Generating Compound Moves for Local Search by Systematic Search

Gustav Björdal,
joint work with Pierre Flener and Justin Pearson

Research Overview

We have previously developed a local-search backend to the modelling language MiniZinc.

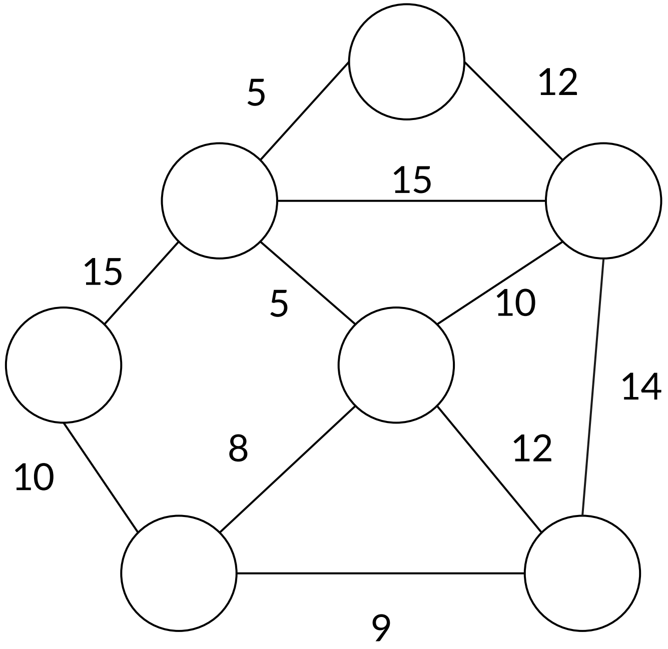
Such a backend must automatically infer a search strategy from a model.

I will here discuss a case where this inference can go wrong and what can be done about it.

TSP with Time Windows

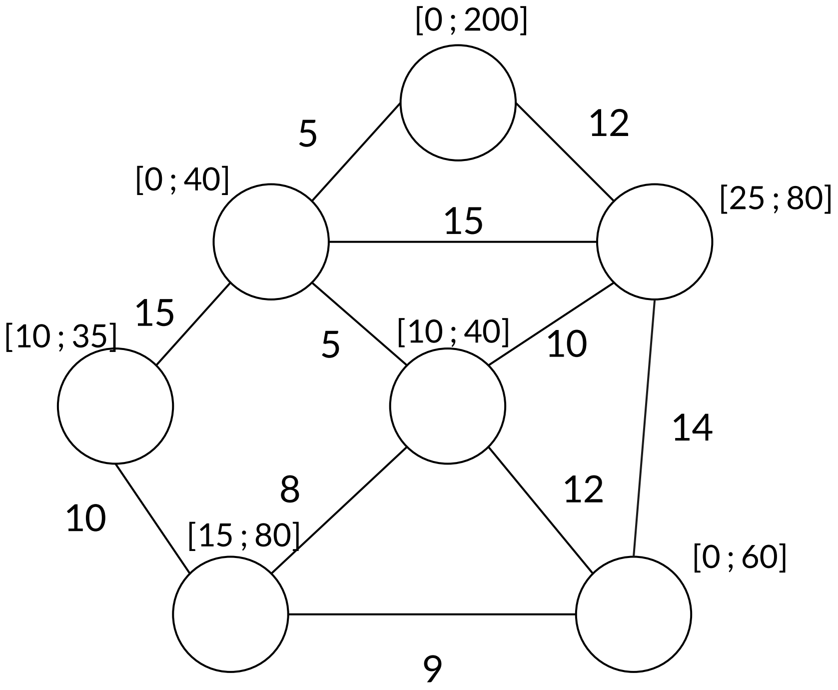
TSP with Time Windows (TSPTW)

Given a graph with weighted edges and a time window for each node:



TSP with Time Windows (TSPTW)

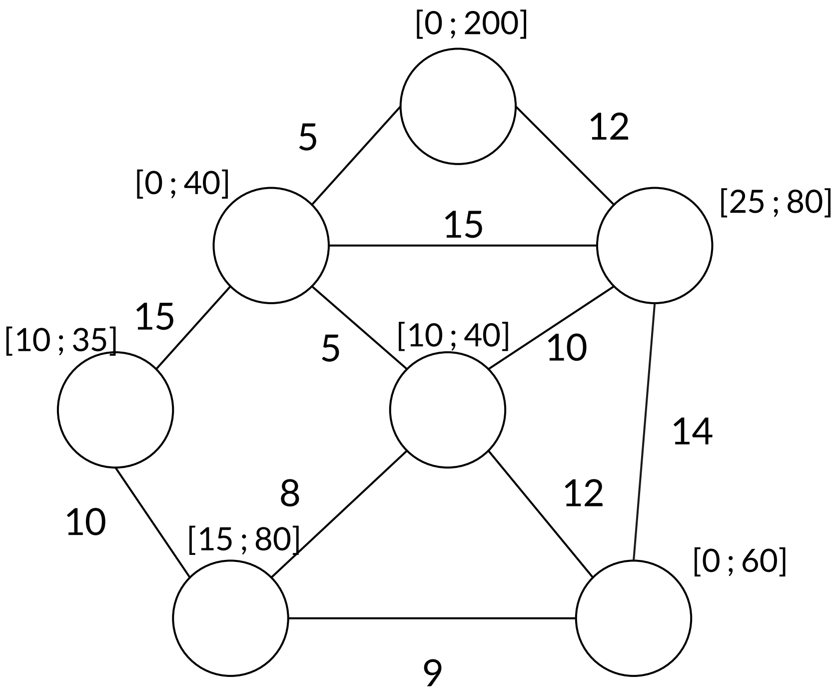
Given a graph with weighted edges and a time window for each node:



TSP with Time Windows (TSPTW)

Given a graph with weighted edges and a time window for each node:

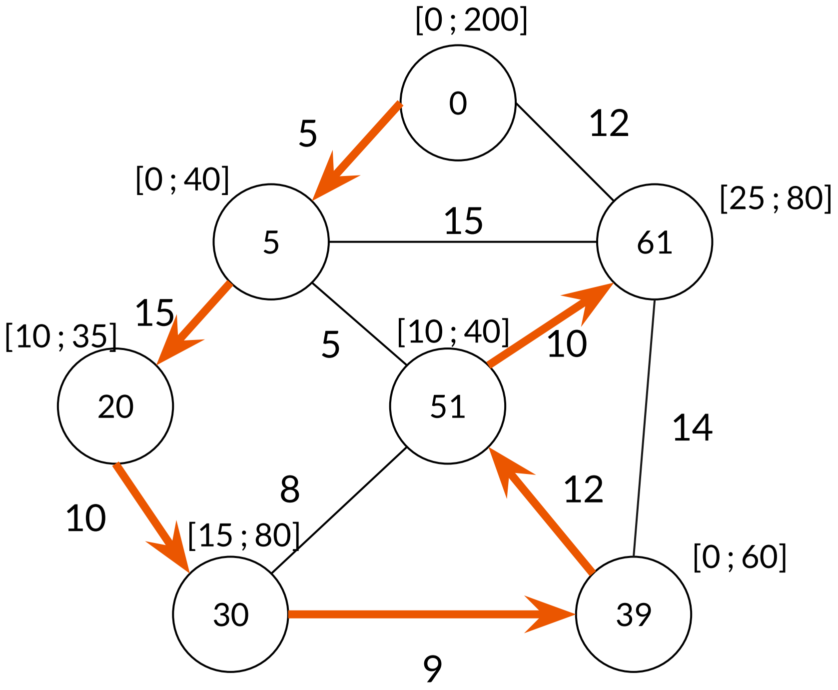
find a shortest Hamiltonian path, such that each node is visited within its time window.



TSP with Time Windows (TSPTW)

Given a graph with weighted edges and a time window for each node:

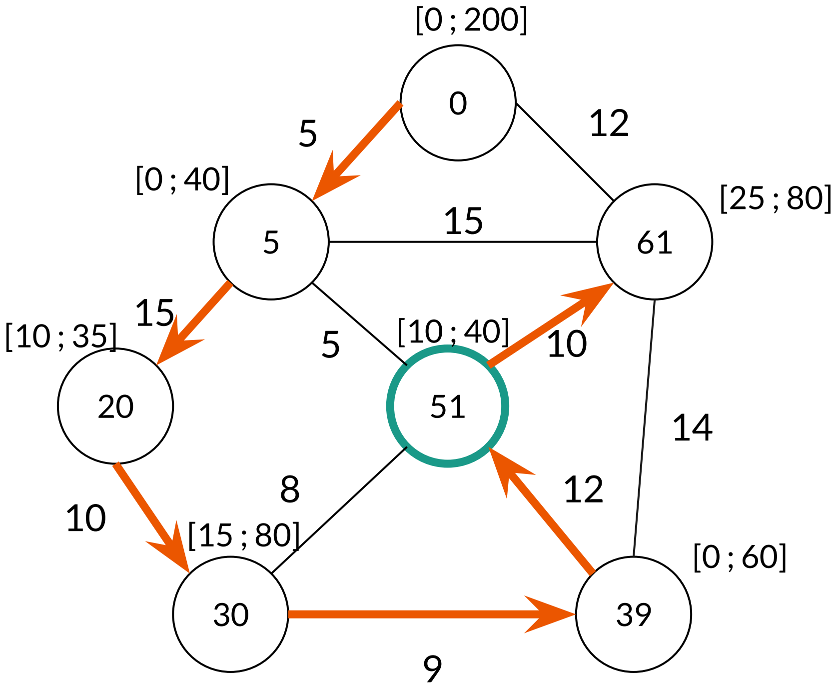
find a shortest Hamiltonian path, such that each node is visited within its time window.



TSP with Time Windows (TSPTW)

Given a graph with weighted edges and a time window for each node:

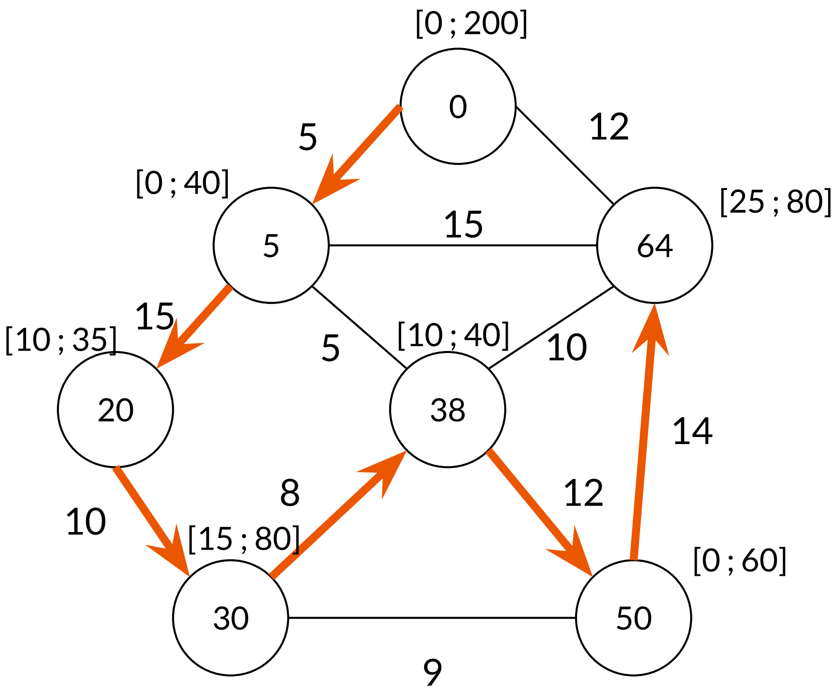
find a shortest Hamiltonian path, such that each node is visited within its time window.



TSP with Time Windows (TSPTW)

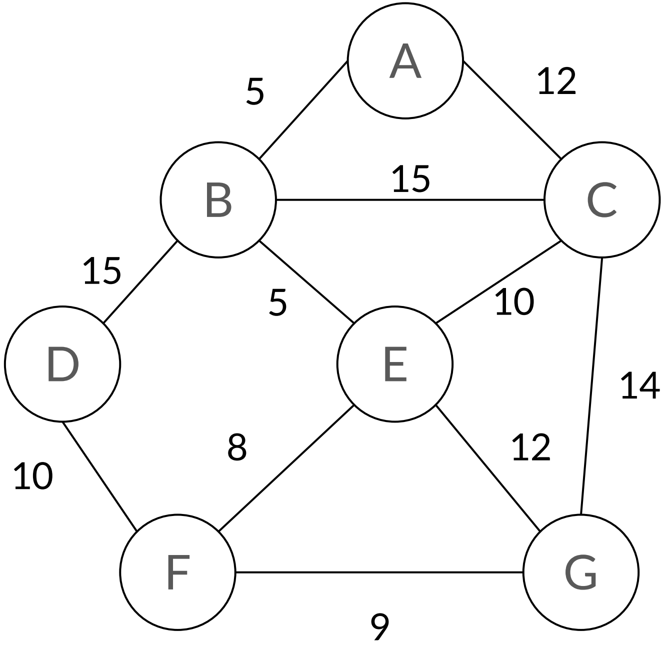
Given a graph with weighted edges and a time window for each node:

find a shortest Hamiltonian path, such that each node is visited within its time window.



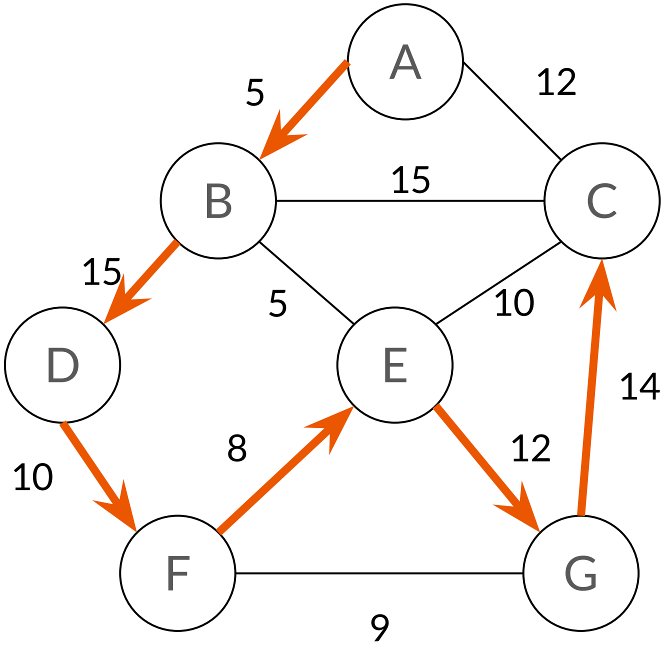
Representing a Route

?	?	?	?	?	?	?
1	2	3	4	5	6	7



Representing a Route

A	B	D	F	E	G	C
1	2	3	4	5	6	7



Modelling TSPTW in MiniZinc

Model Overview

Data

- Nodes
- Distances
- Time windows and duration

Variables

- The route
- Arrival times

Constraints

- Visit each node once
- Within time windows

Objective

- Total distance

Data

```
set of int: Nodes= 1..7;
set of int: Positions = 1..7;   % positions in the route
array[Nodes, Nodes] of int: distance = ...;
array[Nodes] of int: open = ...;
array[Nodes] of int: close = ...;
```

Data

```
set of int: Nodes= 1..7;
set of int: Positions = 1..7;
array[Nodes, Nodes] of int: distance = …;
array[Nodes] of int: open = ...;
array[Nodes] of int: close = ...;
```

distance[2, 5] is the time we spend at node 2 plus the travel time from node 2 to node 5.

Data

```
set of int: Nodes= 1..7;
set of int: Positions = 1..7;
array[Nodes, Nodes] of int: distance = …;
array[Nodes] of int: open = [0,      0, ...];
array[Nodes] of int: close = [200, 40, ...];
```

Variables

```
array[Positions] of var Nodes: route;
array[Positions] of var int: arrivalTime;
```

Variables

```
array[Positions] of var Nodes: route;
array[Positions] of var int: arrivalTime;
```

route =

?	?	?	?	?	?	?
1	2	3	4	5	6	7

Variables

```
array[Positions] of var Nodes: route;
array[Positions] of var int: arrivalTime;
```

route =

?	?	?	?	?	?	?
1	2	3	4	5	6	7

route[1] = first node we visit

route[2] = second node we visit

...

route[13] = last node we visit

Constraints

We do not visit the same node twice:

```
constraint alldifferent(route);
```

Constraints

We do not visit the same node twice:

```
constraint alldifferent(route);
```

We arrive at the node at the first position after it opens:

```
constraint arrivalTime[1] >= open[route[1]];
```

Constraints (cont.)

We arrive at the node at position i after it opens and after we are done at the node at position i–1 plus the travel time:

```
constraint forall(i in Positions where i != 1)(
    arrivalTime[i] >= open[route[i]]
    /\
    arrivalTime[i] >=
    arrivalTime[i-1] + distance[route[i-1], route[i]]
);
```

Constraints (cont.)

We arrive at the node at position i after it opens and after we are done at the node at position i–1 plus the travel time:

```
constraint forall(i in Positions where i != 1)(
    arrivalTime[i] >=
        max(open[route[i]],
            arrivalTime[i-1] + distance[route[i-1], route[i]]
);
```

Constraints (cont.)

We arrive at each node before it closes:

```
constraint forall(i in Positions)(
    arrivalTime[i] <= close[route[i]]
);
```

Minimise the Distance

```
var int: totalDistance;
constraint totalDistance = sum(i in Positions where i != 7)(
    distance[route[i], route[i+1]]
);
solve minimize totalDistance;
```

A TSPTW Model in MiniZinc

```
set of int: Nodes= 1..7;
set of int: Positions = 1..7;
array[Nodes, Nodes] of int: distance = …;
array[Nodes] of int: open = [0,      0, ...];
array[Nodes] of int: close = [200, 40, ...];
```

A TSPTW Model in MiniZinc

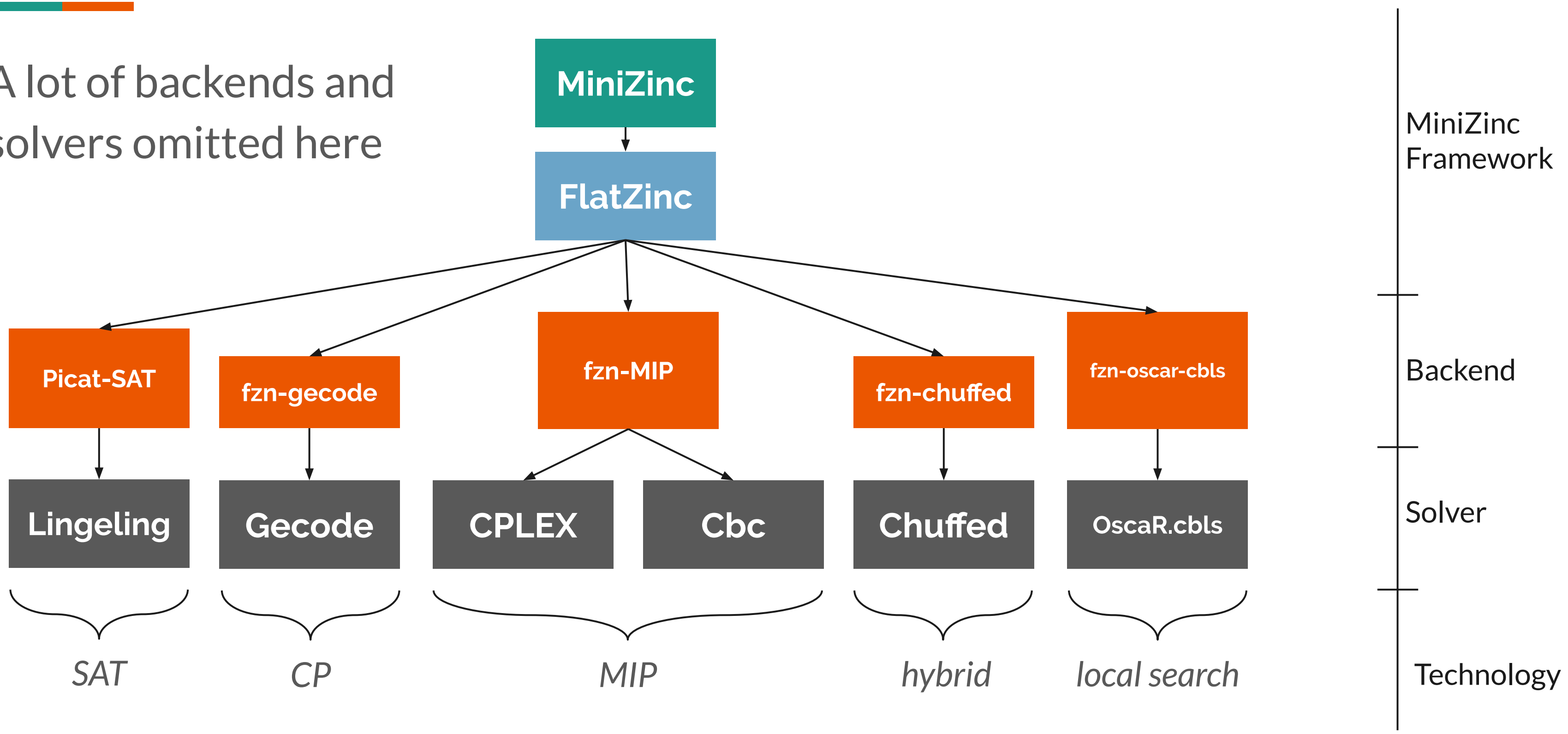
```
…
array[Positions] of var Nodes: route;
array[Positions] of var int: arrivalTime;

constraint alldifferent(route);
constraint arrivalTime[1] >= open[route[1]];
constraint forall(i in Positions where i != 1)(arrivalTime[i] >= … );
constraint forall(i in Positions)(arrivalTime[i] <= ...);
var int: totalDistance;
constraint totalDistance = …;
solve minimize totalDistance;
```

The MiniZinc Framework

The MiniZinc Pipeline

A lot of backends and solvers omitted here



Model Once, Solve Everywhere

MiniZinc offers a unified modelling language for all technologies.

However, backends are not always robust to model variations.

Local-Search Backends on this Model

If we solve the TSPTW model with local search, then we see:

Local-Search Backends on this Model

If we solve the TSPTW model with local search, then we see:

	fzn-oscar-cbls	Yuck	LocalSolver
instance			
n40w120	–	468	–
n40w140	–	391	–
n40w160	–	411	–

Local-Search Backends on This Model

not a MiniZinc backend

If we solve the TSPTW model with local search, then we see:

	fzn-oscar-cbls	Yuck	LocalSolver
instance			
n40w120	–	468	–
n40w140	–	391	–
n40w160	–	411	–

Local-Search Backends on This Model

If we solve the TSPTW model with local search, then we see:

	fzn-oscar-cbls	Yuck	LocalSolver
instance			
n40w120	–	468	–
n40w140	–	391	–
n40w160	–	411	–

Local search is usually good for this kind of problem: why not here?!

Constraints (cont.)

We arrive at the node at position i after it opens and after we are done at the node at position i–1 plus the travel time:

```
constraint forall(i in Positions where i != 1)(
    arrivalTime[i] >=
        max(open[route[i]],
            arrivalTime[i–1] + distance[route[i–1], route[i]]
);
```

Inequality or Equality

```
constraint forall(i in Positions where i != 1)(
    arrivalTime[i] >=
        max(open[route[i]],
            arrivalTime[i-1] + distance[route[i-1], route[i]]);
```

or

```
constraint forall(i in Positions where i != 1)(
    arrivalTime[i] =
        max(open[route[i]],
            arrivalTime[i-1] + distance[route[i-1], route[i]]);
```

Comparing the Equality and Inequality Models

	fzn-oscar-cbls	Yuck	LocalSolver
instance \ model	ineq	ineq	ineq
n40w120	–	468	–
n40w140	–	391	–
n40w160	–	411	–

Comparing the Equality and Inequality Models

	fzn-oscar-cbls		Yuck		LocalSolver	
instance \ model	`eq`	`ineq`	`eq`	`ineq`	`eq`	`ineq`
n40w120	+**434**	–	436	468	+**434**	–
n40w140	+**328**	–	+**328**	391	+**328**	–
n40w160	352	–	+**348**	411	+**348**	–

What's Going on Here?

We significantly improved this model by changing one constraint.

But can this also be done for other models/problems?

What if such a fix cannot be made?

What's actually going on here?

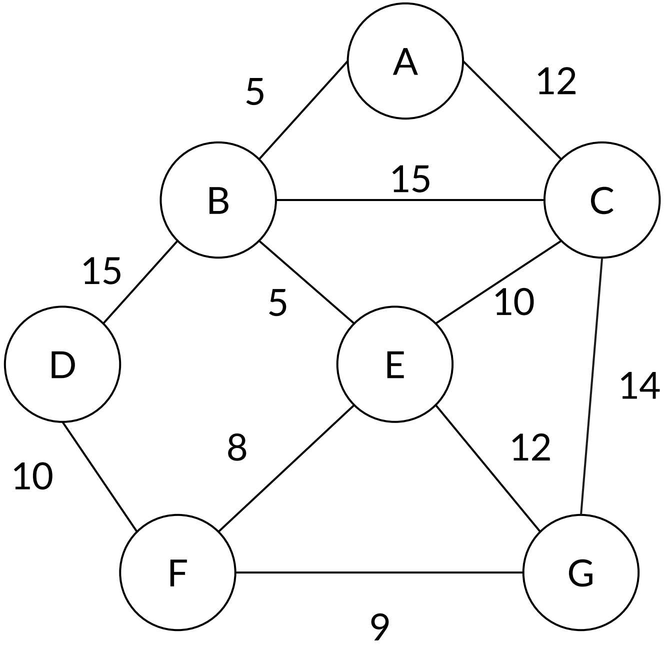
Local Search

Solving TSP with Local Search

We want to search for a best assignment of variables.

Variables

?	?	?	?	?	?	?
1	2	3	4	5	6	7



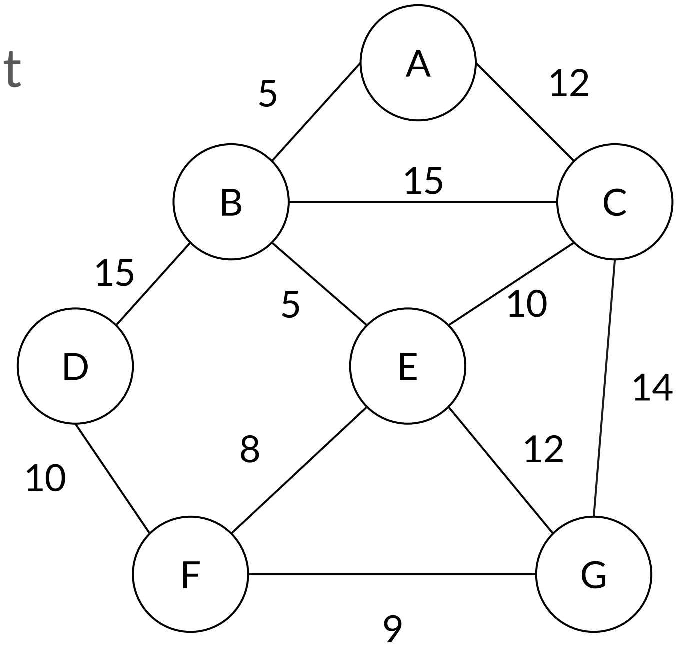
Random Initial Assignment

We want to search for a best assignment of variables.

Pick a random assignment and evaluate it

Variables

A	B	C	D	E	F	G
1	2	3	4	5	6	7



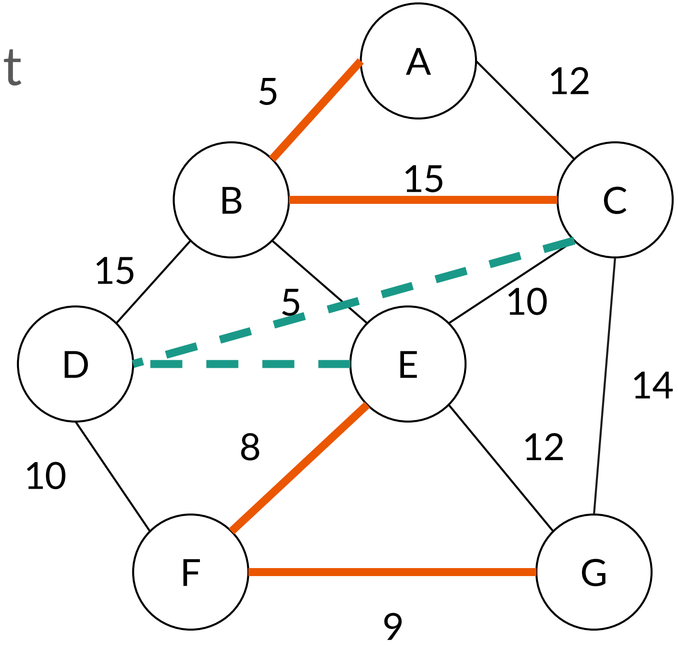
Random Initial Assignment

We want to search for a best assignment of variables.

Pick a random assignment and evaluate it

Variables

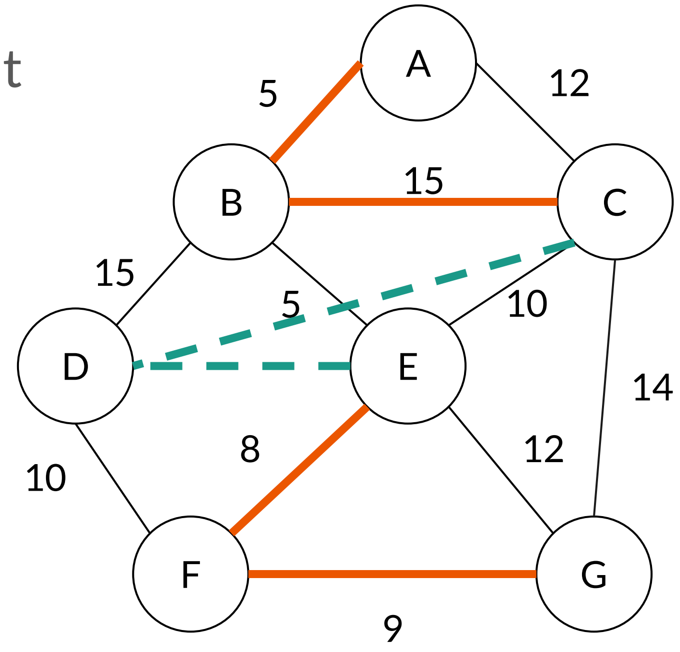
A	B	C	D	E	F	G
1	2	3	4	5	6	7



Random Initial Assignment

We want to search for a best assignment of variables.

Pick a random assignment and evaluate it



Variables

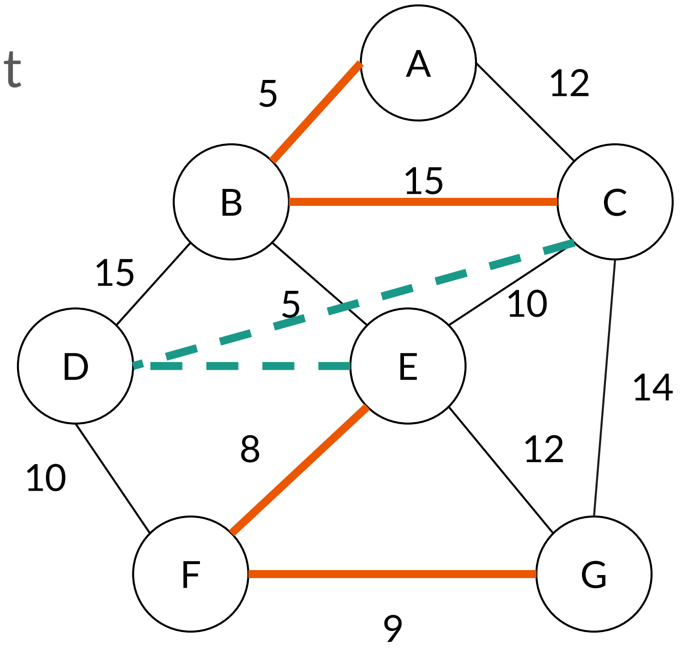
A	B	C	D	E	F	G
1	2	3	4	5	6	7

Cost: 37 + 2

Random Initial Assignment

We want to search for a best assignment of variables.

Pick a random assignment and evaluate it



Variables

A	B	C	D	E	F	G
1	2	3	4	5	6	7

Cost: 37 + 2·1000 = 2037

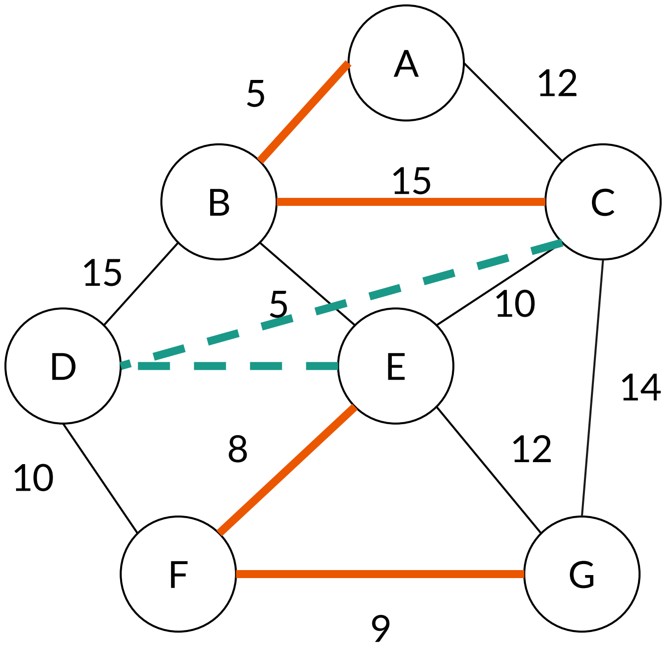
Generate Neighbours

Explore all similar assignments we get upon swapping two values.

Variables

A	B	C	D	E	F	G
1	2	3	4	5	6	7

Cost: 37 + 2·1000 = 2037



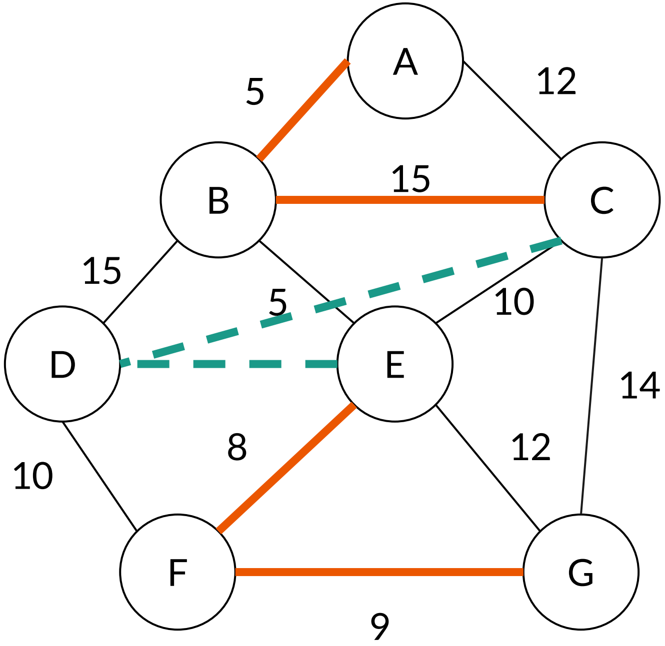
Generate Neighbours

Explore all similar assignments we get upon swapping the values of two variables.

Variables

A	B	C	D	E	F	G
1	2	3	4	5	6	7

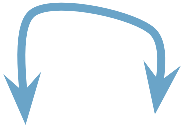
Cost: 37 + 2·1000 = 2037



Generate Neighbours

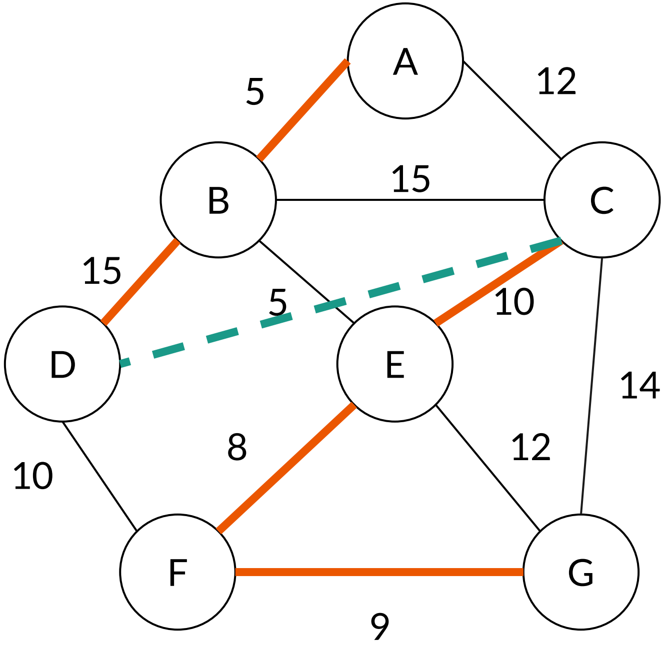
Explore all similar assignments we get upon swapping the values of two variables.

Variables



A	B	D	C	E	F	G
1	2	3	4	5	6	7

Cost: 47 + 1·1000 = 1047



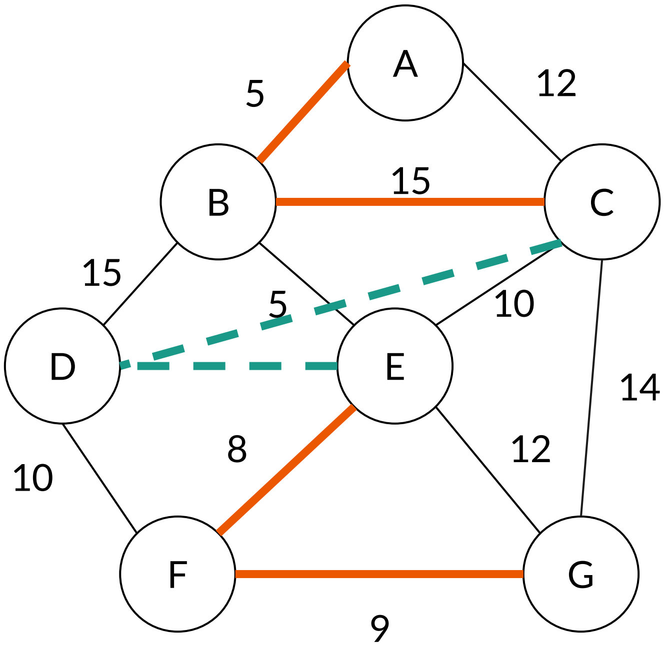
Generate Neighbours

Explore all similar assignments we get upon swapping the values of two variables.

Variables

A	B	C	D	E	F	G
1	2	3	4	5	6	7

Cost: 37 + 2·1000 = 2037



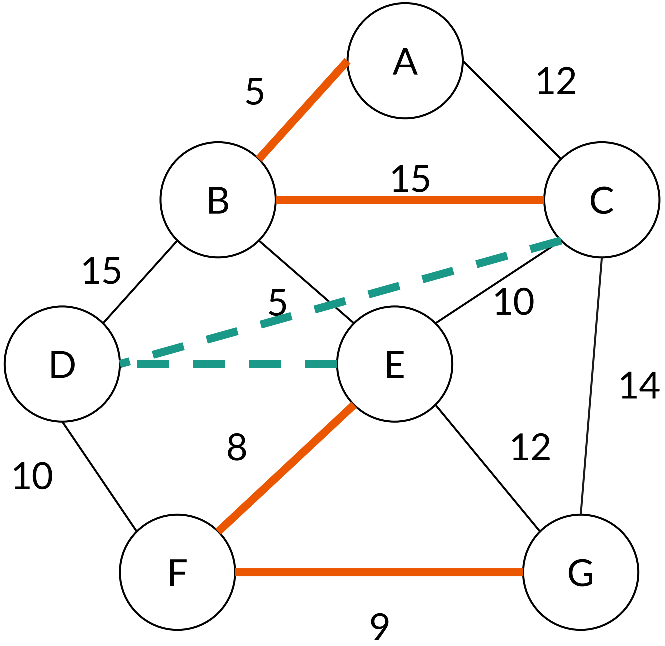
Generate Neighbours

Explore all similar assignments we get from swapping the values of two variables.

Variables

A	B	C	D	E	F	G
1	2	3	4	5	6	7

Cost: $37 + 2 \cdot 1000 = 2037$



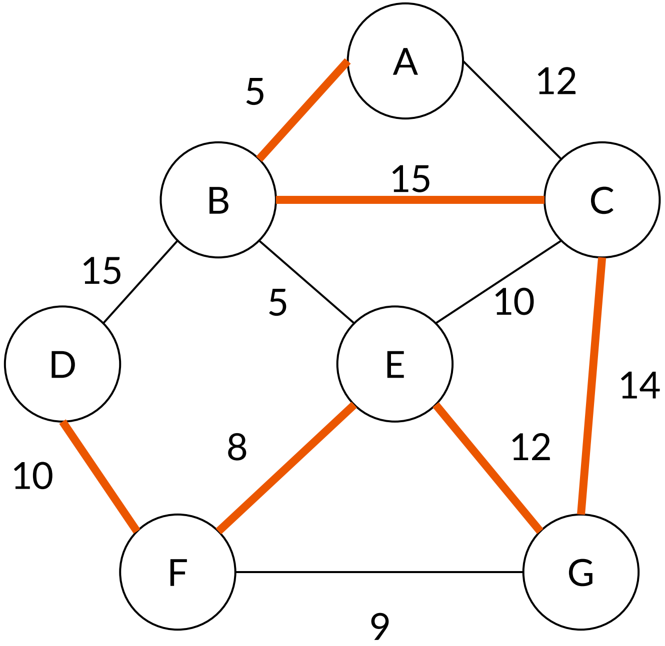
Generate Neighbours

Explore all similar assignments we get from swapping the values of two variables.

Variables

A	B	C	G	E	F	D
1	2	3	4	5	6	7

Cost: 64 + 0·1000 = 64

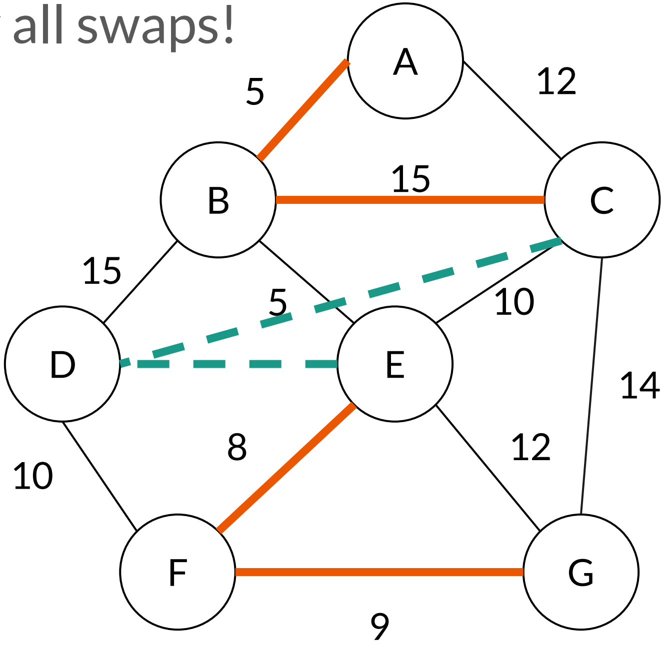


Generate Neighbours

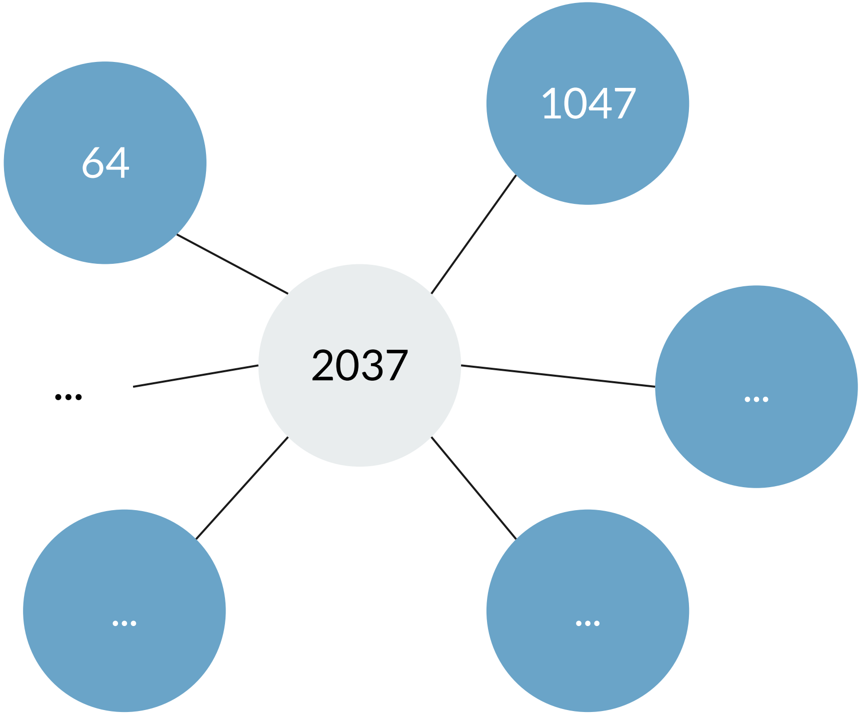
Explore all similar assignments we get from swapping the values of two variables. Try all swaps!

A	B	C	D	E	F	G
1	2	3	4	5	6	7

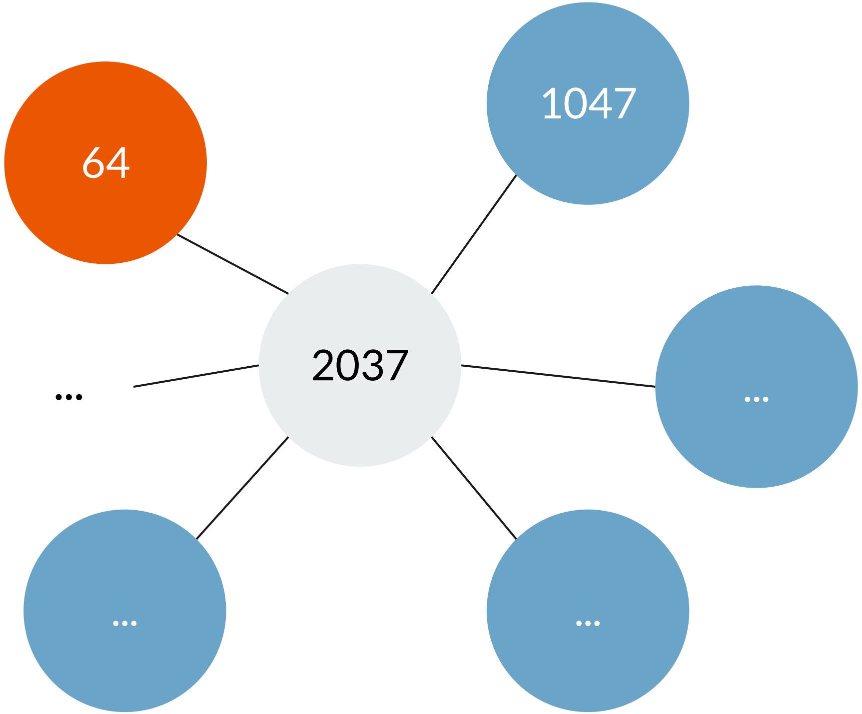
Cost: ?? + ?·1000 = ????



Select a Best Neighbour

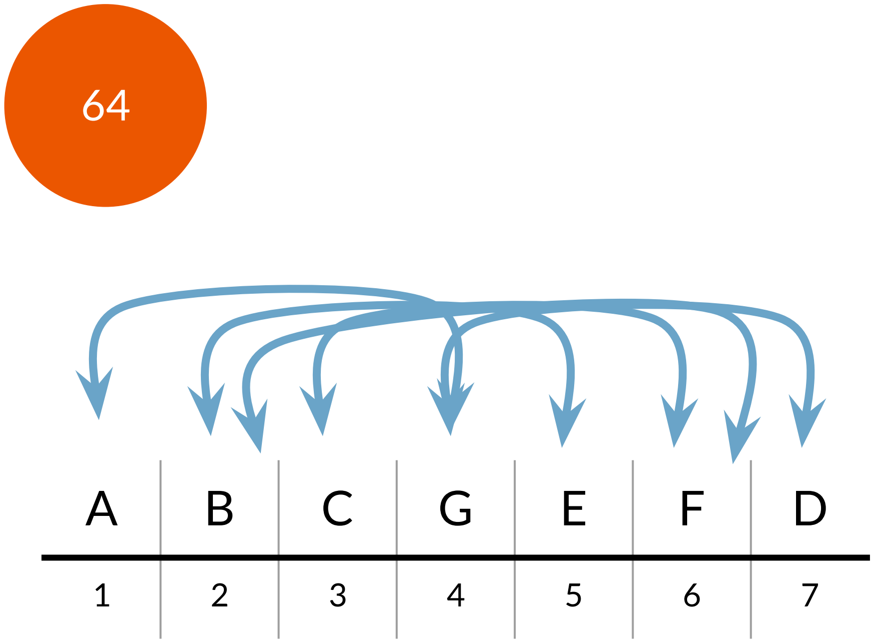


Select a Best Neighbour and Move To It



Select a Best Neighbour and Move To It

Repeat



Design Aspects

Initialisation and neighbourhood

- How do we make the initial assignment?
- How are the neighbours obtained?

Cost function

- How do we evaluate the quality of an assignment?

(Meta-)Heuristic

- Which neighbour is selected?
- How do we prevent the search from getting stuck?

Design Aspects

Initialisation and neighbourhood

- How do we make the initial assignment? Random
- How are the neighbours obtained? Swap

Cost function

- How do we evaluate the quality of an assignment? x+y

(Meta-)Heuristic

- Which neighbour is selected? Best
- How do we prevent the search from getting stuck?

From MiniZinc to Local Search

From This

```
…
array[Positions] of var Nodes: route;
array[Positions] of var int: arrivalTime;

constraint alldifferent(route);
constraint arrivalTime[1] >= open[route[1]];
constraint forall(i in Positions where i != 1)(arrivalTime[i] >= … );
constraint forall(i in Positions)(arrivalTime[i] <= ...);
var int: totalDistance;
constraint totalDistance = …;
solve minimize totalDistance;
```

To This

Initialisation and neighbourhood

- How do we make the initial assignment?
- How are the neighbours obtained?

Cost function

- How do we evaluate the quality of an assignment?

(Meta-)Heuristic

- Which neighbour is selected?
- How do we prevent the search from getting stuck?

How Local-Search Backends to MiniZinc Do It

Determine:

1. which variables to make moves on,
2. which moves to make,
3. how to evaluate the quality of an assignment, and
4. a (meta-)heuristic.

Variables

```
array[Positions] of var Nodes: route;
array[Positions] of var int: arrivalTime;
var int: totalDistance;
```

Variables

```
array[Positions] of var Nodes: route;
array[Positions] of var int: arrivalTime;
var int: totalDistance;
```

Some variables are functionally defined by other variables.

Search variables

Functionally defined variables

Variables

Some variables are functionally defined by other variables.

```
constraint totalDistance = sum(i in Positions where i != 7)(
    distance[route[i], route[i+1]]
);
```

Search variables	Functionally defined variables
	totalDistance

Variables

The rest are search variables.

route[i] arrivalTime[i]	totalDistance
Search variables	Functionally defined variables

Initialisation and Neighbourhood

We must initialise and make moves on the search variables.

Constraints can hint at moves.

Initialisation and Neighbourhood

```
array[Positions] of var Nodes: route;
array[Positions] of var int: arrivalTime;
constraint alldifferent(route);
```

Initialisation and Neighbourhood

```
array[Positions] of var Nodes: route;
array[Positions] of var int: arrivalTime;
constraint alldifferent(route);
```

The alldifferent constraint tells us to initialise ‘route’ to different values and do swap moves.

For arrivalTime[i] we can initialise randomly and do assign moves.

Initialisation and Neighbourhood

```
array[Positions] of var Nodes: route;
array[Positions] of var int: arrivalTime;
constraint alldifferent(route);
```

The alldifferent constraint tells us to initialise ‘route’ to different values and do swap moves.

For arrivalTime we can initialise randomly and do assign moves.

A move is either: swap two values in route
or: change a value in arrivalTime

Cost Function

```
array[Positions] of var Nodes: route;
array[Positions] of var int: arrivalTime;
constraint alldifferent(route);
constraint arrivalTime[1] >= open[route[1]];
constraint forall(i in Positions where i != 1)(arrivalTime[i] >= … );
constraint forall(i in Positions)(arrivalTime[i] <= ...);
var int: totalDistance;
constraint totalDistance = …;
solve minimize totalDistance;
```

Cost Function

```
constraint arrivalTime[1] >= open[route[1]];
constraint forall(i in Positions where i != 1)(arrivalTime[i] >= … );
constraint forall(i in Positions)(arrivalTime[i] <= ...);
var int: totalDistance;
constraint totalDistance = …;
solve minimize totalDistance;
```

Cost: objective + penalty

(Meta-)Heuristic

Tabu search

The Problem with Automatically Inferred Neighbourhoods

Neighbourhood and Moves

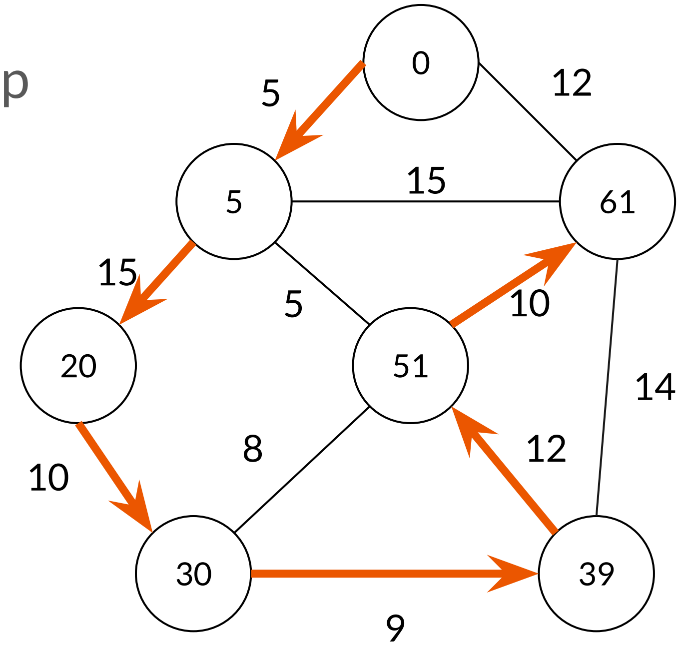
The alldifferent constraint tells us to initialise ‘route’ to different values and do swap moves.

For arrivalTime we can initialise randomly and do assign moves.

But the latter is really bad!

Bad Moves

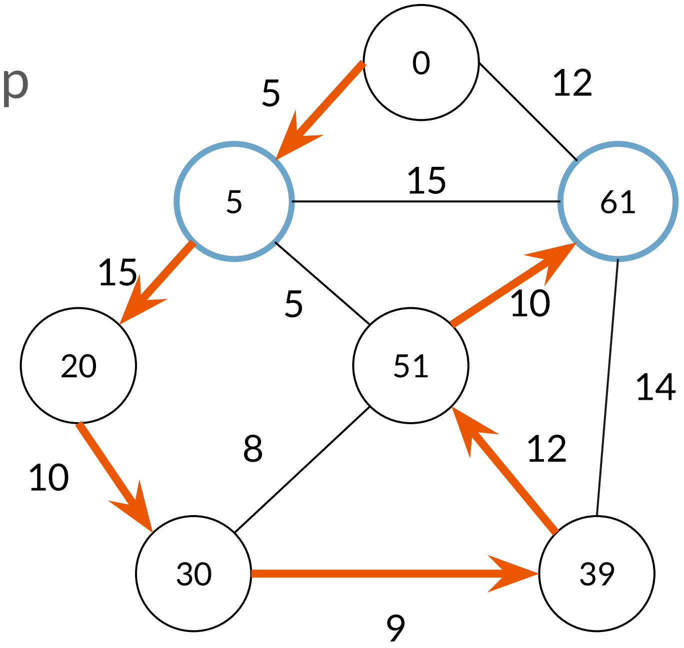
Let's do a swap



Move counter: 0

Bad Moves

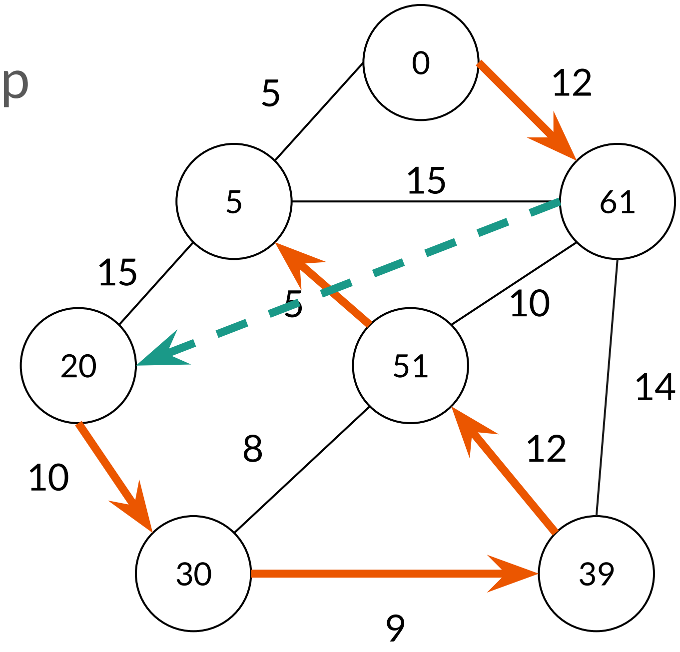
Let's do a swap



Move counter: 0

Bad Moves

Let's do a swap

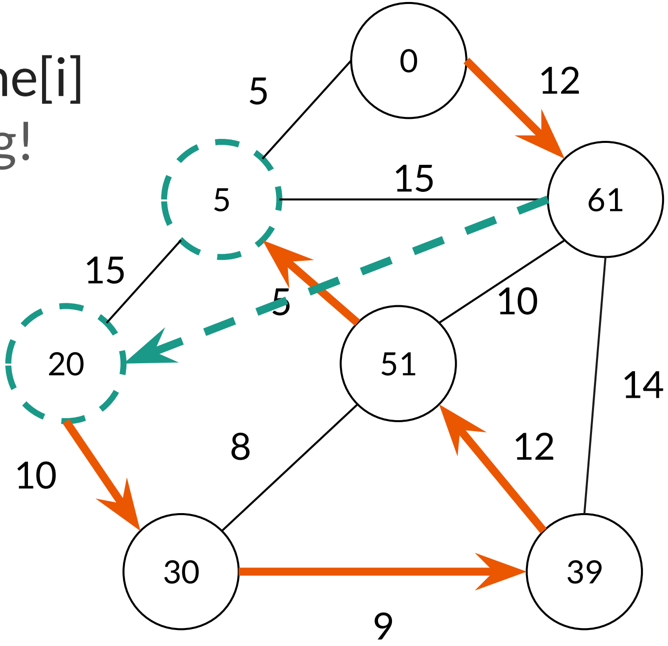


Assume invalid edges have distance 0.

Move counter: 1

Bad Moves

The arrivalTime[i] are now wrong!

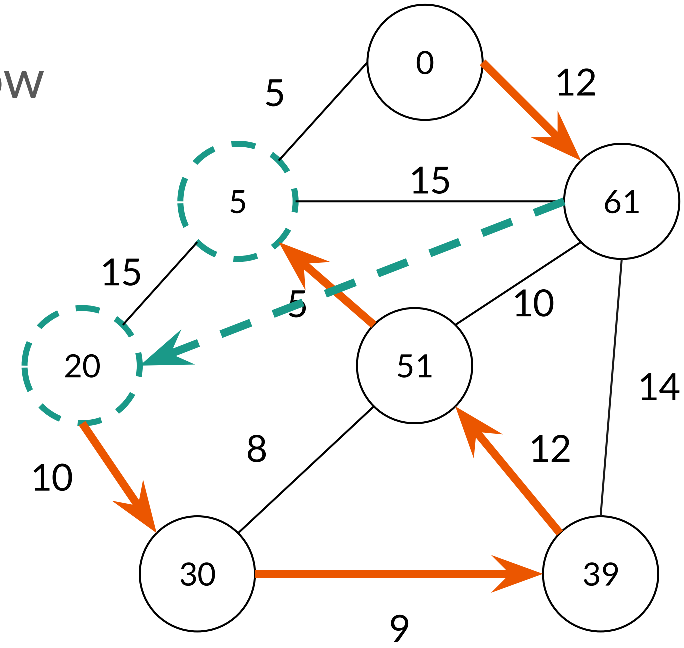


Assume invalid edges have distance 0.

Move counter: 1

Bad Moves

We should now do assign moves...

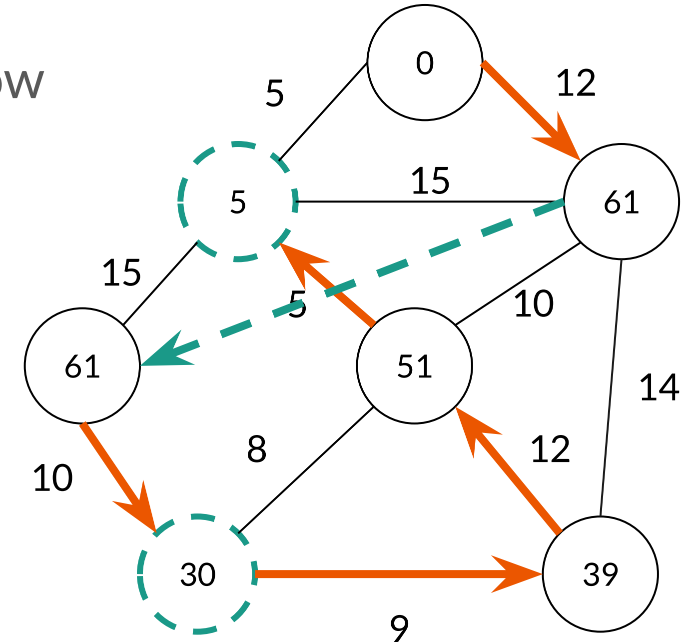


Assume invalid edges have distance 0.

Move counter: 1

Bad Moves

We should now do assign moves...



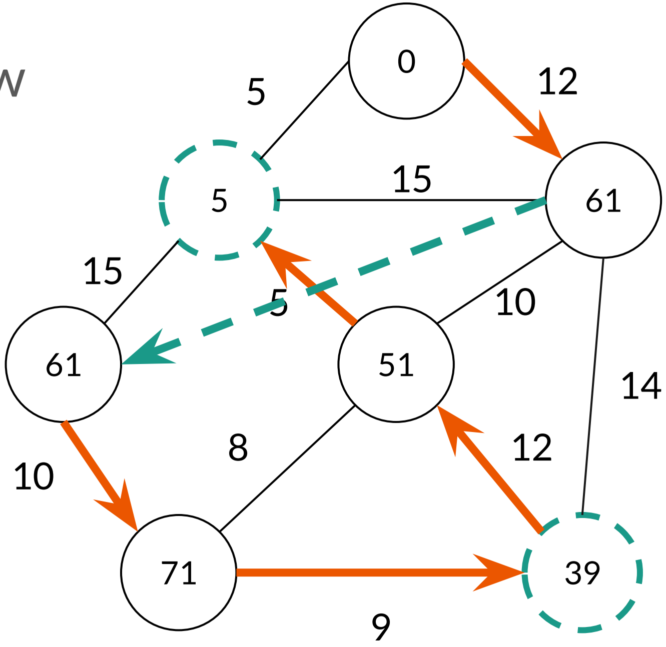
Assume invalid edges have distance 0.

Move counter: 2

Bad Moves

We **should** now do assign moves...

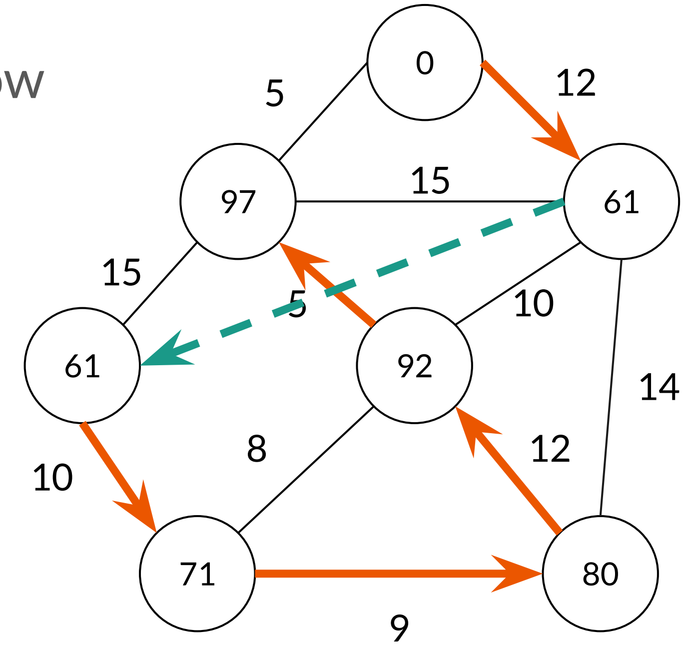
Move counter: 3



Assume invalid edges have distance 0.

Bad Moves

We should now do assign moves...



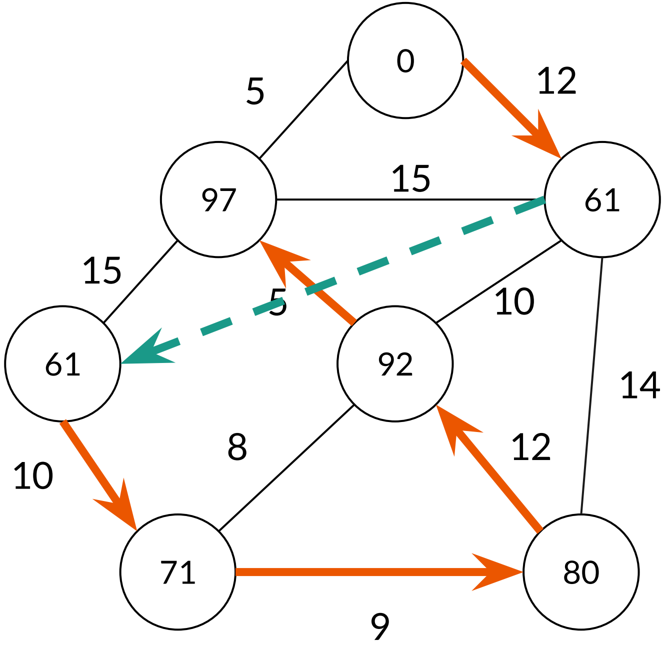
Assume invalid edges have distance 0.

Move counter: 6

Bad Moves

It takes 6 very coordinated moves to repair all the arrivalTime[i] values.

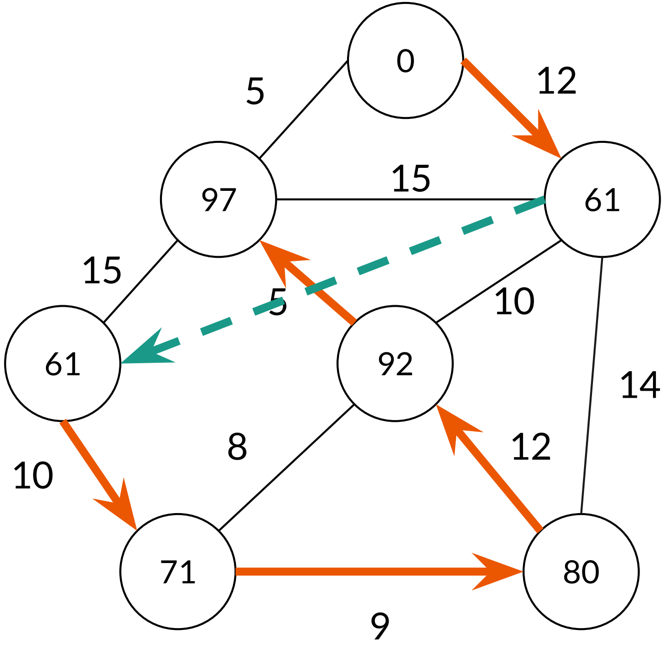
But there is randomness involved in selecting moves, so no hope here!



Bad Moves

We should not search locally for the arrivalTime[i] values!

The arrivalTime[i] are *auxiliary* and mainly represent side information.



What About the Equality Model?

```
constraint forall(i in Positions where i != 1)(
	arrivalTime[i] =
		max(open[route[i]],
			arrivalTime[i-1] + distance[route[i-1], route[i]]);
```

instead of

```
constraint forall(i in Positions where i != 1)(
	arrivalTime[i] >=
		max(open[route[i]],
			arrivalTime[i-1] + distance[route[i-1], route[i]]);
```

More Functionally Defined Variables

route[i]
arrivalTime[i]

Search variables

totalDistance

Functionally defined variables

More Functionally Defined Variables

In the equality model, the arrivalTime[i] are functionally defined!

route[i]	totalDistance arrivalTime[i]
Search variables	Functionally defined variables

Another Way to Look at It

In the inequality model, we actually have a third category: auxiliary variables.

route[i]	arrivalTime[i]	totalDistance
Search variables	Auxiliary variables	Functionally defined variables

Auxiliary Variables

For some models, we cannot do the ineq->eq reformulation in order to eliminate auxiliary variables.

How do we:

1. detect auxiliary variables in a model?
2. avoid searching locally over them?

Generating Compound Moves

Generating Compound Moves

Detect auxiliary variables in a model:

- Add annotation to MiniZinc for specifying search variables.
- Assume variables without a good move are auxiliary.

Avoid searching locally over auxiliary variables:

- Use a constraint programming solver to compute their value after every move on search variables.

We explored many different configurations of this hybridisation.

Numbers!

	fzn-oscar-cbls original		fzn-oscar-cbls CMG config1		fzn-oscar-cbls CMG config2		Yuck		Chuffed
TSPTW	best	median	best	median	best	median	best	median	best
n20w180	377	377^{1}	$^{+}$**253**	253^{10}	261	263^{10}	–	–	***253**
n20w200	347	373^{2}	$^{+}$**233**	233^{10}	$^{+}$**233**	234^{10}	–	–	***233**
n40w120	–	–	$^{+}$**434**	439^{5}	437	464^{9}	–	–	536
n40w140	–	–	$^{+}$**328**	334^{7}	367	388^{10}	–	–	–
n40w160	–	–	$^{+}$**348**	349^{8}	362	393^{10}	–	–	–
CVRP									
A-n37-k5	2614	2870^{9}	2925	2934^{10}	**875**	983^{9}	–	–	1570
A-n64-k9	5431	5659^{4}	5518	5661^{9}	**2868**	3472^{8}	–	–	3667
B-n45-k5	3638	4121^{6}	4201	4207^{10}	**972**	1182^{10}	–	–	2466
P-n16-k8	489	503^{2}	**450**	523^{6}	481	481^{1}	–	–	502

Any questions?

Thank you for listening!