Appendix A

User and System Manual

A.1 Combinatorial Optimization with DISCROPT

DISCROPT is a time-sensitive combinatorial optimizer; it finds the best solution possible within a limited computing budget. These terms need to be clarified.

A (computational) **problem** has an associated parameter, its **size**. For example, here's a problem: *given a map of 100 cities find a tour that visits each city exactly once.* Here, 100 is the size of this particular problem. Algorithms and heuristics are methods that find desirable solutions of all sizes. When we talk about a particular problem, we always have in mind all possible instances and sizes.

An **optimization problem** asks for the "best" solution subjected to an objective function. For example: *given a map (of any number of cities), find the best tour that visits each city exactly once.* The objective function in this case can be the usual Euclidean metric or measure of distance defined by the user. In DISCROPT, "best" always means "minimized"; it attempts to find a solution with the (almost) smallest or minimized objective value.

Combinatorial means the discrete or finite property of the problem. This means each solution to the problem must be made up in a certain manner by a countable, finite number of "elements". In the above examples, a solution is a tour or an ordering of all cities. The elements are cities. Further, a solution (i.e a tour) can be represented as a **circular permutation**. We observe that many optimization problems can be represented by either

"permutations", "subsets", or "set partitions". Hence, these representations together with their slight variants are currently supported.

A user must know and specify which combinatorial representation implied by his optimization problem in order to use DISCROPT. **A user only needs to provide an objective function** which takes as input an already-defined solution representation and provides its objective value. When DISCROPT is run, it uses already-defined search heuristics to find the best solution within *a specified running time*. To be flexible and efficient, users are given the options of defining additional functions that altogether serve as an objective function in a local search framework.

Interestingly, running time is a user input. DISCROPT uses its search strategies to find the most minimized solution within this specified running time. This distinct **time-sensitive** feature is born out of our observation that an abstract optimization problem might have applications with drastically different computing budgets. In one case, a user simply demands for the best possible solution. In others, perhaps more realistic considering the (NP-)hardness of most optimization problems, one can only hope for the best possible within his computing budget.

In the following sections, we will describe by example the process of implementing an optimization problem, a search heuristic, and a solution representation.

A.2 Example: Implementing TSP

We will go through the process of using DISCROPT to optimize the Traveling Saleperson problem (TSP): Given N cities find a shortest tour that visits all vertices, each exactly once. Each instance of this problem can be abstractly represented as a weighted complete graph of N vertices. The weight of edge (i, j) is the distance between city i and j. A solution, i.e a tour, can be abstractly represented as a circular permutation of N numbers. For example, the circular permutation $\pi_0\pi_1 \cdots \pi_{N-1}$ represents the tour starting from city π_0, to π_1, $\cdots$, to π_{N-1}, then coming back to π_0. Then the subjective cost of this solution is:

$$d(\pi_{N-1}, \pi_0) + \sum_{i=0}^{N-2} d(\pi_i, \pi_{i+1})$$

To use DISCROPT, these following functions must be defined:

1. `double ObjectiveFunction::cost(CircularPermutation & sol)` – takes a solution of type circular permutation and return the objective cost of the solution. This function must be defined so that the lower the cost, the better the solution.

2. `double ObjectiveFunction::delta_cost(CircularPermutation & sol, const mutation_element & mut_el)` – takes a solution and a *mutation*, returns the cost of the change from the solution to the neighboring solution obtained by mutating "sol" using the mutation "mut_el". This function is used by such heuristics as simulated annealing and hill climb heuristics.

3. `double ObjectiveFunction::extend_cost(CircularPermutation & sol, const mutation_element & mut_el)` – takes a *partial* solution and a mutation, returns the cost of the extension from the solution to the extended neighbor obtained by extending "sol" using the mutation "mut_el". This is used by the greedy heuristic.

4. `double ObjectiveFunction::correctness(CircularPermutation & sol)` – computes the degree of correctness or feasibility of the given solution. The correctness of a correct solution is 0; a lower number means the solution is more correct.

5. `double ObjectiveFunction::delta_correctness(CircularPermutation & sol, const mutation_element & mut_el)` – is similar to delta_cost. It is used by simulated annealing and hill climb heuristics.

6. `double ObjectiveFunction::extend_correctness(CircularPermutation & sol, const mutation_element & mut_el)` – is similar to extend_cost. It is used by the greedy heuristic.

7. `double ObjectiveFunction::true_cost(CircularPermutation & sol)` – measures the "real" objective cost of a solution. Often, as in TSP, it is the same as cost, but for some problems cost may be different from true cost, and the users may want to trace the values of both during search. DISCROPT, however, tries to find the lowest solution with respect to cost, not true_cost.

8. `double ObjectiveFunction::true_correctness(CircularPermutation & sol)` – measures the "real" correctness of a solution; similarly defined as true_cost.

A.2.1 TSP's Cost

The objective cost of TSP is defined as:

$$d(\pi_{N-1}, \pi_0) + \sum_{i=0}^{N-2} d(\pi_i, \pi_{i+1})$$

In DISCROPT, this is written as:

```
double ObjectiveFunction::cost(CircularPermutation & sol)
{
  int current, next, size = sol.get_permutation_size();
  double weight=0;

  current = sol.first_index();
  for(int i=0; i<size; i++){
    next = sol.next_index(current);
    weight += search_space->edge_weight(sol[current], sol[next]);
    current = next;
  }
  return weight;
}
```

A few observations:

- ObjectiveFunction::search_space stores the data structure of the problem instance, in this case a graph. It contains the method **edge_weight** that gives the weight of an edge. The data structure of search_space must have a *public static* method called **get_size()** that returns the size of a solution (of type circular permutation). In TSP, a circular permutation contains all vertices, and therefore get_size() returns the number of vertices in the graph. This function is mandatory as it is used by DISCROPT to generate a random solution.

  In DISCROPT, we use the Graph Template Library (GTL) to provide the graph data structure for TSP. There is no need for GTL if the user provides his own data structure helped to define the objective function.

- The type CircularPermutation has several public methods accessible to users. In this example, the user uses methods **get_permutation_size()**

to get the size of the permutation, and **first_index()** and **next_index()** to enumerate the indices of a circular permutation solution.

For **permutation, subset, and partition** representations, their elements can still be accessed or enumerated through theindexing mechanism: **first_index(), last_index(), next_index(),** [ ]. For example, the first element of a circular permutation **s** is **s[s.first_index()]**.

It must be reminded that DISCROPT minimizes, i.e. if s_1 is subjectively better than s_2, then cost(s_1) must be lower than cost(s_2).

A.2.2 TSP's delta_cost

Local search heuristics such as simulated annealing and hill climb move from one solution to a neighboring solution to find the best one. Instead of recomputing the cost, we can use **delta_cost** to save a factor of O(n) computation.

Delta_costs takes a solution and a mutation and returns the cost of the change a solution to the neighboring solution obtained by mutating the given solution. Different solution representations have different definitions of mutation (implementation details are in **operator.cpp**):

1. **Circular permutation, permutation** – a mutation means exchanging two random indices. The indices are stored in **mutation_element::first** and **mutation_element::second**.

   Therefore, a neighbor of s has the same permutation as s except at two indices.

   The difference is that a circular_permutation is equivalent (i.e. same objective cost) to all of its cyclic orders; the search space is narrower.

2. **Subset** – a mutation means taking a random item and reversing its position: if the randomly selected element is in the subset solution, it is removed from the subset; conversely, if the randomly selected is not in the subset solution, it is placed in the solution. This random element is stored in **mutation_element::first**.

   Therefore, a neighbor of s has the same subset as s except one element whose position is flipped.

3. **Partition** – a mutation means moving a random element (stored in **mutation_element::first**) to different random parts (stored in **mutation_element::second**).

   Therefore, A neighbor of s, has the same partition as s except an element is moved to another (possibly new) part.

Relevant files to look at are: **combinatorial_solution.h, (circular_)permutation.h, subset.h, partition.h, operator.cpp, basic_types.h**.

For TSP, any two neighboring circular permutations differ by an exchange of two indices. In other words, if s and δs are neighbors, then they are exactly the same except at two indices, say i and j, where $s[i] = \delta s[j]$ and $s[j] = \delta s[i]$. Suppose $|i - j| \neq 1$, then we can define **delta_cost**$(s, \delta s)$ as:

$$-d(s_{\pi(i-1)}, s_{\pi(i)}) - d(s_{\pi(i)}, s_{\pi(i+1)}) - d(s_{\pi(j-1)}, s_{\pi(j)}) - d(s_{\pi(j)}, s_{\pi(j+1)})$$
$$+d(s_{\pi(i-1)}, s_{\pi(j)}) + d(s_{\pi(j)}, s_{\pi(i+1)}) + d(s_{\pi(j-1)}, s_{\pi(i)}) + d(s_{\pi(i)}, s_{\pi(j+1)})$$

We also need to address the special case when $|i - j| = 1$. In DISCROPT, it is written as follows (details omitted):

```
double ObjectiveFunction::delta_cost(CircularPermutation & sol,
                                     const mutation_element &mut_el)
{
  int i = mut_el.first;
  int j = mut_el.second;
  int first = sol[i];
  int before_first = sol[sol.previous_index(i)];
  int after_first = sol[sol.next_index(i)];

  int second = sol[i];
  int after_second = sol[sol.next_index(j)];
  int before_second = sol[sol.previous_index(j)];

  double d = 0;
  if((first != before_second) && (second != before_first)){
    d -= search_space->edge_weight(before_first, first);
    d -= search_space->edge_weight(first, after_first);
    d -= search_space->edge_weight(before_second, second);
    d -= search_space->edge_weight(second, after_second);
```

```
    d += search_space->edge_weight(before_first, second);
    d += search_space->edge_weight(second, after_first);
    d += search_space->edge_weight(before_second, first);
    d += search_space->edge_weight(first, after_second);
  }
  \\ other cases are omitted...
}
```

Delta_cost must be defined so that

$$cost(ns) = cost(s) + delta\_cost(s, m)$$

where ns is a neighbor of s obtained by mutating s by m.

A.2.3 TSP's extend_cost

One of the powerful heuristics we use is based on the traditional **greedy** method, by which a solution is constructed incrementally from most-promising elements selected from a pool of possible extensions. We generalize this abstract greedy strategy to a local-search greedy method, by viewing each possible extension as a neighboring solution so that the act of extending a solution is considered as moving from one solution to its most promising neighbor.

Extend_cost takes a solution and a mutation and returns the cost of the extension from the solution to an extending neighbor obtained by extending the given solution. Extension is defined by *inserting an element e into a position p*; **mutation_element::first** indicates a position p, and **mutation_element::second** indicates the element to be inserted into p. The meaning of this extension (p and e) depends on the solution representation:

1. **Circular permutation, permutation** – There are $n + 1$ possible positions for p. If the solution is $s[s_1 \cdots s_{j-1} s_j \cdots s_n]$, then inserting an element e into a position p creates the solution: $s[s_1 \cdots s_{p-1}\, e\, s_p \cdots s_n]$.

2. **Subset** – There are two possible positions for p: 0 and 1, which means either inserting the element e into the subset solution or not inserting e into the subset solution. In either case, e is inserted into the set that contains the subset solution.

3. **Partition** – If m is the number of parts in the given partition, then there are $m + 1$ possible positions for p. Inserting e into one of the

parts creates a new extension. When $p = m + 1$, e is inserted into a new part.

For TSP, extend_cost is:

$$d(s_{\pi(j-1)}, s_{\pi(i)}) + d(s_{\pi(i)}, s_{\pi(j)}) - d(s_{\pi(j-1)}, s_{\pi(j)})$$

In DISCROPT, it is written as:

```
double ObjectiveFunction::extend_cost(CircularPermutation & sol,
                                      const mutation_element &mut_el)
{
  double weight=0;
  int pos = mut_el.first, prev_pos = sol.previous_index(mut_el.first);
  int new_ele = mut_el.second;

  if(pos > sol.last_index()) pos = sol.first_index();

  weight = search_space->edge_weight(sol[prev_pos], new_ele)
    + search_space->edge_weight(new_ele, sol[pos])
    - search_space->edge_weight(sol[prev_pos], sol[pos]);

  return weight;
}
```

Delta_extend must be defined so that

$$cost(ns) = cost(s) + extend\_cost(s, m)$$

where ns is an extension of s obtained by inserting element m_e into the position m_p of s.

A.2.4 Correctness, delta_correctness, and extend_correctness

For TSP on complete graphs, all solutions are correct (feasible). For problems such as Graph Coloring are may be incorrect solutions, which must be penalized in some way. Our current implementation of Graph Coloring penalizes incorrect solutions (improper colorings) inside the cost function. Therefore, correctness, delta_correctness, extend_correctness need not be defined. Users have the option of defining a measure of correctness separately from the cost function and rely on DISCROPT's ability to combine them. In

such cases, correctness, delta_correctness, and extend_correctness are defined in a similar fashion as cost, delta_cost, and extend_cost.

When these functions are not defined, we must set:

```
const bool ObjectiveFunction::correctness_defined = false;
double ObjectiveFunction::correctness(CircularPermutation & sol)
{
  return 0;
}
double ObjectiveFunction::delta_correctness(CircularPermutation & sol,
                                            const mutation_element &mut_el)
{
  return 0;
}
double ObjectiveFunction::extend_correctness(CircularPermutation & sol,
                                       const mutation_element &mut_el)
{
  return 0;
}
```

A.2.5 Compilation

- For our TSP problem, the subdirectory is called TSP and the objective function is defined in **TSP/tsp.cpp**. The data structures are declared and defined **TSP/tsp_graphs.h** and **TSP/tsp_graph.cpp**.

- **TSP/Makefile** serves as the standard template. Two flags PROBLEM_FLAG and REPRESENTATION_FLAG must be set correctly; look into **problem_defs.h** for detailed information. For TSP, these two flags are set in TSP/Makefile as follows:

  ```
  export PROBLEM_FLAG = TSP
  export REPRESENTATION_FLAG = CIRCULARPERMUTATION
  ```

- To let **TSP/Makefile** know the appropriate files, we specify where the objective function is defined (stored in obj_func_file variable) and where the data structure is defined (stored in the search_space_file variable). The output program name is stored in PROG.

```
obj_func_file = tsp
search_space_file = tsp_graphs
PROG = tsp
```

Note that the files are declared without extension (e.g. .cpp).

- Since we use the Graph Library Template (GTL) to implement the graph data structure, we must specify where the GTL libraries are:

```
LINKING=dynamic
GTL_DIR=/home/phan
ifeq ($(LINKING),static)
GTL_LIB=$(GTL_DIR)/lib/libGTL.a
else
GTL_LIB=-L$(GTL_DIR)/lib -lGTL
endif
```

- To let DISCROPT know about the files, the user must define appropriate entries in **problem_defs.h**. For TSP, it is:

```
#if TSP
#include "TSP/tsp_graphs.h"
typedef Graph InputType;
#endif
```

The general form is:

```
#if PROBLEM_FLAG
#include "ProblemDirectory/search_space_data_structure.h"
typedef DataStructureClass InputType;
#endif
```

Once the files are placed and set up properly, issue the commands:

```
$cd TSP
$make kernel
$make tsp
```

A.2.6 Running TSP

Assuming the output program is called tsp, we run DISCROPT by issuing the command:

```
tsp -i input_file -h which_heuristic -t how_many_seconds
```

This will take the input file, build a data structure based on it using the user-defined data structure, and optimize it using the chosen heuristics. For more information, type:

```
tsp --help
```

A.2.7 Summary

To add a new optimization problem to DISCROPT, the user should:

1. Define **cost**, which computes the objective cost of the solution
2. Define **delta_cost**, which computes the cost of moving from a solution (with known cost) to a neighbor. This function must be defined so that

   $$cost(ns) = cost(s) + delta\_cost(s, m)$$

   where ns is a neighbor of s obtained by mutating s by m.
3. Define **delta_extend**, which computes the cost of extending a solution (with known cost) to another solution with more more element. This function must be defined so that

   $$cost(ns) = cost(s) + extend\_cost(s, m)$$

   where ns is an extension of s obtained by inserting element m_e into the position m_p of s.
4. **correctness**, which computes the degree of correctness of feasibility of the solution. Users can define it separately on inclusively in **cost**.
5. **delta_correctness**, if correctness is defined, this function computes the degree correctness of moving from a solution to a neighbor.
6. **extend_correctness**, if correctness is defined, this function computes the degree of correctness of extending a solution.

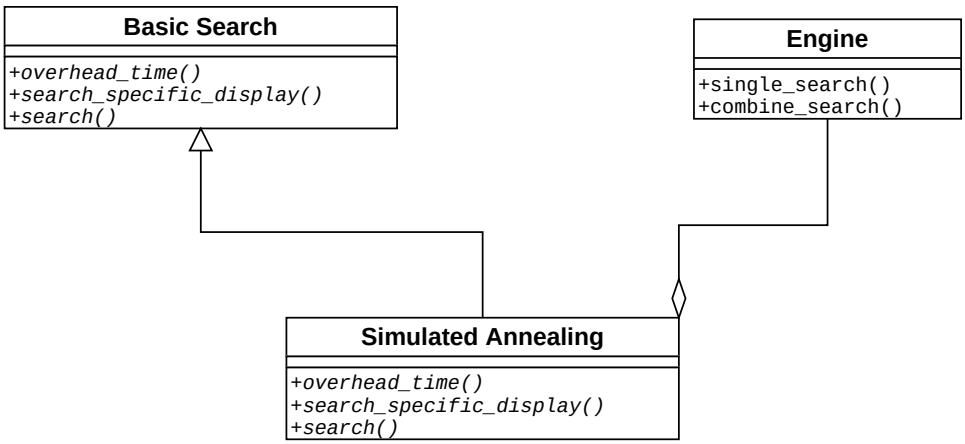

Figure A.1: Heuristic Hierarchy

A.3 Example: Implementing Time-sensitive Simulated Annealing

The Simulated Annealing class, derived from an abstract class Basic Search class, implements three abstract virtual functions:

- **double overhead_time()** – returns the minimal time needed by the search to give meaningful results.
- **void search_specific_display(ostream *out)** – displays information that is specific to the Simulated Annealing heuristic such as *temperature*. The conventional format is **(attribute, value)**.
- **void search()** – this is the actual search begins.

The function **BasicSearch::get_remaining_time()** returns how much time remains whenever it is called. We'll use a simple annealing schedule based on constant reduction of temperature.

Main Makefile

These are partial excerpts from the Makefile that show what need to be done.

```
BASE_OBJECTS = simulated_anneal.o

main.o   : main.cpp ....
         $(HEURISTICS_DIR)/simulated_anneal.cpp \
         $(CC) -c main.cpp

simulated_anneal.o : Heuristics/simulated_anneal.cpp basic_search.h
   $(CC) -c Heuristics/simulated_anneal.cpp $(PSET)
```

basic_types.h, shell_input.cpp

A new type must be added for **HeuristicType** in basic_types.h. Additionally, the input must be parsed correctly when given the flag "-h".

```
      case 'h':
        if(strcmp(argv[i+1], "simulated_annealing")==0)
           heuristic = h_simulated_annealing;
        i++;
        break;
```

engine.h

An instance of **Engine** is instantiated in **main.cpp**. Engine has as its member an instance of **BasicSearch**. The search is then initialized and run.

```
#include "Heuristics/simulated_anneal.cpp"

template<class T>
void Engine<T>::initialize_search()
{
  // Other irrelevant codes omitted
  switch(heuristic_type){

  case h_simulated_annealing:
    the_search =
     new SimulatedAnneal<T>(running_time, r_interval, cp_interval);
    break;

  }
```

```
}
```

Heuristic/simulated_anneal.cpp

This file defines the working of the Simulated Annealing heuristic. We do not intend to describe the details here, but simply specify the requirements. First, **SimulatedAnneal** must be derived from **BasicSearch**, and consequently define three virtual functions: **search() overhead_time(), search_specific_display()**. The last one is meant to output information specific to the heuristic at each report, which in this case, is the current temperature.

The time-sensitive aspect of this particular version of the heuristic is shown in the **update_temperature()** method, when the annealing rate is changed according to how much time is left.

```
template <class T>
class SimulatedAnneal : public BasicSearch<T> {
public:

  SimulatedAnneal(double running_time, double r_interval,
                  double cp_interval);

  //---virtual definition in BasicSearch<T>
  void search();

  void search_specific_display(ostream *out);

  double overhead_time();
};

template<class T>
void SimulatedAnneal<T>::search_specific_display(ostream *out)
{
  *out << "(temperature, "<< temperature << ")\n";
}

template<class T>
```

```
double SimulatedAnneal<T>::overhead_time()
{
  return MinLength * Global<T>::get_neighbor_creation_time() +
    Global<T>::get_solution_creation_time();
}

template<class T>
void SimulatedAnneal<T>::update_temperature()
{
  current_time = get_remaining_time();
  dtime =  prev_time - current_time;
  prev_time = current_time;

  if(current_time > 0 && dtime > 0){
    expected_trials = current_time/dtime;
    dtemp = pow(low_temperature/temperature, 1.0/expected_trials);
    if(dtemp > 0.9999999) dtemp = 0.9999999;
    if(dtemp < 0.0001) dtemp = 0.0001;
  }

  temperature *= dtemp;

  static int count = 0;
  if(temperature <= low_temperature){
    count++;
    temperature = low_temperature;
  }

  if(count > 2) temperature_saturated = true;
}
```

A.3.1 Composition of Different Heuristics

An advanced search may consist of different type of search heuristics depending on several factors: problem, search space, time, etc. To compose different existing heuristics, the user should look at

- **engine.h** to create a new method similarly to **Engine::run_once()**.

- **BasicSearch Engine::the_search**. To create a new search:

  the_search = new SearchHeuristic<T> (running_time, r_interval, cp_interval)

  The variables indicates the amount of time is given to each search, the period of report information, and the period of gathering statistics for various purposes such as determining if the search no longer makes any progress statistically.

A.4 Example: Implementing Circular Permutation

A combinatorial object is the basic entity of a local search; the object over which a search space is defined. Examples are **permutations, subsets**, etc. To work with a search space of solutions whose type has not been included in the system, a user must add a new combinatorial solution type. Its requirements are the same as those of existing types such as permutation, subset, etc. The basic requirements are:

- The new combinatorial object must be derived from the basic CombinatorialObject class, and must implement the pure virtual function specified there.
- Additionally, it must define other methods, in the same manner as other existing types such as permutation. Summarily, these methods include constructors that generate empty, same, and random instances, plus other basic utilities.

A.4.1 Relationships with Neighborhood and Objective Function

Neighborhood Operator: the user must provide parameter-overriding declarations and definitions of two methods for the basic class NeighborOp, as well as all of its derived operators such as **Swap**. They are:

- **gen_neighbor_element()**: generates a random **mutation m** given a solution **s**. This method does not change **s**; it simply generates **m**.

- **commit()**: given a solution and a mutation **s, m** respectively, make the actual physical change specified by **m** on **s**.

Technically, we could commit the mutation in gen_neighbor_element() and eliminate commit(). However, commit allows several neighbors of a solution **s** to exist at the same time, specified only by a mutation and a pointer to the physical solution structure of **s** without actually possessing it. This implementation boots efficiency if we later discard most neighbors and choose only one of them.

These two methods define the essence of a neighborhood structure in a local search search setting. Paired with objective functions and a search method, they traverse the search space from one solution to another.

Objective Function: the user must provide parameter-overriding declarations of **cost, delta_cost, correctness, delta_correctness** member functions of ObjectiveFunction class. These will be used when a new problem is added to the system.

Makefile

Define and direct dependencies.

```
BASE_OBJECT = circular_permutation.o

main.o  :  circular_permutation.h

circular_permutation.o  : circular_permutation.cpp circular_permutation.h \
 combinatorial_object.h
 $(CC) -c circular_permutation.cpp
```

circular_permutation.h, circular_permutation.cpp

Declare and define **CircularPermutation**. The class must be derived from **CombinatorialObject** whose pure virtual functions must also be defined here. The utility methods should be sufficient for implementations of optimization problems.If a special method to manipulate **CircularPermutation** is not defined for a particular problem, users should feel free to add it here. Similar implementations are in **permutation.h, subset.h, partition.h**.

```
#include "basic_types.h"

#include "combinatorial_object.h"

class CircularPermutation : public CombinatorialObject  {

};
```

objective_function.h, operator.h

Simply add an "include" in the header files.

```
#include ‘‘circular_permutation.h’’
```

Add these declarations in objective_functions.h:

```
#ifdef CIRCULARPERMUTATION
static double true_cost(CircularPermutation & solution);
static double true_correctness(CircularPermutation & solution);
static double cost(CircularPermutation & solution) ;
static double correctness(CircularPermutation & solution);
static double delta_cost(CircularPermutation & solution, const mutation_element & mut_el) ;
static double delta_correctness(CircularPermutation & solution, const mutation_element & mut_el);
static double extend_cost(CircularPermutation & solution, const mutation_element & mut_el);
static double extend_correctness(CircularPermutation & solution, const mutation_element & mut_el);
#endif
```

where CIRCULARPERMUTATION is defined in **problem_defs.h**.

operator.cpp

Local search is defined in terms of the neighborhood structure of the search space. This is implemented in **NeighborOp** from which specific operators such as **Swap** are derived. The user must define **gen_mutation()** and **commit()** for CircularPermutation specifically, using *template specialization*.

We illustrate the requirement in the **SwapOp** operator. In **gen_mutation()**, a random pair of indices of the solution, known as a **mutation** is generated. In **commit()**, a pair of "mutation" is applied to the circular permutation solution. The separation of these two methods is meant to make the system more efficient because often several neighbors are generated but only one or few committed, see chapter A.5.

```
template <>
void SwapOp<CircularPermutation>
  ::gen_mutation(const CircularPermutation &s,
                 mutation_element *mut)
{
  element_index random1, random2;
  random1 = (rand() % s.get_size()) + 1 ;
  random2 = (rand() % s.get_size()) + 1;
  while(random2 == random1) random2 = (rand() % s.get_size())+1;

  mut->operation = Swap;
  if(random1<random2) mut->set_mutation(random1, random2);
  else mut->set_mutation(random2, random1);
}

template <>
void SwapOp<CircularPermutation>
  ::commit(CircularPermutation *s,
           const mutation_element & mut)
{
  element first_element, second_element, tmp_element;
  element_index first_index = mut.first_index;
  element_index second_index = mut.second_index;

  first_element = s->element_lookup(first_index);
  second_element = s->element_lookup(second_index);

  tmp_element = s->perm[first_index];
  s->perm[first_index] = s->perm[second_index];
  s->perm[second_index] = tmp_element;

  tmp_element = s->inverse_perm[first_element];
  s->inverse_perm[first_element] = s->inverse_perm[second_element];
  s->inverse_perm[second_element] = tmp_element;
}
```

A.5 An Efficient Implementation of Local-Search Landscapes

A local search traverses its energy landscape from solution to solution. Solutions and their neighbors are generated and evaluated; most are discarded. This is a common phenomenon of local search heuristics. In this system, we implement an efficient way of solution (more precisely neighbors) generation.

The naive creation of random neighbor of a solution takes order $O(n + f(n))$, where n is the size of the solution, and $f(n)$ is the size of changes of going from the solution to its neighbor defined by the neighborhood structure. The more sophisticated implementation takes order $O(f(n))$. Details are discussed below.

This implementation is accomplished in **Operator::gen_mut(), Operator::gen_commit()**, and **ObjectiveFunction::delta_cost()**.

A.5.1 Lazy Implementation of Solution Objects

In a local search, it is often the case that several neighbors of a given solution are generated, evaluated to select of a few and discard the rest. To take advantage of this characteristic, we design a **Solution** object that contains a pointer to **Combinatorial** object and a **Mutation** object.

An unmutated solution **s** has an empty mutation, and a pointer to a combinatorial object. Neighbors of **s** have non-empty mutation objects, and their combinatorial object point to that of **s**. The discarded neighbors are simply deleted, while the selected ones will now copy the combinatorial object to which they point, and individualize them by **committing** their own mutations.

The saving in copying the actual combinatorial object is usually significant. The improvement depends on the improvement ratio of the **delta_objective** and **objective** functions. This in turn depends on the continuity of objective function acting on the neighborhood defined by the neighborhood operator. The smoother on the search space the objective function, the better: small difference in structural solution, small difference in value.

In a more sophisticated implementation, we can have higher order of referencing: a mutated solution of a mutated solution. However, I don't think that the implementation complexity justifies the saving.

A.5.2 Detailed Implementation of the Solution class

A **Solution** object contains a a mutation object, a combinatorial object (CO) – permutation, subset, or partition – and its costs, computed by the specified objective functions. Each combinatorial object has a reference count which is manipulated by the **Solution** class to know when to copy, commit, and discard the combinatorial object.

Since the system is time sensitive, generation of **Solution** objects are stopped by a **monitoring object** when the system meets a time constraint (e.g. reporting constraint, or deadline constraint). If the system meets the reporting constraint, the search that requests a solution to be generated must satisfy this constraint by reporting and then ask the monitoring object to lift the block. To ease a specific search (e.g. simulated annealing) to deal with these activities, they are handled in a **BasicSearch** class from which a specific search is derived. If the system meets the deadline constraint, it will stop.

The **Solution** class **generate solutions** in three distinct manners, which account for all the need of a typical local-search heuristic. Assuming time constraints have been satisfied,

1. generate random solution:
    - increment the new CO reference count (which was 0).
    - set mutation attribute to NoOp.
    - compute costs via user-defined objective functions.
2. generate a copy of a solution
    - increment the new object reference count.
    - set mutation attribute to the same as the one from which it is copied.

- compute costs.

3. generate a random neighbor of a solution. This is where the real work is achieved, carefully. When we want to create a random mutation of a **Solution** object A,

   - if it is not a mutated object, then simply create a random mutation whose CO is pointed to A whose reference count is consequently incremented.
   - if it is a mutated object, then first its mutated CO must be committed before a new random mutation is created. Things must be handled properly in both cases where the count is greater than 1 or not.

Deletion of a Solution object occurs only if the deleted object is the sole possessor of the CO. Otherwise, the CO's reference count is decremented.

In **Assignment of a Solution object**, the object to be assigned to decrements its reference count and delete the CO if its no longer being referred to, before incrementing and pointing to the new referenced CO.