
th2 = nodes[matching[kj]]->coord;

                /* Identical coordinates: draw no arc */
if (th1 == th2) {
        continue;
}

                /* Get coordinates of the two nodes. */
x1 = CENTX + CIRRAD * cos(th1);
y1 = CENTY + CIRRAD * sin(th1);
x2 = CENTX + CIRRAD * cos(th2);
y2 = CENTY + CIRRAD * sin(th2);

                /* If points are not 180 degrees apart */
if (fabs((th2-th1) - M_PI) >= ROUNDOFF) {

                        /* Get slopes of the tangent lines.*/
        m1 = tan(th1) ? -1/tan(th1) : BIGSLOPE;
        m2 = tan(th2) ? -1/tan(th2) : BIGSLOPE;

                        /* Get coordinates of the intersection
                         * of the tangents.
                         */
        x3 = (y2 + m1*x1 - (y1 + m2*x2))/(m1-m2);
        y3 = m1 * (x3 - x1) + y1;

                        /* Get the radius of the arc. */
        radius = pow((y3-y1)*(y3-y1) + (x3-x1)*(x3-x1), 0.5);

                        /* Get the arc angles. */
        if (th2-th1 < M_PI) {   /* Sweep from th2 to th1. */
                a1 =  90 + (th2 * 180 / M_PI);
                a2 = 270 + (th1 * 180 / M_PI);

        } else {                /* Sweep from th1 to th2. */
                a1 =  90 + (th1 * 180 / M_PI);
                a2 = -90 + (th2 * 180 / M_PI);
        }

        printf("%6f  %6f  %6f   %6f   %6f   VARARC\n",
                              x3, y3, radius, a1, a2);

                } else {        /* 180 degs apart: join with line segment. */
                        printf("%6f  %6f  %6f  %6f   LINE\n", x1, y1, x2, y2);
                }
        }

                                /* Draw the node circles */
        for (kj = 0; kj < n; kj++) {

                printf("%6f  %6f  %6f  %sCIRCLE\n",
                       CENTX + CIRRAD * cos(nodes[kj]->coord),
                       CENTY + CIRRAD * sin(nodes[kj]->coord),
                       NODESIZE,
                       (colors[kj] == BLUE) ? "H" : "S");
        }

        printf("QCend\n");
}
```
This code is written to file `qcgrapharc.c`.

For didactic purposes, we also include the following alternative version of `qcarc.c`, which contains an Ω function that matches more closely the pseudocode in the theoretical paper.

⟨*qcarc.journal.c* 58⟩≡

```
#include <stdio.h>
#include <math.h>
#include <assert.h>
#include "qcmatch.h"

#ifndef M_PI
#define M_PI    3.14159265358979323846
#endif

typedef struct {
        REAL    coord;
} NODE;

REAL    euclid_cost( void *nx, void *ny );

BOOLEAN
Circle_Omega ( MNODE *x1, MNODE *x2, MNODE *x3, void **input, COSTFUNC cost,
        int psi );

static void Circle_Get_Xover( REAL x1, REAL y1, REAL x2, REAL y2,
                REAL delta, REAL *x3, REAL *y3 );

COSTFUNC        CostFunc  = euclid_cost;
OMEGAFUNC       OmegaFunc = Circle_Omega;

REAL
euclid_cost( void *nx, void *ny )
{
        REAL   x1, x2, y1, y2;

        x1 = cos(((NODE *)nx)->coord);
        y1 = sin(((NODE *)nx)->coord);

        x2 = cos(((NODE *)ny)->coord);
        y2 = sin(((NODE *)ny)->coord);

        return sqrt((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2));
}

BOOLEAN
Circle_Omega ( MNODE *x1, MNODE *x2, MNODE *x3, void **input, COSTFUNC cost,
        int psi )
```

```
{
        REAL a1,b1;     /* x-y coordinates of MNODE x1 */
        REAL a2,b2;     /* x-y coordinates of MNODE x2 */
        REAL a3,b3;     /* x-y coordinates of MNODE x3 */
        REAL a4,b4;     /* x-y coordinates of crossover of x1 and x2 */
        REAL a5,b5;     /* x-y coordinates of crossover of x1 and x2 */

        REAL c4,d4;     /* x-y coordinates for rotation of (a4,b4) */
        REAL c5,d5;     /* x-y coordinates for rotation of (a5,b5) */

        REAL del, eps;  /* Delta and Epsilon values for crossovers */


        a1 = cos(((NODE **)input)[x1->x]->coord);
        b1 = sin(((NODE **)input)[x1->x]->coord);
        a2 = cos(((NODE **)input)[x2->x]->coord);
        b2 = sin(((NODE **)input)[x2->x]->coord);
        a3 = cos(((NODE **)input)[x3->x]->coord);
        b3 = sin(((NODE **)input)[x3->x]->coord);

        del = psi * (x1->i - x2->i);
        eps = psi * (x2->i - x3->i);

        Circle_Get_Xover(a1, b1, a2, b2, del, &a4, &b4);
        Circle_Get_Xover(a2, b2, a3, b3, eps, &a5, &b5);


        /* Rotate the crossover points w.r.t. MNODE x3, and then determine
         * the relative ordering of x3 and the two crossover points.  For
         * this, interpret crossover (a4,b4)=(c4,d4) as occurring in the
         * interval [x2,x1] and the second crossover (a5,b5)=(c5,d5) as
         * occurring in the interval [x3,x2].
         */
        c4 = a4*a2 + b4*b2;
        d4 = b4*a2 - a4*b2;
        c5 = a5*a2 + b5*b2;
        d5 = b5*a2 - a5*b2;

        if (d4 >= 0) {
                return (d5 <= 0) ? TRUE : (c5 <= c4);

        } else {
                return (d5 > 0) ? FALSE : (c4 <= c5);
        }
}

#define norm(a,b) (sqrt((a)*(a) + (b)*(b)))
```

```
void
Circle_Get_Xover( REAL x1, REAL y1, REAL x2, REAL y2, REAL delta,
                  REAL *x3, REAL *y3 )
{
        REAL a;                 /* Half of delta */
        REAL c;                 /* Half of distance from (x1,y1) to (x2,y2) */
        REAL d;                 /* Distance from origin to line through
                                   (x1,y1) and (x2,y2) */
        REAL u,v;               /* (u,v) coordinates of the crossover point */
        REAL alpha,beta;        /* Midpt between (x1,y1) and (x2,y2) on circle */


        REAL hip;               /* Half of inner product */
        REAL asqr, csqr;        /* Intermediate values, for clarity */


        a = 0.5 * delta;

        hip = (x1*x2 + y1*y2)/2;
        if (hip > 0.5) {
                hip = 0.5;
        } else if (hip < -0.5) {
                hip = -0.5;
        }

        c = sqrt(0.5 - hip);
        d = sqrt(0.5 + hip);

        if (-x2*y1 + x1*y2 < 0) {
                d = -d;
        }

        asqr = a*a;
        csqr = c*c;

        if (asqr > csqr) {
                if ((-a) >= 0) {        /* if (-a >= c) */
                        *x3 = x1;
                        *y3 = y1;

                } else {                /* if ( a >= c) */
                        *x3 = x2;
                        *y3 = y2;
                }
                return;
        }
        u = -(1+d) * (1-(asqr/csqr));
        v = (asqr == csqr) ? a : (a * sqrt(1+(u*u)/(csqr-asqr)));

        if (hip > 0) {
                alpha = (x1 + x2)/(2*d);
                beta  = (y1 + y2)/(2*d);
        } else {
                alpha = (y2 - y1)/(2*c);
                beta  = (x1 - x2)/(2*c);
        }

        *x3 = (u+d)*alpha -     v*beta;
        *y3 =     v*alpha + (u+d)*beta;

        /*
        assert(fabs(norm(*x3,*y3) - 1.0) < 1E-6);
        assert(fabs(-norm(*x3-x1,*y3-y1) + norm(*x3-x2,*y3-y2) + delta) < 1E-6);
        */
}
```

This code is written to file `qcarc.journal.c`.

13 APPENDIX 3: Testing Linear Tours

In order to test linear tours, the test driver module `qctest.c` must be linked with either `qcsqrtline.c`, and also with `qcgraphline.c`. The source listings follow for these two files.

The legal tour types (`-t` command-line option) for testing linear tours are `BRL`, `BAL`, `URL`, or `UAL`. If the test program is run with tour types `BRC` or `BAC`, the tests will work, but the performance optimization for line-like tours in the Buss-Yianilos algorithm (elimination of the second scan) will be lost.

The file `qcsqrtline.c` contains a cost function for linear tours that returns the square root of the Euclidean distance between two nodes. It also contains a constant-time Ω function implementation.

⟨*qcsqrtline.c* 62⟩≡

```
#include <stdio.h>
#include <math.h>
#include "qcmatch.h"

#define squareit(A) ((A)*(A))

typedef struct {
        REAL    coord;
} NODE;

REAL    sqrtcost( void *nx, void *ny );
BOOLEAN Line_Omega( MNODE *x1, MNODE *x2, MNODE *x3, void *nodes[],
                COSTFUNC cost, int psi );

COSTFUNC        CostFunc  = sqrtcost;
OMEGAFUNC       OmegaFunc = Line_Omega;


REAL
sqrtcost( void *nx, void *ny )
{
        return sqrt(fabs(((NODE *)nx)->coord - ((NODE *)ny)->coord));
}

/* This version of omega works only for the linear/sqrt case */

BOOLEAN
Line_Omega( MNODE *x1, MNODE *x2, MNODE *x3, void *input[], COSTFUNC cost,
        int psi )
{
        REAL    del, eps;
        NODE    **nodes;
        nodes = (NODE **)input;

        del = psi * (x1->i - x2->i);
        eps = psi * (x2->i - x3->i);
        if ((eps <= 0) || (squareit(eps * ((nodes[x2->x]->coord -
                                            nodes[x1->x]->coord) -
                                           squareit(del))) <
                          squareit(del * ((nodes[x3->x]->coord -
                                            nodes[x2->x]->coord) +
                                           squareit(eps)))
                          && (del > 0))) {
                return TRUE;
        } else {
                return FALSE;
        }
}
```

This code is written to file `qcsqrtline.c`.

The file `qcgraphline.c` contains the function `graphics_output` for linear tours that writes a PostScript figure for each tour to standard output.

⟨*qcgraphline.c* 64⟩≡

```
#include <stdio.h>
#include <math.h>
#include "qcmatch.h"

#ifndef M_PI
#define M_PI    3.14159265358979323846
#endif

#define NODESIZE  .01   /* Size of circles used to represent nodes. */
#define YL         0.1   /* The y value of the line. */
#define YC         0.1   /* The y value of the centers of the edge semicircles. */

#define XLEFT    0.0
#define XRIGHT   1.0

#define MAXVAL  M_PI

typedef struct {
        REAL    coord;
} NODE;

void
graphics_output( int n, NODE *nodes[], int colors[], int matching[])
{
        int     kj;
        REAL    center, radius;


        printf("\nQCbegin\n");
        printf("PROOF\n");

                                /* Draw the coordinate line */
        printf("%6f  %6f  %6f  %6f  LINE\n", XLEFT, YL, XRIGHT, YL);

                                /* Draw the edge semicircles */
        for (kj=0 ; kj < n ; kj++) {

                                /* Unmatched or already-matched nodes */
                if (matching[kj] < 0 || kj >= matching[kj]) {
                        continue;
                }

                center  = (nodes[matching[kj]]->coord + nodes[kj]->coord)/2;
                center /= MAXVAL;

                radius = center - nodes[kj]->coord/MAXVAL;

                printf("%6f  %6f  %6f  UPPERSEMI\n", center, YC, radius);
        }

                                /* Draw the node circles */
        for (kj = 0; kj < n; kj++) {

                printf("%6f  %6f  %6f  %sCIRCLE\n", nodes[kj]->coord/MAXVAL,
                       YL, NODESIZE, (colors[kj] == BLUE) ? "H" : "S");
        }

        printf("QCend\n");
}
```
This code is written to file `qcgraphline.c`.

14 APPENDIX 4: PostScript Header File

This is the header file `qcheader.ps`, to which the test program output is appended in order to produce a valid PostScript EPS file. Note that the file includes self-documentation of the commands it defines.

⟨*qcheader.ps* 66⟩≡

```
%!
%%Title: Assignment Experiments Header File
%%Creator: PNY
%%CreationDate: October 1993
%%For: pny@minerva (Peter N. Yianilos)
%%Pages: 0
%%BoundingBox: 0 0 432 432
%%EndComments

/QCdict 32 dict def

 QCdict begin

        QCdict /mtrx matrix put

        /CIRCLE {
                gsave newpath 0 360 arc stroke grestore
        } def

        /UPPERSEMI {
                gsave newpath 0 180 arc stroke grestore
        } def

        /VARARC {
                gsave newpath arc stroke grestore
        } def

        /LOWERSEMI {
                gsave newpath 180 360 arc stroke grestore
        } def

        /LINE {
                gsave newpath moveto lineto stroke grestore
        } def

        /HCIRCLE {
                gsave 3 copy
                gsave newpath 0 360 arc 1 setgray fill grestore
                gsave newpath 0 360 arc stroke grestore
                grestore
        } def
        /SCIRCLE {
                gsave newpath 0 360 arc 0 setgray fill grestore
        } def

        /PROOF {
                .2 .2 translate
                gsave [ 1 1000 div 5 1000 div] 0 setdash newpath
                0 0 moveto 0 1 lineto 1 1 lineto 1 0 lineto closepath
                stroke grestore
                /FinalShow {showpage} def
        } def

 0 setlinewidth
 432 432 scale

 end

/FinalShow { } def
/QCbegin {QCdict begin /EnteredState save def .002 setlinewidth } def
/QCend   {FinalShow EnteredState restore end} def

%%EndProlog

% ------------------ INSERT PLOT COMMANDS BELOW -------------------

%  Coordinate System:
%
%       Design your image within the unit square.  It will actually appear
%       3" by 3" but may be scaled again when it is included in the TeX file.
%
%       Before producing a figure you must invoke: QCbegin
%
%       To produce a "proof" copy which may be sent to a printer,
%       the next command must be "PROOF".
%
%       Now include plotting commands from the list below.
%
%       At the end of the file invoke: QCend
%
%  <x> <y> <r> CIRCLE
%       Draws a circle centered at <x> <y> of radius <r>.
%
%  <x> <y> <r> LOWERSEMI
%       Draws the lower semicircle of the circle centered at <x> <y> of
%       radius <r>.
%
```

```
%  <x> <y> <r> UPPERSEMI
%       Draws the upper semicircle of the circle centered at <x> <y> of
%       radius <r>.
%
%  <x> <y> <r> <angle1> <angle2> VARARC
%       Draws a counterclockwise arc centered at <x> <y> or radius <r>
%       from angular position <angle1> to <angle2> measured in degrees
%       from the x-axis.
%
%  <x1> <y1> <x2> <y2> LINE
%       Draws a line from <x1>,<y1> to <x2>,<y2>
%
%  <x> <y> <r> HCIRCLE
%       Draws a hollow circle centered at <x>,<y> of radius <r>
%
%  <x> <y> <r> SCIRCLE
%       Draws a solid (black) circle centered at <x>,<y> of radius <r>
%
```

This code is written to file `qcheader.ps`.

References

[1] S. R. Buss and P. N. Yianilos, *Linear and $O(n \log n)$ Time Minimum-Cost Matching Algorithms For Quasi-convex Tours (Extended Abstract)*, in Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms, 1994, pp. 65–76.