# Project report - Integer Programming

Klaas Fiete Krutein

March 3, 2020

The objective of this project is to assess the effect of different solver methods on the solution time and the complexity of the algorithm. This report is presented as a combined code and report script. Its structure follows the following logic. After defining the model formulations using the pyomo modeling interface package and importing the source data, instances of each data set and each model are generated that are tested with different parameter settings. The solver package used is Gurobi 9.0. Following this the report continues with an analysis section and finishes with a comparative analysis of the model outputs.

## Model formulations

[1]:
```
from pyomo.environ import *
import glob
import pandas as pd
import time
import numpy as np
```

We begin by modeling the assigment relaxation formulation. The data, parameter, variable and constraint formulations correspond to the formulation as given in the project description.

[2]:
```
## Modeling framework assignment relaxation formulation (0)

def assignment_relax(dist):

    # Create model
    m = ConcreteModel()

    ## DATA

    # initialize number of points in problem
    m.P = Param(within=PositiveIntegers, initialize=len(dist)-1)
    # initialize indices for points in problem
    m.V = RangeSet(0,m.P)

    # Transform matrix to dictionary
    dist.to_dict()
    dist = dist.stack().to_dict()
    #print(dist)
    ## PARAMETERS

    # initialize cost parameters
    m.c = Param(m.V, m.V, initialize = dist)
    #m.c.pprint()

    ## VARIABLES

    # connection variable
    m.x = Var(m.V, m.V, within = Binary, initialize = 0)
    #m.x.pprint()

    ## CONSTRAINTS

    # Constraint 1b
    def single_connect1(model, i, j):
        return(sum(m.x[i,j] for j in m.V) == 1)
    m.s_con1 = Constraint(m.V, m.V, rule = single_connect1)
    #m.s_con1.pprint()

    # Constraint 1c
    def single_connect2(model, i, j):
        return(sum(m.x[i,j] for i in m.V) == 1)
    m.s_con2 = Constraint(m.V, m.V, rule = single_connect2)
    #m.s_con2.pprint()

    # Constraint 1d
    def no_self_routes(model, i):
        return(m.x[i,i] == 0)
    m.zeros = Constraint(m.V, rule = no_self_routes)
    #m.zeros.pprint()

    ## OBJECTIVE

    def objective_rule(m):
        return(sum(sum(m.c[i,j] * m.x[i,j] for j in m.V) for i in m.V))
    m.objective = Objective(rule = objective_rule, sense = minimize,␣
↪doc='Define objective function')

    return(m)
```

We continue with the MTZ formulation. As for the assignment relaxation formulation, the definitions follow the definitions from the project description.

[3]:
```
## Modeling framework MTZ formulation

# For testing purposes of random starting vertex
```

```
np.random.seed(123)

def MTZ(dist):

    # Create model
    m = ConcreteModel()

    ## DATA

    # initialize number of points in problem
    m.P = Param(within=PositiveIntegers, initialize=len(dist)-1)
    # initialize indices for points in problem
    m.V = RangeSet(0,m.P)

    # Transform matrix to dictionary
    dist.to_dict()
    dist = dist.stack().to_dict()
    #print(dist)

    ## PARAMETERS

    # initialize cost parameters
    m.c = Param(m.V, m.V, initialize = dist)
    #m.c.pprint()

    ## VARIABLES

    # connection variable
    m.x = Var(m.V, m.V, within = Binary, initialize = 0)
    #m.x.pprint()

    # tour position variable
    m.u = Var(m.V, within = NonNegativeReals, bounds = (0,(len(m.V)-1)),␣
 ↪initialize = 0)
    #m.u.pprint()

    ## CONSTRAINTS

    # generate random starting vertex
    r = int(np.random.uniform(0,28,1))
    #print(r)

    # Constraint 1b
    def single_connect1(model, i, j):
        return(sum(m.x[i,j] for j in m.V) == 1)
    m.s_con1 = Constraint(m.V, m.V, rule = single_connect1)
    #m.s_con1.pprint()
```

```
# Constraint 1c
def single_connect2(model, i, j):
    return(sum(m.x[i,j] for i in m.V) == 1)
m.s_con2 = Constraint(m.V, m.V, rule = single_connect2)
#m.s_con2.pprint()

# Constraint 1d
def no_self_routes(model, i):
    return(m.x[i,i] == 0)
m.zeros = Constraint(m.V, rule = no_self_routes)
#m.zeros.pprint()

# Constraint 3a
def order(model, i, j):
    if i == r or j == r:
        return(Constraint.Skip)
    else:
        return(m.u[i] - m.u[j] + 1 <= (len(m.V)-1) * (1 - m.x[i,j]))
m.order_con = Constraint(m.V, m.V, rule = order)
#m.order_con.pprint()

# Constraint 3b
def initial_vertex(model, i):
    if i == r:
        return(m.u[i] == 0) # using python syntax we start indexing at␣
↪position zero
    else:
        return(Constraint.Skip)
m.start_at = Constraint(m.V, rule = initial_vertex)
#m.start_at.pprint()

# Constraint 3c
def later_vertex(model, i):
    if i == r:
        return(Constraint.Skip)
    else:
        return(inequality(1, m.u[i], len(m.V)-1)) # again we need to␣
↪consider the indexing change
m.continue_at = Constraint(m.V, rule = later_vertex)
#m.continue_at.pprint()

## OBJECTIVE

def objective_rule(m):
    return(sum(sum(m.c[i,j] * m.x[i,j] for j in m.V) for i in m.V))
    m.objective = Objective(rule = objective_rule, sense = minimize,␣
↪doc='Define objective function')

    return(m)
```

Lastly, we create the function for the MCF formulation, which also follows the same description as the project task.

[4]:
```
## Modeling framework MCF formulation (3)

# For testing purposes of random starting vertex
np.random.seed(123)

def MCF(dist):

    # Create model
    m = ConcreteModel()

    ## DATA

    # initialize number of points in problem
    m.P = Param(within=PositiveIntegers, initialize=len(dist)-1)
    # initialize indices for points in problem
    m.V = RangeSet(0,m.P)
    #m.V.pprint()

    # Transform matrix to dictionary
    dist.to_dict()
    dist = dist.stack().to_dict()
    #print(dist)

    ## PARAMETERS

    # initialize cost parameters
    m.c = Param(m.V, m.V, initialize = dist)
    #m.c.pprint()

    ## VARIABLES

    # connection variable
    m.x = Var(m.V, m.V, within = Binary, initialize = 0)
    #m.x.pprint()

    # flow variable
    m.f = Var(m.V, m.V, m.V, within = NonNegativeReals, bounds = (0, 1),␣
↪initialize = 0)
```

```python
    ## CONSTRAINTS

    # generate random starting vertex
    r = int(np.random.uniform(0,28,1))
    #print(r)

    # generate a subset of vertices that excludes r
    m.sub_V = Set(dimen = 1)
    for k in m.V:
        if k != r:
            m.sub_V.add(k)
    #m.sub_V.pprint()

    # Constraint 1b
    def single_connect1(model, i, j):
        return(sum(m.x[i,j] for j in m.V) == 1)
    m.s_con1 = Constraint(m.V, m.V, rule = single_connect1)
    #m.s_con1.pprint()

    # Constraint 1c
    def single_connect2(model, i, j):
        return(sum(m.x[i,j] for i in m.V) == 1)
    m.s_con2 = Constraint(m.V, m.V, rule = single_connect2)
    #m.s_con2.pprint()

    # Constraint 1d
    def no_self_routes(model, i):
        return(m.x[i,i] == 0)
    m.zeros = Constraint(m.V, rule = no_self_routes)
    #m.zeros.pprint()

    # Constraint 4a
    def flows_from_r(model, v):
        if v == r:
            return(Constraint.Skip)
        else:
            return(sum(m.f[v,r,j] for j in m.sub_V) - sum(m.f[v,j,r] for j in m.
↪sub_V) == 1)
    m.out_flows = Constraint(m.V, rule = flows_from_r)
    #m.out_flows.pprint()

    # Constraint 4b
    def flow_conservation(model, i, v):
        if i == r or v == r or i == v:
            return(Constraint.Skip)
        else:
            # generate a subset of vertices that excludes i
            m.sub_V1 = Set(dimen = 1)
            for k in m.V:
                if k != i:
                    m.sub_V1.add(k)
            #m.sub_V1.pprint()
            constr = (sum(m.f[v,i,j] for j in m.sub_V1) - sum(m.f[v,j,i] for j␣
↪in m.sub_V1) == 0)
            m.del_component(m.sub_V1)
            return(constr)
    m.flow_con = Constraint(m.V, m.V, rule = flow_conservation)
    #m.flow_con.pprint()

    # Constraint 4c
    def link(model, i, j, v):
        if v == r:
            return(Constraint.Skip)
        else:
            return(m.f[v,i,j] <= m.x[i,j])
    m.link_con = Constraint(m.V, m.V, m.V, rule = link)

    ## OBJECTIVE

    def objective_rule(m):
        return(sum(sum(m.c[i,j] * m.x[i,j] for j in m.V) for i in m.V))
    m.objective = Objective(rule = objective_rule, sense = minimize,␣
↪doc='Define objective function')

    return(m)
```

## Data

Having defined the model formulations, we can continue with reading in the data and creating model formulation instances for each formulation for each testing dataset. We begin by reading in the data.

[5]:
```
# read in source files for test data
source = r'/Users/fietekrutein/Documents/University/University of Washington/
↪Courses/2020 Q1/IND E 599 Integer Programming/Project' # use your path
all_files = glob.glob(source + "/*.txt")

# test datasets
bays29 = pd.read_csv(source + '/bays29.txt', delimiter = "\t", skiprows = 1,␣
↪header = None)
dantzig42 = pd.read_csv(source + '/dantzig42.txt', delimiter = "\t", skiprows =␣
↪1, header = None)
pr76 = pd.read_csv(source + '/pr76.txt', delimiter = "\t", skiprows = 1, header␣
↪= None)
```

```
rat99 = pd.read_csv(source + '/rat99.txt', delimiter = "\t", skiprows = 1,␣
↪header = None)
```

We now generate instances of the assignment relaxation formulation for each dataset.

[6]:
```
# create assignment relaxation formulations for each dataset
bays29_assignment_relax_m = assignment_relax(bays29)
dantzig42_assignment_relax_m = assignment_relax(dantzig42)
pr76_assignment_relax_m = assignment_relax(pr76)
rat99_assignment_relax_m = assignment_relax(rat99)
```

We further generate instances for the MTZ formulation.

[7]:
```
# create MTZ formulations for each dataset
bays29_MTZ_m = MTZ(bays29)
dantzig42_MTZ_m = MTZ(dantzig42)
pr76_MTZ_m = MTZ(pr76)
rat99_MTZ_m = MTZ(rat99)
```

Lastly, we create model instances for the MCF formulation.

[8]:
```
# create MCF formulations for each dataset
bays29_MCF_m = MCF(bays29)
dantzig42_MCF_m = MCF(dantzig42)
pr76_MCF_m = MCF(pr76)
rat99_MCF_m = MCF(rat99)
```

## Solver Definition

We can now proceed with the solver definition. It was decided to create a single solver function that, once called, iterates through the solving process for the passed instance and changing the parameter settings on *Presolve* and *Cuts* and solving them accordingly for each instance. The function therefore returns four results:

1. Default settings
2. No pre-solve
3. No cuts
4. No pre-solve and no cuts

We further pass it an indicator whether to only solve the default solver, since we will not need the alternative solver configurations for the assignment relaxation formulation. We use Gurobi 9.0 to solve the problems.

[9]:
```
# Define solving instance
from gurobipy import *

timelimit = 3600 # in seconds
```

```
def solve(m, default):

    ## SOLVE

    if __name__ == '__main__':
        from pyomo.opt import SolverFactory
        import pyomo.environ

        # Create instances for each model setting
        m_default = m
        if default == False:
            m_no_presolve = m
            m_no_cuts = m
            m_no_presolve_no_cuts = m

        opt = SolverFactory('gurobi')
        opt.options['IntFeasTol']= 10e-10
        opt.options['MIPGap'] = 0
        opt.options['TimeLimit'] = timelimit

        # Solve default
        print('')
        print('DEFAULT')
        print('')
        results_default = opt.solve(m_default, load_solutions=False, tee = True)
        print(results_default)


        if default == False:
            # Solve without presolve if enabled
            print('')
            print('NO PRESOLVE')
            print('')
            opt.options['Presolve'] = 0 # disable presolve
            results_no_presolve = opt.solve(m_no_presolve,␣
 ↪load_solutions=False, tee = True)
            print(results_no_presolve)


            # Solve without cuts if enabled
            print('')
            print('NO CUTS')
            print('')
            opt.options['Presolve'] = -1
            opt.options['Cuts'] = 0 # disable cuts
            results_no_cuts = opt.solve(m_no_cuts, load_solutions=False, tee =␣
 ↪True)
```

```
            print(results_no_cuts)


            # Solve without cuts and without presolve if enabled
            print('')
            print('NO PRESOLVE NO CUTS')
            print('')
            opt.options['Presolve'] = 0 # disable presolve
            opt.options['Cuts'] = 0 # disable cuts
            results_no_presolve_no_cuts = opt.solve(m_no_presolve_no_cuts,␣
↪load_solutions=False, tee = True)
            print(results_no_presolve_no_cuts)
```

### Results

We continue by solving each model instance using the solver function defined above. Since the model output is quite large in size and would not fit the format of a report very well, we comment out the solver command in this script and report the model results in table format below. We start with the assignment relaxation formulation.

[10]:
```
## Assignment relaxation formulation
# solve each model instances
#print('Assignment relaxation formulation')
#print('Bays29 Dataset:')
#solve(bays29_assignment_relax_m, default = True)
```

[11]:
```
#print('Assignment relaxation formulation')
#print('Dantzig42 Dataset:')
#solve(dantzig42_assignment_relax_m, default = True)
```

[12]:
```
#print('Assignment relaxation formulation')
#print('Pr76 Dataset:')
#solve(pr76_assignment_relax_m, default = True)
```

[13]:
```
#print('Assignment relaxation formulation')
#print('Rat99 Dataset:')
#solve(rat99_assignment_relax_m, default = True)
```

The obtained results from the assignment relaxation formulation are presented in Table 1.

Table 1: Results for the assignment relaxation formulation

| Dataset | Objective Value | Solution time in seconds |
|---|---:|---:|
| Bays29 | 1,764 | 0.29 |
| Dantzig42 | 532 | 0.29 |
| Pr76 | 77,119 | 1.29 |
| Rat99 | 1,089 | 2.63 |

The results show a low computation time. However, we know that the results from the assignment relaxation formulation can contain subtours. The formulation is therefore not feasible to the actual TSP problem without subtours. We therefore continue with our code analysis towards the MTZ and the MCF formulation.

[14]:
```
## MTZ formulation
# solve each model instances
#print('MTZ formulation')
#print('Bays29 Dataset:')
#solve(bays29_MTZ_m, default = False)
```

[15]:
```
#print('MTZ formulation')
#print('Dantzig42 Dataset:')
#solve(dantzig42_MTZ_m, default = False)
```

[16]:
```
#print('MTZ formulation')
#print('Pr76 Dataset:')
#solve(pr76_MTZ_m, default = False)
```

[17]:
```
#print('MTZ formulation')
#print('Rat99 Dataset:')
#solve(rat99_MTZ_m, default = False)
```

We obtain the results presented in Table 2 from the MTZ formulation.

Table 2: Results for the MTZ formulation

| **Dataset** | **Settings** | | **Opt.** | **Obj.** | **UB** | **LB** | **Nodes** | **Cuts** | **Time** |
|---|---|---|---|---|---|---|---|---|---|
| | Presolve | Cuts | | | | | | | |
| Bays29 | Yes | Yes | Yes | 2,020 | | | 1,192 | 55 | 2.71 |
| | No | Yes | Yes | 2,020 | | | 1,690 | 164 | 9.05 |
| | Yes | No | Yes | 2,020 | | | 4,689 | 0 | 6.21 |
| | No | No | Yes | 2,020 | | | 5431 | 0 | 13.04 |
| Dantzig42 | Yes | Yes | Yes | 699 | | | 8,147 | 118 | 24.16 |
| | No | Yes | Yes | 699 | | | 5,679 | 155 | 54.94 |
| | Yes | No | Yes | 699 | | | 1,192,740 | 0 | 875.67 |
| | No | No | No | | 699 | 665 | 1,419,533 | 0 | 3,601.70 |
| Pr76 | Yes | Yes | No | | 108,159 | 107,222 | 535,655 | 749 | 3,601.57 |
| | No | Yes | No | | 108,159 | 106,810 | 188,056 | 1,789 | 3,601.24 |
| | Yes | No | No | | 110,314 | 97,686 | 1,204,286 | 0 | 3,600.10 |
| | No | No | No | | 108,159 | 92,076 | 320,975 | 0 | 3,600.03 |
| Rat99 | Yes | Yes | Yes | 1,211 | | | 2,558 | 339 | 59.26 |
| | No | Yes | Yes | 1,211 | | | 2,614 | 495 | 87.24 |
| | Yes | No | No | | 1,211 | 1,183 | 696,304 | 0 | 3,603.04 |
| | No | No | No | | 1,211 | 1,173 | 133,105 | 0 | 3,601.97 |

Before further diving into the analysis, we can already identify that larger problems create a significantly longer solution time for the MTZ formulation. Further, under default settings and without pre-solve the problem using the larger Rat99 dataset appears to be easier to solve than the Pr76 dataset.

[18]:
```
## MCF formulation
# solve each model instances
#print('MCF formulation')
#print('Bays29 Dataset:')
#solve(bays29_MCF_m, default = False)
```

[19]:
```
#print('MCF formulation')
#print('Dantzig42 Dataset:')
#solve(dantzig42_MCF_m, default = False)
```

[20]:
```
#print('MCF formulation')
#print('Pr76 Dataset:')
#solve(pr76_MCF_m, default = False)
```

[21]:
```
#print('MCF formulation')
#print('Rat99 Dataset:')
#solve(rat99_MCF_m, default = False)
```

The results for the MCF formulation are presented in Table 3.

Table 3: Results for the MCF formulation

| **Dataset** | **Settings** | | **Opt.** | **Obj.** | **UB** | **LB** | **Nodes** | **Cuts** | **Time** |
|---|---|---|---|---|---|---|---|---|---|
| | Presolve | Cuts | | | | | | | |
| Bays29 | Yes | Yes | Yes | 2,020 | | | 15 | 2 | 7.21 |
| | No | Yes | Yes | 2,020 | | | 13 | 0 | 9.45 |
| | Yes | No | Yes | 2,020 | | | 1 | 0 | 5.57 |
| | No | No | Yes | 2,020 | | | 21 | 0 | 8.80 |
| Dantzig42 | Yes | Yes | Yes | 699 | | | 1 | 1 | 61.56 |
| | No | Yes | Yes | 699 | | | 29 | 5 | 67.95 |
| | Yes | No | Yes | 699 | | | 1 | 0 | 61.32 |
| | No | No | No | 699 | | | 27 | 0 | 69.06 |
| Pr76 | Yes | Yes | No | | $\infty$ | 105,149 | 7 | 3 | 3,604.84 |
| | No | Yes | No | | 108,613 | 105,120 | 9 | 2 | 3,607.20 |
| | Yes | No | No | | $\infty$ | 105,160 | 26 | 0 | 3,604.87 |
| | No | No | No | | 108,613 | 105,344 | 332 | 0 | 3,606.30 |
| Rat99 | Yes | Yes | Yes | 1,211 | | | 63 | 9 | 3,056.52 |
| | No | Yes | No | | 4,394 | 0 | 0 | 0 | 3,602.00 |
| | Yes | No | Yes | 1,211 | | | 65 | 0 | 3,030.84 |
| | No | No | No | | 4,394 | 0 | 0 | 0 | 3,602.03 |

Similarly to the results from the MTZ formulation, we can see that the Pr76 dataset seems to be more difficult to solve than the Rat99 dataset. We now proceed by answering the questions provided in the project instructions in the analysis section.

## Analysis

### The relative strength of the LP relaxations of the MTZ and the MCF formulations

From the model description in the project document we already know that the MCF formulation describes the integer set more accurately and is stronger than the MTZ formulation. It is furthermore as strong as the cut formulation. However, to illustrate this point, we can investigate the optimal solutions of the LP relaxations as returned by the solver. These are reported in Table 4.

Table 4: LP relaxation optima

| **Dataset** | **Optimum** | **LP Relaxation optimum** | |
|---|---|---|---|
| | | MTZ | MCF |
| Bays29 | 2,020 | 1175.07 | 2,013.50 |
| Dantzig42 | 699 | 535.44 | 697 |
| Pr76 | 108,159 | 77,648.79 | 105,120 |
| Rat99 | 1,211 | 1,090.92 | 1,206 |

It is clearly visible that the MCF formulation is describing the feasible integer set better than the MTZ formulation since the optimal solution of the relaxation is already very close to the optimum of the integer problem. This proximity also explains why the number of explored nodes is a lot lower during the solution process for the MCF formulation than for the MTZ formulation. Therefore, the MCF formulation is a lot stronger than the MTZ formulation.

### Effects of the stronger formulation on the number of Branch-and-Bound nodes

Similarly, following the logic from the previous sub section, the MCF formulation does not allow to explore many nodes in the area around the optimal solutions since the space is tightly constrained. We can see in Table 3 that in comparison to Table 2 the number of explored nodes is only a tiny fraction of the nodes explored in the MTZ formulation. This also leads to a significantly lower number of cutting planes introduced, since there is just not that much potential for valid inequalities, given the large number of existing constraints.

### Effects of the stronger but larger formulation on the total runtime

Tables 2 and 3 show that the run time is comparable for small data sets and if all solver settings are enabled. However, the larger MCF formulation generates a large number of constraints that already take quite a while to be initialized. Since their number is large, with increasing problem size, the time required to even check whether a solution is feasible to all the constraints takes a lot of time. Table 3 shows that for the larger datasets Pr76 and Rat66, most solver settings cause the solver not to converge within the given maximum run time of one hour. While that is also true for the MTZ formulation, this increase is more extreme and results in some problems not even finding a single feasible solution within one hour, despite the small number of nodes explored. This effect is not clearly visible in the reported model output, but it can be obtained through running the code included in this script. In conclusion, we can derive that the larger formulation is stronger but does not provide as much potential to use smart solver techniques such as cutting planes and pre-solve and in general makes it difficult for a solver to evaluate feasible solutions since the constraint set grows exponentially.

## Importance of presolve and good cuts

As mentioned previously, the results from both the MTZ and the MCF formulation show that enabling pre-solving, which reduces the problem set by redundant constraints before starting the core branch and bound algorithm, causes different effects on the two formulations. The MTZ formulation gets solved significantly quicker with presolve enabled, as Table 2 shows. For instances, which converge to the optimum within the time limit, this reveals itself through a lower total run time and a lower number of cuts added. For instances that do not converge within one hour, only the reduced number of cuts reveals the effect of the presolve importance. For the MCF formulation, the effect of disabling presolve has a much smaller effect since the feasible region is already pretty well constrained to approximate the integer set. This is valid both for the run time and the number of cuts. While the performance improvement is still noticeable, especially for larger datasets, it is less significant then for the MTZ formulation.

For cuts, we can observe similar dynamics as for presolve. For the MTZ formulation, enabling cuts leads to much lower run times for instances that are of relatively large size. These cuts cause much narrower bounds, as can be observed in Table 2 for the Pt76 dataset. As with presolve, the effect of enabling cuts in the MCF formulation is smaller than for the MTZ formulation, as visible in Table 3. The effects are so small, such that for smaller instances it even appears that enabling cuts makes the algorithm even slightly slower. This could be caused by a higher complexity of the solving algorithm.

In summary, it is visible that enabling presolve and cutting planes have a larger effect on the run times of the weaker formulation. With these solver capabilities enabled, the MTZ formulation actually becomes faster in run time than the MCF formulation. Without these capabilities, the run time of the MCF is faster as long as the problem dimensions are not too large.

## Comparison to Concorde and conclusions

When we compare the results obtained to the benchmark in formation from the Concorde solver we receive the results presented in Table 5.

Table 5: Results from Concorde

| **Dataset** | **Solution time in seconds**[1] |
|---|---:|
| Bays29 | 0.04 |
| Dantzig42 | 0.09 |
| Pr76 | 0.60 |
| Rat99 | 0.40 |

We can see that the run times are a lot faster than any of the result obtained form the formulations tested for this project. We know that the cut formulation that is used by the Concorde solver is as strong as the MCF formulation in the $x$-space. However, it achieves this with a much smaller constraint set and is therefore avoiding the issues that the MCF formulation is facing in evaluating a new solution over the constraint set for larger datasets. Hence, it does not require a large number of cuts or branch and bound iterations and still has the properties that the constraint set is so tightly defined that only very few solutions need to be evaluated.

In conclusion, we can draw from this project that the formulation of an integer program strongly influences its complexity and thus the computational effort necessary to solve it within a limited time horizon. The project results further show that efficient solvers, using a variety of solution techniques, such as cutting planes and presolves, can reduce the computational effort significantly. Since weaker formulations with less constraints have more potential to tighten the constraint set it is an obvious consequence that using these techniques improves the solution time more significantly than on strong formulations. Since some strong formulations also constrain the feasible space very tightly far away from the optimal solution, the large number of constraints can therefore be a burden to the solver and is the reason that efficient solution techniques applied to weaker formulations can outrun stronger formulations through strategically applying a combination of smart solution techniques in regions close to the optimal solution. A combination of a strong formulation with a small number of constraints is therefore obviously the best choice, which is confirmed by the results from the Concorde solver.

## References

[1] *Benchmark Information*, University of Waterloo, 2003.