Experiments

February 2, 2018

```
In [1]: %load_ext autoreload
        %autoreload 2
        %matplotlib inline
```

1 Random Graphs

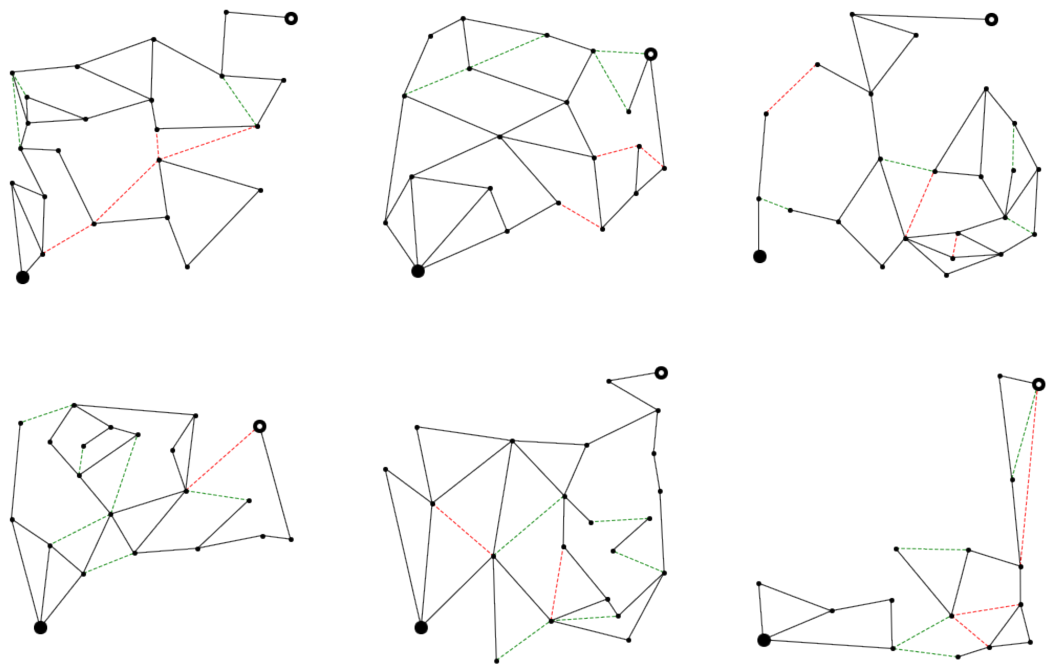
Draw random Delaunay triangulations

```
In [10]: from graphs import draw_graph, random_delaunay, random_realization
         from matplotlib import pyplot as plt
         import numpy as np

         def draw_random_graph( g, s, t, pos, hidden_state, realization, cut, pruned, ax=None,
             if ax is None:
                 ax = plt.subplot(111)
             hidden_edges = {l[0]: k for l, k in zip(hidden_state, realization)}
             if with_removed:
                 removed_edges = cut.edges() + pruned.edges()
             else:
                 removed_edges = []
             return draw_graph(g, s, t, pos, hidden_edges=hidden_edges,
                                 removed_edges=removed_edges, ax=ax)

         def draw_lot_of_graphs(rows, columns, width=5, with_removed=False):
             f, axs = plt.subplots(rows, columns, sharey=False, figsize=(columns * width, rows*
             if columns == 1:
                 axs = np.array([axs]).T
             if rows == 1:
                 axs = np.array([axs])
             for i in range(rows):
                 for j in range(columns):
                     g,  hidden_state, s, t, cut, pruned = random_delaunay(30, 0.5, 7, iters=1)
                     realization = random_realization(g, hidden_state, s, t)
                     draw_random_graph(g, s, t, g.pos, hidden_state=hidden_state, realization=r
                                       cut=cut, pruned=pruned, ax=axs[i,j], with_removed=with_r
             plt.tight_layout()
```

In [11]:
```
draw_lot_of_graphs(2,3)
```



2 Experiments

One Experiment is defined by a type of map, a set of classifiers and a set of policies

In [23]:
```
from experiment import RandomDelaunayExperiment, all_classifier_samples

exp_config = {'description': 'Random triangulations test',
              'map': {
                  'type': 'delaunay',
                  'number': 10,
                  'p_not_traversable': 0.5,
                  'n_hidden':7,
                  'size': 30,
                  'iters': 1
              },
              'classifier': {
                  'sigma': [0.5],
                  'samples': 1
              },
              'policy': {
                  'thresholds': [0, 0.5, 1]
```

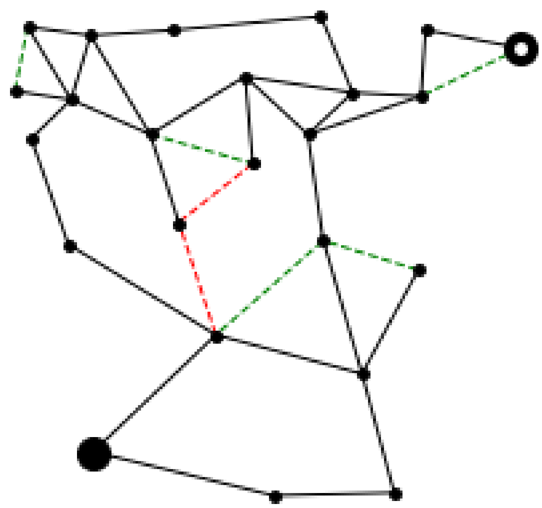
```
                    }
                   }

        exp = RandomDelaunayExperiment('test', exp_config, save=False, pool=1)
```

3 Sample a graph

An experiment repeatedly samples a planning instance made by - a graph - source and target nodes - a list of hidden states - a realization

```
In [16]: realization, planner, sources = exp.sample(0)
         draw_random_graph(planner._graph, sources[0], planner.target, planner._graph.pos,
                           planner.hidden_state, realization, None, None)
```



4 Compute all policy cost using classifiers

For a planning instance, we compute the competitive ration of all policies when using all classifiers

```
In [22]: classifiers = exp_config['classifier']
         policies = exp_config['policy']

         all_classifier_samples(realization, planner, sources=sources,
                                classifier_config=classifiers,
                                policy_config=policies)
```

```
Out[22]:    source  sigma  gamma                                     classification  \
         0      18    0.5    0.5  (0.83612828421, 0.598778096742, 0.985847385614...

            optimal  optimistic@0  optimistic@0.5  optimistic@1
         0      1.0           1.0             1.0           1.0
```

which is the same as the experiment method

```
In [19]: exp.compute_sample(2)
```

```
Out[19]:    source  sigma  gamma                                     classification  \
         0      13    0.5    0.5  (0.276369446103, 0.123315830072, 0.18544202570...

            optimal  optimistic@0  optimistic@0.5  optimistic@1
         0      1.0      1.424568             1.0      1.105254
```

that is applied to all maps to compute the final result.

```
In [20]: exp.compute()
```

```
Experiment test: 100%|| 10/10 [00:01<00:00,  4.97it/s]
```

```
Out[20]:    source  sigma  gamma                                     classification  \
         0       7    0.5    0.5  (0.923898161252, 0.204363043834, 0.56977803012...
         0      13    0.5    0.5  (0.978629766594, 0.0552927977009, 0.2865663388...
         0      13    0.5    0.5  (0.723630553897, 0.876684169928, 0.18544202570...
         0       3    0.5    0.5  (0.537193733993, 0.908166616615, 0.97802344131...
         0      12    0.5    0.5  (0.111833354785, 0.276035185364, 0.32134685316...
         0      26    0.5    0.5  (0.574146386314, 0.527225616736, 0.95470753753...
         0      28    0.5    0.5  (0.513772525978, 0.4754519308, 0.681791381714,...
         0       6    0.5    0.5  (0.711935502413, 0.326045018449, 0.46847478866...
         0      27    0.5    0.5  (0.528838722212, 0.822530737545, 0.93352337491...
         0      27    0.5    0.5  (0.170100475008, 0.478621172842, 0.68578413452...

             optimal  optimistic@0  optimistic@0.5  optimistic@1
         0  1.000000      1.000000        1.000000      1.000000
         0  1.000000      1.000000        1.000000      1.000000
         0  1.000000      1.000000        1.000000      1.000000
         0  1.000000      1.000000        1.000000      1.072915
         0  1.000000      1.000000        1.000000      1.000000
         0  1.366423      1.000000        1.366423      1.368792
         0  1.092566      1.000000        1.000000      1.092566
         0  1.016892      1.233201        1.000000      1.000000
         0  1.000000      1.000000        1.000000      1.000000
         0  1.000000      1.000000        1.000000      1.000000
```