Mobility Meeting Scheduler

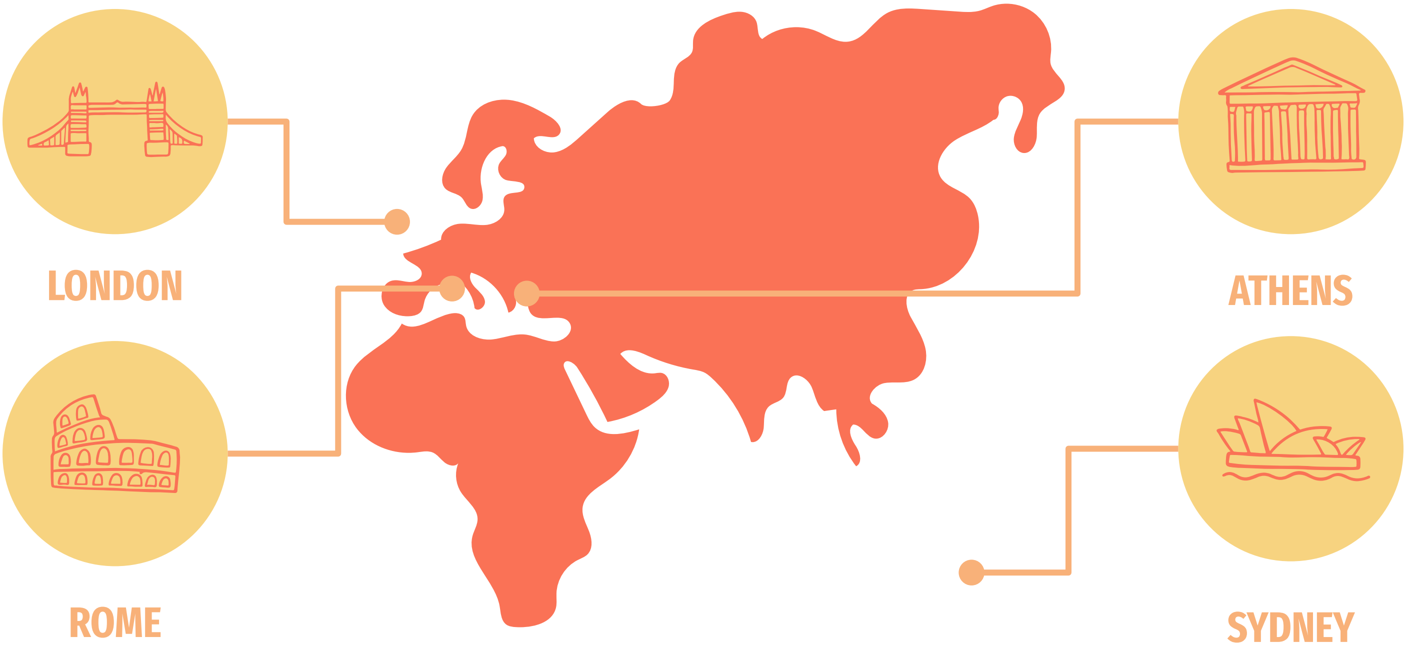
Motivation

Background

Initially, we thought about exchange students:

- Friends can go to different cities in the same semester
- But they encounter themselves in a time dedicated to visiting different places
- So, they can decide to make a trip together
- This implies some variables and constraints that will be discussed in the next slides

List of possible destinations



Individual constraints

TIME

Each person has different available days.

ORIGIN

Everyone needs to start and finish the trip in their city.

PREFERENCES

Each person has their own flight preferences.

The plan for each person

OUTGOING FLIGHT

Go to your destination

DESTINATION

The most important part of the plan

INCOMING FLIGHT

Get back home

Goals

TIME TOGETHER

Spend more time in the destination together.

COST

Spend less money, considering both the total and individual costs.

SEPARATED TIME

Spend less time in the destination waiting for others to arrive.

Formalization

Data models

FLIGHT

- Origin
- Destination
- Duration
- Departure
- Arrival
- Price
- Number of connections

STUDENT

- Current City
- Max connections
- Max duration
- Earliest flight
- Latest flight
- Availability

INPUT

- Students
- Minimum useful time
- List of destinations

SOLUTION

- List of pairs, having, for each student:
  - Outgoing trip
  - Incoming trip

Constraints

1

2

3

DESTINATION

Destination must be in the list of possible destinations

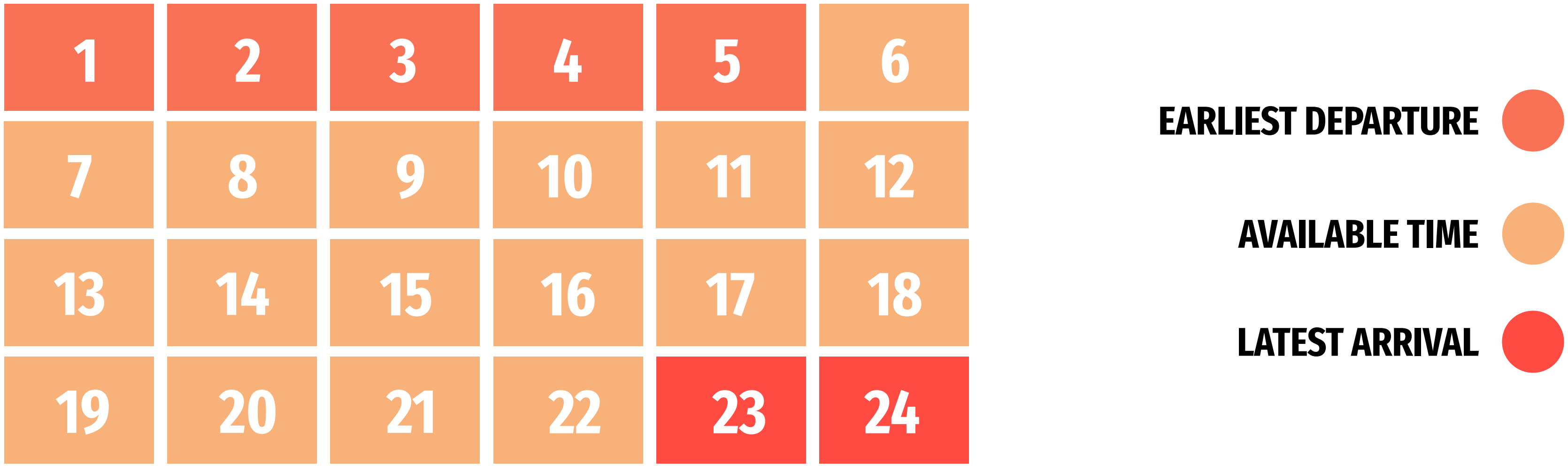
FLIGHT

Outgoing and incoming flights must obey to the following constraints

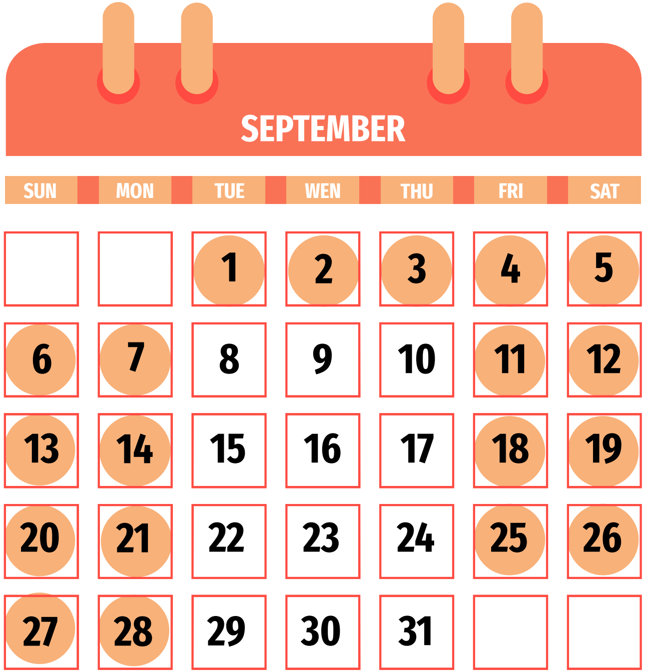
TIME

Group members must be together for a minimum useful time

Flight Constraints



Flight Constraints



AVAILABILITY

Cost function

X * COST + Y * TIME TOGETHER + Z * TIME WAITING

$$\frac{X * \text{COST}}{Y * \text{TIME TOGETHER} + Z * \text{TIME WAITING}}$$

Solutions

Input

In order to properly compare the different solvers, the same input needed to be used, which included:

- **3** students (city, availability, maxConnections, maxDuration, earliestDeparture, latestArrival)
- **113 / 9311** flights (origin, destination, departure, arrival, duration, price, stops)
- Minimum Time
- Destinations

```json
[
    {
        "origin": "budapest",
        "destination": "zagreb",
        "departure": "01/06/2022, 08:50:00",
        "arrival": "01/06/2022, 16:55:00",
        "duration": 485,
        "price": "130",
        "stops": 1
    },
    ...
]
```

```json
{
    "students": [
        {
            "city": "budapest",
            "availability": [
                ["01/06/2022", "17/06/2022"],
                ["24/06/2022", "01/07/2022"]
            ],
            "maxConnections": 1,
            "maxDuration": 600,
            "earliestDeparture": "05:30:00",
            "latestArrival": "23:30:00"
        },
        ...
    ],
    "minimumTime": 1440,
    "destinations": [
        "budapest",
        "zagreb",
        "milano",
        "wien",
        "krakow",
        "rome",
        "paris"
    ]
}
```

DOcplex

Decision Variables

To solve this problem, the following decision variables were created:

```python
# Chosen Destination
Destination = model.integer_var(0, len(DESTINATIONS) - 1, "Destination")

# Indexes of the flights each student has to take
StudentsFlights = [model.integer_var_list(2, 0, len(FLIGHTS) - 1) for i in range(N_STUDENTS)]

# Student avaliability interval
StudentsAvailabilityIntervals = model.integer_var_list(N_STUDENTS, 0, N_MAX_INTERVALS, "Interval")

# Cost for each of the students
StudentsCosts = model.integer_var_list(N_STUDENTS, 0, MAXIMUM_FLIGHT_COST, "StudentCost")

# Total trip cost
TotalCost = model.integer_var(0, MAXIMUM_FLIGHT_COST * N_STUDENTS, "TotalCost")

# Useful time
UsefulTime = model.integer_var(0, MAXIMUM_USEFUL_TIME, "UsefulTime")

# Separated Time
SeparatedTime = model.integer_var(0, MAXIMUM_USEFUL_TIME, "SeparatedTime")
```

Constraints

For each of the students, the following constraints were applied:

```
# Origin of the flights
model.add(model.if_then(StudentOrigin != Destination, model.element(FLIGHTS_ORIGINS, Outgoing) == StudentOrigin))
model.add(model.if_then(StudentOrigin != Destination, model.element(FLIGHTS_ORIGINS, Incoming) == Destination))

# Destination of the flights
model.add(model.if_then(StudentOrigin != Destination, model.element(FLIGHTS_DESTINATIONS, Outgoing) == Destination))
model.add(model.if_then(StudentOrigin != Destination, model.element(FLIGHTS_DESTINATIONS, Incoming) == StudentOrigin))
```

Origin / Destination

Constraints

For each of the students, the following constraints were applied:

```
# Availability
startAvailability = model.element(STUDENS_AVAILABILITIES[i], StudentsAvailabilityIntervals[i] * 2)
endAvailability = model.element(STUDENS_AVAILABILITIES[i], StudentsAvailabilityIntervals[i] * 2 + 1)
model.add(model.if_then(StudentOrigin != Destination, model.element(FLIGHTS_DEPARTURES, Outgoing) >= startAvailability))
model.add(model.if_then(StudentOrigin != Destination, model.element(FLIGHTS_ARRIVALS, Incoming) <= endAvailability))
```

Availability

Constraints

For each of the students, the following constraints were applied:

```
# Outgoing arrival time must be before Incoming departure time
model.add(model.if_then(StudentOrigin != Destination, model.element(FLIGHTS_ARRIVALS, Outgoing) < model.element(FLIGHTS_DEPARTURES, Incoming)))

# Earliest departure
model.add(model.if_then(StudentOrigin != Destination, model.element(FLIGHTS_DEPARTURE_TIMES, Outgoing) >= model.element(STUDENTS_DEPARTURES, i)))
model.add(model.if_then(StudentOrigin != Destination, model.element(FLIGHTS_DEPARTURE_TIMES, Incoming) >= model.element(STUDENTS_DEPARTURES, i)))

# Latest arrival
model.add(model.if_then(StudentOrigin != Destination, model.element(FLIGHTS_ARRIVAL_TIMES, Outgoing) <= model.element(STUDENTS_ARRIVALS, i)))
model.add(model.if_then(StudentOrigin != Destination, model.element(FLIGHTS_ARRIVAL_TIMES, Incoming) <= model.element(STUDENTS_ARRIVALS, i)))
```

Flight Times

Constraints

For each of the students, the following constraints were applied:

```
# Maximum number of stops
model.add(model.if_then(StudentOrigin != Destination, model.element(FLIGHTS_STOPS, Outgoing) <= model.element(STUDENTS_STOPS, i)))
model.add(model.if_then(StudentOrigin != Destination, model.element(FLIGHTS_STOPS, Incoming) <= model.element(STUDENTS_STOPS, i)))

# Maximum flight duration
model.add(model.if_then(StudentOrigin != Destination, model.element(FLIGHTS_DURATIONS, Outgoing) <= model.element(STUDENTS_DURATIONS, i)))
model.add(model.if_then(StudentOrigin != Destination, model.element(FLIGHTS_DURATIONS, Incoming) <= model.element(STUDENTS_DURATIONS, i)))

# Student Cost
studentCost = model.element(FLIGHTS_COSTS, StudentsFlights[i][0]) + model.element(FLIGHTS_COSTS, StudentsFlights[i][1])
model.add(model.conditional(StudentOrigin == Destination, 0, studentCost) == StudentsCosts[i])
```

Stops / Duration / Student Cost

Constraints

The following constraints were also applied to the model:

```python
# Useful Time
model.add(UsefulTime == firstIncomingTime - lastOutgoingTime)
model.add(UsefulTime >= MINIMUM_USEFUL_TIME)

# Separated Time
model.add(SeparatedTime == (lastOutgoingTime - firstOutgoingTime) + (lastIncomingTime - firstIncomingTime))

# Total cost
model.add(TotalCost == model.sum(StudentsCosts))
```

Useful Time / Separated Time / Total Cost

Objective

A minimization function was applied:

```
# Minimize function
model.add(model.minimize(TotalCost / UsefulTime))
```

Conclusion

- DOcplex implementation is very intuitive and easy to understand.
- The only downside is the extra parsing needed since the *model.element()* function can only access arrays of integers or floats, and not objects.
- After parsing every flight attribute to a list of integers, where the flight variable was the index, the solution was easily attained.

OR-Tools

Decision Variables

Flights

Each flight has a number which corresponds to the student who used that flight

Destinations

One variable for each of the possible destinations

Optimization Variables

Variables that were used only for the optimization process

Constraints

For each of the students:

```python

# origin must be student city
if (flights[i].origin != students[j].city):
    model.Add(output_flights[i] != outgoing)
# destination must be student city
if (flights[i].destination != students[j].city):
    model.Add(output_flights[i] != incoming)
# if destination is student city it can't be outgoing flight
if (flights[i].destination == students[j].city):
    model.Add(output_flights[i] != outgoing)
# if origin is student city it can't be incoming flight
if (flights[i].origin == students[j].city):
    model.Add(output_flights[i] != incoming)

```

```python

# flight must have no more than max_connections
if (flights[i].stops > students[j].max_connections):
    model.Add(output_flights[i] != outgoing)
    model.Add(output_flights[i] != incoming)

# flight duration must be less or equal to max_duration
if (flights[i].duration > students[j].max_duration):
    model.Add(output_flights[i] != outgoing)
    model.Add(output_flights[i] != incoming)
```

Constraints

For each of the possible destinations:

```

# All outgoing flights must have the same destination
if (flights[i].destination != destinations[j]):
    group_destinations = model.NewBoolVar(f'group_destination{str(i)}-{str(j)}')
    model.Add(chosen_destinations[j] == 1).OnlyEnforceIf(group_destinations)
    model.Add(chosen_destinations[j] == 0).OnlyEnforceIf(group_destinations.Not())
    model.Add(output_flights[i] <= 0).OnlyEnforceIf(group_destinations)

# All incoming flights must have the same origin
if (flights[i].origin != destinations[j]):
    group_destinations = model.NewBoolVar(f'group_return{str(i)}-{str(j)}')
    model.Add(chosen_destinations[j] == 1).OnlyEnforceIf(group_destinations)
    model.Add(chosen_destinations[j] == 0).OnlyEnforceIf(group_destinations.Not())
    model.Add(output_flights[i] >= 0).OnlyEnforceIf(group_destinations)
```

Constraints

For each pair of flights:

```python

# True if flights have absolute equality i.e. either they are not used or are used by the same person
same_person_flights = model.NewBoolVar(f'same_person_fligths_{str(i)}-{str(j)}')
model.Add(output_flights[i] == -output_flights[j]).OnlyEnforceIf(same_person_flights)
model.Add(output_flights[i] != -output_flights[j]).OnlyEnforceIf(same_person_flights.Not())

```

```python

# If minimum time is not respected
if (flights[j].departure - flights[i].arrival).total_seconds() < minimum_time*60:

    model.Add(output_flights[i] == 0).OnlyEnforceIf(same_person_flights)
else:
    # Flight i must be positive and flight j negative, or both 0
    model.Add(output_flights[i] >= output_flights[j]).OnlyEnforceIf(same_person_flights)
```

Objective

In order to calculate the cost function:

```

model.Add(separated_time == sum(separated_times))
model.Add(useful_time == first_incoming - last_outgoing)
total_cost = sum(students_cost)
model.AddDivisionEquality(cost_function, total_cost*1000000000, useful_time)
model.Minimize(cost_function)
```

Conclusions

- The used method is too slow and not scalable. (Results will be shown later)
- OR-Tools solver is lacking some important constraints that would make the implementation easier.

```python

def count(model: cp_model.CpModel, vars, value):
    '''
    Returns the count of 'value' in 'vars'
    '''
    value = model.NewConstant(value)
    trues = [model.NewIntVar(0, 1, 'true' + str(i)) for i in range(len(vars))]
    for i in range(len(vars)):
        equal = model.NewBoolVar('equal')
        model.Add(vars[i] == value).OnlyEnforceIf(equal)
        model.Add(vars[i] != value).OnlyEnforceIf(equal.Not())

        model.Add(trues[i] == 1).OnlyEnforceIf(equal)
        model.Add(trues[i] == 0).OnlyEnforceIf(equal.Not())

    return sum(trues)

```

Conversion from DOcplex

Sample of the converted code:

```python

def element(model, vars, index):
    t = model.NewIntVar(-MAX_INT, MAX_INT, 'temp')
    model.AddElement(index, vars, t)
    return t

```

```python

student_from_outside_destination = model.NewBoolVar('student_from_outside_destination')
model.Add(StudentOrigin != Destination).OnlyEnforceIf(student_from_outside_destination)
model.Add(StudentOrigin == Destination).OnlyEnforceIf(student_from_outside_destination.Not())

# Origin of the flights
out = element(model, FLIGHTS_ORIGINS, Outgoing)
inc = element(model, FLIGHTS_ORIGINS, Incoming)
model.Add(out == StudentOrigin).OnlyEnforceIf(student_from_outside_destination)
model.Add(inc == Destination).OnlyEnforceIf(student_from_outside_destination)

```

SICStus

Program

In a very high-level style, the main predicate has the following before the enumeration:

main.pl

```prolog
main :-
    read_data(Data),
    create_variables(Data, Variables),
    constrain_variables(Data, Variables),
```

Data - Format

The first thing done was read the input in the following format, which included parsing:

data.pl

```
data(
    [
        Origins, Destinations, Departures, Arrivals,
        Durations, Prices, Connections
    ],
    PossibleDestinations,
    [
        Cities, AvailabilityStarts, AvailabilityEnds,
        MaximumConnections, EarliestDepartures, LatestArrivals
    ],
    MinimumUsefulTime, Locations
).
```

The function element/3 from clpfd only works with lists of integers, that's why the flights and students were split into multiple lists, where we can find the information of one row by using element/3 with the same index in the different lists.

Data - Dummy Trips

To get some constraints in a more consistent form, dummy trips were added, for each student's available interval:

- A trip that begins and ends at the start of the interval
- A trip that begins and ends at the end of the interval

These two trips have cost and durations of 0 and the current city of the student as origin and destination. These allow us to have no conditional restrictions, resulting in better propagation of constraints.

Decision Variables

The decision variables are the plans for the students. But some dependant variables were created, resulting in a structure with the following format:

variables.pl

```
variables(Destination, OutgoingPlans, IncomingPlans,
    IndividualCosts, TotalCost, UsefulTime, Goal).
```

Destination is the index of the location chosen as destination.

OutgoingPlans and IncomingPlans are lists of indices of trips.

The other variables are used to calculate Goal (the cost function).

Decision Variables

And the variables are created using the following predicate:

variables.pl

```prolog
create_variables(Data, [Destination, OutgoingPlans, IncomingPlans, IndividualCosts, TotalCost, UsefulTime, Goal]) :-
    % Final destination variable
    data_possible_destinations(Data, PossibleDestinations),
    Destination in_set PossibleDestinations,
    % List of pairs with indices of outgoing and incoming trip, 0 represents none
    create_plans(Data, OutgoingPlans-IncomingPlans),
    % Cost of each student
    create_individual_costs(Data, OutgoingPlans-IncomingPlans, IndividualCosts),
    % Total costs
    sum(IndividualCosts, #=, TotalCost),
    % Useful time
    data_flight_departures(Data, Departures),
    data_flight_arrivals(Data, Arrivals),
    create_useful_time(Departures-Arrivals, OutgoingPlans-IncomingPlans, _, UsefulTime),
    % Minimization goal
    Goal #= (86400 * TotalCost) / UsefulTime.
```

Restrictions

The list of restrictions used for each student are:

constraints.pl

```
        constrain_student_flight_locations(Origins-Destinations, Destination,
            OutgoingTrip-IncomingTrip, City),
        constrain_student_availability(Departures-Arrivals, OutgoingTrip-IncomingTrip,
            AvailabilityStarts, AvailabilityEnds, EarliestDeparture, LatestArrival),
        constrain_student_maximum_number_of_connections(Stops, OutgoingTrip-IncomingTrip,
            MaximumConnections),
        constrain_student_maximum_duration(Durations, OutgoingTrip-IncomingTrip,
            MaximumDuration),
```

For optimization purposes, the lists of students attributes are traversed at the same time.

Restrictions

As an example of a constraint function:

constraints.pl

```prolog
% The trips needs to start on the student city and end in it again, and they
% also need to lead the student to the destination.
constrain_student_flight_locations(Origins-Destinations, Destination,
    OutgoingTrip-IncomingTrip, City) :-
    element(OutgoingTrip, Origins, OutgoingOrigin),
    element(OutgoingTrip, Destinations, OutgoingDestination),
    element(IncomingTrip, Origins, IncomingOrigin),
    element(IncomingTrip, Destinations, IncomingDestination),
    OutgoingOrigin #= City,
    IncomingDestination #= City,
    OutgoingDestination #= Destination,
    IncomingOrigin #= Destination.
```

Enumeration

- Timeout of 10 seconds
- Minimize the cost

main.pl

```
    flatten_variables(Variables, LabelVariables),
    variables_cost(Variables, Cost),
    labeling([time_out(10000, Flag), minimize(Cost)], LabelVariables),
```

Conclusion

- The first implementation in SICStus was too complex and tried to use too much data structures, which ended up worsening the understanding of the program and ease of development. This resulted in a program that could not find a solution for our input file;
- We also tried a solution similar to the use used for the other implementations, but that resulted in too many conditionals and an even worse performance;
- The final solution tried to remove all conditionals by depending on dummy trips that make all the students have homogeneous plan formats and restrictions.

Results and Conclusion

DOcplex Results

Sample Dataset

Search completed, **9** solutions found.

Best objective:	0.001870622 **(Optimal)**
Number of branches:	25430
Number of fails:	13174
Total memory usage:	8.6 MB
Time spent in solve:	**0.31s**
Search speed (br. / s):	84766.7

Full Dataset

Search completed, **42** solutions found.

Best objective:	0.00005684341 **(Optimal)**
Number of branches:	52063
Number of fails:	26019
Total memory usage:	196.0 MB
Time spent in solve:	**41.56s**
Search speed (br. / s):	1269.2

OR-Tools Results

Sample Dataset

On the first resolution:

Number of fails: 9677
Number of branches: 10356
Number of propagations: 3,036,194
Time spent: **10s**

On the DOcplex conversion:

Number of fails: 6333
Number of branches: 6734
Number of propagations: 2,021,097
Time spent: **1.7s**

Full Dataset

On the first resolution:

Unable to finish

On the DOcplex conversion:

Number of fails: 191
Number of branches: 2251
Number of propagations: 17,913,097
Time spent: **140s**

SICStus Results

Sample Dataset	
Resumptions:	6983
Entailments:	1854
Prunings:	5077
Backtracks:	53
Constraints created:	84

========

0.088 sec. file parsing runtime
0.051 sec. program runtime

Full Dataset	
Resumptions:	245116
Entailments:	25887
Prunings:	180360
Backtracks:	1402
Constraints created:	84

========

4.172 sec. file parsing runtime
2.723 sec. program runtime

Optimal Solution

Destination: **Milano**
Total Cost: **64€**
Useful Time: 1125900 (13 days, 0:45:00)
Separated Time: 116400 (1 day, 8:20:00)

Student 1 - Budapest

Departure: 01/06/2022, 08:15:00
Arrival: 01/06/2022, 09:50:00
Duration: 1:35:00
Price: 13€ | Stops: 0

Departure: 14/06/2022, 21:00:00
Arrival: 14/06/2022, 22:30:00
Duration: 1:30:00
Price: 5€ | Stops: 0

Student 2 - Zagreb

Departure: 14/06/2022, 22:30:00
Arrival: 01/06/2022, 20:15:00
Duration: 1:15:00
Price: 10€ | Stops: 0

Departure: 15/06/2022, 17:20:00
Arrival: 15/06/2022, 18:35:00
Duration: 1:15:00
Price: 10€ | Stops: 0

Student 3 - Wien

Departure: 01/06/2022, 06:50:00
Arrival: 01/06/2022, 08:15:00
Duration: 1:25:00
Price: 16€ | Stops: 0

Departure: 15/06/2022, 09:05:00
Arrival: 15/06/2022, 10:30:00
Duration: 1:25:00
Price: 10€ | Stops: 0

Conclusion

- The SICStus solver outperformed both OR-Tools and DOcplex, but they use slightly different approaches;
- This solution could be used as a part of a final product, as it is fully working with real-life data in a reasonable time. The full dataset contained more than 9000 flights distributed across 7 different European cities during the whole month of June 2022.

Questions