Assignment report (Nicolò Rossi 129807)

1 Quick overview of the problem

The proprosed problem (that we will refer to as the "Trains" problem) consists in finding the best posible timetable for trains moving on a rail system.
The rail system is modelled as a multigraph, $G = (V, E)$, in which the nodes ($v \in V$) are the stations and the edges ($e \in E$) the railroads. The maximum number of railroads among two stations, so the maximum number of edges connecting two nodes, is two. Each train ($t \in T$) is indistiguishible and must be in one of theese three states at any time:

1. **Moving**: The train is moving from a station to another following a given route
2. **Stopping**: The train is stopping at a station to load/unload passengers
3. **Parking**: The train is parked in the deposit of a given station

Each node is labelled with number $c(v \in V) \in \mathbb{N}$ indicating the maximum amount of trains that can park in that station and each edge is labelled whith another number $c(e \in E) \in \mathbb{N}$ showing the number of time units necessary to complete the route. We say that T_{max} is the length of the timetable in time units and it should be made so it maximizes the number of connections among stations without having two trains at the same stop or route at the same time.
It can be also optionally required that the timetable satisfies certain constraints:

1. **Direct trains**: there must be a train connecting two stations without any stop
2. **Minimum connections**: there must be at least k connections among two stations
3. **Paths**: there must be a train connecting two stations and making a list of intermediate stops (in any order)

The starting configuration must have all the trains in a valid "parking" status and the number of trains parked in each station must be equal to that in the final configuration.

2 Assuptions and changes to the assignemnt

In the interpretation of this problem the following assumptions were made:

- Due to the high complexity of the problem (see NP-hardness section), it was almost never possible to generate solutions with $T_{max} = 144$ so it was decided to treat this parameter as an input rather than as a constant.
- It was considered necessary for every train leaving or entering a deposit of a station to stop at the current station.
- Two stations (a and b) are considered to be connected when a train stops at a and then at b without **parking** in any station in the meantime.
- Connections do not involve train changes
- To sataisfy a **path request** (or link request) is necessary that the train does not park when servicing the request.

- Instead of preparing ten graphs with ten test cases for each it was decided to prepare nine graphs with nine test cases for each and an additional collection of twenty test cases to show experimentally how it is possible to solve the Hamiltonian circuit problem using these models.

3 Model structure and implementation in Minizinc and ASP

3.1 Search space generation

3.2 Input

Input is prepared and formatted by external python scripts that will be described in the next section. The information used are:

- the duration of the plan in time units
- the identifiers of the stations and the associated capacity of their deposit
- the routes with their length, source, destination and id (used to understeand if two routes are actually the same rail but taken in the opposite direction)
- the direct requests
- the minimum connection requests
- the path request with a given identification number

3.3 Train number constraint

In minizinc we set an integer variable, usedTrains so that it must be is less or equal then the maximum number of trains that can be stored in any station (and greater than zero). Each train with an id greater than usedTrains is forced to be unused and can't be in any state.

```
var 1..maxTrains: usedTrains;
constraint (usedTrains>0 /\ usedTrains<=maxTrains);
...
% do not use more than usedTrains
constraint forall(time in TIME, train in TRAINS)
      ((train > usedTrains)->(Parking[time,train] = 0 /\
        Stopping[time,train] = 0 /\
        Moving[time,train] = 0 /\
        Distance[time, train] = 0));
```

In ASP the concept is similar but unused trains are simply ignored.

```
# look for the best number of trains
CI.addProgramLine("1{ max_trains(T) : usable_trains(T) }1.")
CI.addProgramLine("train(TRAIN) :- usable_trains(TRAIN), TRAIN<=MAX, max_trains(MAX).")
```

3.4 Unique train state constraint

In minizinc we encode that any train can be in a parking, moving or stopping state using the Parking, Moving and Stopping arrays. At any time for any train only one of them can have a value grater than one indicating where the train is in the rail system.

```
% constraint the trains to be in a single place at a time
constraint forall(time in TIME, train in TRAINS)
      ((train <= usedTrains)->
      (Stopping[time,train]>0 <-> Parking[time,train]=0 /\ Moving[time,train]=0));
constraint forall(time in TIME, train in TRAINS)
      ((train < usedTrains)->(Parking[time,train] >0 <->
       Stopping[time,train]=0 /\ Moving[time,train]=0));
constraint forall(time in TIME, train in TRAINS)
      ((train <= usedTrains)->(Moving[time,train] >0
      <-> Parking[time,train]=0 /\ Stopping[time,train]=0));
constraint forall(time in TIME, train in TRAINS)
      ((train <= usedTrains)->(Stopping[time,train] > 0 ->
      Distance[time,train] = 0));
constraint forall(time in TIME, train in TRAINS)
      ((train < usedTrains)->(Parking[time,train] > 0 ->
       Distance[time,train]  = 0));
constraint forall(time in TIME, train in TRAINS)
      ((train <= usedTrains)->(Distance[time,train] = 0 <->
       Moving[time,train]  = 0));
```

In ASP this is encoded directly in the search space generation.

3.5 Valid train actions definition

In MiniZinc, for each used train, we set that:

- if it is parking in a station it can either remain parked or be loaded on the rail
- if it is stopping at a station it can take an adjacent route or park in that station
- if it was moving on a rail it can either continue moving on it or, only if it is arrivng at a station, stop at it or take another route leaving that station.

```
% movment constraints

constraint forall(time in 1..Final-1, train in TRAINS)
    ((train <= usedTrains)->(
       Parking[time,train] > 0 ->
      ((Parking[time+1,train] = Parking[time,train]) \/
       (Stopping[time+1,train] = Parking[time,train]))
    ));

constraint forall(time in 1..Final-1, train in TRAINS)
```

```
    ((train <= usedTrains)->(
       Stopping[time,train] > 0 ->
        ((Parking[time+1,train] = Stopping[time,train]) \/
          exists(r in ROUTES where Connected[Stopping[time,train], r]
          /\ time+Length[r]<Final)
          (Moving[time+1,train] = r /\ Distance[time+1,train] = 1))
    ));

constraint forall(time in 1..Final-1, train in TRAINS)
    ((train <= usedTrains)->(
       Moving[time,train] > 0 ->
        if Distance[time,train] < Length[Moving[time,train]] then
              Moving[time+1,train]=Moving[time,train] /\
              Distance[time+1,train] = Distance[time,train]+1
        else
            (exists(r in ROUTES where Connected[To[Moving[time,train]], r] /\
                    time+Length[r]<Final)
            (Moving[time+1,train] = r /\ Distance[time+1,train] = 1))
            \/ (Stopping[time+1,train] = To[Moving[time,train]])
        endif
    ));
```

In ASP the model is similar; it was added a rule that checks if it is possible to take a given route and end it in time to reduce the search space.

```
# prepares the following steps
# from the deposit
CI.addProgramLine("1{ stopping(TIME+1,TRAIN,STATION);
                  parking(TIME+1,TRAIN,STATION) }1 :-
                  TIME<FIN, final(FIN), parking(TIME, TRAIN, STATION),
                  time(TIME), train(TRAIN), park(STATION).")
# from a stop (if a station has a deposit or not)
CI.addProgramLine("1{ parking(TIME+1,TRAIN,STATION);
                  moving(TIME+1, TRAIN, ROUTE, 1) :
                  local_routes(STATION, ROUTE), TIME+LEN<FIN,
                  len(ROUTE, LEN)   }1 :- TIME<FIN, final(FIN),
                  stopping(TIME, TRAIN, STATION), time(TIME),
                  train(TRAIN), park(STATION).")
# note: the train does not move if it is impossible to end the route in time
CI.addProgramLine("1{ moving(TIME+1, TRAIN, ROUTE, 1) :
         local_routes(STATION, ROUTE), TIME+LEN<FIN, len(ROUTE, LEN) }1 :-
         TIME<FIN, final(FIN), stopping(TIME, TRAIN, STATION),
         time(TIME), train(TRAIN), stop_only(STATION).")
# moving
# while moving
CI.addProgramLine("moving(TIME+1, TRAIN, ROUTE, POSITION+1) :- ")
```

```
CI.addProgramLine("TIME<FIN, final(FIN), moving(TIME, TRAIN, ROUTE, POSITION),
                   time(POSITION), time(TIME), train(TRAIN),
                  route(ROUTE), POSITION<LENGTH, len(ROUTE, LENGTH).")
# when arriving
CI.addProgramLine("1{ stopping(TIME+1, TRAIN, NEXT_STATION) :
                   to(ROUTE, NEXT_STATION); ")
CI.addProgramLine("   moving(TIME+1, TRAIN, NEXT_ROUTE, 1) :
          to(ROUTE, NEXT_STATION), local_routes(NEXT_STATION, NEXT_ROUTE),
          TIME+LEN<FIN, len(NEXT_ROUTE, LEN)  }1:- ")
CI.addProgramLine("TIME<FIN, final(FIN), moving(TIME, TRAIN, ROUTE, POSITION),
     time(POSITION), time(TIME), train(TRAIN), route(ROUTE),
     POSITION=LENGTH, len(ROUTE, LENGTH).")
```

3.6 Initial and final configurations preparation

In MiniZinc we simply state that any used tain must be parked in a station with a deposit.

```
% starting configuration
constraint forall(train in TRAINS)
           ((train<=usedTrains)->
           (exists(station in STATIONS)(Parking[1, train]=station /\
            Capacities[station]>0)));
```

In ASP we similarly force each train to be parked at time 1. To remove symmetries made by train identifiers we force them to follow the order of station identifiers.

```
CI.addProgramLine("1{ parking(1, TRAIN, STATION): park(STATION)  }1 :- train(TRAIN).")
# symmetry breaking rule, sort identifiers
CI.addProgramLine(":- parking(1, TR1, S1), parking(1, TR2, S2),
                      train(TR1), train(TR2), park(S1), park(S2),
                      TR1>TR2, S1<S2.")
```

To constraint the final state so that the number of parked train in each station is the same as in the initial state in MiniZinc as in ASP we count the number of parked trains for each station at both times ad require them to be equal.

```
% stating conf = stop conf
constraint forall(station in STATIONS)
           (sum(train in TRAINS)
           (Parking[1, train]==station) == sum(train in TRAINS)
               (Parking[Final, train]==station));
```

In ASP it is also required that each train is in a station (as a cut).

```
# --- FINAL STATE
# 1 - all the trains are in a station (additional cut)
CI.addProgramLine(":- moving(FIN, TRAIN, ROUTE, DUR), final(FIN), time(DUR),
                      train(TRAIN), route(ROUTE).")
```

```
CI.addProgramLine(":- stopping(FIN, TRAIN, STATION), final(FIN),
                      train(TRAIN), station(STATION).")
# 2 - the number of train in each deposit is equal at the initial and final steps
CI.addProgramLine(":- CINI!=CFIN, park(STAT), CINI = #count{ TRAIN :
                      parking(1,TRAIN,STATION), train(TRAIN),
                      station(STATION), STATION=STAT },")
CI.addProgramLine("  CFIN = #count{ TRAIN : parking(FIN,TRAIN,STATION),
                     train(TRAIN), station(STATION),
                     final(FIN), STATION=STAT }.")
```

3.7 Five minutes constraint

The "five minutes" constraint requires that a station must be free for a time unit when a train stops at it. This is implemented directly in MiniZinc and ASP in a natural way.

```
% cannot have another train stopping at a
% station or another train passing by it within
% the next time unit a train stopped at this station
% within the next time unit
constraint forall(time in TIME, t1 in TRAINS)
     (Stopping[time,t1] > 0 -> not exists(t2 in TRAINS)
     ( t1!=t2 /\ Moving[time,t2] > 0 /\
       Distance[time,t2] == 1 /\
       From[Moving[time,t2]] == Stopping[time,t1]));
constraint forall(time in TIME, t1 in TRAINS)
                 (Stopping[time,t1] > 0 -> not exists(t2 in TRAINS)
                 ( t1!=t2 /\ Stopping[time,t1] == Stopping[time+1,t2]));
```

The same in the ASP encoding:

```
# - eliminate cases in which a train enters a station before the temporal unit gap
CI.addProgramLine(":- moving(TIME, TRAIN1, ROUTE1, POS1),
     moving(TIME+1,TRAIN1,ROUTE2,POS1),
     stopping(TIME, TRAIN2, DES), to(ROUTE1, DES), from(ROUTE2, DES), ")
CI.addProgramLine("   train(TRAIN1), train(TRAIN2),
       route(ROUTE1), route(ROUTE2), time(POS1), time(POS2),
       route(DES), time(TIME).")
# - eliminate cases in which a train stops at
# a station before the temporal unit gap
CI.addProgramLine(":- stopping(TIME, TRAIN1, STATION),
           stopping(TIME+1, TRAIN2, STATION), station(STATION),
           time(TIME), train(TRAIN1), train(TRAIN2).")
```

3.8 Collision detection

In MiniZinc collison ar detected by chcking if the Stopping, Parking and Moving array have different values at any time. In this case the global constraint `alldifferent_except_0` is quite useful.

```
% cannot have two trains leaving a station at the same time
```

```
constraint forall(time in TIME, t1 in TRAINS)
                 (Moving[time,t1] > 0 -> not exists(t2 in TRAINS)
                 ( t1!=t2 /\
                   From[Moving[time,t1]] == From[Moving[time,t2]] /\
                   Distance[time,t1] == Distance[time,t2] /\
                   Distance[time,t1] == 1 ));

% cannot have two trains at the same station at the same time
constraint forall(time in TIME)
   (alldifferent_except_0(
         ([ Stopping[time,train] | train in TRAINS])
    ));

% cannot have two trains on the same route at the same time
        (alldifferent_except_0(
        ([ if Moving[time,train]>0
           then Id[Moving[time,train]]
           else 0 endif | train in TRAINS])));
```

In ASP this is done by checking that two trains are never in the same state.

```
# eliminate cases in which a train is leaving
# the station and another one is arriving
CI.addProgramLine(":- stopping(TIME, TRAIN, STATION),
        moving(TIME, ROUTE, STATION, 1), from(ROUTE,STATION),
        route(ROUTE), station(STATION), train(TRAIN).")
# eliminate the case in which two trains are leaving the same station
 CI.addProgramLine(":- moving(TIME, TRAIN1, ROUTE1, 1),
          moving(TIME, TRAIN2, ROUTE2, 1),
          from(ROUTE1,STATION), from(ROUTE2,STATION),
          route(ROUTE1), route(ROUTE2),
          station(STATION), train(TRAIN1),
          train(TRAIN2), TRAIN1!=TRAIN2.")

# - eliminate cases in which two trains are
# on the same route at the same time

CI.addProgramLine(":- moving(TIME, TRAIN1, ROUTE1, POS1),
           moving(TIME, TRAIN2, ROUTE2, POS2), time(POS1),
           time(POS2), route(ROUTE1), route(ROUTE2),")
CI.addProgramLine("id(ROUTE1,ID1), id(ROUTE2,ID2),
                   ID1=ID2, train(TRAIN1), train(TRAIN2),
                   TRAIN1!=TRAIN2.")

# - eliminate cases in which two trains are stopping at the same station

CI.addProgramLine(":- stopping(TIME, TRAIN1, STATION),
```

```
        stopping(TIME, TRAIN2, STATION), station(STATION),
        train(TRAIN1), train(TRAIN2),
        time(TIME), TRAIN1 != TRAIN2.")
```

3.9 Manage direct trains

Direct trains are required to stop at two given station without making any itermediate stop. In MiniZinc this is expressed in the natural way:

```
% DIRECTS
constraint forall(a in STATIONS, b in STATIONS)
    (Directs[a,b] ->
             exists(train in TRAINS, t1,t2 in TIME)
                   ( t1<=t2 /\
                     Stopping[t1,train]=a /\
                     Stopping[t2,train]=b /\
                     not exists(t3 in TIME)
                       ( t3<t2 /\
                         t3>t1 /\
                         Stopping[t3,train]>0) ));
```

In ASP it is required the the number of intermediate stops is zero:

```
# --- MANAGE DIRECTS
CI.addProgramLine("ok_direct(A,B) :-  station(A),
        station(B), direct(A,B), train(TRAIN),
        time(T1), time(T2), T1<T2, stopping(T1, TRAIN, A),
        stopping(T2, TRAIN, B),")
CI.addProgramLine("CNT=#count{ C : stopping(T3, T, C),
        T3>T1, T3<T2, time(T3), station(C),
        train(T), T=TRAIN }, CNT=0.")
CI.addProgramLine(":- direct(A,B), not ok_direct(A,B).")
```

3.10 Manage minimum connection constraint

Two stations are connected by a train when it stops at both without parking or stopping again at one of them in the middle. In MiniZinc this is implemented by counting the number of times every train connects two stations:

```
% MINIMUM
constraint forall(a in STATIONS, b in STATIONS)
      ( (Min[a,b] > 0) ->
         Min[a,b]<=sum(train in TRAINS, t1,t2 in TIME)
           (bool2int(t1<=t2 /\ Stopping[t1,train]=a /\ Stopping[t2,train]=b /\
           ( not exists(t3 in TIME)(t3<t2 /\ t3>t1 /\ (Parking[t3,train]>0 \/
             Stopping[t3,train]=a \/ Stopping[t3,train]=b))))));
```

In ASP it is necessary to compute the time intervals in which a train is moving to connect the two stations and look if they are acceptable:

```
# --- MANAGE MINIMUM NUMBER OF CONNECTIONS
# compute return intervals for each train (if required)
CI.addProgramLine("interval(TRAIN, A, START, STOP) :- station(A), station(B),
       min_conn(A,B,K), stopping(START, TRAIN, A), train(TRAIN),
       time(START), time(STOP), ")
CI.addProgramLine("STOP=#min{ TIME : time(TIME), TIME>START,
                    stopping(TIME, TRAIN, A) ; FIN : final(FIN) }.")
CI.addProgramLine(":-  min_conn(A,B,K), station(A),
      station(B), K>C, C=#count{ START, TRAIN :
      interval(TRAIN, A, START, STOP ),   ")
CI.addProgramLine(" train(TRAIN), time(START),
        time(STOP), stopping(STOPTIME, TRAIN, B),
        time(STOPTIME), STOPTIME>START, STOPTIME<STOP}.")
```

3.11 Manage path constraints

Path constraints require that there exists a train connecting two stations and passing by a list of other stations. This is done in MiniZinc in the natural way (additional cuts were added to speed up the search):

```
% PATHS

constraint forall(a in STATIONS, b in STATIONS)
   ( (PathRQ[a,b] > 0) ->
      exists(train in TRAINS, t1,t2 in TIME)
        (Stopping[t1,train]=a /\
         Stopping[t2,train]=b /\
         t1<t2 /\
         (forall(inter in STATIONS)
            ((Services[PathRQ[a,b],inter]=1) -> exists(t3 in TIME)
            (Stopping[t3,train]=inter /\ t3>t1 /\ t3<t2)))));

% additional cuts
constraint forall(a in STATIONS, b in STATIONS)
    ( (PathRQ[a,b] > 0) ->
       exists(train in TRAINS, t1,t2 in TIME)
       (Stopping[t1,train]=a /\
        Stopping[t2,train]=b /\
        t1<t2));

constraint forall(idt in 1..idLimit, inter in STATIONS)
      ( (Services[idt,inter]=1) ->
        exists(train in TRAINS, t in TIME)
        (Stopping[t,train]=inter));
```

In ASP this contraint is implemented in a more complex way. This method involves the computation of the time intervals at which any train is connecting the first and last stations and checks if one of them is acceptable. The ASP code necessary for the intermediate stops check is prepared using python so that if one of these stops is missing the solution is discarded.

```
# --- MANAGE INTERMEDIATE STOPS
# compute return intervals for each train (if required)
CI.addProgramLine("interval(TRAIN, A, START, STOP) :-
                station(A), station(B), pathReq(A,B,ID),
                stopping(START, TRAIN, A),
                train(TRAIN), time(START), time(STOP), ")
CI.addProgramLine("STOP=#min{ TIME : time(TIME),
                   TIME>START, stopping(TIME, TRAIN, A)
                   ; FIN : final(FIN) }.")

# compute the pass by intervals for two stations (if necessary)
CI.addProgramLine("in_path( TRAIN, A, B, START, STOP) :-
                   train(TRAIN), pathReq(A,B,ID), station(A),
                   station(B), interval(TRAIN, A, START, MAXRANGE), ")
CI.addProgramLine("STOP=#min{ STP : stopping(STP, TRAIN, B),
                   STP>START, STP<=MAXRANGE }.")

# prepare path constraints
for constraint in Reader.pathRequests:
# parte generale
    CI.addProgramLine("ok(" + constraint["ID"] + ")
        :- pathReq(" + constraint["SOURCE"] + "," +
        constraint["DESTINATION"] +"," + constraint["ID"] + "), ")
    CI.addProgramLine("station(" + constraint["SOURCE"] +
        "), station(" + constraint["DESTINATION"] + "),")
    CI.addProgramLine("in_path(TRAIN," +
        constraint["SOURCE"] + "," + constraint["DESTINATION"]
        + ",START,STOP), train(TRAIN), time(START), time(STOP)")
    # manage intermediate steps
    ID = 1
    for stop in constraint["STOPS"]:
        CI.addProgramLine(", stopping(TIME"+ str(ID)
        + ", TRAIN, " + str(stop) + "),
        time(TIME"+ str(ID) + "), TIME"+ str(ID)
        + "<STOP, TIME"+ str(ID) + ">START")
        ID+=1
        CI.addProgramLine(".")
        # remove cases without the required path
        CI.addProgramLine(":- not ok(" + constraint["ID"] + "),
           pathReq(" + constraint["SOURCE"] + "," +
           constraint["DESTINATION"] + "," + constraint["ID"]
           + "), station(" + constraint["SOURCE"]
           + "), station(" + constraint["DESTINATION"] + ").")
```

3.12 Objective function computation

The objective function maximizes the number of connected stations. In this case it is admissible that a train parked during the connection trip. In both MiniZinc and ASP it is fistly computed

the number of connected stations and then required its maximization, so that it is easy to read the answer from the output of the solvers.

```
array [STATIONS, STATIONS] of var bool: Linked;
constraint forall(s in STATIONS)(not Linked[s,s]);
constraint forall(s1,s2 in STATIONS)
  (
    (s1!=s2) -> ( (not exists(train in TRAINS, t1,t2 in TIME)
                  (t1<t2 /\
                   Stopping[t1,train]>0 /\
                   Stopping[t1,train]==s1 /\
                   Stopping[t2,train]==s2)) -> not Linked[s1,s2])
  );


var int: ans;
constraint ans = sum(Linked);
solve maximize ans;

# --- OBJECTIVE FUNCTION
CI.addProgramLine("linked(A,B) :- A!=B, station(A),
    station(B), train(TRAIN), time(T1), time(T2), T1<T2,
    stopping(T1, TRAIN, A), stopping(T2, TRAIN, B).")
CI.addProgramLine("answer(S) :- S=#count{ A,B : linked(A,B),
                   station(A), station(B), A!=B }.")
CI.addProgramLine("#maximize{ S : answer(S) }.")
```

4 Brief description of the pre and post processing utilities

The Pre and Post processing activities for the input management and model creation where made by using python scripts to interface this programming language to the Clingo and MiniZinc solvers. They require that you have `clingo` and `minizinc` in your path and they let you create, import or modify models written ASP and MiniZinc, run the solvers on them and evaluate the given answers directly in a single program. The `InputReader.py` utility checks if the input format is correct and formats it for the use in `clingo` and `minizinc`.

5 Proof of NP-Hardness

5.1 The "train" problem is NP-hard

In this section we aim to show that the train problem is NP-hard and we do that by reducing the NP-complete **Hamiltonian circuit** problem to the train problem.

5.1.1 Instance translation:

Consider a generic Hamiltonian circuit instance. Let $G(V, E)$ be the undirected graph associated to this instance and let $v \in E$ be the starting node. We create an instance of the train problem as follows:

1. We set $G(V, E)$ to be the graph describing all the possible routes of the trains, so V is the set of stations and E the set of the railroads.

2. We set $v \in V$ to be the only station with a train deposit, with a maximum capacity of 1, hence, the maximum number of trains is 1.

3. The weight of each edge is set to 1

4. We set $T = 2|V| + 3$ as the total number of time slots, so that is possible to move and stop at each station iff at any time it is possible to reach a never visited station (or to return to v) by traversing a single edge.

5. Even if not strictly necessary, we add the request of having a train from v to v and passing by $V \setminus \{v\}$

5.1.2 Solution:

By point number 5, we have that the minimum amount of time necessary to serve such a request is the sum of:

1. 1 time slot for putting the train onto the rails at the beginning of the operations

2. $|V| + 1$ time slots for stopping in each station and two times in v

3. $|V|$ time slots for reaching the next stop

4. 1 time slot for putting the train in the parking of v

Having set, by point 4, the maximum number of time slots exactly to $2|V| + 3$ this means that the solver might yield a solution iff condition 5 is satisfied and there is not a the train does pass exactly one time by any station (as the amount of time slots is not sufficient to permit repetitions of them). This means that the routes taken by the train exactly describe an Hamiltonian circuit. We can finally conclude that this problem admits a solution iff there is an Hamiltonian circuit in the graph $G = (V, E)$ and this proves the NP-hardness of this problem.

5.2 Considerations on admissibility verification

Consider now a generic instance of a train problem:

1. the number of trains that are not still in a park is less than $|V| * T$ as any train must use a time slot to be prepared on a stop.

2. the maximum number of times a train can reach a station is again bounded by $|V| * T$ for the same reasons as before

3. the requests are at in a linear relation with the total length of the program. Let R be the set of the requests

points 1 and 2 combined states that the number occurring in the definition of the program are bounded by values polynomially related to the input.
If we consider a generic solution to a generic instance of the train problem we can evaluate if it is admissible in polynomial time (let O be the output, by the previous consideration it has polynomial size and is structured as a timetable stating at each time where each train is):

1. we can check in polynomial time (in respect to $|O|$) whether two trains are in the same location (route or stop) at the same time or if there are too many trains stopping at a train station.
2. we can check a direct request (eg. from A to B) easily looking for a train that stops at A at a certain time and then at B without any stop or park action in the middle, the way of reasoning is similar for the request with intermediate stops.
3. to test minimum number of connections request (eg. from A to B) we simply scan the timetables for each train and count the total number of times a train stops at A and then at B.

Any other request is easily solved in polynomial time with similar strategies. We can then state that it is possible to evaluate admissibility in polynomial time if we assume that the number of time unit is polinomially relataed with the size of the program, otherwise the solution would be of course exponential in size.

6 Testing configuration and results

Many different test cases were perpared to test the performance of the MiniZinc and ASP models. We considered nine different rail system in nine possible configuration each and another twenty graphs to show how it is possible to reduce the Hamiltomian Circuit problem to the "Trains" one:

- The first graph is a five nodes linear graph with one-binary rails
- The second graph is a five nodes linear graph with two-binary rails
- The third graph is a small tree of five nodes
- The fourth graph is the four nodes complete graph
- The fifth graph is composed of two disjointed linear graphs of three nodes
- The sixth graph is a tree of six nodes with a single branch
- the seventh graph is made of five nodes connected in a star-like topology
- the eighth graph is a cycle of six nodes with an high weight edge connecting node 1 to 4
- the ninth graph is that given as example in the assignment
- folder `noHam` contains ten examples of graphs that have not Hamiltonian Circuits
- folder `yesHam` contains ten examples of graphs that have Hamiltonian Circuits

It is possible to use the script `getInfo.sh <testcase.in>` to print its details in a more readable format and have a .dot representation of the rail system.
The testing machine is configured with 16GB of RAM, a quad core (eight threads) Intel i7-2630QM processor and Ubuntu 16.04. Clingo version is 5.3.1 and MiniZinc is version 2.2.3. The timeout was set after five minutes, see `run.sh` and `runAll.sh` scripts for more information.
The results shown how the ASP encoding is far superior than MiniZinc when elaborating longer plans with many trains; on the other end, in the most complex cases of the Hamiltonian circuit tests, the MiniZinc encoding is capable of finding the solution while ASP tends to timeout or to run out of memory. This might be due to the implementation of the path constraint that is far more complex in ASP.

7 Considerations on solver settings

We used the test cases `es9_t20.in` and `es8_t40_dDA_dAD` for MiniZinc and ASP respectively, the solution in the first example was found in almost ten seconds, while in the second there were necessary two minutes and ten seconds.

7.1 MiniZinc

Changing the search options for the "Moving" variables only has a huge negative impact in the execution time, if compared with the default method. The folowing table shows the results of the MiniZinc execution for all the possible combinations of variable and constraint choiches:

```
max_regret,indomain_median:          TIMEOUT!
occurrence,indomain_median:          TIMEOUT!
occurrence,indomain:                 TIMEOUT!
smallest,indomain_min:               54154.359
max_regret,indomain:                 TIMEOUT!
input_order,indomain:                TIMEOUT!
anti_first_fail,indomain_reverse_split:      TIMEOUT!
smallest,indomain_reverse_split:     TIMEOUT!
input_order,indomain_reverse_split: TIMEOUT!
first_fail,indomain_median:          TIMEOUT!
max_regret,indomain_split:         TIMEOUT!
occurrence,indomain_min:             TIMEOUT!
dom_w_deg,indomain_split:            TIMEOUT!
input_order,indomain_min:            258131.556
largest,indomain_max:                TIMEOUT!
largest,indomain:                    TIMEOUT!
max_regret,indomain_reverse_split:   TIMEOUT!
largest,indomain_min:                TIMEOUT!
anti_first_fail,indomain_median:     TIMEOUT!
dom_w_deg,indomain_median:           TIMEOUT!
input_order,indomain_split:          TIMEOUT!
max_regret,indomain_random:          TIMEOUT!
largest,indomain_reverse_split:      TIMEOUT!
most_constrained,indomain:           TIMEOUT!
most_constrained,indomain_random:    TIMEOUT!
most_constrained,indomain_split:     TIMEOUT!
dom_w_deg,indomain_reverse_split:    TIMEOUT!
input_order,indomain_max:            TIMEOUT!
first_fail,indomain_random:          TIMEOUT!
anti_first_fail,indomain_random:     TIMEOUT!
most_constrained,indomain_min:       TIMEOUT!
most_constrained,indomain_reverse_split:   TIMEOUT!
dom_w_deg,indomain:                  TIMEOUT!
occurrence,indomain_reverse_split: TIMEOUT!
largest,indomain_median:             TIMEOUT!
dom_w_deg,indomain_min:              TIMEOUT!
```

```
anti_first_fail,indomain_min:        57600.29
largest,indomain_split:              TIMEOUT!
anti_first_fail,indomain_max:        TIMEOUT!
smallest,indomain:                   TIMEOUT!
most_constrained,indomain_median:    TIMEOUT!
first_fail,indomain_max:             TIMEOUT!
dom_w_deg,indomain_random:           TIMEOUT!
anti_first_fail,indomain:            TIMEOUT!
smallest,indomain_random:            TIMEOUT!
smallest,indomain_split:             TIMEOUT!
most_constrained,indomain_max:       TIMEOUT!
first_fail,indomain_reverse_split:   TIMEOUT!
largest,indomain_random:             TIMEOUT!
input_order,indomain_random:         TIMEOUT!
smallest,indomain_median:            TIMEOUT!
anti_first_fail,indomain_split:      TIMEOUT!
max_regret,indomain_min:             TIMEOUT!
dom_w_deg,indomain_max:              TIMEOUT!
smallest,indomain_max:               TIMEOUT!
occurrence,indomain_max:             TIMEOUT!
first_fail,indomain_split:           TIMEOUT!
occurrence,indomain_split:           TIMEOUT!
max_regret,indomain_max:             TIMEOUT!
first_fail,indomain_min:             TIMEOUT!
occurrence,indomain_random:          TIMEOUT!
input_order,indomain_median:         TIMEOUT!
first_fail,indomain:                 TIMEOUT!
```

We get the best result with the combination `smallest,indomain_min`, however, this result is far form that of the default settings. For ASP the situation is different and it was seen that the best search mode was `handy` (large problems optimization):

```
many: 167846
auto: 172416
frumpy: 158683
jumpy: 149859
tweety: 145054
handy: 125162
crafty: 137595
trendy: 190945
many: 167846
```

8 Conclusions and possible imporovements

Even if these models are not solved efficiently enough to solve the proposed problem in it original formulation (if not for very easy cases) it is often possible to cope with plans with reasonable length, in particular with the ASP implementation. There are a few possible improvement for the models:

- Add the symmery breaking rule for identifiers also in the MiniZinc model
- Add additional cuts in the MiniZinc model as in ASP
- Modify the implementation of path constraint in ASP to make it more efficient

It could also be useful to improve the python interfaces to Clingo and MiniZinc to make them more flexible and complete.