Ancri Carlo matricola n°29264A A.A. 2024/2025

Informatica Applicata al Suono

C++ project

Project Structure

This project provides a modular C++ framework for solving three combinatorial optimization problems (AP, GAP and UFLP) built on three abstract base classes: ProblemInstance, ProblemSolver, ProblemSolution. Its primary goal is to provide a clean, extensible framework in which new problem types or alternative solution methods can be plugged in as fast as possible. The interactions between the user and the program are outlined in the following 5 points:

1. Select a problem for which a solver will be made available.
2. Import an instance of the previously selected problem type.
3. Print the solution currently in memory to the terminal.
4. Export the solution currently in memory to a file.
5. Free the memory and terminate the program.

General design choices

I personally chose to use smart pointers (specifically std::unique_ptr) to manage dynamic memory safely and avoid manual deallocation. The use of a unique pointer ensures that there is only one owner of each object (the current problem instance, solver or solution) at any given time, which eliminates the risk of memory leaks or unintended copies during execution. Since smart pointers take ownership of objects upon creation, I initialized them at the beginning of the main() function to guarantee clear and centralized control over all major components of the application; as a result, the constructors of all the classes are left near-empty. Any setup or data loading is instead handled through explicitly defined member functions, which fully replace the role of constructors in terms of functionality and object readiness.

For a less discursive explanation of the code written, see the Doxigen documentation over every function/class.

Global functions

CreateProblemInstance(...)

First of all, a unique_pointer is created for the current instance of a problem. Then, using ‘problem’ passed as a parameter, the newly created pointer is made specific to the selected problem. Using ‘filename’, the file describing the instance of a particular problem is opened: if it corresponds to the correct problem, the instance is created and populated correctly; otherwise, a runtime_error is thrown. Finally, the created pointer is returned.

CreateProblemSolver(...)

Using the ‘problem’ parameter of the function, a unique_ptr to a solver selected by the user is simply created and returned. If the user enters a problem number not among the specified ones, a runtime_error is thrown.

checkIfRightInstance(...)

This function is responsible for ensuring that the user is trying to solve an instance of the problem for which the solver has already been made available. In fact, it compares the previously selected problem number ‘currentProblem’ with a character located at a fixed position inside the instance file, which determines its type (‘A’ for AP, ‘G’ for GAP, ‘U’ for UFLP). Then returns ‘true’ if everything is fine, ‘false’ otherwise.

safeReadInt(...)

This function ensures that the number entered by the user from the terminal is actually selectable from the menu displayed on the screen. If the entered character is not among the possible choices, the function will continue to prompt the user for input until a valid one is entered.

clearScreen(...)

It clears the terminal after a delay (‘seconds’) given in the function parameters, whether you are using Windows or a Unix systems.

ProblemInstance class

The ProblemInstance class defines the interface for loading and accessing information about a particular optimization problem instance, sush as cost matrices, capacity constraints, or resource requirements. It is a virtual class useful for creating a "starting point" for the derived classes APInstance, GAPInstance, and UFLPInstance. These classes, in fact, implement a specific ‘loadFromFile’ method for the solvable problem instance.

In addition to this member function, a virtual destructor is implemented and left as default, in order to ensure safer memory deallocation handled by the derived classes (if necessary). If a destructor is not present in the derived classes, it means it has been left implicit.

APInstance

The class APInstance implements the code of the member function loadFromFile (inherited from ProblemInstance); in addition to this, the class also adds the private attributes 'costMatrix' and 'numAgents'.

Private attributes:

- `costMatrix` → Cost matrix is a 2D matrix that represent the costs associated with each agent and task
- `numAgents` → Number of agents in the problem instance

Public member functions:

- `void loadFromFile(const std::string& filename)` → This function first ensures that the instance file chosen by the user exists, opens it, and scans it line by line; as it progresses, the program expects a series of data in a specific order (as described in the project instructions) and populates the current problem instance. In particular, it first looks for a line "n:" and then a line "c_ij:".
- `const std::vector<std::vector<int>>& getCostMatrix() const` → Getter method that returns the cost matrix (private attribute) as constant value.
- `int getNumAgents() const` → Getter method that returns the number of agents (private attribute).

GAPInstance

The class GAPInstance inherits from the virtual class ProblemInstance the previously mentioned methods; it also implements some attributes capable of storing essential information for solving GAP instances.

Private attributes:

- `numTasks` → Number of tasks in the problem instance
- `numAgents` → Number of agents in the problem instance
- `cost` → A 2D matrix representing the costs associated with each agent and task
- `resource` → A 2D resource matrix representing the resources required by each agent for each task
- `capacities` → Array of capacities for each agent, representing the maximum resources they can provide

Public member functions:

- `void loadFromFile(const std::string& filename) override` → This function follows the same process as the one with the same name in the APInstance class, differing in the type of strings it looks for. In particular, it first looks for a line "m:", then the line "n:", then "c_ij:", then "r_ij:" and last "b_i:".
- `const std::vector<std::vector<int>>& getCostMatrix() const` → Getter method that returns the cost matrix (private attribute) as constant value.
- `const std::vector<std::vector<int>>& getResourceMatrix() const` → Getter method that returns the resource matrix (private attribute) as constant value.
- `const std::vector<int>& getCapacities() const` → Getter method that returns the capacities for each agent (private attribute) as constant value.
- `int getNumTasks() const` → Getter method that returns the number of tasks (private attribute).
- `int getNumAgents() const` → Getter method that returns the number of agents (private attribute).

UFLPInstance

The class UFLPInstance inherits from the virtual class ProblemInstance the previously mentioned methods; it also implements some attributes capable of storing essential information for solving UFLP instances.

Private attributes:

- `numFacilities` → Number of facilities
- `numClients` → Number of clients
- `openingCosts` → An array representing the opening costs for each facility
- `serviceCosts` → A 2D matrix representing the transportation costs from facilities to every client

Public member functions:

- `void loadFromFile(const std::string& filename) override` → This function follows the same process as the one with the same name in the APInstance class, differing in the type of strings it looks for. In particular, it first looks for a line "m:", then the line "n:", then "c_ij:" and last "f_i:".
- `int getNumFacilities() const` → Getter method that returns the number of facilities (private attribute).
- `int getNumClients() const` → Getter method that returns the number of clients (private attribute).
- `const std::vector<int>& getOpeningCosts() const` → Getter method that returns an array representing the opening cost for each fecility (private attribute) as constant value.
- `const std::vector<std::vector<int>>& getServiceCosts() const` → Getter method that returns the service cost from each client to evety facility (private attribute) as constant value.

ProblemSolver class

The ProblemSolver class serves as the abstract base for all algorithmic components that operate on a problem instance to produce a solution. Its role is to define a common interface for solving different types of problems, without prescribing how the solution is computed. The only method avaible on this interface is the pure virtual function 'solve(const ProblemInstance* instance)', which takes a pointer to a generic problem instance and returns a smart pointer to a solution object.

This abstraction allows for great flexibility and extensibility: each specific solver such as APSolver, GAPSolver, or UFLPSolver inherits from ProblemSolver and implements its own logic to interpret the data from the corresponding ProblemInstance subclass and produce a valid assignment, allocation, or facility configuration.

Solvers interact only through the abstract ProblemSolver interface, letting the application's main logic free to invoke the 'solve()' method on any problem type using polymorphism. This makes it easy to plug in new algorithms or replace existing ones without having to rewrite the entire hierrarchy.

APSolver

This is a simple greedy algorithm that assigns agents to tasks one by one: for each agent, it selects the cheapest available task; once a task is assigned, it becomes unavailable for other agents (it proceeds sequentially through the agents and does not backtrack). This approach does not guarantee optimality, but it provides a feasible assignment.

GAPSolver

In thit algotirhm the strategy used is critically drive, where more difficult tasks are assigned first, in order to avoid infeasible solutions. First of all, the tasks are ordered by criticality, which is defined as the number of agents that can perform the task. Then the algorithm assigns tasks to agents in the order of criticality, assigning each task to the agent that can perform it with the lowest cost. If no agent has enough residual capacity, it chooses the agent with the leas capacity overflow (as a fallback); then it updates the residual capacities of the agents. After attempting to assign all tasks, it checks if all tasks have been assigned;

if not, throws a runtime error. Once all tasks are assigned, it computes the total resources used and the total cost of the assignment and builds a GAPSolution object (It sets the assignment, computes resources and cost, and returns it).

UFLPSolver

- `void sortStructure(std::vector<std::pair<double, int>>& score)` → This function sorts a list of [score, index] pairs in ascending order by score. This algorithm orders the array in place.

The strategy used to resolve an UFLP instance is a greedy local search algorithm. This function first constructs an initial facility opening configuration by selecting approximately one third of all facilities with the best combined opening and average service cost. It then applies a local search that toggles each facility open/closed, accepting any change that lowers the total cost (opening + service) until no further improvement is possible. Finally, each client is assigned to its nearest open facility. Once all clients are assigned, it computes the total cost of the solution and builds a UFLPSolution object (it sets the assignment, computes the total cost, and returns it).

ProblemSolution class

The ‘ProblemSolution’ class is the abstract base designed that provides a unified interface for representing and displaying the output given by the ‘ProblemSolver’ class, regardless of the specific problem type. This includes storing decision variables (such as assignments or facility selections), calculating objective values (like total cost), and formatting the results for user-friendly display or file export.

Each subclass extends ‘ProblemSolution’ to handle the specific structure and evaluation logic of its corresponding problem. A solver object returns ‘std::unique_ptr<ProblemSolution>’ objects, allowing the main program to handle the result generically, regardless of its concrete type. Each solution class typically provides a ‘print(std::ostream&)’ method to output its content, in addition to introducing an overrided ‘<<’ operator, which allows printing the solution to an output stream of choice (whether it is to the terminal or to a file).

Public member function:

```
friend std::ostream& operator<<(std::ostream& os, const ProblemSolution& sol) {
      sol.print(os);
      return os;
}
```

This function aims to standardize the method of printing the solution to an instance. Using the passed parameters, it invokes the print method on the ProblemSolution ‘sol’ object, redirecting the output to a chosen output stream. Finally, it returns the output stream.

APSolution

The class APSolution stores all the information regarding which tasks are assigned to which agents and the total cost of the solution obtained by APSolver. As mentioned above, it inherits the ‘print’ method from the ProblemSolution class, which will be used through the overrided ‘<<’ operator.

Private attributes:

- `assignment` → Assignment vector where assignment[i] is the task assigned to agent i
- `totalCost` → Total cost of the assignments

Protected member function:

- `std::ostream& APSolution::print(std::ostream& os) const` → Function that simply prints on the output stream first the total cost to solve the instance, then the series of assignments chosen by the solver between agent and task.

Public member functions:

- `APSolution::APSolution()` → Constructor that sets ‘totalCost’ to a default value (0)
- `const std::vector<int>& getAssignment() const` → Getter method that returns an array representing the task assigned to each agent, where assignment[i] is the task assigned to agent i, as constant value.
- `void setAssignment(const std::vector<int>& assign)` → Setter method that assigns a vector to APSolution::assignment where assign[i] is the task assigned to agent i.

- `void computeCost(const std::vector<std::vector<int>>& costMatrix)` → Computes the total cost of the assignment based on the provided cost matrix; the total cost is computed as the sum of costs for each agent-task assignment.
- `int getTotalCost() const` → Getter method that returns the total cost of every assignment (private attribute).

GAPSolution

Class implemented following the same guidelines used for APSolution, replacing the various members/member functions with those useful for a GAP instance.

Private attributes:

- `assignment` → Assignment vector where assignment[i] is the task assigned to agent i
- `capacityUsed` → Capacity used by each agent, where capacityUsed[i] is the total resources used by agent i
- `totalCost` → Total cost of the assignment

Protected member function:

- `std::ostream &print(std::ostream &os) const` → Function that simply prints on the output stream first the total cost to solve the instance, then the series of assignments chosen by the solver between agent and task and last the capacity used for every agent.

Public member functions:

- `GAPSolution::GAPSolution()` → Constructor that sets ‘totalCost’ to a default value (0)
- `void setAssignment(const std::vector<int>& assign)` → Function that assigns a vector where assign[i] is the task assigned to agent i.
- `const std::vector<int>& getAssignment() const` → Getter method that returns the total cost of every assignment, where assignment[i] is the task assigned to agent *i* (private attribute) as constant value.
- `void computeResourcesAndCost(const std::vector<std::vector<int>>& cost, const std::vector<std::vector<int>>& resource)` → Computes the total cost of the assignment based on the provided cost matrix. The

total cost is computed as the sum of costs for each agent-task assignment, and the capacity used by each agent is updated accordingly.

- `int getTotalCost() const` → Getter method that returns the total cost of the instance resolved (private attribute).
- `const std::vector<int>& getCapacityUsed() const` → Getter method that returns the capacity used by every agent (private attribute) as constant value.

UFLPSolution

Class implemented following the same guidelines used for APSolution, replacing the various members/member functions with those useful for a UFLP instance.

Private attributes:

- `openFacilities` → An array tha indicates if a facility is open (true) or closed (false)
- `assignment` → An array representing all the facility-clients assignments
- `totalCost` → Total cost of the solution

Protected member function:

- `std::ostream& print(std::ostream &) const` → Function that prints the total cost for solving the instance, which facilities are open, and then all the assignments between facility and client that have been established.

Public member functions:

- `UFLPSolution::UFLPSolution()` → Constructor that sets 'totalCost' to a default value (0)
- `void setOpenFacilities(const std::vector<bool>& opens)` → Given a parameter vector opens, it populates the solution attribute openFacilities identically to opens.
- `const std::vector<bool>& getOpenFacilities() const` → Returns a constant reference to a vector of booleans indicating which facilities are open.

- `void setAssignment(const std::vector<int>& assign)` → This function initializes the assignment attribute by setting it equal to an array passed as a parameter.
- `const std::vector<int>& getAssignment() const` → Getter method for the assignment attribute.
- `void computeCost(const std::vector<int>& openingCost, const std::vector<std::vector<int>>& serviceCost)` → Function that returns the total cost of the solution. This number is calculated as the sum of the opening costs for open facilities and the service costs for assigned clients.
- `int getTotalCost() const` → Getter method for the totalCost attribute.

MakeFile - building the project

This Makefile is designed to efficiently manage the compilation of a modular C++ project, where source files and header files are organized into separate directories. The final executable is named ‘project’, and it is built from all ‘.cpp’ source files located in the ‘src/’ directory. Header files are included from the include/’ directory, which is specified using the ‘-I$(INC_DIR)’ compiler flag. The compiler used is ‘g++’, with the C++17 standard enabled (-std=c++17) and all warnings activated via the ‘-Wall’ flag. The Makefile automatically handles both the compilation of individual object files (.o) and the linking step that produces the final executable. The source files are dynamically discovered using the ‘wildcard’ function, and the corresponding object files are generated using ‘patsubst’. This means that adding new source files to the ‘src/’ directory requires no manual updates to the Makefile. Use ‘make clean’ for removing the object files and executable (useful for a full rebuild). The ‘.PHONY’ declaration ensures that ‘make’ always treats ‘clean’ as command, even if files with those names exist in the directory.

Bibliography

https://cppreference.com/

https://stackoverflow.com/questions

https://makefiletutorial.com/