Constraint Programming M – Assignment 2: Solver comparison option
Student GUID: **2381169g**

COIN-BC ILP solver - Minizinc VS. PuLP (python).
Before deciding on what approaches to use, I first spent some time studying the ILP encoding of the firefighter problem from lecture 4. Once I figured that I understood the problem reasonably well, I decided to look for ILP solvers and frameworks online. I ended up picking PuLP package since it seemed to have a very friendly syntax for defining ILP's and because I also feel very comfortable with Python (2x convenience points). The package is tailored specifically for solving ILP's, using the COIN-BC solver by default. Given that PuLP is considered to be relatively slow, I became curious – how slow can it really be? My idea became, to test how well an ILP problem defined and compiled in Minizinc (a general CSP solver), can perform VS python and PuLP (which is tailored specifically for ILP's).
As an attempt to keep the test as fair as possible, I tried my best to encode the problems as similar as possible to each other, within the different languages. I made sure to use the same domains, same data types, the same logic in my function/constraint definitions and even the same solvers. Note that the variables, domains, constraints and the objective function, were all taken from the mathematical definitions from lecture 4, (section 'An ILP for Firefighting') and translated to both Minizinc and PuLP. Below is a thorough description of the translation process.

Transferring the notation (constants and variables) *[note that all variables used in the source code will be in italics]*
1. Graph definition $G = (V, E)$ is the graph : required variables: int: *n_nodes*, int: *n_edges*, array[int]: *from*, array[int] *to*. Respectively, variables *NODES* and *EDGES* are initialized as arrays, [1..*n_nodes*] and [1..*n_edges*]. The mapping between the nodes, via the edges, is represented by arrays *from* and *to* where the index is the **edge**, and the from/to **nodes** can be obtained from the value at the edge index (*from*[i] -> *to*[i]).
2. T is some maximum time is *MAX_TIME*, r_i are the places the fire starts is *START_FIRES* and d is our budget of defenders is *budet_defenders*.
3. $N(x)$ is the neighbourhood of vertex x : this was an interesting challenge to implement in the ILP version of Minizinc. The way I achieved it can be seen in **table 1**, in constraints 3 and 5. As we go over every node, my idea was to check IF a node is in the FROM list -> it means that it will have a neighbour at the TO list, at that same index. The same logic follows in the opposite direction - if a node is in the TO list, it means its neighbour is at the FROM list at the same index. For each node, going over every possible edge and checking for neighbours using both the FROM and TO lists, guarantees that every single neighbour pair is covered, and thus can be included in the constraints. Edges that are not neighbours, we simply ignore (return True in forall constraint or add 0 in sum function).

4. $b_{x,t}$ gets 1 if vertex x is burned at or before t, 0 otherwise
$d_{x,t}$ gets 1 if vertex x is burned at or before t, 0 otherwise : is represented by 2d arrays, *burn_n_t* and *defend_n_t*, with domains [0,1]

Constraint translation (Table 1.)

Constraint	Domain/Range	Minizinc	PuLP

$b_{x,t} \geq b_{x,t-1}$	$x \in V, 1 \leq t \leq T$	`constraint forall(t in time_range, n in NODES)`
		`  (burn_n_t[n,t] >= burn_n_t[n,t-1]);`
			`for t in time_range:`
			`    for n in NODES:`
			`        lp_firefighter += burn_n_t[n][t] >= burn_n_t[n][t-1]`

$d_{x,t} \geq d_{x,t-1}$	$x \in V, 1 \leq t \leq T$	`constraint forall(t in time_range, n in NODES)`
		`  (defend_n_t[n,t] >= defend_n_t[n,t-1]);`
			`for t in time_range:`
			`    for n in NODES:`
			`        lp_firefighter += defend_n_t[n][t] >= defend_n_t[n][t-1]`

$b_{x,t} + d_{x,t} \geq b_{y,t-1}$	$x \in V, y \in N(x),$ $1 \leq t \leq T$	`constraint forall(t in time_range, n in NODES) (`
		`  forall(e in EDGES) (`
		`    if n = FROM[e] then`
		`      bool2int(burn_n_t[n,t]) + bool2int(defend_n_t[n,t]) >= bool2int(burn_n_t[TO[e],t-1])`
		`    elseif n = TO[e] then`
		`      bool2int(burn_n_t[n,t]) + bool2int(defend_n_t[n,t]) >= bool2int(burn_n_t[FROM[e],t-1])`
		`    else true endif`
		`  )`
		`);`
			`for t in time_range:`
			`    for n in NODES:`
			`        for e in EDGES:`
			`            if n == FROM[e]:`
			`                lp_firefighter += burn_n_t[n][t] + defend_n_t[n][t] >= burn_n_t[TO[e]][t-1]`
			`            elif n == TO[e]:`
			`                lp_firefighter += burn_n_t[n][t] + defend_n_t[n][t] >= burn_n_t[FROM[e]][t-1]`
			`            else: lp_firefighter += True`

$b_{x,t} + d_{x,t} \leq 1$	$x \in V, 1 \leq t \leq T$	`constraint forall(t in time_range, n in NODES)`
		`  (burn_n_t[n,t] + defend_n_t[n,t] <= 1);`
			`for t in time_range:`
			`    for n in NODES:`
			`        lp_firefighter += defend_n_t[n][t] + burn_n_t[n][t] <= 1`

$\sum_{y \in N(x)} b_{y,t-1} \geq b_{x,t}$	$x \in V, 1 \leq t \leq T$	`constraint forall(t in time_range, n in NODES) (`
		`  burn_n_t[n,t-1] + sum(e in EDGES) (`
		`    if n = FROM[e] then`
		`      burn_n_t[TO[e],t-1]`
		`    elseif n = TO[e] then`
		`      burn_n_t[FROM[e],t-1]`
		`    else 0 endif`
		`  ) >= burn_n_t[n,t]`
		`);`
			`for t in time_range:`
			`    for n in NODES:`
			`        total_sum = burn_n_t[n][t-1] #initialize sum`
			`        for e in EDGES: #account for neighbours`
			`            if n == FROM[e]:`
			`                total_sum += burn_n_t[TO[e]][t-1]`
			`            elif n == TO[e]:`
			`                total_sum += burn_n_t[FROM[e]][t-1]`
			`            else: total_sum += 0`
			`        lp_firefighter += total_sum >= burn_n_t[n][t]`

$\sum_{x \in V} (d_{x,t} - d_{x,t-1}) \leq d$	$1 \leq t \leq T$	`constraint forall(t in time_range)`
		`  (sum(n in NODES)(defend_n_t[n,t] - defend_n_t[n,t-1]) <= budget_defenders);`
			`for t in time_range:`
			`    lp_firefighter += lpSum(defend_n_t[n][t] - defend_n_t[n][t-1] for n in NODES) <= budget_defenders`

$d_{x,0} = 0$	$x \in V$	`constraint forall(n in NODES)`
		`  (defend_n_t[n,0] = 0);`
			`for n in NODES:`
			`    lp_firefighter += defend_n_t[n][0] == 0`

$b_{x,0} = \begin{cases} 1 & \text{if } x = r_i \text{ for some } i, 1 \leq i \leq f \\ 0 & \text{otherwise} \end{cases}$	$x \in V$	`constraint forall(n in NODES)(`
		`  if START_FIRES[n] = 1 then`
		`    burn_n_t[n,0] = 1`
		`  else`
		`    burn_n_t[n,0] = 0 endif`
		`);`
			`for n in NODES:`
			`    if START_FIRES[n] == 1:`
			`        lp_firefighter += burn_n_t[n][0] == 1`
			`    else:`
			`        lp_firefighter += burn_n_t[n][0] == 0`

Instance generation or sourcing

Decided to create my own system of generating graph instances, using python package NetworkX that can generate hundreds of different types of graphs (parameterized). After playing with and plotting a quite few of them, for my final setup, I ended up with 5 groups of graphs, some containing multiple variants.
Overall, my goal was to make sure that the generated data had some variance, but was also balanced, in terms of the firefighter budget vs defender budget vs the starting locations of the fires. I put a fair amount of time in tweaking the parameter generation code, so it usually generates reasonably balanced data for this problem, for various graph types and node counts. Most of the generation is done using basic weighted randomness and ratios. (no complex math)

Experimental pipeline

M pipeline is setup fully in python and can be run from a single script *pipeline.py, by* instantiating the Pipeline class. What the pipeline does is described below:
1. Firstly, the 2 scripts I created must be present: *firefighter_minizinc.py* and *firefighter_pulp.py* – they have functions that take in the exact same parameters: *MAX_TIME, budget_defenders, n_nodes, n_edges, FROM, TO, START_FIRES* and will instantiate the models for the firefighter problem, execute the solver and return the results (execution_time, status and the result).
Note that the minizinc version is compiled using the official minizinc python bindings – I chose this option for convenience, and my lack of experience with command-line heavy operations. For this to work, minizinc compiler must be installed on the computer, and my *firefighter.mzn* solution file must be present in the same directory, as *pipeline.py*.
2. To instantiate the Pipeline, we provide the node_range (eg. (25,75) means the generated graphs will have between 25-75 nodes), the n_runs (how many firefighter graphs will be generated, and each solver will have to solve) and timeout (time given to solver before it is timed out). Once the pipeline is instantiated, n_runs number of random graphs are created and stored in state, ready to be used by different solvers. We can then run our solvers on our graphs, which will store the results that will be used to automatically generate a comparison graph (Fig 1.) Note that for instances that timeout, we simply set their execution time = timeout limit. I chose this method, because it will not skew the averages (as it would if we chose to ignore the timeouts) and because it should make it very clear in the graphs, where the timeouts have occurred.

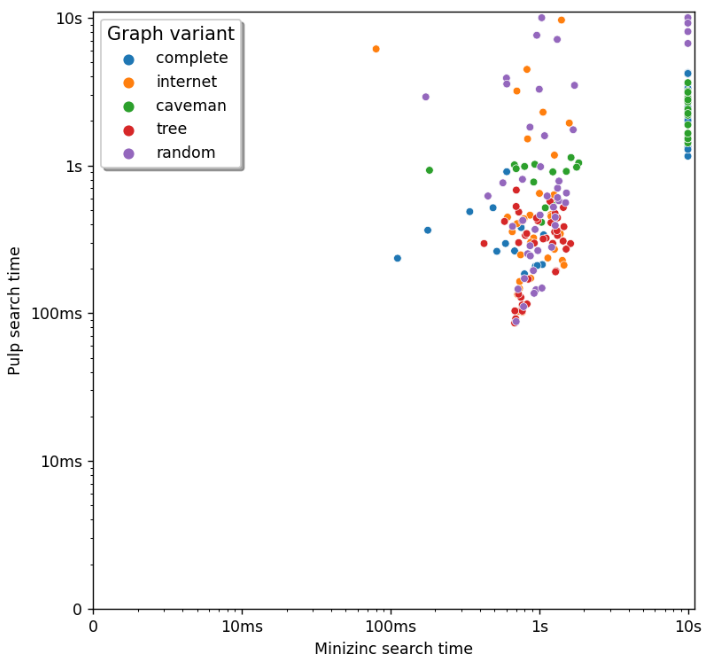
```
pipeline = Pipeline(n_nodes_range=(25, 100), n_runs=200, timeout=10)
pipeline.run("minizinc")
pipeline.run("pulp")
pipeline.generate_graph()
```

-> code used to run the pipeline, that will generate the output graph (fig 1.). 200 graphs will be generated and ran on both minizinc and pulp, and each instance given 10s to execute, before it times out.

Results

Inspiration for the results graph format, taken from week 4 lecture material, 'How do we compare methods?' section.
To my surprise, we can see that pulp performs better on almost all instances of the graphs! Additionally, timeouts are much more frequent in Minizinc (can be seen by the grouping of points at time 10s). Another interesting observation, is that all the minizinc solutions are either completed in under 2s, or result in a timeout – there are no inbetweens. For pulp however, those instances were oftentimes solved, and oftentimes between 2s and 10s.
I was curious why that is happening, so I ran some of the timed-out graphs on the minzinc IDE, and discovered that Minizinc would find and print a candidate solution quite quickly (around the same time as PuLP, between 2s and 10s), and it also happened to be the same (best) solution as PuLP returned, however, it would keep searching for other solutions for a very long time and could not establish that it is fact the best solution. With pulp however, such timeouts are almost nonexistent in fig 1., and I suspect it is because the solver is somehow able to figure out much quicker if a discovered solution is optimal or not.

Fig 1.



Extras/Acknowledgements

1. I worked on this project alone. There was a lot of coding and googling through documentation involved (mainly for plotting, graphs and minizinc binding) – used resources are included in comments, in the source code.
2. For the bonus section, I thought one way the minizinc ILP can be optimized is by setting the domains of defend_n_t and burn_n_t to inbuilt data type boolean, instead of 0..1. I also had to use the inbuilt bool2int() function, in places where I was comparing or adding the defend and burn values.
I still used the original version when running the pipeline, since that was most appropriate for my objective, however I saved a copy of the bool version in file called 'firefighter_bonus.mzn'. Sadly, this did not yield any significant increase in performance. ☹