1. HyperFun

Implementation of a GRASP to the hyperbolic function problem.

Contents

2. This is a GRASP for the hyperbolic function problem. The GRASP metaheuristic is a simple way of giving good heuristic results for the problem. We want to compare these results with other solution methods.

```
#include "common.h"
#include "parsecml.h"
#include "heap.h"
#include "timer.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define N 1000
  int print_best = 0;
```

3. Structure fpi_inst

Data stored for each problem instance.

```
  typedef struct {
    int n;     /* number of variables */
    int m;     /* number of fractions */
    int r;     /* number of constraints */
    int **A;     /* matrix of coeficients a_ij */
    int *A0;     /* vector of coeficients a_i0 */
    int **B;     /* matrix of coeficients b_ij */
    int *B0;     /* vector of coeficients b_i0 */
    int **R;     /* matrix of coeficients r_ij */
    int *R0;     /* vector of coeficients r_i0 */
    int knapsack;     /* 1 if the problem is has only a knapsack constraint */
  } fpi_inst;
```

4. Function fpi_new_from_file

Returns a new instance from the file given in the parameter.

• **parameter** f. The file with information about the instance.

```
  fpi_inst *fpi_new_from_file(FILE *f) {
      ⟨read the number of variables 5⟩
      ⟨read the number of fractions 6⟩
      ⟨read all coefficients 7⟩
```

5. ⟨read the number of variables 5⟩ ≡

```
  char str[32];
  int n, m, mm, r, i, j;
  fpi_inst *fi;
  int *A0, *B0, *R0, **A, **B, **R;
```

This code is used in chunk 4.

6. ⟨ read the number of fractions 6 ⟩ ≡

```
fscanf(f, "ai␣%d", &m);
if (m < 1) {
   printf("error:␣invalid␣number␣(%d)␣of␣fractions\n", m);
   return 0;
}
```

This code is used in chunk 4.

7. ⟨ read all coefficients 7 ⟩ ≡

```
⟨ read A_0 8 ⟩
⟨ read B_0 9 ⟩
⟨ read R_0 10 ⟩
⟨ read matrices A, B, and R 11 ⟩
⟨ initialize other instance variables 12 ⟩
⟨ delete memory in case of error 13 ⟩
```

This code is used in chunk 4.

8. ⟨ read A_0 8 ⟩ ≡

```
A0 = NewVec(int, m);
if (¬A0) return 0;
for (i = 0; i < m; i++) fscanf(f, "%d", A0 + i);
fscanf(f, "\nbi␣%d", &mm);
if (mm ≠ m) goto error1;
```

This code is used in chunk 7.

9. ⟨ read B_0 9 ⟩ ≡

```
B0 = NewVec(int, m);
if (¬B0) goto error1;
for (i = 0; i < m; i++) fscanf(f, "%d", B0 + i);
fscanf(f, "\nri␣%d", &r);
if (r < 1) {
   printf("error:␣invalid␣number␣of␣constraints\n");
   goto error2;
}
```

This code is used in chunk 7.

10. ⟨ read R_0 10 ⟩ ≡

```
R0 = NewVec(int, r);
if (¬R0) goto error2;
for (i = 0; i < m; i++) fscanf(f, "%d", R0 + i);
fscanf(f, "\nnvar␣%d", &n);
if (n < 1) {
   printf("error:␣invalid␣number␣of␣variables\n");
   goto error3;
}
fscanf(f, "%32s", str);
```

This code is used in chunk 7.

11. $\langle$ read matrices A, B, and R 11 $\rangle \equiv$

```
  NewVec2(A, int, m, n);
  if (¬A) goto error3;
  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++) fscanf(f, "%d", A[i] + j);
  fscanf(f, "%32s", str);
  NewVec2(B, int, m, n);
  if (¬B) goto error4;
  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++) fscanf(f, "%d", B[i] + j);
  fscanf(f, "%32s", str);
  NewVec2(R, int, r, n);
  if (¬R) goto error5;
  for (i = 0; i < r; i++)
    for (j = 0; j < n; j++) fscanf(f, "%d", R[i] + j);
```

This code is used in chunk 7.

12. $\langle$ initialize other instance variables 12 $\rangle \equiv$

```
  fi = New(fpi_inst);
  if (¬fi) goto error6;
  fi→m = m;
  fi→n = n;
  fi→r = r;
  fi→A0 = A0;
  fi→B0 = B0;
  fi→R0 = R0;
  fi→A = A;
  fi→B = B;
  fi→R = R;
  fi→knapsack = 0;
  return fi;
```

This code is used in chunk 7.

13. $\langle$ delete memory in case of error 13 $\rangle \equiv$

```
error6: Delete(R);
error5: Delete(B);
error4: Delete(A);
error3: Delete(R0);
error2: Delete(B0);
error1: Delete(A0);
  return 0; }
```

This code is used in chunk 7.

14. Function fpi_delete

Deletes the memory associated with the problem instance.

```
void fpi_delete(fpi_inst *pfi)
{
  Delete(pfi→R);
  Delete(pfi→B);
  Delete(pfi→A);
  Delete(pfi→R0);
  Delete(pfi→B0);
  Delete(pfi→A0);
  Delete(pfi);
}
```

15. Function fpi_print

Shows the contents of the instance on standard output.

```
void fpi_print(fpi_inst *pfi)
{
  int i, j;

  printf("entrei␣aqui.␣m=%d,␣n=%d,␣r=%d\n", pfi→m, pfi→n, pfi→r);
  printf("ai␣");
  for (i = 0; i < pfi→m; i++) printf("%d␣", pfi→A0[i]);
  printf("\nbi␣");
  for (i = 0; i < pfi→m; i++) printf("%d␣", pfi→B0[i]);
  printf("\nri␣");
  for (i = 0; i < pfi→r; i++) printf("%d␣", pfi→R0[i]);
  printf("\n");
  printf("aij\n");
  for (i = 0; i < pfi→m; i++) {
    for (j = 0; j < pfi→n; j++) printf("%d␣", pfi→A[i][j]);
    printf("\n");
  }
  printf("bij\n");
  for (i = 0; i < pfi→m; i++) {
    for (j = 0; j < pfi→n; j++) printf("%d␣", pfi→B[i][j]);
    printf("\n");
  }
  printf("rij\n");
  for (i = 0; i < pfi→r; i++) {
    for (j = 0; j < pfi→n; j++) printf("%d␣", pfi→R[i][j]);
    printf("\n");
  }
}
```

16. Function fpi_print_sol

Prints a solution to the standard output.

```
void fpi_print_sol(fpi_inst *ins, int *x)
{
  int i, n = ins→n;
  for (i = 0; i < n; i++) printf("%d␣", x[i]);
  printf("\n");
}
```

17. The objective function.

The first issue that must be dealt with in the GRASP is the objective function. The normal objetive function of the problem is simply the value of

$$\sum_{i=1}^{m} \frac{a_0 + \sum_{j=1}^{n} a_{ij} x_j}{b_0 + \sum_{j=1}^{n} b_{ij} x_j}.$$

However, an extended objective function can be used to handle unfeasibility in the constraints.

A first method to handle the linear constraints is to create only solutions that are guaranteed to be feasible. This can be made possible by carefully checking each candidate solution to make sure that the following conditions are true: 1) the value of any of the denominators in the objective function is not equal to zero; 2) each of the r constraints is satisfied. This is the approach that we developed initially.

18. Function fpi_cost

Computes the cost of a solution to the problem.

```
double fpi_cost(fpi_inst *ins, int *x)
{
  int i, j;
  double val1, val2;
  double val = 0;
  int m = ins→m;
  int n = ins→n;
  for (i = 0; i < m; i++) {
    val1 = ins→A0[i];
    val2 = ins→B0[i];
    for (j = 0; j < n; j++) {
      if (x[j] ≠ −1) {      /* -1 marks an unassigned variable */
        val1 += ins→A[i][j] * x[j];
        val2 += ins→B[i][j] * x[j];
      }
    }
    if (val2 ≡ 0) {
      printf("========>␣error:␣the␣solution␣is␣not␣feasible\n");
      fpi_print_sol(ins, x);
    }
    val += val1/val2;
  }
  return val;
}
```

19. Function fpi_feasible

Returns 1 if the solution is feasible, and 0 otherwise.

```
int fpi_feasible(fpi_inst *ins, int *x) { int i, j, n, m, r, val;
    r = ins→r;
    n = ins→n;
    m = ins→m;
    ⟨ check for violated constraints 20 ⟩
    ⟨ check for invalid entries in the solution vector 21 ⟩
    ⟨ check for invalid denominator 22 ⟩
```

20. ⟨ check for violated constraints 20 ⟩ ≡

```
  if (¬ins→knapsack)      /* check if all constraints are satisfied */
    for (i = 0; i < r; i++) {
      val = ins→R0[i];
      for (j = 0; j < n; j++)
        if (x[j] ≠ −1) val += ins→R[i][j] * x[j];
      if (val > 0) {      /* solution violates constraints */
        printf("solution␣violates␣constraints\n");
        return 0;
      }
    }
```

This code is used in chunk 19.

21. ⟨ check for invalid entries in the solution vector 21 ⟩ ≡

```
  for (j = 0; j < n; j++) {
    if (x[j] ≠ 0 ∧ x[j] ≠ 1 ∧ x[j] ≠ −1) {      /* solution has wrong entry */
      printf("solution␣has␣wrong␣entry\n");
      return 0;
    }
  }
```

This code is used in chunk 19.

22. check if $\sum b_{ij} x_j \geq 1$

⟨ check for invalid denominator 22 ⟩ ≡

```
  for (i = 0; i < m; i++) {
    val = ins→B0[i];
    for (j = 0; j < n; j++) {
      if (x[j] ≠ −1) {
        val += ins→B[i][j] * x[j];
      }
    }
    if (val ≤ 0) {      /* solution has ∑ B_{ij} x_j ≤ 0 */
      return 0;
    }
  }
  return 1; }
```

This code is used in chunk 19.

23. Function fpi_difference_cost

The variable components of a solution are ordered in the construction phase of GRASP according to the values of their contribution to the objective function $f(x)$. In this function, represented in the paper as $f'(x)$, we compute this improvement.

```
double fpi_difference_cost(fpi_inst *ins, int *x, int i)
{
  double icost;     /* initial cost */
  double ncost;     /* new cost */
  int n = ins→n;
  int *xc, j;
  if (x[i] ≡ −1) return 0;
  xc = NewVec(int, n);
  if (¬xc) return 0;     /* copy x to xc */
  for (j = 0; j < n; j++) xc[j] = x[j];     /* change unset variables to -1 */
  for (j = 0; j < n; j++) {
    xc[j] = (xc[j] ≡ 0) ? −1 : (xc[j] ≡ −1 ? 0 : xc[j]);
  }     /* find the initial cost */
  icost = fpi_cost(ins, xc);     /* change the solution */
  xc[i] = 1 − x[i];
  if (¬fpi_feasible(ins, xc)) {     /* avoid using an unfeasible solution */
    Delete(xc);
    return 0;
  }     /* compute new cost */
  ncost = fpi_cost(ins, xc);
  Delete(xc);
  return ncost − icost;
}
```

24. Structure hyp_grasp

This is the information used by GRASP.

```
typedef struct {
  fpi_inst *ins;
  int *x;     /* a vector with a solution to the problem */
  int *best;     /* set of edges in the best solution (must be a tree) */
  double cost;     /* cost of the solution */
  double best_cost;     /* cost of the best solution */
  double best_time;     /* time at which it was found */
  int no_improv_it;     /* number of iterations without improvement */
} hyp_grasp;
```

25. Function grasp_new

Construct a new GRASP data structure.

```
hyp_grasp *grasp_new(fpi_inst *ins)
{
  hyp_grasp *g = New(hyp_grasp);
```

```
    if (¬g) return 0;
    g→x = NewVec(int, ins→n);
    if (¬g→x) goto error1;
    g→best = NewVec(int, ins→n);
    if (¬g→best) goto error2;
    g→best_cost = 99999999;      /* large number */
    g→ins = ins;
    g→no_improv_it = 0;
    return g;
  error2: Delete(g→x);
  error1: Delete(g);
    return 0;
  }
```

26. Function grasp_delete

Delete that GRASP data.

```
  void grasp_delete(hyp_grasp *g)
  {
    if (g→best) Delete(g→best);
    if (g→x) Delete(g→x);
    Delete(g);
  }
```

27. Function construction_phase_rand

Call the constructor to create a new solution.

```
  int construction_phase_rand(hyp_grasp *g) {
      ⟨ create a random solution 28 ⟩
      ⟨ save information from construction phase 29 ⟩
```

28. ⟨ create a random solution 28 ⟩ ≡

```
  int i;
  for (i = 0; i < g→ins→n; i++) {      /* copy destinations and source to a vector */
    g→x[i] = rand() % 2;
  }
```

This code is used in chunk 27.

29. ⟨ save information from construction phase 29 ⟩ ≡

```
  g→cost = fpi_cost(g→ins, g→x);
  return 1; }
```

This code is used in chunk 27.

30. Structure diff_var

Used to hold information about variables during the sorting process.

```
typedef struct {
   int var;
   int diff;
   int pos;
} diff_var;
```

31. These are macros used to implement the heap sort and auxiliary functions.

```
#define heap_compare(i, j) (a[i].diff > a[j].diff)
#define heap_swap(p, q)
  {
    diff_var __x = a[p];
    a[p] = a[q];
    a[q] = __x;
    a[a[p].var].pos = p;
    a[a[q].var].pos = q;
  }
  define_heap_operation(diff_var)
#undef heap_compare
#undef heap_swap
```

32. Function construct_solution

The construction phase of our GRASP consists of two subphases. Initially, we define an initial assignment of the value 1 to one the variables. The target variable is chosen according to the amount of initial contribution to the solution. Then, all other variables are assigned values in $\{0, 1\}$ using a greedy function that tries to maximize the partial objective function corresponding to the values already assigned.

In the second subphase, we order the remaining variables acording to its contribution to the objective function. i.e., select one of the best elements in the RCL according to the value

$$\sum_{i \in S'} \frac{a_{ij} x_j}{b_{ij} x_j} - \sum_{i \in S} \frac{a_{ij} x_j}{b_{ij} x_j},$$

where S is the original set of selected indices and S' is the new set of selected indices. The RCL is composed of all indices i such that after making variable $x_i = 1$ the solution remains feasible.

This is implemented as follows.

```
int construct_solution(hyp_grasp *g) { int alpha, elem, feas, i, ii, j;
    int n = g→ins→n;
    int nn = n;
    int n_one_vars = 0;
    diff_var *a;
    a = NewVec(diff_var, n);
    if (¬a) return 0;
    for (i = 0; i < n; i++) {      /* init all variables to 0 */
      g→x[i] = 0;
    }
    for (j = 0; j < n; j++) {
        ⟨ find assignments for all variables 33 ⟩
    }
    ⟨ check solution after all variables set 37 ⟩
```

33. ⟨ find assignments for all variables 33 ⟩ ≡

```
  ⟨ initialize heap structure 34 ⟩
  ⟨ select a variable to be part of the solution 35 ⟩
  ⟨ check feasibility of the partial solution 36 ⟩
```

This code is used in chunk 32.

34. We need to initialize variables to its initial positions and differences.

⟨ initialize heap structure 34 ⟩ ≡

```
  for $(i = 0, ii = 0;\ i < n;\ i{+}{+})$ {
    if $(g{\to}x[i] \equiv 0)$ {
      $a[ii].var = i$;
      $a[ii].pos = ii$;
      $a[ii].diff = fpi\_difference\_cost(g{\to}ins, g{\to}x, i)$;
      $ii{+}{+}$;
    }
  }
  $nn = ii$;
  if $(nn \equiv 0)$ break;
```

This code is used in chunk 33.

35. Now, we we select one of the best elements to become part of the solution.

⟨ select a variable to be part of the solution 35 ⟩ ≡

```
  $alpha = nn \equiv 1 \ ?\ 1 : (rand(\,) \% (nn - 1)) + 1$;      /* random $\alpha$, s.t. $1 \le \alpha \le n$ */
  $elem = rand(\,) \% alpha$;
  $heap\_sort\_diff\_var(a, 0, nn)$;
  $g{\to}x[a[elem].var] = 1$;      /* try to set variable to 1 */
```

This code is used in chunk 33.

36. Finally, we have to check if the (partial) solution created in this way is feasible to the formulation.

⟨ check feasibility of the partial solution 36 ⟩ ≡

```
  $feas = fpi\_feasible(g{\to}ins, g{\to}x)$;
  if $(\neg feas)$ {      /* this variable cannot be part of the solution */
    $g{\to}x[a[elem].var] = -1$;
  }
  else {
    $g{\to}x[a[elem].var] = 1$;
    $n\_one\_vars{+}{+}$;
    if $(g{\to}ins{\to}knapsack \land n\_one\_vars \equiv g{\to}ins{\to}\mathtt{RO}[1])$ {
          /* we found the number of variables, thus we can end the construction phase */
      break;
    }
  }      /* remove variable elem from consideration */
  $a[elem] = a[nn - 1]$;
  $nn{-}{-}$; }
```

This code is used in chunk 33.

37. It is well possible that a solution is not feasible after this construction phase, due to the random choices made. Then, we should return -1 to flag this.

```
⟨ check solution after all variables set 37 ⟩ ≡
  if (¬fpi_feasible(g→ins, g→x) ∨ (g→ins→knapsack ∧ n_one_vars ≠ g→ins→RO[1])) {
    Delete(a);
    return −1;
  }     /* set all excluded variables to value 0. */
  for (j = 0; j < n; j++)
    if (g→x[j] ≡ −1) g→x[j] = 0;
  g→cost = fpi_cost(g→ins, g→x);
#if 0
  printf("end␣of␣construction␣fase\n");
  fflush(stdout);
  printf("the␣solution␣is:");
  for (j = 0; j < n; j++) printf("␣%d", g→x[j]);
  printf("\nCost␣is␣%lf\n", g→cost);
#endif
  Delete(a);
  return 1; }
```

This code is used in chunk 32.

38. Function construction_phase

Create a solution for the hyperbolic fraction problem.

```
#define MAX_CONST_TRY 100
  int construction_phase(hyp_grasp *g)
  {
    int i = 0, ret;     /* try to construct a feasible solution */
    do {
      ret = construct_solution(g);     /* call the GRASP constructor */
    } while (ret ≡ −1 ∧ i++ < MAX_CONST_TRY);
    if (i ≥ MAX_CONST_TRY) {
      return 0;
    }
#if 0
    construction_phase_rand(g);     /* performed a randomized construction */
    printf("initial␣solution␣has␣cost␣%lf\n", fpi_cost(g→ins, g→x));
#endif
    return 1;
  }
```

39. Forward declaration:

```
  void improvement_phase_knapsack(hyp_grasp *g);
```

40. Function improvement_phase

The improvement phase of GRASP has the objective of finding a local optimum solution according to a local neighborhood. The neighborhood of our problem is defined by pertubations on the incumbent solution. The perturbation consists of selecting one of the variables and flipping its value to zero or one. A variable is selected randomly, and after a flipping in the selected variable is performed, the resulting solution is tested for feasibility. If feasibility is achieved, then the solution is accepted if its cost is better than the previous one. Otherwise, a new random perturbation of the solution is done. This phase ends after N iterations without improvement.

```
void improvement_phase(hyp_grasp *g) {
    int i = 0;
    double cost;
    if (g→ins→knapsack) {
      improvement_phase_knapsack(g);
      return;
    }
    while (i < N)
    ⟨ perturb solution 41 ⟩
    ⟨ save perturbed solution if necessary 42 ⟩
```

41. ⟨ perturb solution 41 ⟩ ≡

```
{ int pos = rand() % g→ins→n;
g→x[pos] = 1 − g→x[pos];
if (¬fpi_feasible(g→ins, g→x)) {
  g→x[pos] = 1 − g→x[pos];
  i++;
  continue;      /* try another change */
}
cost = fpi_cost(g→ins, g→x);
```

This code is used in chunk 40.

42. ⟨ save perturbed solution if necessary 42 ⟩ ≡

```
if (cost < g→cost)
⟨ accept a good solution 43 ⟩
⟨ else reject a worse solution 44 ⟩
```

This code is used in chunk 40.

43. ⟨ accept a good solution 43 ⟩ ≡

```
{
  g→cost = cost;
  i = 0;
}
```

This code is used in chunk 42.

44. $\langle$ else reject a worse solution 44 $\rangle \equiv$

```
else {
    g→x[pos] = 1 − g→x[pos];
    i++;
}
} }
```

This code is used in chunk 42.

45. Function sum_x

Returns $\sum_i x_i$

```
double sum_x(hyp_grasp *g, int *x)
{
    int i;
    double s = 0;
    for (i = 0; i < g→ins→n; i++) s += x[i];
    return s;
}
```

46. Function improvement_phase_knapsack

This function gives an alternative implementation of local search for the case of knapsack instances. The idea is to exchange the values of *two* * variables (with different values), instead of changing just one (since this would make the knapsack solution infeasible).

```
void improvement_phase_knapsack(hyp_grasp *g) {
        int i = 0;
        double cost;
        if (sum_x(g, g→x) ≠ g→ins→R0[1]) {
            fpi_print_sol(g→ins, g→x);
            printf("error␣␣sum=%lf,␣%d\n", sum_x(g, g→x), g→ins→R0[1]);
        }
        while (i < N)
        ⟨ perturb knapsack solution 47 ⟩
        ⟨ save perturbed knapsack solution if necessary 48 ⟩
```

47. $\langle$ perturb knapsack solution 47 $\rangle \equiv$

```
{ int pos1 = rand() % g→ins→n;
int pos2;
do {
    pos2 = rand() % g→ins→n;
} while (pos1 ≡ pos2 ∨ g→x[pos1] ≡ g→x[pos2]);
g→x[pos1] = 1 − g→x[pos1];
g→x[pos2] = 1 − g→x[pos2];
if (¬fpi_feasible(g→ins, g→x)) {       /* revert changes */
    g→x[pos1] = 1 − g→x[pos1];
```

```
        g→x[pos2] = 1 − g→x[pos2];
        i++;
        continue;       /* try another change */
    }
    cost = fpi_cost(g→ins, g→x);
```
This code is used in chunk 46.

48. ⟨ save perturbed knapsack solution if necessary 48 ⟩ ≡
```
    if (cost < g→cost)
    ⟨ accept a good knapsack solution 49 ⟩
    ⟨ else reject a worse knapsack solution 50 ⟩
```
This code is used in chunk 46.

49. ⟨ accept a good knapsack solution 49 ⟩ ≡
```
    {
        g→cost = cost;
        i = 0;
    }
```
This code is used in chunk 48.

50. ⟨ else reject a worse knapsack solution 50 ⟩ ≡
```
    else {
        g→x[pos1] = 1 − g→x[pos1];
        g→x[pos2] = 1 − g→x[pos2];
        i++;
    }
    if (sum_x(g, g→x) ≠ g→ins→R0[1]) {
        fpi_print_sol(g→ins, g→x);
        printf("error␣pos(%d,%d),␣values(%d,%d),␣sum=%lf,␣%d\n", pos1, pos2, g→x[pos1], g→x[pos2],
            sum_x(g, g→x), g→ins→R0[1]);
    }
    } }
```
This code is used in chunk 48.

51. Function save_results

If necessary, save current solution.
```
    void save_results(hyp_grasp *g)
    {
        int i;
        if (g→cost < g→best_cost) {
            g→best = memcpy(g→best, g→x, sizeof (g→x[1]) * g→ins→n);
            g→best_cost = g→cost;
            g→best_time = get_time( );
            if (print_best) {
                printf("best␣solution␣is␣%lf,␣", g→cost);
                printf("at␣time␣=␣%.2lf\n", g→best_time);
```

```
      for (i = 0; i < g→ins→n; i++) printf("%d␣", g→best[i]);
      printf("\n");
      if (sum_x(g, g→best) ≠ g→ins→R0[1]) printf("error:␣it␣is␣wrong:%lf!\n", sum_x(g, g→best));
    }
    g→no_improv_it = 0;
  }
  else {
    g→no_improv_it++;
  }
}
```

52. Function grasp_end_condition

Returns 1 if GRASP must stop, 0 otherwise.

```
int grasp_end_condition(hyp_grasp *g)
{
  static int n = 10000;
  if (g→no_improv_it > n) return 1;
  return 0;
}
```

53. Function grasp_run

GRASP is a sampling procedure, proposed by Feo and Resende [1], which tries to create good solutions with high probability. The main tools to make this possible are the construction and the improvement phases. In the construction phase, GRASP creates a complete solution by adding pieces (or elements) that compose the solution with the help of a greedy function, used to perform the selection. In the hyperbolic function problem, a solution is composed by a set of 0-1 variables. The improvement phase then takes the incumbent solution and performs local perburbations in order to get a local optimal solution, with respect to some predefined neighborhood. The general algorithm can therefore be described as follows:

```
void grasp_run(hyp_grasp *g)
{
  int i = 1;
  set_initial_time();
  while (¬grasp_end_condition(g)) {
    int res = construction_phase(g);
    if (¬res) {
      continue;     /* a solution could not be find this time, so try again. */
    }
    improvement_phase(g);
    save_results(g);
    i++;
  }
  printf("total␣time␣=␣%.2lf,␣end␣time␣=␣%.2lf\n", g→best_time, get_time());
  printf("best␣solution␣is␣%lf\n", g→best_cost);
}
```

54. Structure App

Stores important data for the application.

```
typedef struct {
   int argc;
   char **argv;
   int delay;
   int seed;
   int knapsack;
} App;
```

55. Function app_new

Creates a new application data structure.

```
App *app_new(int argc, char **argv)
{
   App *a = New(App);
   if (¬a) return 0;
   a→delay = 100;
   a→argc = argc;
   a→argv = argv;
   a→seed = 0;
   a→knapsack = 0;
   return a;
}
```

56. Function app_delete

Frees memory used by application data.

```
void app_delete(App *a)
{
   Delete(a);
}
```

57. Command line processing. The parameters given in the command line must be read and parsed.

58. Function print_usage

Shows the possible parameters for the program.

```
void print_usage(char **argv, cmdl_st * cml)
{
   printf("%s:␣solve␣the␣hyperbolic␣function␣problem.\n", argv[0]);
   printf("usage:\n\t%s␣<filename>␣[options]\n", argv[0]);
   cmdl_print_options(cml);
}
```

59. Function app_process_args

Initial processing of command line arguments by the application

```
void app_process_args(App *a)
{
    int l4 = −1, nrun = 1;
    int p = 0, p2 = 0, h;
    cmdl_st cml[] = {      /* solution type */
    {"No.␣program␣iterations", "-n", &nrun, CLT_INT, 0}, {"random␣seed", "-s", &(a→seed), CLT_INT,
        0}, {"print␣each␣improvement", "-p", &p, CLT_OPT, 0}, {"print␣each␣2^i␣iter", "-p2", &p2,
        CLT_OPT, 0}, {"print␣best␣solution␣found", "-b", &print_best, CLT_OPT, 0},
        {"knapsack␣instance", "-k", &(a→knapsack), CLT_OPT, 0}, {"print␣help", "-h", &h, CLT_OPT,
        0}, {"maximum␣delay", "-d", &(a→delay), CLT_INT, 0}, {"Look␣for␣this␣value", "-l", &l4,
        CLT_INT, 0}};      /* initialize the comand line structure */
    cmdl_init(cml, sizeof (cml)/sizeof (cmdl_st));
    if (a→argc ≡ 1 ∨ (a→argc ≡ 2 ∧ ¬strcmp(a→argv[1], "-h"))) {
        print_usage(a→argv, cml);
        exit(0);
    }
    if (¬cmdl_process(a→argc, a→argv, cml)) {
        printf("error␣in␣command␣line.␣Try␣-help\n");
        exit(1);
    }
    if (a→delay < 1) {
        printf("error:␣delay␣must␣be␣positive\n");
        exit(1);
    }
}
```

60. Function main

The main entry point of the program. Perform initialization and call the solving routine.

```
int main(int argc, char **argv)
{
    hyp_grasp *g;
    fpi_inst *inst;
    App *a = app_new(argc, argv);
    if (¬a) exit(1);
    printf("GRASP␣for␣hyperbolic␣functions\n");
    app_process_args(a);
    inst = fpi_new_from_file(fopen(argv[1], "r"));
    if (inst) {
        printf("success␣reading␣file␣%s\n", argv[1]);
    }
    else {
        printf("error␣reading␣file␣%s\n", argv[1]);
        exit(1);
    }
    inst→knapsack = a→knapsack;
    g = grasp_new(inst);
    if $(\neg g)$ goto error1;
    grasp_run(g);
    grasp_delete(g);
    fpi_delete(inst);
    return 0;
  error1: fpi_delete(inst);
    return 1;
  }
```

61.

References

[1] T.A. Feo and M.G.C. Resende. Greedy randomized adaptive search procedures. *J. of Global Optimization*, 6:109–133, 1995.