Overview of available functions

In this section we give an overview of the variables and functions, implemented in the framework, that the user interacts with. We will go over their inputs and outputs as well as an explanation of what each variable or function does and their purpose.

The framework uses only one structure, `HVS` to store all the necessary data. This structure holds information about all the user implemented functions and basic information about the problem such as the number of nondominated points to find, if set, the objective direction, either maximization or minimization, the initial reference point and the data structure created by the user. It is required as input by all the frameworks functions.
We define a point, `HVS_Point`, as a vector of doubles. We also define a set `HVS_Set`, as a set of pointers to points (`HVS_Point`). The set also has a comparator, which keeps the points ordered in relation to the first objective. Lastly we define two enumerates, `HVS_ObjectiveDir` and `HVS_SolveType`. The former defines the objective direction of the problem (`HVS_MIN` and `HVS_MAX`) and the latter defines the whether the goal is to find all nondominated points (`HVS_2D_ALL`) or to find a subset of all nondominated points (`HVS_2D_NUM`).
The variable `data` and functions `input`, `init`, `solve` and `close` refer to user implemented data structure and functions, respectively. The variable `numNdPnts` refers to an integer and variables `verbose` refers to a boolean. More details are given below.

The available functions are the following:

- `HVSstart(HVS)`: Receives a pointer to the framework data structure, initialized it and sets default values.
- `HVSfree(HVS)`: Receives a pointer to the framework data structure and frees all associated data.
- `HVSsetProblemDS(HVS, data)`: Receives a user implemented data structure that contains the problems variables, saving a reference to it on the framework data structure.
- `HVSsetInput(HVS, input)`: Receives a user implemented function that reads the problems input, saving a reference to it on the framework data structure.
- `HVSsetInit(HVS, init)`: Receives a user implemented function that initializes the problem and all the necessary variables, saving a reference to it on the framework data structure.
- `HVSsetSolve(HVS, solve)`: Receives a user implemented function that solves a version of the problem and returns the best point calculated, saving a reference to it on the framework data structure.
- `HVSsetClose(HVS, close)`: Receives a user implemented function that frees all variables associated with the problem, saving a reference to it on the framework data structure.
- `HVSsetObjectiveDir(HVS, HVS_ObjectiveDir)`: Receives the value representing the objective of the function, `HVS_MAX` for maximization and `HVS_MIN` for minimization, setting it on the frameworks data structure.
- `HVSsetNumNdPoints(HVS, numNdPnts)`: Receives an integer value indicating how many nondominated points the framework needs to find before stopping, only used when `HVS_2D_NUM` is selected as the solving method.

- `HVSsetIRefPoint(HVS, HVS_Point)`: Receives a point that is set as the initial reference point for the frameworks solving process.
- `HVSsetVerbose(HVS, verbose)`: Receives a boolean. If set to *True*, the framework will output additional information during the solving process.
- `HVSsolve(HVS, HVS_SolveType)`: Receives and executes the solving method desired. This function will execute all the user implemented functions in the following order: `input`, `init`, `solve` and `close`. If called with `HVS_2D_ALL`, it will calculate all nondominated points, while with `HVS_2D_NUM`, calculates the number of nondominated points indicated by `HVSsetNumNdPoints()`, defined above. If the number of solutions desired is not set, `HVS_2D_NUM` will calculate all nondominated points.
- `HVSgetSols(HVS)`: Returns the set of nondominated points found.
- `HVSprintSet(HVS_Set)`: Receives a `HVS_Set` of points and print them to the console.

User Guide

In this section we present a step-by-step guide on how a user can use the framework to implement a problem, run the desired algorithm and print the resulting nondominated set on the console. The flow of the framework can be seen in Figure 1, in which the colored words represent what the user must implement, and is presented in the following steps:

1. Implement functions `input`, `init`, `solve` and `close`. These functions are responsible for reading all the inputs necessary, initializing the problem, solving an instance of the problem and freeing all the resources allocated to the problem, respectively.
2. If, in order to solve a subproblem additional variables are needed, excluding the reference point, create a data structure that holds all the required variables associated to the implemented problem.
3. On the `main()` function, declare the main data structure `HVS` and initialize it by calling `HVSstart(HVS)`.
4. Set all the user implemented functions on the framework by calling, in any order, `HVSsetInput(HVS, input)`, `HVSsetInit(HVS, init)`, `HVSsetSolve(HVS, solve)`, `HVSsetClose(HVS, close)`, `HVSsetDataStructure(HVS, data)`.
5. In order to set the initial reference point, the function `HVSsetIRefPoint(HVS, HVS_Point)` should be called. The objective of the problem should also be set using function `HVSsetObjectiveDir(HVS, HVS_ObjectiveDir)`.
6. If the user wants to calculate a specific number of nondominated points, they must call function `HVSsetNumNdPoints(HVS, numNdPnts)`, with `numNdPnts` being an integer of how many points to calculate.
7. With all necessary variables set, function `HVSsolve(HVS, HVS_SolveType)` should be called. This will run the algorithm chosen.
8. The results can be obtained by using function `HVSgetSols(HVS)`, which returns a `HVS_Set`, and displayed using function `HVSprintSet(HVS_Set)`.
9. Lastly, the function `HVSfree(HVS)` must be called to release all the frameworks variables.

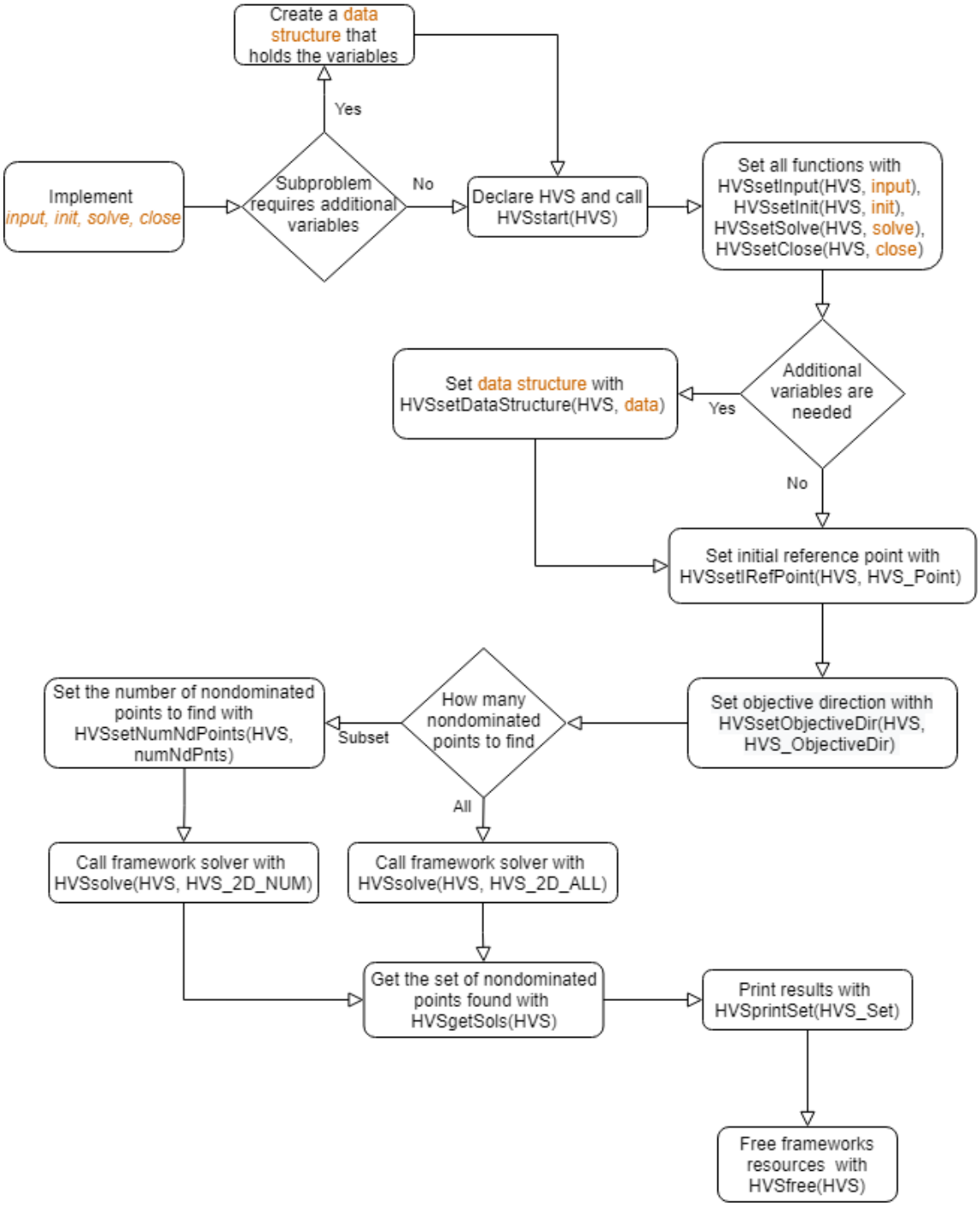

Figure 1: Flow diagram of the steps required to use the framework. The highlighted words represent functions or a data structure that the user must implement

Implementation example

In this section we will follow the steps given in Section and give example of an implementation of the ILP linearization of the hypervolume scalarized bi-objective knapsack with cardinality constraints, presented in Section ?? in `C++` using Solving Constraint Integer Programs (SCIP). We will give examples of the implementation of functions `init, solve` and `close` and an example of a data structure that holds all the variables needed by the problem. The examples presented are simplified. Throughout the examples `SCIP_CALL()`

is used. This is a built in define of SCIP and ensures that, in case of error, it is caught and the user is warned of the function that gave an error.

Figures 2, 3, 4 and 5 are parts of function `init`. Figures 6 and 7 refer to function `solve`. Figure 8 presents a snippet of the implementation of function `close`. Figure 10 refers to the main function of the program.

As shown in Section the first steps consists of implementing the functions input, init, solve and close, and a data structure if necessary. We will not shown an example of function input.

In order to create a problem with SCIP, we first need to start the SCIP environment, add the plugins and set the problems objective. This is illustrated in Figure 2. Variable `scip` will be used throughout the examples and refers to the one initialized in this figure.

```
/* init SCIP */
SCIPcreate(scip);

/* include default plugins */
SCIP_CALL(SCIPincludeDefaultPlugins(scip));

/* define problem */
SCIP_CALL(SCIPcreateProbBasic(scip, "Bi-Objective Knapsack"));

/* set objective function direction */
SCIP_CALL(SCIPsetObjsense(scip, SCIP_OBJSENSE_MAXIMIZE));
```

Figure 2: Declaration of the SCIP environment, add addition of the default plugins, definition of a problem and setting of the objective direction

```
std::vector<std::vector<SCIP_Real>> matrixQ;
std::vector<SCIP_VAR *> matrixY;

/* calculate matrixQ[0][1] and add it to SCIP as a variable */
matrixQ[0][1] = arrayA[0] * arrayB[1];
SCIP_CALL(SCIPcreateVarBasic(scip, &(matrixY.at(1)), ("y01"), 0.0,
↪  1.0, matrixQ[0][1], SCIP_VARTYPE_BINARY));
SCIP_CALL(SCIPaddVar(scip, matrixY.at(1)));

/* calculate r1r2 and add it to SCIP as a variable */
SCIP_CALL(SCIPcreateVarBasic(scip, &(matrixY.back()), ("r1r2"), 1.0,
↪  1.0, iRefPoint[0] * iRefPoint[1], SCIP_VARTYPE_BINARY));
SCIP_CALL(SCIPaddVar(scip, matrixY.back()));
```

Figure 3: Calculation of two binary constraints, creation of the corresponding SCIP variables and addition the variable to SCIP

```
SCIP_CONS *cons1 = nullptr;
std::vector<SCIP_CONS *> cons4;
/* init constraints */
/* cons1 -> sum{i=1,..,n} (a{i}y{ii}) >= r1 */
/* cons4 -> sum{i=1,..,n; i!=j} (y{ij}) <= (k-1)y{jj} for all j in
↪  {1,..,n} */
SCIP_CALL(SCIPcreateConsBasicLinear(scip, &cons1, "cons1", 0, nullptr,
↪  nullptr, iRefPoint.first, SCIPinfinity(scip)));

for (int i = 0; i < numElems; i++) {
    SCIP_CALL(SCIPaddCoefLinear(scip, cons1, matrixY.at(numElems * i +
    ↪  i), arrayA[i]));
}

for (int j = 0; j < numElems; j++) {
    SCIP_CALL(SCIPcreateConsBasicLinear(scip, &cons4.at(j), ("cons4_"
    ↪  + std::to_string(j + 1)).c_str(), 0, nullptr, nullptr,
    ↪  -SCIPinfinity(scip), 0));
    for (int i = 0; i < numElems; i++) {
        if (i != j) {
            SCIP_CALL(SCIPaddCoefLinear(scip, cons4.at(j),
            ↪  matrixY.at(numElems * i + j), 1));
        }
    }
    SCIP_CALL(SCIPaddCoefLinear(scip, cons4.at(j), matrixY.at(numElems
    ↪  * j + j), - (coef - 1)));
}
```

Figure 4: Creation of a simple constraint and a vector of constraints, setting of their coefficients

The next step consists of adding the binary variables to the problem. Figure 3 shows the calculation of two coefficients, `Q[0][1]` and r^1r^2.Matrix `MatrixQ` holds the coefficients and `MatrixY` hold the associated SCIP variables. `iRefPoint` is the initial reference point. As coefficient `Q[0][1]` is associated to a binary variable, the lower bound is set to 0.0 and the upper bound is set to 1.0. In contrast, as r^1r^2 must always be considered, both lower and upper bounds are set to 1.0.

```
/* add constraints to scip */
SCIP_CALL(SCIPaddCons(scip, cons1));

for (int i = 0; i < cons4.size(); i++) {
    SCIP_CALL(SCIPaddCons(scip, cons4.at(i)));
}
```

Figure 5: Addition of the constraints to SCIP

```
SCIP *scipCp = nullptr;

/* creating the SCIP environment that will be used to solve the
↪  problem  */
SCIPcreate(&scipCp);

/* updating matrixQ in position (i,i) due to new reference point  */
for(int i = 0; i < numElems; i++) {
    matrixQ[i][i] = arrayA[i] * arrayB[i] - (refPoint[HVS_Y] *
    ↪  arrayA[i] + refPoint[HVS_X] * arrayB[i]);
    SCIPchgVarObj(scip, matrixY.at(numElems * i + i), matrixQ[i][i]);
}
/* updating r1r2 due to new reference point */
SCIPchgVarObj(scip, matrixY.back(), refPoint[HVS_X] *
↪  refPoint[HVS_Y]);

/* updating constraints due to new reference point */
SCIPchgLhsLinear(scip, cons1, refPoint[HVS_X]);
SCIPchgLhsLinear(scip, cons2, refPoint[HVS_Y]);

/* copying the problem to the new SCIP env */
SCIPcopy(scip, scipCp, nullptr, nullptr,
↪  (std::to_string(refPoint[HVS_X]) + "," +
↪  std::to_string(refPoint[HVS_Y])).c_str(), 1, 1, 0, 1, nullptr);
```

Figure 6: Modification of the necessary problem variables and constraints due to new reference point and creation of a copy of the SCIP environment in other to be solved

After that, we need to create the constraints as well as to set their coefficient. If a constraint is an equality, the upper bound and the lower bound must be set to the same value. If a constraint is an inequality with one bound being infinite, `SCIPinfinity(scip)` must be used. Figure 4 shows examples of how to create basic constraints in SCIP and how to add coefficients to each constraint. In this figure, `numElems` refers to the number of values for each objective and `coef` refers to the coefficient constraint of the problem.

The last step necessary in order to initialize a SCIP problem is to add the created constraints to the environment. This is illustrated in Figure 5, in which the two constraints created previously are added.

In order to solve a subproblem, we need to take in consideration that, because each subproblems has a different reference point, all the problems variables and constraints that have reference to the reference point must be updated. This is illustrated in Figure 6. In this example we also create a copy of the SCIP environment after all the modifications are done. This is done, because in order to modify the problems variables and constraints it is required to have a reference of the container that holds them. Therefore, as we have the information of the variables and constraints of the problem created previously we do the necessary alterations on the main SCIP environment. We also create a copy of the problem because it makes it easier to solve multiple subproblems in a row, as we do not need to free the solved and transformed state in order to go back to the problem specification, as shown in Section **??**.

```
    /* solving the problem */
    SCIPsolve(scipCp);

    /* getting the best solution  */
    SCIP_SOL *sol = SCIPgetBestSol(scipCp);

    int numVars = SCIPgetNOrigVars(scipCp);
    SCIP_VAR **vars = SCIPgetOrigVars(scipCp);
    double *results = (double *)malloc(sizeof(double)*numVars);
    SCIPgetSolVals(scipCp, sol, numVars, vars, results);

    /* calculating the hypervolume coordinates of the solution  */
    SCIP_Real hvX = 0;
    SCIP_Real hvY = 0;
    for(int i = 0; i < numElems; i++) {
        hvX += arrayA[i] * results[i * numElems + i];
        hvY += arrayB[i] * results[i * numElems + i];
    }
    free(results);
    /* freeing the SCIP environment */
    SCIPfree(&scipCp);
```

Figure 7: Solving of a SCIP subproblem, extraction of the results and calculation of the new nondominated point

With the copy of the modified SCIP environment we can call the function to solve the subproblem, extract the results and free the copied SCIP. This is shown in Figure 7, in which the copied environment is solved. We use `SCIPgetBestSol`, `SCIPgetNOrigVars`, `SCIPgetOrigVars` and `SCIPgetSolVals` to extract the value of the binary variables of the problem. From their output, it is possible to calculate the resulting nondominated point.

```
    /* freeing the variables */
    for (int i = 0; i < matrixY.size(); i++) {
        SCIP_CALL(SCIPreleaseVar(scip, &matrixY.at(i)));
    }

    /* freeing the constraints */
    SCIP_CALL(SCIPreleaseCons(scip, &cons1));

    for (int i = 0; i < cons4.size(); i++) {
        SCIP_CALL(SCIPreleaseCons(scip, &cons4.at(i)));
    }

    /* freeing the SCIP environment */
    SCIP_CALL(SCIPfree(&scip));
```

Figure 8: Freeing the variables, constraints and the SCIP environment.

SCIP requires all variables and constraints to be freed. Figure 8 shows how to free these variables and constraints and finally how to free SCIP itself. If some variable of constraint

is not freed before calling `SCIPfree`, an error is given.

As the problem presented requires additional variables, like for example `matrixQ`, we need to create a data structure that stores them. This is illustrated in Figure 9. The data structure shown only has the variables that were used in these examples. Therefore, other necessary variables, like for the other constraints, are omitted.

```
struct problemVariables {
    int numElems; /**< Number of elements for each objective */
    int maxElems; /**< Max number of elements in each solution */
    std::vector<SCIP_Real> arrayA; /**< Gain for objective A */
    std::vector<SCIP_Real> arrayB; /**< Gain for objective B */
    SCIP *scip; /**< Pointer for SCIP*/
    std::vector<std::vector<SCIP_Real>> matrixQ;
    std::vector<SCIP_VAR *> matrixY;
    SCIP_CONS *cons1 = nullptr;
    std::vector<SCIP_CONS *> cons4;
};
```

Figure 9: Demonstration of the data structure required for the biobjective knapsack problem

With all required functions implemented, we can call all the required functions on `main()`, solve the problem and print the resulting set of nondominated points found, which corresponds to steps 3 to 9 presented in Section . Figure 10 contains an example of how the `main()` function could be implemented. Although we call `setInitialRefPoint()` on the main function, it can be called inside the user defined function `input`, allowing the reference point to be passed as input alongside the problems data. In this example we use `HVS_2D_NUM` in order to find 5 nondominated points, as set by `HVSsetNumNdPoints()`.

```
HVS *hvs;
HVSstart(&hvs);

HVSsetInput(hvs, readInput);
HVSsetInit(hvs, initProblem);
HVSsetSolve(hvs, solveProblem);
HVSsetClose(hvs, closeProblem);
HVSsetDataStructure(hvs, new problemVariables());
HVSsetNumNdPoints(hvs, 5);
setInitialRefPoint(data, {0.0,0.0});
HVSsetVerbose(hvs, true);
HVSsolve(HVS_2D_NUM, hvs);
HVS_Set results = HVSgetSols(hvs);
HVSprintSet(results);

HVSfree(&hvs);
```

Figure 10: Example of the `main()` function