The Bin Packing Problem

Like the multiple knapsack problem, the bin packing problem also involves packing items into bins. However, the bin packing problem has a different objective: find the fewest bins that will hold all the items.

The following summarizes the differences between the two problems:

- Multiple knapsack problem: Pack a subset of the items into a fixed number of bins, with varying capacities, so that the total value of the packed items is a maximum.
- Bin packing problem: Given as many bins with a common capacity as necessary, find the fewest that will hold all the items. In this problem, the items aren't assigned values, because the objective doesn't involve value.

The next example shows how to solve a bin packing problem.

Example

In this example, items of various weights need to be packed into a set of bins with a common capacity. Assuming that there are enough bins to hold all the items, the problem is to find the fewest that will suffice.

The following sections present programs that solve this problem. For the full programs, see Complete programs (#complete_programs).

This example uses the MPSolver wrapper (/optimization/lp/mpsolver).

Import the libraries

The code below imports the required libraries.

Python (#python) C++ Java (#java) C# (#c) (#c++)

```
#include <iostream>
#include <memory>
#include <numeric>
#include <ostream>
#include <vector>
```

```
#include "ortools/linear_solver/linear_expr.h"
#include "ortools/linear_solver/linear_solver.h"
```

Create the data

The code below creates the data for the example.

Python (#python) C++ (#c++) Java (#java) C# (#c)

```
struct DataModel {
  const std::vector<double> weights = {48, 30, 19, 36, 36, 27,
                                       42, 42, 36, 24, 30};
  const int num_items = weights.size();
  const int num_bins = weights.size();
  const int bin_capacity = 100;
};
```

The data includes the following:

- `weights`: A vector containing the weights of the items.
- `bin_capacity`: A single number giving the capacity of the bins.

There are no values assigned to the items because the goal of minimizing the number of bins doesn't involve value.

Note that `num_bins` is set to the number of items. This is because if the problem has a solution, then the weight of every item must be less than or equal to the bin capacity. In that case, the maximum number of bins you could need is the number of items, because you could always put each item in a separate bin.

Declare the solver

The following code declares the solver.

Python (#python) C++ (#c++) Java (#java) C# (#c)

```
  // Create the mip solver with the SCIP backend.
  std::unique_ptr<MPSolver> solver(MPSolver::CreateSolver("SCIP"));
  if (!solver) {
```

```
    LOG(WARNING) << "SCIP solver unavailable.";
    return;
  }
```

Create the variables

The following code creates the variables for the program.

Python (#python) C++ Java (#java) C# (#c) (#c++)

```
std::vector<std::vector<const MPVariable*>> x(
    data.num_items, std::vector<const MPVariable*>(data.num_bins));
for (int i = 0; i < data.num_items; ++i) {
  for (int j = 0; j < data.num_bins; ++j) {
    x[i][j] = solver->MakeIntVar(0.0, 1.0, "");
  }
}
// y[j] = 1 if bin j is used.
std::vector<const MPVariable*> y(data.num_bins);
for (int j = 0; j < data.num_bins; ++j) {
  y[j] = solver->MakeIntVar(0.0, 1.0, "");
}
```

As in the multiple knapsack example, you define an array of variables `x[(i, j)]`, whose value is 1 if item `i` is placed in bin `j`, and 0 otherwise.

For bin packing, you also define an array of variables, `y[j]`, whose value is 1 if bin `j` is used —that is, if any items are packed in it—and 0 otherwise. The sum of the `y[j]` will be the number of bins used.

Define the constraints

The following code defines the constraints for the problem:

Python (#python) C++ Java (#java) C# (#c) (#c++)

```
// Create the constraints.
// Each item is in exactly one bin.
for (int i = 0; i < data.num_items; ++i) {
  LinearExpr sum;
```

```
  for (int j = 0; j < data.num_bins; ++j) {
    sum += x[i][j];
  }
  solver->MakeRowConstraint(sum == 1.0);
}
// For each bin that is used, the total packed weight can be at most
// the bin capacity.
for (int j = 0; j < data.num_bins; ++j) {
  LinearExpr weight;
  for (int i = 0; i < data.num_items; ++i) {
    weight += data.weights[i] * LinearExpr(x[i][j]);
  }
  solver->MakeRowConstraint(weight <= LinearExpr(y[j]) * data.bin_capaci
}
```

The constraints are the following:

- Each item must be placed in exactly one bin. This constraint is set by requiring that the sum of `x[i][j]` over all bins `j` is equal to 1. Note that this differs from the multiple knapsack problem, in which the sum is only required to be less than or equal to 1, because not all items have to be packed.
- The total weight packed in each bin can't exceed its capacity. This is the same constraint as in the multiple knapsack problem, but in this case you multiply the bin capacity on the right side of the inequalities by `y[j]`.

  Why multiply by `y[j]`? Because it forces `y[j]` to equal 1 if any item is packed in bin `j`. This is so because if `y[j]` were 0, the right side of the inequality would be 0, while the bin weight on the left side would be greater than 0, violating the constraint. This connects the variables `y[j]` to the objective of the problem, for now the solver will try to minimize the number of bins for which `y[j]` is 1.

Define the objective

The following code defines the objective function for the problem.

Python (#python) C++ Java (#java) C# (#c) (#c++)

```
// Create the objective function.
MPObjective* const objective = solver->MutableObjective();
LinearExpr num_bins_used;
for (int j = 0; j < data.num_bins; ++j) {
  num_bins_used += y[j];
```

```
  }
  objective->MinimizeLinearExpr(num_bins_used);
```

Since `y[j]` is 1 if bin j is used, and 0 otherwise, the sum of the `y[j]` is the number of bins used. The objective is to minimize the sum.

Call the solver and print the solution

The following code calls the solver and prints the solution.

Python (#python) C++ (#c++) Java (#java) C# (#c)

```
const MPSolver::ResultStatus result_status = solver->Solve();
// Check that the problem has an optimal solution.
if (result_status != MPSolver::OPTIMAL) {
  std::cerr << "The problem does not have an optimal solution!";
  return;
}
std::cout << "Number of bins used: " << objective->Value() << std::endl
          << std::endl;
double total_weight = 0;
for (int j = 0; j < data.num_bins; ++j) {
  if (y[j]->solution_value() == 1) {
    std::cout << "Bin " << j << std::endl << std::endl;
    double bin_weight = 0;
    for (int i = 0; i < data.num_items; ++i) {
      if (x[i][j]->solution_value() == 1) {
        std::cout << "Item " << i << " - Weight: " << data.weights[i]
                  << std::endl;
        bin_weight += data.weights[i];
      }
    }
    std::cout << "Packed bin weight: " << bin_weight << std::endl
              << std::endl;
    total_weight += bin_weight;
  }
}
std::cout << "Total packed weight: " << total_weight << std::endl;
```

The solution shows the minimum number of bins required to pack all the items. For each bin that is used, the solution shows the items packed in it, and the total bin weight.

Output of the program

When you run the program, it displays the following output.

```
Bin number 0
  Items packed: [1, 5, 10]
  Total weight: 87

Bin number 1
  Items packed: [0, 6]
  Total weight: 90

Bin number 2
  Items packed: [2, 4, 7]
  Total weight: 97

Bin number 3
  Items packed: [3, 8, 9]
  Total weight: 96


Number of bins used: 4.0
```

Complete programs

The complete programs for the bin packing problem are shown below.

Python (#python) C++ (#c++) Java (#java) C# (#c)

```
#include <iostream>
#include <memory>
#include <numeric>
#include <ostream>
#include <vector>

#include "ortools/linear_solver/linear_expr.h"
#include "ortools/linear_solver/linear_solver.h"

namespace operations_research {
struct DataModel {
  const std::vector<double> weights = {48, 30, 19, 36, 36, 27,
                                       42, 42, 36, 24, 30};
  const int num_items = weights.size();
```

```
  const int num_bins = weights.size();
  const int bin_capacity = 100;
};

void BinPackingMip() {
  DataModel data;

  // Create the mip solver with the SCIP backend.
  std::unique_ptr<MPSolver> solver(MPSolver::CreateSolver("SCIP"));
  if (!solver) {
    LOG(WARNING) << "SCIP solver unavailable.";
    return;
  }

  std::vector<std::vector<const MPVariable*>> x(
      data.num_items, std::vector<const MPVariable*>(data.num_bins));
  for (int i = 0; i < data.num_items; ++i) {
    for (int j = 0; j < data.num_bins; ++j) {
      x[i][j] = solver->MakeIntVar(0.0, 1.0, "");
    }
  }
  // y[j] = 1 if bin j is used.
  std::vector<const MPVariable*> y(data.num_bins);
  for (int j = 0; j < data.num_bins; ++j) {
    y[j] = solver->MakeIntVar(0.0, 1.0, "");
  }

  // Create the constraints.
  // Each item is in exactly one bin.
  for (int i = 0; i < data.num_items; ++i) {
    LinearExpr sum;
    for (int j = 0; j < data.num_bins; ++j) {
      sum += x[i][j];
    }
    solver->MakeRowConstraint(sum == 1.0);
  }
  // For each bin that is used, the total packed weight can be at most
  // the bin capacity.
  for (int j = 0; j < data.num_bins; ++j) {
    LinearExpr weight;
    for (int i = 0; i < data.num_items; ++i) {
      weight += data.weights[i] * LinearExpr(x[i][j]);
    }
    solver->MakeRowConstraint(weight <= LinearExpr(y[j]) * data.bin_capa
  }

  // Create the objective function.
  MPObjective* const objective = solver->MutableObjective();
```

```
  LinearExpr num_bins_used;
  for (int j = 0; j < data.num_bins; ++j) {
    num_bins_used += y[j];
  }
  objective->MinimizeLinearExpr(num_bins_used);

  const MPSolver::ResultStatus result_status = solver->Solve();

  // Check that the problem has an optimal solution.
  if (result_status != MPSolver::OPTIMAL) {
    std::cerr << "The problem does not have an optimal solution!";
    return;
  }
  std::cout << "Number of bins used: " << objective->Value() << std::enc
            << std::endl;
  double total_weight = 0;
  for (int j = 0; j < data.num_bins; ++j) {
    if (y[j]->solution_value() == 1) {
      std::cout << "Bin " << j << std::endl << std::endl;
      double bin_weight = 0;
      for (int i = 0; i < data.num_items; ++i) {
        if (x[i][j]->solution_value() == 1) {
          std::cout << "Item " << i << " - Weight: " << data.weights[i]
                    << std::endl;
          bin_weight += data.weights[i];
        }
      }
      std::cout << "Packed bin weight: " << bin_weight << std::endl
                << std::endl;
      total_weight += bin_weight;
    }
  }
  std::cout << "Total packed weight: " << total_weight << std::endl;
}
}  // namespace operations_research

int main(int argc, char** argv) {
  operations_research::BinPackingMip();
  return EXIT_SUCCESS;
}
```

Last updated 2023-01-18 UTC.