Conference Scheduler Documentation

Release ”

PyCon UK

Oct 29, 2017

Contents

1 Tutorial **3**
- 1.1 Installing the conference scheduler . . . 3
- 1.2 Inputting the data . . . 3
- 1.3 Creating a schedule . . . 5
- 1.4 Avoiding room overcrowding . . . 6
- 1.5 Coping with new information . . . 7
- 1.6 Spotting the Changes . . . 8
- 1.7 Scheduling chairs . . . 9
- 1.8 Validating a schedule . . . 10

2 How To... **13**
- 2.1 Define a conference . . . 13
- 2.2 Modify event tags and unavailabiity . . . 19
- 2.3 Use different solvers . . . 20
- 2.4 Use heuristics . . . 20
- 2.5 Obtain the mathematical representation of a schedule . . . 21

3 Background **23**
- 3.1 Mathematical model . . . 23
- 3.2 Heuristics . . . 26
- 3.3 Bibliography . . . 27

4 Reference **29**
- 4.1 conference_scheduler.scheduler . . . 29
- 4.2 conference_scheduler.resources . . . 33

5 Indices and tables **37**

Bibliography **39**

Python Module Index **41**

Contents:

CHAPTER 1

Tutorial

Welcome! You are the head scheduler for the upcoming `name` con. You have a venue, you have talks and you have a week to schedule everything!

Your venue has two rooms in which you will schedule talks and workshops in parallel but you're also going to want to find time for some social events.

You have organised your time slots as follows:

- The first day will have 2 sessions (morning and afternoon) with two 30 minute time slots in each room.
- The second day will have 1 room used for longer 1 hour workshops, the other room used for more talks and 2 long sessions set aside for the social events.

Installing the conference scheduler

The conference scheduler is compatible with Python 3.6+ only.

You can install the latest version of `conference_scheduler` from PyPI:

```
$ pip install conference_scheduler
```

If you want to, you can also install a development version from source:

```
$ git clone https://github.com/PyconUK/ConferenceScheduler
$ cd ConferenceScheduler
$ python setup.py develop
```

Inputting the data

Let us create these time slots using the `conference_scheduler`:

```
>>> from datetime import datetime
>>> from conference_scheduler.resources import Slot, Event

>>> talk_slots  = [Slot(venue='Big', starts_at=datetime(2016, 9, 15, 9, 30),␣
↪duration=30, session="A", capacity=200),
...                Slot(venue='Big', starts_at=datetime(2016, 9, 15, 10, 0),␣
↪duration=30, session="A", capacity=200),
...                Slot(venue='Small', starts_at=datetime(2016, 9, 15, 9, 30),␣
↪duration=30, session="B", capacity=50),
...                Slot(venue='Small', starts_at=datetime(2016, 9, 15, 10, 0),␣
↪duration=30, session="B", capacity=50),
...                Slot(venue='Big', starts_at=datetime(2016, 9, 15, 12, 30),␣
↪duration=30, session="C", capacity=200),
...                Slot(venue='Big', starts_at=datetime(2016, 9, 15, 13, 0),␣
↪duration=30, session="C", capacity=200),
...                Slot(venue='Small', starts_at=datetime(2016, 9, 15, 12, 30),␣
↪duration=30, session="D", capacity=50),
...                Slot(venue='Small', starts_at=datetime(2016, 9, 15, 13, 0),␣
↪duration=30, session="D", capacity=50),
...                Slot(venue='Big', starts_at=datetime(2016, 9, 16, 9, 30),␣
↪duration=30, session="E", capacity=50),
...                Slot(venue='Big', starts_at=datetime(2016, 9, 16, 10, 00),␣
↪duration=30, session="E", capacity=50),
...                Slot(venue='Big', starts_at=datetime(2016, 9, 16, 12, 30),␣
↪duration=30, session="F", capacity=50),
...                Slot(venue='Big', starts_at=datetime(2016, 9, 16, 13, 0),␣
↪duration=30, session="F", capacity=50)]
>>> workshop_slots = [Slot(venue='Small', starts_at=datetime(2016, 9, 16, 9, 30),␣
↪duration=60, session="G", capacity=50),
...                   Slot(venue='Small', starts_at=datetime(2016, 9, 16, 13, 0),␣
↪duration=60, session="H", capacity=50)]
>>> outside_slots = [Slot(venue='Outside', starts_at=datetime(2016, 9, 16, 12, 30),␣
↪duration=90, session="I", capacity=1000),
...                  Slot(venue='Outside', starts_at=datetime(2016, 9, 16, 13, 0),␣
↪duration=90, session="J", capacity=1000)]
>>> slots = talk_slots + workshop_slots + outside_slots
```

Note that the `duration` must be given in minutes.

We also have a number of talks and workshops to schedule, because of the duration/location of the slots we know some of them are unavailable for a given slot:

```
>>> events = [Event(name='Talk 1', duration=30, tags=['beginner'],␣
↪unavailability=outside_slots[:], demand=50),
...           Event(name='Talk 2', duration=30, tags=['beginner'],␣
↪unavailability=outside_slots[:], demand=130),
...           Event(name='Talk 3', duration=30, tags=['beginner'],␣
↪unavailability=outside_slots[:], demand=200),
...           Event(name='Talk 4', duration=30, tags=['beginner'],␣
↪unavailability=outside_slots[:], demand=30),
...           Event(name='Talk 5', duration=30, tags=['intermediate'],␣
↪unavailability=outside_slots[:], demand=60),
...           Event(name='Talk 6', duration=30, tags=['intermediate'],␣
↪unavailability=outside_slots[:], demand=30),
...           Event(name='Talk 7', duration=30, tags=['intermediate', 'advanced'],␣
↪unavailability=outside_slots[:], demand=60),
...           Event(name='Talk 8', duration=30, tags=['intermediate', 'advanced'],␣
↪unavailability=outside_slots[:], demand=60),
```

```
...             Event(name='Talk 9', duration=30, tags=['advanced'],␣
↪unavailability=outside_slots[:], demand=60),
...             Event(name='Talk 10', duration=30, tags=['advanced'],␣
↪unavailability=outside_slots[:], demand=30),
...             Event(name='Talk 11', duration=30, tags=['advanced'],␣
↪unavailability=outside_slots[:], demand=30),
...             Event(name='Talk 12', duration=30, tags=['advanced'],␣
↪unavailability=outside_slots[:], demand=30),
...             Event(name='Workshop 1', duration=60, tags=['testing'],␣
↪unavailability=outside_slots[:], demand=40),
...             Event(name='Workshop 2', duration=60, tags=['testing'],␣
↪unavailability=outside_slots[:], demand=40),
...             Event(name='City tour', duration=90, tags=[], unavailability=talk_
↪slots[:] + workshop_slots[:], demand=100),
...             Event(name='Boardgames', duration=90, tags=[], unavailability=talk_
↪slots[:] + workshop_slots[:], demand=20)]
```

Further to this we have a couple of other constraints:

- The speaker for `Talk 1` is also the person delivering `Workshop 1`:

```
>>> events[0].add_unavailability(events[6])
```

- Also, the person running `Workshop 2` is the person hosting the `Boardgames`:

```
>>> events[13].add_unavailability(events[-1])
```

Note that we haven't indicated the workshops cannot happen in the talk slots but this will automatically be taken care of because of the duration of the workshops (60mins) and the duration of the talk slots (30mins).

Creating a schedule

Now that we have `slots` and `events` we can schedule our event:

```
>>> from conference_scheduler import scheduler
>>> schedule = scheduler.schedule(events, slots)

>>> schedule.sort(key=lambda item: item.slot.starts_at)
>>> for item in schedule:
...     print(f"{item.event.name} at {item.slot.starts_at} in {item.slot.venue}")
Talk 5 at 2016-09-15 09:30:00 in Small
Talk 11 at 2016-09-15 09:30:00 in Big
Talk 4 at 2016-09-15 10:00:00 in Small
Talk 10 at 2016-09-15 10:00:00 in Big
Talk 1 at 2016-09-15 12:30:00 in Small
Talk 6 at 2016-09-15 12:30:00 in Big
Talk 3 at 2016-09-15 13:00:00 in Small
Talk 8 at 2016-09-15 13:00:00 in Big
Talk 2 at 2016-09-16 09:30:00 in Big
Workshop 2 at 2016-09-16 09:30:00 in Small
Talk 9 at 2016-09-16 10:00:00 in Big
Talk 12 at 2016-09-16 12:30:00 in Big
Boardgames at 2016-09-16 12:30:00 in Outside
Talk 7 at 2016-09-16 13:00:00 in Big
Workshop 1 at 2016-09-16 13:00:00 in Small
City tour at 2016-09-16 13:00:00 in Outside
```

We see that all the events are scheduled in appropriate rooms (as indicated by the unavailability attribute for the events). Also we have that `Talk 1` doesn't clash with `Workshop 1`. Similarly, the `Boardgame` does not clash with `Workshop 2`.

You will also note that no two events with the same tags are on at the same time. Tags allow for a quick way to batch define unavailability.

Avoiding room overcrowding

The data we input in to the model included information about demand for a talk; this could be approximated from previous popularity for a talk. However, the scheduler has put `Talk 3` (which have high demand) in the small room (which has capacity 50). We can include an objective function in our scheduler to minimise the difference between room capacity and demand:

```
>>> from conference_scheduler.lp_problem import objective_functions
>>> func = objective_functions.efficiency_capacity_demand_difference
>>> schedule = scheduler.schedule(events, slots, objective_function=func)

>>> schedule.sort(key=lambda item: item.slot.starts_at)
>>> for item in schedule:
...     print(f"{item.event.name} at {item.slot.starts_at} in {item.slot.venue}")
Talk 4 at 2016-09-15 09:30:00 in Big
Talk 5 at 2016-09-15 09:30:00 in Small
Talk 3 at 2016-09-15 10:00:00 in Big
Talk 9 at 2016-09-15 10:00:00 in Small
Talk 6 at 2016-09-15 12:30:00 in Big
Talk 11 at 2016-09-15 12:30:00 in Small
Talk 2 at 2016-09-15 13:00:00 in Small
Talk 7 at 2016-09-15 13:00:00 in Big
Talk 8 at 2016-09-16 09:30:00 in Big
Workshop 2 at 2016-09-16 09:30:00 in Small
Talk 12 at 2016-09-16 10:00:00 in Big
Talk 1 at 2016-09-16 12:30:00 in Big
Boardgames at 2016-09-16 12:30:00 in Outside
Talk 10 at 2016-09-16 13:00:00 in Big
Workshop 1 at 2016-09-16 13:00:00 in Small
City tour at 2016-09-16 13:00:00 in Outside
```

We see that `Talk 3` has moved to the bigger room but that all other constraints still hold. Note however that this has also moved `Talk 2` (which has relatively high demand) to a small room. This is because we have minimised the overall overcrowding. This can have the negative effect of leaving one slot with a high overcrowding for the benefit of overall efficiency. We can however include a different objective function to minimise the maximum overcrowding in any given slot:

```
>>> from conference_scheduler.lp_problem import objective_functions
>>> func = objective_functions.equity_capacity_demand_difference
>>> schedule = scheduler.schedule(events, slots, objective_function=func)

>>> schedule.sort(key=lambda item: item.slot.starts_at)
>>> for item in schedule:
...     print(f"{item.event.name} at {item.slot.starts_at} in {item.slot.venue}")
Talk 1 at 2016-09-15 09:30:00 in Small
Talk 9 at 2016-09-15 09:30:00 in Big
Talk 3 at 2016-09-15 10:00:00 in Big
Talk 10 at 2016-09-15 10:00:00 in Small
Talk 4 at 2016-09-15 12:30:00 in Small
```

```
Talk 7 at 2016-09-15 12:30:00 in Big
Talk 2 at 2016-09-15 13:00:00 in Big
Talk 8 at 2016-09-15 13:00:00 in Small
Talk 6 at 2016-09-16 09:30:00 in Big
Workshop 2 at 2016-09-16 09:30:00 in Small
Talk 12 at 2016-09-16 10:00:00 in Big
Talk 11 at 2016-09-16 12:30:00 in Big
Boardgames at 2016-09-16 12:30:00 in Outside
Talk 5 at 2016-09-16 13:00:00 in Big
Workshop 1 at 2016-09-16 13:00:00 in Small
City tour at 2016-09-16 13:00:00 in Outside
```

Now, both `Talk 2` and `Talk 3` are in the bigger rooms.

Coping with new information

This is fantastic! Our schedule has now been published and everyone is excited about the conference. However, as can often happen, one of the speakers now informs us of a particular new constraints. For example, the speaker for `Talk 11` is unable to speak on the second day.

We can enter this new constraint:

```
>>> events[10].add_unavailability(*slots[9:])
```

We can now solve the problem one more time from scratch just as before:

```
>>> alt_schedule = scheduler.schedule(events, slots, objective_function=func)

>>> alt_schedule.sort(key=lambda item: item.slot.starts_at)
>>> for item in alt_schedule:
...     print(f"{item.event.name} at {item.slot.starts_at} in {item.slot.venue}")
Talk 3 at 2016-09-15 09:30:00 in Big
Talk 12 at 2016-09-15 09:30:00 in Small
Talk 2 at 2016-09-15 10:00:00 in Big
Talk 10 at 2016-09-15 10:00:00 in Small
Talk 1 at 2016-09-15 12:30:00 in Big
Talk 8 at 2016-09-15 12:30:00 in Small
Talk 5 at 2016-09-15 13:00:00 in Big
Talk 9 at 2016-09-15 13:00:00 in Small
Talk 11 at 2016-09-16 09:30:00 in Big
Workshop 2 at 2016-09-16 09:30:00 in Small
Talk 4 at 2016-09-16 10:00:00 in Big
Talk 7 at 2016-09-16 12:30:00 in Big
Boardgames at 2016-09-16 12:30:00 in Outside
Talk 6 at 2016-09-16 13:00:00 in Big
Workshop 1 at 2016-09-16 13:00:00 in Small
City tour at 2016-09-16 13:00:00 in Outside
```

This has resulted in a completely different schedule with a number of changes. We can however solve the problem with a new objective function which is to minimise the changes from the old schedule:

```
>>> func = objective_functions.number_of_changes
>>> similar_schedule = scheduler.schedule(events, slots, objective_function=func,␣
↪original_schedule=schedule)

>>> similar_schedule.sort(key=lambda item: item.slot.starts_at)
```

```
>>> for item in similar_schedule:
...     print(f"{item.event.name} at {item.slot.starts_at} in {item.slot.venue}")
Talk 1 at 2016-09-15 09:30:00 in Small
Talk 9 at 2016-09-15 09:30:00 in Big
Talk 3 at 2016-09-15 10:00:00 in Big
Talk 10 at 2016-09-15 10:00:00 in Small
Talk 4 at 2016-09-15 12:30:00 in Small
Talk 7 at 2016-09-15 12:30:00 in Big
Talk 2 at 2016-09-15 13:00:00 in Big
Talk 8 at 2016-09-15 13:00:00 in Small
Talk 11 at 2016-09-16 09:30:00 in Big
Workshop 2 at 2016-09-16 09:30:00 in Small
Talk 12 at 2016-09-16 10:00:00 in Big
Talk 6 at 2016-09-16 12:30:00 in Big
Boardgames at 2016-09-16 12:30:00 in Outside
Talk 5 at 2016-09-16 13:00:00 in Big
Workshop 1 at 2016-09-16 13:00:00 in Small
City tour at 2016-09-16 13:00:00 in Outside
```

Spotting the Changes

It can be a little difficult to spot what has changed when we compute a new schedule and so there are two functions which can help. Let's take our `alt_schedule` and compare it with the original. Firstly, we can see which events moved to different slots:

```
>>> event_diff = scheduler.event_schedule_difference(schedule, alt_schedule)
>>> for item in event_diff:
...     print(f"{item.event.name} has moved from {item.old_slot.venue} at {item.old_
↪slot.starts_at} to {item.new_slot.venue} at {item.new_slot.starts_at}")
Talk 1 has moved from Small at 2016-09-15 09:30:00 to Big at 2016-09-15 12:30:00
Talk 11 has moved from Big at 2016-09-16 12:30:00 to Big at 2016-09-16 09:30:00
Talk 12 has moved from Big at 2016-09-16 10:00:00 to Small at 2016-09-15 09:30:00
Talk 2 has moved from Big at 2016-09-15 13:00:00 to Big at 2016-09-15 10:00:00
Talk 3 has moved from Big at 2016-09-15 10:00:00 to Big at 2016-09-15 09:30:00
Talk 4 has moved from Small at 2016-09-15 12:30:00 to Big at 2016-09-16 10:00:00
Talk 5 has moved from Big at 2016-09-16 13:00:00 to Big at 2016-09-15 13:00:00
Talk 6 has moved from Big at 2016-09-16 09:30:00 to Big at 2016-09-16 13:00:00
Talk 7 has moved from Big at 2016-09-15 12:30:00 to Big at 2016-09-16 12:30:00
Talk 8 has moved from Small at 2016-09-15 13:00:00 to Small at 2016-09-15 12:30:00
Talk 9 has moved from Big at 2016-09-15 09:30:00 to Small at 2016-09-15 13:00:00
```

We can also look at slots to see which now have a different event scheduled:

```
>>> slot_diff = scheduler.slot_schedule_difference(schedule, alt_schedule)
>>> for item in slot_diff:
...     print(f"{item.slot.venue} at {item.slot.starts_at} will now host {item.new_
↪event.name} rather than {item.old_event.name}" )
Big at 2016-09-15 09:30:00 will now host Talk 3 rather than Talk 9
Big at 2016-09-15 10:00:00 will now host Talk 2 rather than Talk 3
Big at 2016-09-15 12:30:00 will now host Talk 1 rather than Talk 7
Big at 2016-09-15 13:00:00 will now host Talk 5 rather than Talk 2
Big at 2016-09-16 09:30:00 will now host Talk 11 rather than Talk 6
Big at 2016-09-16 10:00:00 will now host Talk 4 rather than Talk 12
Big at 2016-09-16 12:30:00 will now host Talk 7 rather than Talk 11
Big at 2016-09-16 13:00:00 will now host Talk 6 rather than Talk 5
```

```
Small at 2016-09-15 09:30:00 will now host Talk 12 rather than Talk 1
Small at 2016-09-15 12:30:00 will now host Talk 8 rather than Talk 4
Small at 2016-09-15 13:00:00 will now host Talk 9 rather than Talk 8
```

We can use this facility to show how using `number_of_changes` as our objective function resulted in far fewer changes:

```
>>> event_diff = scheduler.event_schedule_difference(schedule, similar_schedule)
>>> for item in event_diff:
...     print(f"{item.event.name} has moved from {item.old_slot.venue} at {item.old_
↪slot.starts_at} to {item.new_slot.venue} at {item.new_slot.starts_at}")
Talk 11 has moved from Big at 2016-09-16 12:30:00 to Big at 2016-09-16 09:30:00
Talk 6 has moved from Big at 2016-09-16 09:30:00 to Big at 2016-09-16 12:30:00
```

Scheduling chairs

Once we have a schedule for our talks, workshops and social events, we have the last task which is to schedule chairs for the talk sessions.

We have 6 different sessions of talks to chair:

```
Talk 4 at 2016-09-15 09:30:00 in Big
Talk 1 at 2016-09-15 10:00:00 in Big

Talk 7 at 2016-09-15 09:30:00 in Small
Talk 6 at 2016-09-15 10:00:00 in Small

Talk 8 at 2016-09-15 12:30:00 in Big
Talk 5 at 2016-09-15 13:00:00 in Big

Talk 11 at 2016-09-15 12:30:00 in Small
Talk 10 at 2016-09-15 13:00:00 in Small

Talk 3 at 2016-09-16 09:30:00 in Big
Talk 2 at 2016-09-16 10:00:00 in Big

Talk 12 at 2016-09-16 12:30:00 in Big
Talk 9 at 2016-09-16 13:00:00 in Big
```

We will use the conference scheduler, with these sessions corresponding to slots:

```
>>> chair_slots  = [Slot(venue='Big', starts_at=datetime(2016, 9, 15, 9, 30),␣
↪duration=60, session="A", capacity=200),
...                 Slot(venue='Small', starts_at=datetime(2016, 9, 15, 9, 30),␣
↪duration=60, session="B", capacity=50),
...                 Slot(venue='Big', starts_at=datetime(2016, 9, 15, 12, 30),␣
↪duration=60, session="C", capacity=200),
...                 Slot(venue='Small', starts_at=datetime(2016, 9, 15, 12, 30),␣
↪duration=60, session="D", capacity=50),
...                 Slot(venue='Big', starts_at=datetime(2016, 9, 16, 12, 30),␣
↪duration=60, session="E", capacity=200),
...                 Slot(venue='Small', starts_at=datetime(2016, 9, 16, 12, 30),␣
↪duration=60, session="F", capacity=50)]
```

We will need 6 chairpersons for these slots and we will use events as chairs. In practice, all chairing will be taken care of by 3 people, with each person chairing 2 sessions:

```
>>> events = [Event(name='Chair A-1', duration=60, demand=0),
...           Event(name='Chair A-2', duration=60, demand=0),
...           Event(name='Chair B-1', duration=60, demand=0),
...           Event(name='Chair B-2', duration=60, demand=0),
...           Event(name='Chair C-1', duration=60, demand=0),
...           Event(name='Chair D-2', duration=60, demand=0)]
```

As you can see, we have set all unavailabilities to be empty however `Chair A` is in fact the speaker for `Talk 11`. Also `Chair B` has informed us that they are not present on the first day. We can include these constraints:

```
>>> events[0].add_unavailability(chair_slots[4])
>>> events[1].add_unavailability(chair_slots[4])
>>> events[2].add_unavailability(*chair_slots[4:])
>>> events[3].add_unavailability(*chair_slots[4:])
```

Finally, each chair cannot chair more than one session at a time:

```
>>> events[0].add_unavailability(events[1])
>>> events[2].add_unavailability(events[3])
>>> events[4].add_unavailability(events[5])
```

Now let us get the chair schedule:

```
>>> chair_schedule = scheduler.schedule(events, chair_slots)

>>> chair_schedule.sort(key=lambda item: item.slot.starts_at)
>>> for item in chair_schedule:
...     print(f"{item.event.name} chairing {item.slot.starts_at} in {item.slot.venue}
↪")
Chair A-2 chairing 2016-09-15 09:30:00 in Big
Chair B-1 chairing 2016-09-15 09:30:00 in Small
Chair B-2 chairing 2016-09-15 12:30:00 in Small
Chair C-1 chairing 2016-09-15 12:30:00 in Big
Chair A-1 chairing 2016-09-16 12:30:00 in Small
Chair D-2 chairing 2016-09-16 12:30:00 in Big
```

Validating a schedule

It might of course be helpful to use the tool simply to check if a given schedule is correct: perhaps someone makes a manual change and it is desirable to verify that this is still a valid schedule. Let us first check that our schedule obtained from the algorithm is correct:

```
>>> from conference_scheduler.validator import is_valid_schedule, schedule_violations
>>> is_valid_schedule(chair_schedule, events=events, slots=chair_slots)
True
```

Let us modify our schedule so that it schedules an event twice:

```
>>> from conference_scheduler.resources import ScheduledItem
>>> chair_schedule[0] = ScheduledItem(event=events[2], slot=chair_slots[0])
>>> for item in chair_schedule[:2]:
...     print(f"{item.event.name} chairing {item.slot.starts_at} in {item.slot.venue}
↪")
Chair B-1 chairing 2016-09-15 09:30:00 in Big
Chair B-1 chairing 2016-09-15 09:30:00 in Small
```

We now see that we have an invalid schedule:

```
>>> is_valid_schedule(chair_schedule, events=events, slots=chair_slots)
False
```

We can furthermore identify which constraints were broken:

```
>>> for v in schedule_violations(chair_schedule, events=events, slots=chair_slots):
...     print(v)
Event either not scheduled or scheduled multiple times - event: 1
Event either not scheduled or scheduled multiple times - event: 2
```

CHAPTER 2

How To...

Define a conference

Using PyCon UK 2016 for example data, let us consider how we might define a conference and pass the necessary data to the scheduler.

The aim here is to show how we might use simple data structures and formats to hold the information we require and then parse and process those structures into the necessary form for the scheduler.

We'll use a variety of simple Python data types, YAML and JSON documents and we'll include both inline data and external files. This is simply by way of example and it's likely that a real application would standardise on far fewer of those options than are shown here.

Data Structures

Slots

In 2016, there were three types of event: talks, workshops and plenary sessions and these were held in five rooms at Cardiff City Hall. Not all the rooms were suitable for all types of event.

We can capture this information using a Python list and dictionary:

```
>>> event_types = ['talk', 'workshop', 'plenary']

>>> venues = {
...     'Assembly Room': {'capacity': 500, 'suitable_for': ['talk', 'plenary']},
...     'Room A': {'capacity': 80, 'suitable_for': ['workshop']},
...     'Ferrier Hall': {'capacity': 200, 'suitable_for': ['talk']},
...     'Room C': {'capacity': 80, 'suitable_for': ['talk', 'workshop']},
...     'Room D': {'capacity': 100, 'suitable_for': ['talk']}}
```

The events took place over the course of five days in September but, for this example, we are not considering the first day of the conference (Thursday) or the last (Monday). There were talks given on all three of those days (Friday to Sunday) but the workshops only took place on the Sunday.

Here is how we might represent this information using JSON:

```
>>> import json

>>> json_days = """{
...     "16-Sep-2016": {"event_types": ["talk", "plenary"]},
...     "17-Sep-2016": {"event_types": ["talk", "plenary"]},
...     "18-Sep-2016": {"event_types": ["talk", "plenary", "workshop"]}}"""
```

The time periods available for the three event types were not the same and they were also grouped for talks but not for the other two.

This time using YAML, here is how we might represent that information:

```
>>> yaml_session_times = """
...     talk:
...         morning:
...         -
...             starts_at: 10:15:00
...             duration: 30
...         -
...             starts_at: 11:15:00
...             duration: 45
...         -
...             starts_at: 12:00:00
...             duration: 30
...         afternoon:
...         -
...             starts_at: 12:30:00
...             duration: 30
...         -
...             starts_at: 14:30:00
...             duration: 30
...         -
...             starts_at: 15:00:00
...             duration: 30
...         -
...             starts_at: 15:30:00
...             duration: 30
...         evening:
...         -
...             starts_at: 16:30:00
...             duration: 30
...         -
...             starts_at: 17:00:00
...             duration: 30
...     workshop:
...         None:
...         -
...             starts_at: 10:15:00
...             duration: 90
...         -
...             starts_at: 11:15:00
...             duration: 105
...         -
...             starts_at: 14:30:00
...             duration: 90
...         -
...             starts_at: 16:30:00
```

```
...             duration: 60
...     plenary:
...         None:
...         -
...             starts_at: 9:10:00
...             duration: 50"""
```

Events

For the events which need to be scheduled, we have the talks that were accepted for PyCon UK 2016 in a `YAML file`

Unavailability

When organising any conference, it is common that speakers might be unavailable to attend on certain days or for certain time periods.

Although we don't have the real information from PyCon UK 2016, for this example, we can create some fictitious data. Let's say that Alex Chan was not available on either the Friday or the Sunday morning:

```
>>> yaml_speaker_unavailability =  """
...     Alex Chan:
...     -   unavailable_from: 2016-09-16 00:00:00
...         unavailable_until: 2016-09-16 23:59:59
...     -   unavailable_from: 2016-09-18 00:00:00
...         unavailable_until: 2016-09-18 12:00:00"""
```

It's also common for speakers to request that their talk not be scheduled at the same time as somebody else's that they would like to attend. For example, in 2016, Owen Campbell asked that his talk not be scheduled opposite his son's, Thomas Campbell. Let's also say that Owen wanted to attend David R. MacIver's talk 'Easy solutions to hard problems' (because he wanted to be inspired to create this library):

```
>>> yaml_speaker_clashes = """
...     Owen Campbell:
...         - Thomas Campbell
...         - David R. MacIver"""
```

Loading into Python

Since we used a Python list and dictionary for the event types and venues, those are already available to us.

Next, we need to load the JSON and YAML data so that it too becomes available as lists and dictionaries. First, let's load the JSON document which holds the 'days' information. We'll include a function to convert the strings representing the dates into Python datetime objects:

```
>>> import json
>>> from datetime import datetime
>>> from pprint import pprint

>>> def date_decoder(day):
...     for key in day.keys():
...         try:
...             new_key = datetime.strptime(key, '%d-%b-%Y')
...             day[new_key] = day[key]
...             del day[key]
```

```
...         except:
...             pass
...     return day
>>>
>>> days = json.loads(json_days, object_hook=date_decoder)

>>> pprint(days)
{datetime.datetime(2016, 9, 16, 0, 0): {'event_types': ['talk', 'plenary']},
 datetime.datetime(2016, 9, 17, 0, 0): {'event_types': ['talk', 'plenary']},
 datetime.datetime(2016, 9, 18, 0, 0): {'event_types': ['talk',
                                                        'plenary',
                                                        'workshop']}}
```

We can load the YAML document containing the ‘session times’ information in a similar fashion. Again, the data is loaded into a Python dictionary with each event type as a key mapping to a further dictionary with the session name as key and a list of slot times as its values. The start times are converted to an integer representing the number of seconds since midnight:

```
>>> import yaml

>>> session_times = yaml.load(yaml_session_times)

>>> pprint(session_times['workshop'])
{'None': [{'duration': 90, 'starts_at': 36900},
          {'duration': 105, 'starts_at': 40500},
          {'duration': 90, 'starts_at': 52200},
          {'duration': 60, 'starts_at': 59400}]}
```

And also the file containing the talks:

```
>>> with open('docs/howto/pyconuk-2016-talks.yml', 'r') as file:
...     talks = yaml.load(file)

>>> pprint(talks[0:3])
[{'duration': 30,
  'speaker': 'Kevin Keenoy',
  'title': 'Transforming the government’s Digital Marketplace from portal to '
           'platform'},
 {'duration': 45,
  'speaker': 'Tom Christie',
  'title': 'Django REST framework: Schemas, Hypermedia & Client libraries.'},
 {'duration': 30,
  'speaker': 'Iacopo Spalletti',
  'title': 'django CMS in the real time web: how to mix CMS, websockets, REST '
           'for a fully real time experience'}]
```

Finally, the unavailability and clashes:

```
>>> speaker_unavailability = yaml.load(yaml_speaker_unavailability)

>>> pprint(speaker_unavailability)
{'Alex Chan': [{'unavailable_from': datetime.datetime(2016, 9, 16, 0, 0),
                'unavailable_until': datetime.datetime(2016, 9, 16, 23, 59, 59)},
               {'unavailable_from': datetime.datetime(2016, 9, 18, 0, 0),
                'unavailable_until': datetime.datetime(2016, 9, 18, 12, 0)}]}


>>> speaker_clashes = yaml.load(yaml_speaker_clashes)
```

```
>>> pprint(speaker_clashes)
{'Owen Campbell': ['Thomas Campbell', 'David R. MacIver']}
```

Processing

Before we can compute a schedule for our conference, we need to create the `Event` and `Slot` objects required by the scheduler.

Slots

The nested structure we have used to define our session times is convenient and readable, but it's not the structure required by the scheduler. Instead, we need to flatten it so that we have the start time, duration and session name at the same level. We'll create a dictionary of these with the event type as a key as we'll need each associated list separately later on:

```
>>> slot_times = {
...     event_type: [{
...         'starts_at': slot_time['starts_at'],
...         'duration': slot_time['duration'],
...         'session_name': session_name}
...         for session_name, slot_times in session_times[event_type].items()
...         for slot_time in slot_times]
...     for event_type in event_types}

>>> pprint(slot_times['workshop'])
[{'duration': 90, 'session_name': 'None', 'starts_at': 36900},
 {'duration': 105, 'session_name': 'None', 'starts_at': 40500},
 {'duration': 90, 'session_name': 'None', 'starts_at': 52200},
 {'duration': 60, 'session_name': 'None', 'starts_at': 59400}]
```

Now, we can use that flattened structure to create instances of `conference_scheduler.resources.Slot`. A `Slot` instance represents a time and a place into which an event can be scheduled. We'll combine the `slot_times` dictionary with the `days` list and the `venues` dictionary to give us all the possible combinations.

Again, we'll create a dictionary of those with the event type as key because we'll need each list of `Slots` separately later on:

```
>>> import itertools as it
>>> from datetime import timedelta
>>> from conference_scheduler.resources import Slot

>>> slots = {
...     event_type: [
...         Slot(
...             venue=venue,
...             starts_at=(
...                    day + timedelta(0, slot_time['starts_at'])
...             ).strftime('%d-%b-%Y %H:%M'),
...             duration=slot_time['duration'],
...             session=f"{day.date()} {slot_time['session_name']}",
...             capacity=venues[venue]['capacity'])
...         for venue, day, slot_time in it.product(
...             venues, days, slot_times[event_type])
...         if (event_type in venues[venue]['suitable_for'] and
```

```
...             event_type in days[day]['event_types'])]
...     for event_type in event_types}

>>> pprint(slots['talk'][0:5])
[Slot(venue='Assembly Room', starts_at='16-Sep-2016 10:15', duration=30, capacity=500,
↪ session='2016-09-16 morning'),
 Slot(venue='Assembly Room', starts_at='16-Sep-2016 11:15', duration=45, capacity=500,
↪ session='2016-09-16 morning'),
 Slot(venue='Assembly Room', starts_at='16-Sep-2016 12:00', duration=30, capacity=500,
↪ session='2016-09-16 morning'),
 Slot(venue='Assembly Room', starts_at='16-Sep-2016 12:30', duration=30, capacity=500,
↪ session='2016-09-16 afternoon'),
 Slot(venue='Assembly Room', starts_at='16-Sep-2016 14:30', duration=30, capacity=500,
↪ session='2016-09-16 afternoon')]
```

Events

We can now create instances of `conference_scheduler.resources.Event` using the talks dictionary we loaded from the YAML file. Once again, we'll create a dictionary with the event type as the keys:

```
>>> from conference_scheduler.resources import Event
>>>
>>> events = {'talk': [
...     Event(
...         talk['title'], talk['duration'], demand=None,
...         tags=talk.get('tags', None))
...     for talk in talks]}

>>> pprint(events['talk'][0:3])
[Event(name='Transforming the government's Digital Marketplace from portal to platform
↪', duration=30, demand=None, tags=(), unavailability=()),
 Event(name='Django REST framework: Schemas, Hypermedia & Client libraries.',␣
↪duration=45, demand=None, tags=(), unavailability=()),
 Event(name='django CMS in the real time web: how to mix CMS, websockets, REST for a␣
↪fully real time experience', duration=30, demand=None, tags=(), unavailability=())]
```

We then need to add our unavailability information to those Event objects. It's currently in a form based on the speaker so that it's easy to maintain but we need it based on the talks those speakers will give. We'll create a dictionary with the talk index as its key and a list of slots in which it must not be scheduled. (This will give us a dictionary with the index of Alex Chan's talk as the key mapping to a list of all slots on Friday and Sunday morning):

```
>>> talk_unavailability = {talks.index(talk): [
...      slots['talk'].index(slot)
...      for period in periods
...      for slot in slots['talk']
...      if period['unavailable_from'] <= datetime.strptime(slot.starts_at, '%d-%b-%Y
↪%H:%M') and
...      period['unavailable_until'] >= datetime.strptime(slot.starts_at, '%d-%b-%Y
↪%H:%M') + timedelta(0, slot.duration * 60)]
... for speaker, periods in speaker_unavailability.items()
... for talk in talks if talk['speaker'] == speaker}

>>> pprint(talk_unavailability[55][0:3])
[0, 1, 2]
```

And then add those entries to our `events` dictionary:

```
>>> for talk, unavailable_slots in talk_unavailability.items():
...     events['talk'][talk].add_unavailability(*[slots['talk'][s] for s in␣
↪unavailable_slots])

>>> pprint(events['talk'][55].unavailability[0:3])
(Slot(venue='Assembly Room', starts_at='16-Sep-2016 10:15', duration=30, capacity=500,
↪ session='2016-09-16 morning'),
 Slot(venue='Assembly Room', starts_at='16-Sep-2016 11:15', duration=45, capacity=500,
↪ session='2016-09-16 morning'),
 Slot(venue='Assembly Room', starts_at='16-Sep-2016 12:00', duration=30, capacity=500,
↪ session='2016-09-16 morning'))
```

To complete our Event objects, we'll need to add the information from our `speaker_clashes` dictionary to their unavailability. First, let's map the speaker entries in that dictionary to the relevant talks:

```
>>> talk_clashes = {talks.index(talk): [
...     talks.index(t) for s in clashing_speakers
...     for t in talks if t['speaker'] == s]
... for speaker, clashing_speakers in speaker_clashes.items()
... for talk in talks if talk['speaker'] == speaker}

>>> pprint(talk_clashes)
{50: [19, 63]}
```

And now we can add those entries to our `events` dictionary:

```
>>> for talk, clashing_talks in talk_clashes.items():
...     events['talk'][talk].add_unavailability(*[events['talk'][t] for t in clashing_
↪talks])

>>> pprint(events['talk'][50])
Event(name='Ancient Greek Philosophy, Medieval Mental Models and 21st Century␣
↪Technology', duration=30, demand=None, tags=(), unavailability=(Event(name='Using␣
↪Python for National Cipher Challenge', duration=30, demand=None, tags=(),␣
↪unavailability=()), Event(name='Easy solutions to hard problems', duration=30,␣
↪demand=None, tags=(), unavailability=())))
```

Modify event tags and unavailabiity

As shown in *Tutorial* it is possible to add to the `unavailability` of an `Event` by using the `Event.add_unavailability` method:

```
>>> from conference_scheduler.resources import Slot, Event
>>> from pprint import pprint

>>> slots  = [Slot(venue='Big', starts_at='15-Sep-2016 09:30', duration=30, session="A
↪", capacity=200),
...           Slot(venue='Big', starts_at='15-Sep-2016 10:00', duration=30, session="A
↪", capacity=200)]
>>> events = [Event(name='Talk 1', duration=30, demand=50),
...           Event(name='Talk 2', duration=30, demand=130)]
```

Let us note the first event as unavailable for all the slots. We do this by unpacking the `slots` list (`*slots`) and passing it to the `add_unavailability` method:

```
>>> events[0].add_unavailability(*slots)
>>> pprint(events[0].unavailability)
(Slot(venue='Big', starts_at='15-Sep-2016 09:30', duration=30, capacity=200, session=
↪'A'),
 Slot(venue='Big', starts_at='15-Sep-2016 10:00', duration=30, capacity=200, session=
↪'A'))
```

We can remove a specific item:

```
>>> events[0].remove_unavailability(slots[0])
>>> events[0].unavailability
(Slot(venue='Big', starts_at='15-Sep-2016 10:00', duration=30, capacity=200, session=
↪'A'),)
```

We can also completely clear the unavailability:

```
>>> events[0].clear_unavailability()
>>> events[0].unavailability
()
```

Similar methods exist for modifying event `tags`:

```
>>> events[0].add_tags('Python', 'Ruby', 'Javascript')
>>> events[0].tags
('Python', 'Ruby', 'Javascript')
>>> events[0].remove_tag("Python")
>>> events[0].tags
('Ruby', 'Javascript')
>>> events[0].clear_tags()
>>> events[0].tags
()
```

Use different solvers

The Pulp library is a Python interface to underlying solution engines. By default it comes packaged with CBC

It is however possible to use a number of other solvers (see the Pulp documentation for details) and these can be passed to the scheduler. For example here is how we would use the GLPK solver for a given set of `events` and `slots`:

```
>>> scheduler.schedule(events=events, slots=slots, solver=pulp.GLPK())
```

Different solvers can have major impact on the performance of the scheduler. This can be an important consideration when scheduling large or highly constrained problems.

Use heuristics

The linear programming approach is guaranteed to give an optimal solution, however this can be quite expensive computationally. It is possible to use a heuristic approach that probabilistically searches through the solution space.

Here is how to do this with a hill climbing algorithm:

```
>>> import conference_scheduler.heuristics as heu
>>> heuristic = heu.hill_climber
>>> scheduler.heuristic(events=events, slots=slots, algorithm=heuristic)
```

A simulated annealing algorithm is also implemented:

```
>>> heuristic = heu.simulated_annealing
>>> scheduler.heuristic(events=events, slots=slots, algorithm=heuristic)
```

Obtain the mathematical representation of a schedule

When scheduling a conference, it might be desirable to recover the schedule in a different format. Let us schedule a simple conference as described in *Tutorial*:

```
>>> from datetime import datetime
>>> from conference_scheduler.resources import Slot, Event
>>> from conference_scheduler import scheduler, converter

>>> slots  = [Slot(venue='Big', starts_at=datetime(2016, 9, 15, 9, 30), duration=30,␣
↪session="A", capacity=200),
...           Slot(venue='Big', starts_at=datetime(2016, 9, 15, 10, 0), duration=30,␣
↪session="A", capacity=200)]
>>> events = [Event(name='Talk 1', duration=30, demand=50),
...           Event(name='Talk 2', duration=30, demand=130)]

>>> schedule = scheduler.schedule(events, slots)
```

We can view this schedule as before:

```
>>> for item in schedule:
...     print(f"{item.event.name} at {item.slot.starts_at} in {item.slot.venue}")
Talk 1 at 2016-09-15 09:30:00 in Big
Talk 2 at 2016-09-15 10:00:00 in Big
```

If we want to recover the mathematical array form of our solution (as described in *Mathematical model*), we use the `scheduler.schedule_to_array` function:

```
>>> array = converter.schedule_to_array(schedule, events=events, slots=slots)
>>> array
array([[1, 0],
       [0, 1]], dtype=int8)
```

We can also return from a mathematical array to the schedule using the `scheduler.array_to_schedule` function:

```
>>> for item in converter.array_to_schedule(array, events=events, slots=slots):
...     print(f"{item.event.name} at {item.slot.starts_at} in {item.slot.venue}")
Talk 1 at 2016-09-15 09:30:00 in Big
Talk 2 at 2016-09-15 10:00:00 in Big
```

CHAPTER 3

Background

Mathematical model

The scheduler works by solving a well understood mathematical problem called an integer linear program *[Dantzig1963]*, *[Schaerf1999]*.

If we assume that we have M events and N slots, then the schedule is mathematically represented by a binary matrix $X \in \{0, 1\}^{M \times N}$. Every row corresponds to an event and every column to a slot:

$$X_{ij} = \begin{cases} 1, & \text{if event } i \text{ is scheduled in slot } j \\ 0, & \text{otherwise} \end{cases} \tag{3.1}$$

From that we can build up various constraints on what is a valid schedule.

The constraints

All events must be scheduled

For every row (event), we sum over every column of X and must have total sum 1.

$$\sum_{j=1}^{N} X_{ij} = 1 \text{ for all } 1 \leq i \leq M \tag{3.2}$$

All slots cannot have more than one event

For every column (slot), we sum over every row of X and must have total sum at most 1.

$$\sum_{i=1}^{M} X_{ij} \leq 1 \text{ for all } 1 \leq j \leq N \tag{3.3}$$

This in itself would give a basic schedule for a conference however things can be a bit more complicated:

- Some slots can be in parallel with each other (for example because of multiple rooms);
- Slots and events might have different duration;
- It might be desirable to have some common thread for talks in a collections of slots (for example: the morning session)

The mathematical representation for these constraints will be described below.

Events are only scheduled in slots for which they are available

There are multiple reasons for which an event might not be available in a given slot: perhaps the speaker is unavailable on a given day.

These constraints can be captured using a matrix $C_s \in \{0,1\}^{M \times N}$:

$$C_{sij} = \begin{cases} 1, & \text{if event } i \text{ is available in slot } j \\ 0, & \text{otherwise} \end{cases} \tag{3.4}$$

This gives the following constraint:

$$X_{ij} \leq C_{sij} \text{ for all } 1 \leq i \leq M,\ 1 \leq j \leq N \tag{3.5}$$

Two events are scheduled at the same time only if they are available to do so

Any two given events might not be able to occur concurrently: two events could be delivered by the same speaker, or they could be about a similar topic.

This constraint is captured using a matrix $C_e \in \{0,1\}^{M \times M}$:

$$C_{eii'} = \begin{cases} 1, & \text{if event } i \text{ is available during event } i' \\ 0, & \text{otherwise} \end{cases} \tag{3.6}$$

Using this, we define the following set for every slot j:

$$S_j = \{1 \leq j' \leq N \mid \text{ if } j \text{ and } j' \text{ are at the same time}\} \tag{3.7}$$

Using this we have the following constraint:

$$X_{ij} + X_{i'j'} \leq 1 + C_{eii'} \text{ for all } j' \in S_j \text{ for all } 1 \leq j \leq N \text{ for all } 1 \leq i, i' \leq M \tag{3.8}$$

We see that if $C_{eii'} = 0$ then at most one of the two events can be scheduled across the two slots j, j'.

Expressions (3.2), (3.3), (3.5), and (3.8) define a valid schedule and can be used by themselves.

However, it might be desirable to also optimise a given objective function.

Objective functions

Efficiency: Optimising to avoid total room overflow

Demand for events might be known: this will be captured using a vector $d \in \mathbb{R}^M_{\geq 0}$. Similarly capacity for rooms might be known, captured using another vector $c \in \mathbb{R}^N_{\geq 0}$. Whilst it might not be possible to stick to those constraints strongly

(when dealing with parallel sessions delegates might not go where they originally intended) we can aim to minimise the expected overflow given by the following expression:

$$\sum_{i=1}^{M}\sum_{j=1}^{N} X_{ij}(d_i - c_j) \tag{3.9}$$

Using this, our optimisation problem to give a desirable schedule is obtained by solving the following problem:

Minimise (3.9) subject to (3.2), (3.3), (3.5) and (3.8).

Equity: Optimising to avoid worse room overflow

Minimising (3.9) might still leave a given slot with a very large overflow relative to all over slots. We can aim to minimise the maximum overflow in a given slot given by the following expression:

$$\max_{i,j} X_{ij}(d_i - c_j)$$

Note that it is not possible to use $\max$ in the objective function for a linear program (it is none linear). However, instead we can define another variable: β as the upper bound for the overflow in each slot:

$$X_{ij}(d_i - c_j) \leq \beta \text{ for all } 0 \leq i \leq N \text{ for all } 1 \leq j \leq M \tag{3.10}$$

The objective function then becomes to minimize:

$$\beta \tag{3.11}$$

Using this, our optimisation problem to give a desirable schedule is obtained by solving the following problem:

Minimise (3.11) subject to (3.2), (3.3), (3.5), (3.8) and (3.10).

Consistency: Minimise change from a previous schedule

Once a schedule has been obtained and publicised to all delegates, a new constraint might arise (modifying (3.2), (3.3), (3.5) and (3.8). At this point the original optimisation problem can be solved again leading to a potentially completely different schedule. An alternative to this is to use distance from an original schedule X_o as the objective function. Norms on matrix spaces are usually non linear however, given the boolean nature of our variables, the following function can be used to measure the number of changes:

$$\sum_{i=1}^{M}\sum_{j=1}^{N} \delta(X_{oij}, X_{ij}) \tag{3.12}$$

where $\delta : \{0, 1\}^2 \to \{0, 1\}$ is given by:

$$\delta(x_o, x) = \begin{cases} x, & \text{if } x_o = 0 \\ 1 - x, & \text{if } x_o = 1 \end{cases} \tag{3.13}$$

Using this it is possible to obtain a schedule that is least disruptive from another schedule when presented with new constraints by solving the following problem:

Minimise (3.12) subject to (3.2), (3.3), (3.5), and (3.8).

Heuristics

The mathematical problem described in *Mathematical model* can be solved using linear programming which is guaranteed to give the optimal solution. This can be computationally expensive, thus `conference_scheduler` includes the ability to search the solution space in an efficient manner to obtain if not the best solution: a very good one.

A good overview of the main ideas behind these types of algorithms can be found in *[Aarts1997]*.

One subset of these types of algorithms is called neighbourhood searches. This involves defining a neighbourhood of a candidate solution. In `conference_scheduler`, the neighbourhood of a matrix X is defined as the matrix that corresponds to moving a single event to another slot. If the new slot is currently being used by another event the two events are swapped.

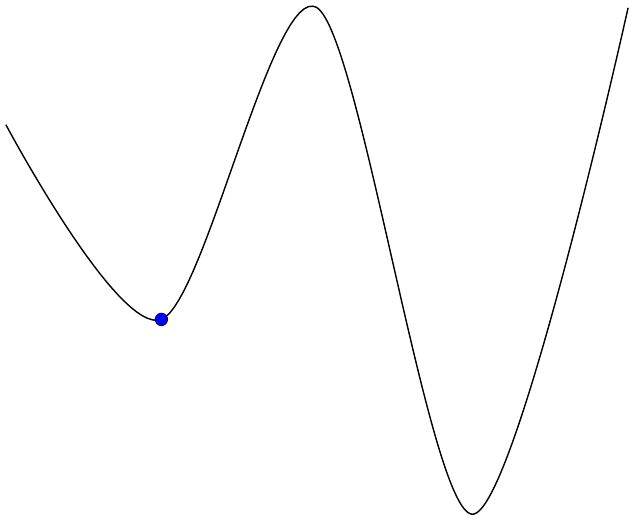
The general format of a neighbourhood search is to randomly select an element in the neighbourhood of X and either accept it or not as the new current solution according to a given criteria.

Hill climbing

The Hill climbing algorithm has the most elementary acceptance criteria for a new solution. If the new solution has a better score then it is accepted.

The downside to this greedy approach is that by immediately accepting the better solution the algorithm can arrive at a local optimal: a hill top that is not the highest hill.

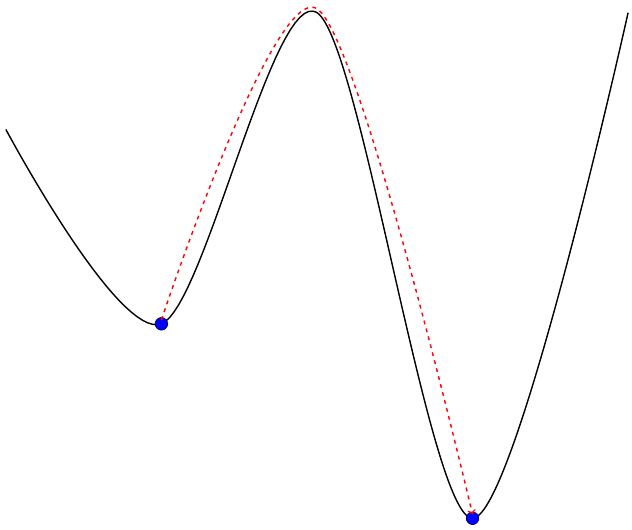
Note that the objective functions used in `conference_scheduler` are minimised. Whilst, mathematically this is equivalent, the analogy of finding the highest hill through hill climbing fails and we are in fact looking for the lowest valley.



Simulated annealing

This algorithm will accept a worse solution with a given probability. However this probability will decrease over time. Thus, the algorithm spends time at the beginning exploring the search space before beginning to hone in to a better solution. The name of this algorithm comes from the concept of annealing in metallurgy. A good overview of this is given in *[Henderson2003]*.

This in essence allows our search to “jump” away from local optima.



Bibliography

CHAPTER 4

Reference

conference_scheduler.scheduler

Compute a schedule in one of three forms, convert a schedule between forms and compute the difference between .

A schedule can be represented in one of three forms:

- solution: a list of tuples of event index and slot index for each scheduled item
- array: a numpy array with rows for events and columns for slots
- schedule: a generator for a list of ScheduledItem instances

`conference_scheduler.scheduler.`**`solution`**(*events*, *slots*, *objective_function=None*, *solver=None*, ***kwargs*)

Compute a schedule in solution form

param events of `resources.Event` instances

type events list or tuple

param slots of `resources.Slot` instances

type slots list or tuple

param solver a pulp solver

type solver pulp.solver

param objective_function from lp_problem.objective_functions

type objective_function callable

param kwargs arguments for the objective function

type kwargs keyword arguments

returns A list of tuples giving the event and slot index (for the given events and slots lists) for all scheduled items.

rtype list

Example

For a solution where

- event 0 is scheduled in slot 1
- event 1 is scheduled in slot 4
- event 2 is scheduled in slot 5

the resulting list would be:

```
[(0, 1), (1, 4), (2, 5)]
```

conference_scheduler.scheduler.**array**(*events*, *slots*, *objective_function=None*)

Compute a schedule in array form

param events of resources.Event instances

type events list or tuple

param slots of resources.Slot instances

type slots list or tuple

param objective_function from lp_problem.objective_functions

type objective_function callable

returns An E by S array (X) where E is the number of events and S the number of slots. Xij is 1 if event i is scheduled in slot j and zero otherwise

rtype np.array

Example

For 3 events, 7 slots and a solution where

- event 0 is scheduled in slot 1
- event 1 is scheduled in slot 4
- event 2 is scheduled in slot 5

the resulting array would be:

```
[[0, 1, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 1, 0, 0],
 [0, 0, 0, 0, 0, 1, 0]]
```

conference_scheduler.scheduler.**schedule**(*events*, *slots*, *objective_function=None*, *solver=None*, ***kwargs*)

Compute a schedule in schedule form

param events of resources.Event instances

type events list or tuple

param slots of resources.Slot instances

type slots list or tuple

param solver a pulp solver

type solver pulp.solver

param objective_function from lp_problem.objective_functions

type objective_function callable

param kwargs arguments for the objective function

type kwargs keyword arguments

returns A list of instances of `resources.ScheduledItem`

rtype list

`conference_scheduler.scheduler.`**`solution_to_array`**(*solution*, *events*, *slots*)

Convert a schedule from solution to array form

param solution of tuples of event index and slot index for each scheduled item

type solution list or tuple

param events of `resources.Event` instances

type events list or tuple

param slots of `resources.Slot` instances

type slots list or tuple

returns An E by S array (X) where E is the number of events and S the number of slots. Xij is 1 if event i is scheduled in slot j and zero otherwise

rtype np.array

Example

For For 3 events, 7 slots and the solution:

```
[(0, 1), (1, 4), (2, 5)]
```

The resulting array would be:

```
[[0, 1, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 1, 0, 0],
 [0, 0, 0, 0, 0, 1, 0]]
```

`conference_scheduler.scheduler.`**`solution_to_schedule`**(*solution*, *events*, *slots*)

Convert a schedule from solution to schedule form

param solution of tuples of event index and slot index for each scheduled item

type solution list or tuple

param events of `resources.Event` instances

type events list or tuple

param slots of `resources.Slot` instances

type slots list or tuple

returns A list of instances of `resources.ScheduledItem`

rtype list

`conference_scheduler.scheduler.`**`schedule_to_array`**(*schedule*, *events*, *slots*)

Convert a schedule from schedule to array form

param schedule of instances of `resources.ScheduledItem`

type schedule list or tuple

param events of `resources.Event` instances

type events list or tuple

param slots of `resources.Slot` instances

type slots list or tuple

returns An E by S array (X) where E is the number of events and S the number of slots. Xij is 1 if event i is scheduled in slot j and zero otherwise

rtype np.array

`conference_scheduler.scheduler.`**`array_to_schedule`**(*array*, *events*, *slots*)

Convert a schedule from array to schedule form

param array An E by S array (X) where E is the number of events and S the number of slots. Xij is 1 if event i is scheduled in slot j and zero otherwise

type array np.array

param events of `resources.Event` instances

type events list or tuple

param slots of `resources.Slot` instances

type slots list or tuple

returns A list of instances of `resources.ScheduledItem`

rtype list

`conference_scheduler.scheduler.`**`event_schedule_difference`**(*old_schedule*, *new_schedule*)

Compute the difference between two schedules from an event perspective

param old_schedule of `resources.ScheduledItem` objects

type old_schedule list or tuple

param new_schedule of `resources.ScheduledItem` objects

type new_schedule list or tuple

returns A list of `resources.ChangedEventScheduledItem` objects

rtype list

Example

```
>>> from conference_scheduler.resources import Event, Slot, ScheduledItem
>>> from conference_scheduler.scheduler import event_schedule_difference
>>> events = [Event(f'event_{i}', 30, 0) for i in range(5)]
>>> slots = [Slot(f'venue_{i}', '', 30, 100, None) for i in range(5)]
>>> old_schedule = (
...     ScheduledItem(events[0], slots[0]),
...     ScheduledItem(events[1], slots[1]),
...     ScheduledItem(events[2], slots[2]))
>>> new_schedule = (
...     ScheduledItem(events[0], slots[0]),
...     ScheduledItem(events[1], slots[2]),
...     ScheduledItem(events[2], slots[3]),
...     ScheduledItem(events[3], slots[4]))
>>> diff = (event_schedule_difference(old_schedule, new_schedule))
>>> print([item.event.name for item in diff])
['event_1', 'event_2', 'event_3']
```

conference_scheduler.scheduler.**slot_schedule_difference**(*old_schedule*, *new_schedule*)

Compute the difference between two schedules from a slot perspective

param old_schedule of `resources.ScheduledItem` objects

type old_schedule list or tuple

param new_schedule of `resources.ScheduledItem` objects

type new_schedule list or tuple

returns A list of `resources.ChangedSlotScheduledItem` objects

rtype list

Example

```
>>> from conference_scheduler.resources import Event, Slot, ScheduledItem
>>> from conference_scheduler.scheduler import slot_schedule_difference
>>> events = [Event(f'event_{i}', 30, 0) for i in range(5)]
>>> slots = [Slot(f'venue_{i}', '', 30, 100, None) for i in range(5)]
>>> old_schedule = (
...     ScheduledItem(events[0], slots[0]),
...     ScheduledItem(events[1], slots[1]),
...     ScheduledItem(events[2], slots[2]))
>>> new_schedule = (
...     ScheduledItem(events[0], slots[0]),
...     ScheduledItem(events[1], slots[2]),
...     ScheduledItem(events[2], slots[3]),
...     ScheduledItem(events[3], slots[4]))
>>> diff = slot_schedule_difference(old_schedule, new_schedule)
>>> print([item.slot.venue for item in diff])
['venue_1', 'venue_2', 'venue_3', 'venue_4']
```

conference_scheduler.resources

class conference_scheduler.resources.**Event**(*name*, *duration*, *demand*, *tags=None*, *unavailability=None*)

An event (e.g. a talk or a workshop) that needs to be scheduled

param name A human readable string

type name str

param duration The expected duration of the event in minutes

type duration int

param demand The anticipated demand - e.g. the number of attendees expected This will be compared with `Slot.capacity` during computation of the schedule to ensure that events are only scheduled in slots that can accommodate them. Use 0 if the event could be scheduled in any slot.

type demand int

param tags of human readable strings

type tags list or tuple, optional

param unavailability of `resources.Slot` or `resources.Event`

type unavailability list or tuple, optional

class `conference_scheduler.resources.`**`Slot`**

A period of time at a venue in which an event can be scheduled

param venue A human readable string

type venue str

param starts_at The starting time for the time period

type starts_at datetime

param duration The duration of the time period in minutes

type duration int

param capacity This will be compared with `Event.demand` during computation of the schedule to ensure that events are only scheduled in slots that can accommodate them.

type capacity int

param session A human readable string which serves as a tag for similar time periods e.g. ‘morning’, ‘afternoon’

type session str

Example

For a conference where:

- It will take place on 2016-09-17
- There are two rooms - ‘Main Hall’ and ‘Small Room’
- The Main Hall can seat 500 people and the Small Room, 50
- It is intended to hold two 30 minute talks in the morning (from 09:30 to 10:00 and from 11:00 to 11:30) and two more in the afternoon (from 14:00 to 14:30 and 15:00 to 15:30)

We would create the following eight objects:

```
>>> from conference_scheduler.resources import Slot
>>> Slot(
...     venue='Main Hall', starts_at=datetime(2016, 09, 17, 09, 30),
...      duration=30, capacity=500, session='morning')
>>> Slot(
...      venue='Main Hall', starts_at=datetime(2016, 09, 17, 10, 00),
...      duration=30, capacity=500, session='morning')
>>> Slot(
...      venue='Main Hall', starts_at=datetime(2016, 09, 17, 14, 00),
...      duration=30, capacity=500, session='afternoon')
>>> Slot(
...      venue='Main Hall', starts_at=datetime(2016, 09, 17, 15, 00),
...      duration=30, capacity=500, session='afternoon')
>>> Slot(
...      venue='Small Room', starts_at=datetime(2016, 09, 17, 09, 30),
...      duration=30, capacity=50, session='morning')
>>> Slot(
...      venue='Small Room', starts_at=datetime(2016, 09, 17, 10, 00),
...      duration=30, capacity=50, session='morning')
>>> Slot(
...      venue='Small Room', starts_at=datetime(2016, 09, 17, 14, 00),
...      duration=30, capacity=50, session='afternoon')
>>> Slot(
...      venue='Small Room', starts_at=datetime(2016, 09, 17, 15, 00),
...      duration=30, capacity=50, session='afternoon')
```

class conference_scheduler.resources.**ScheduledItem**

Represents that an event has been scheduled to occur in a slot

param event

type event resources.Event

param slot

type slot resources.Slot

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Bibliography

[Aarts1997] Aarts, Emile HL, and Jan Karel Lenstra, eds. Local search in combinatorial optimization. Princeton University Press, 1997.

[Dantzig1963] Dantzig, George B. “Linear programming and extensions.” (1963).

[Henderson2003] Henderson, Darrall, Sheldon H. Jacobson, and Alan W. Johnson. “The theory and practice of simulated annealing.” Handbook of metaheuristics. Springer US, 2003. 287-319.

[Schaerf1999] Schaerf, Andrea. “A survey of automated timetabling.” Artificial intelligence review 13.2 (1999): 87-127. APA

Python Module Index

C

conference_scheduler.resources, 33
conference_scheduler.scheduler, 29

Index

A

array() (in module conference_scheduler.scheduler), 30
array_to_schedule() (in module conference_scheduler.scheduler), 32

C

conference_scheduler.resources (module), 33
conference_scheduler.scheduler (module), 29

E

Event (class in conference_scheduler.resources), 33
event_schedule_difference() (in module conference_scheduler.scheduler), 32

S

schedule() (in module conference_scheduler.scheduler), 30
schedule_to_array() (in module conference_scheduler.scheduler), 32
ScheduledItem (class in conference_scheduler.resources), 35
Slot (class in conference_scheduler.resources), 34
slot_schedule_difference() (in module conference_scheduler.scheduler), 33
solution() (in module conference_scheduler.scheduler), 29
solution_to_array() (in module conference_scheduler.scheduler), 31
solution_to_schedule() (in module conference_scheduler.scheduler), 31