Parametrized Complexity
Vertex Cover

*Based on the book **Parameterized Algorithms** by Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Daniel Marx, Marcin Pilipczuk, Michal Pilipczuk and Saket Saurabh*

Chapter 3 - Bounded Search Trees

Ilan Naiman & Yoan Shimoni

Dr. Meirav Zehavi

July 19, 2020

Source Code

Ilan Naiman – 206291098
Yoan Shimoni - 317756609

Table of Contents

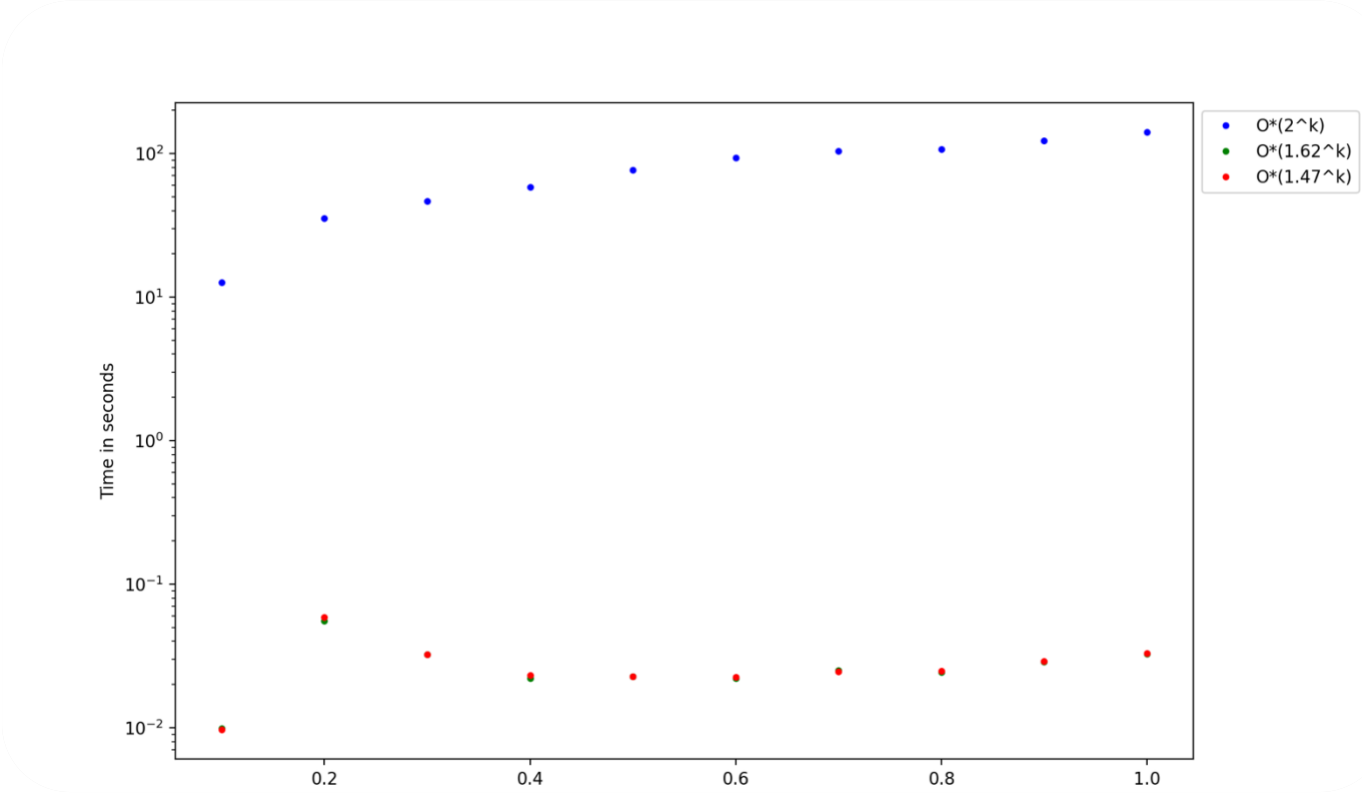
Ilan Naiman & Yoan Shimoni

Experiment 1

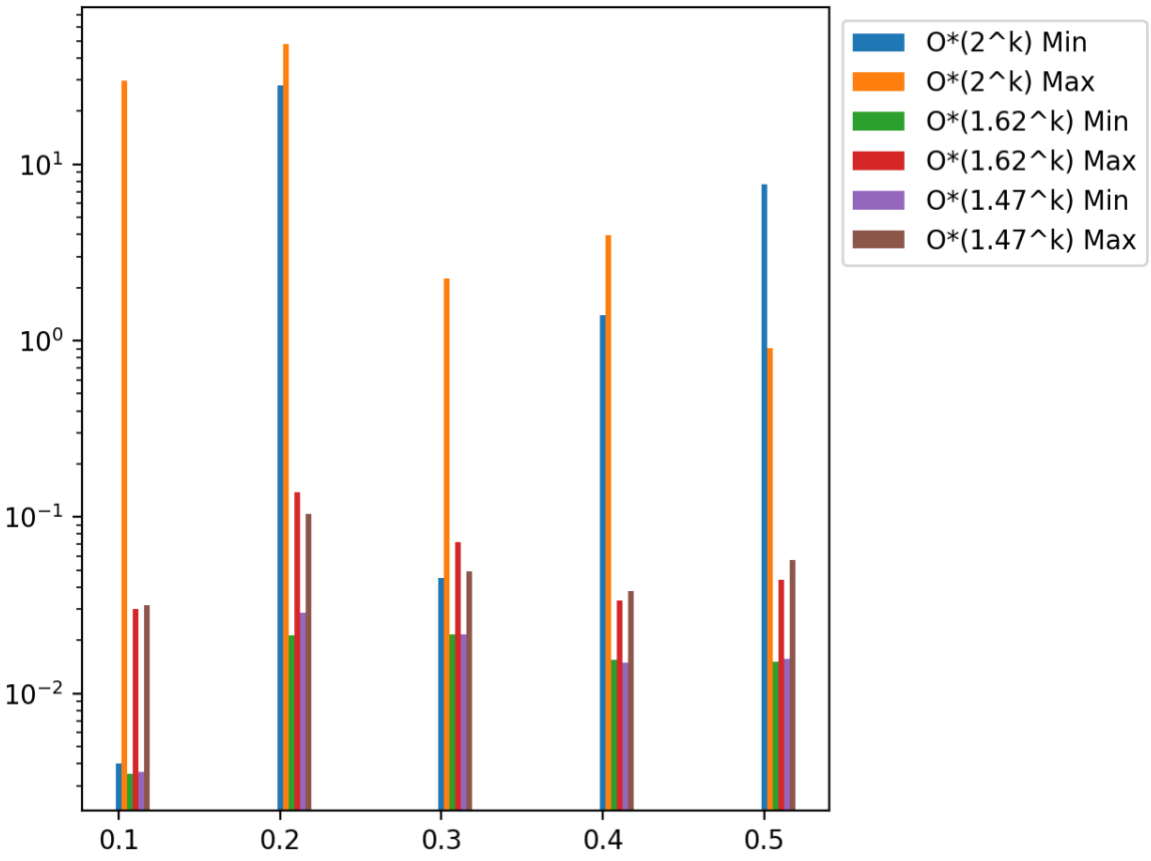
In this experiment we will randomly generate graphs with 35 vertices and the probability of edge between every 2 vertices will be changed in each time we run the algorithms. For each probability, $(p = 0.1, p = 0.2, \dots, p = 1)$, We will run all 3 algorithms for 50 iterations on each. Then we collect the result, such that for every p, we take the mean of all the 50 iterations for each algorithm, plot the results in a graph and then compare them all.
First, we want to examine what happens to the run time of each algorithm as the probability for edge changes.
Also, we want to see if there is difference in run time not only in theory but also in practice.
We run the algorithm with $k = 17$.

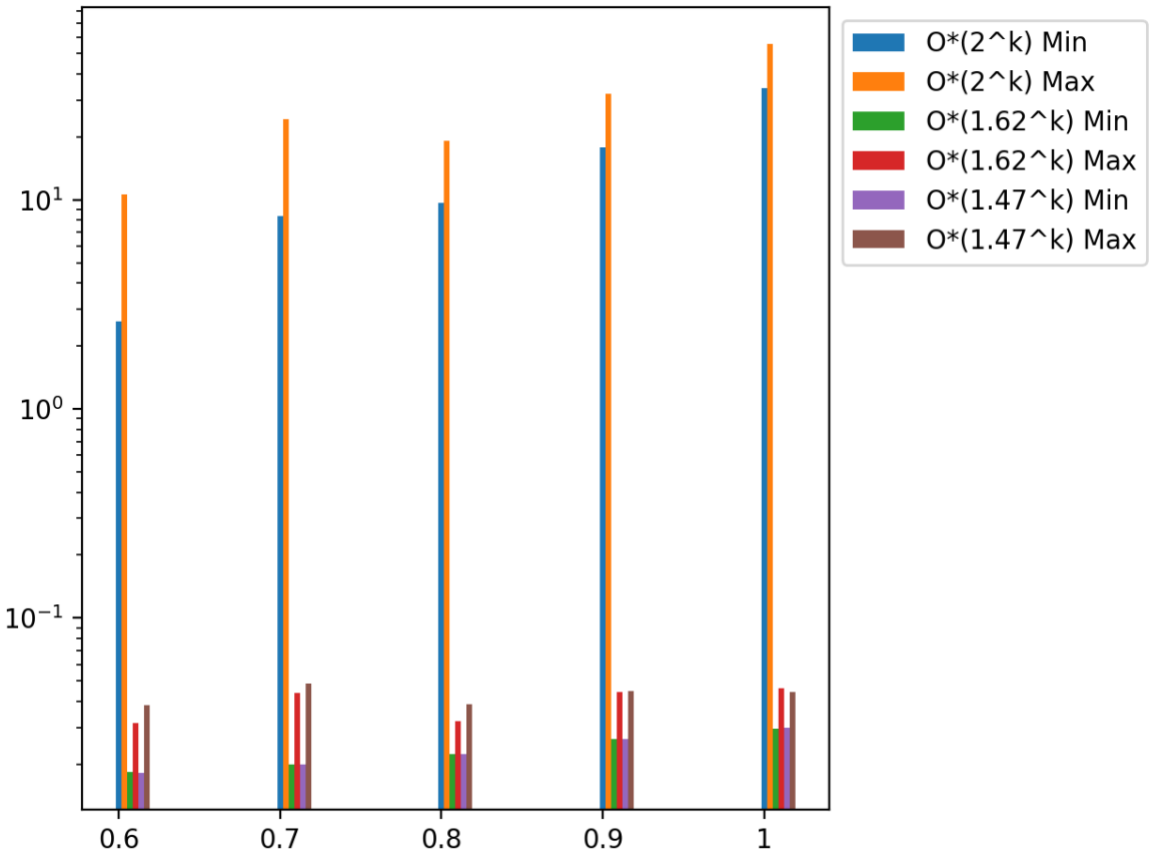
The results:

Plot of the results (Time in seconds of each algorithm for probability of an arc between 2 vertices):



The average results for each and every p can be seen in the plot. We also looked at the max and min run time of each algorithm for each and every p.





Conclusions:

- First, from the first plot, we instantly observe there are much difference between the algorithm that runs in time complexity $O^*(2^k)$, to the 2 others algorithms which run in $O^*(1.62^k)$ and $O^*(1.47^k)$ respectively. The $O^*(2^k)$ time algorithm runs very much slower.
- Another interesting result, is that for random graphs of size 35 vertices, and k=17, we can hardly see any difference between the $O^*(1.62^k)$ and $O^*(1.47^k)$ time complexity algorithms in practice (the red dots almost merge with the green ones).
- From the first plot, we observe that graphs with low probability such as $p = 0.1$, has better run time, but worse run time in average was for $p = 0.2$ and then became better as p were growing towards $p = 0.6$. Then again, for $p > 0.6$ the runtime in average started to slow down again.
- By the second plot and the third plot we learn that for sparse graphs the red bars are slightly lower than the brown ones, which generally means that in the worst case the run time of $O^*(1.47^k)$ algorithm was better, but as the graphs became dense, the $O^*(1.62^k)$ algorithm was better. This is only really small difference but it's worth further investigating whether the $O^*(1.62^k)$ algorithm has better runtime in practice for dense graphs than $O^*(1.47^k)$ algorithm.
- From the last 2 bullets, we conclude we want to further investigate how the vertex cover algorithm works for sparse and dense graphs. In order to investigate this topic, we will run the $O^*(1.62^k)$ and $O^*(1.47^k)$ algorithms on bigger v and k. (Running $O^*(2^k)$ on bigger inputs will be really heavy computation and will take a lot of time, as we already conclude both other algorithms are better in practice in general)

In order to repeat this experiment, using our implementations of all of the 3 algorithms, one should run the following code (each time saving the time results in some list and the use different statistics method to calculate average, min and max time):

```
for p in numpy.arange(0.1, 1.1, 0.1):
for i in range(1, 51):
graph_1 = nx.erdos_renyi_graph(v, p)
start = timeit.default_timer()
VertexCoverAlg3.run_vertex_cover_alg_3(graph_1, k)  # run algorithm O*(1.47^k)
stop = timeit.default_timer()
start = timeit.default_timer()
VertexCoverAlg2.run_vertex_cover_alg_2(graph_1, k) # run algorithm O*(1.62^k)
stop = timeit.default_timer()
start = timeit.default_timer()
VertexCoverAlg1.run_vertex_cover_alg_1(graph_1, k) # run algorithm O*(2^k)
stop = timeit.default_timer()
```

in order to plot our results we used matplotlib.pyplot library.

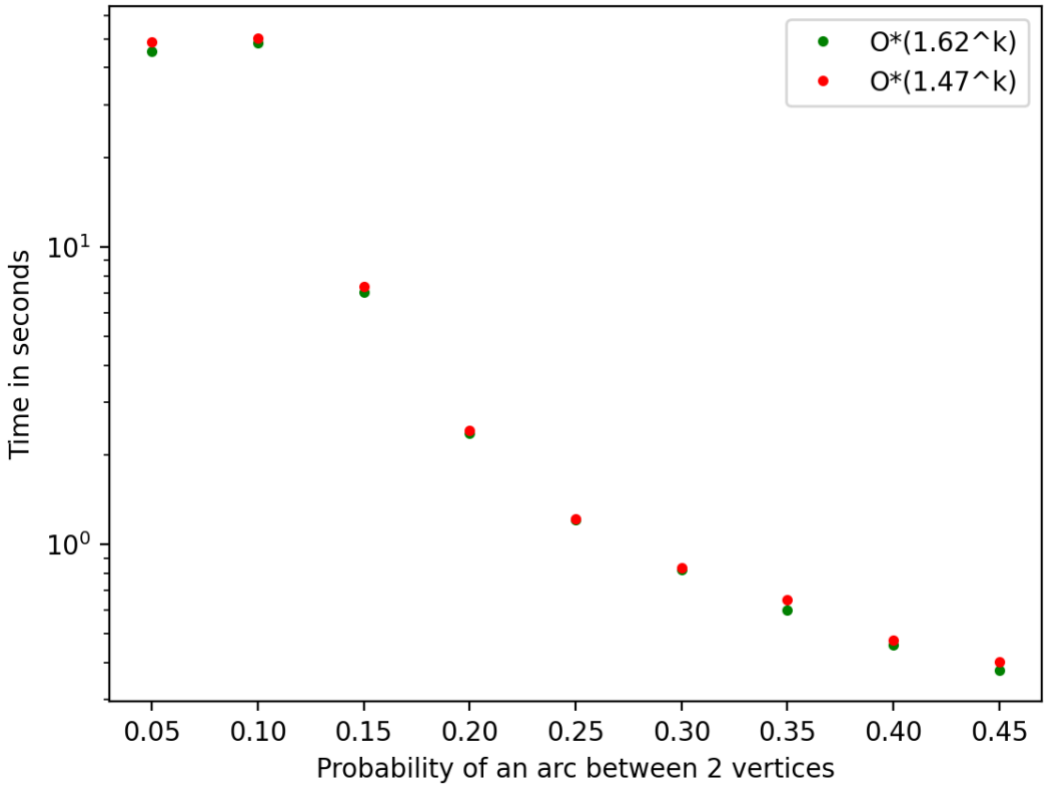
Ilan Naiman & Yoan Shimoni

Experiment 2

In this experiment we will randomly generate graphs with 100 vertices and the probability of edge between every 2 vertices will be changed in each time we run the algorithms. For each probability, $(p = 0.05, p = 0.1, \dots, p = 0.45)$, we will run 2 algorithms, the $O^*(1.47^k)$ and the $O^*(1.62^k)$ time complexity algorithms, for 50 iterations on each.
Then we collect the result, such that for every p, we take the mean of all the 50 iterations for each algorithm, plot the results in a graph and then compare them all.
We want to generate sparse graphs with linear number of edges and to check whether the $O^*(1.47^k)$ time complexity algorithms is really better in practice on some sparse random graph. We run the algorithm with $k = 45$.
Then, we also will try run both algorithms with low fixed p ($p = 0.05$) and changing v, number of vertices. (Plot3 shows the results for this experiment which is also running the algorithms on sparse graphs)

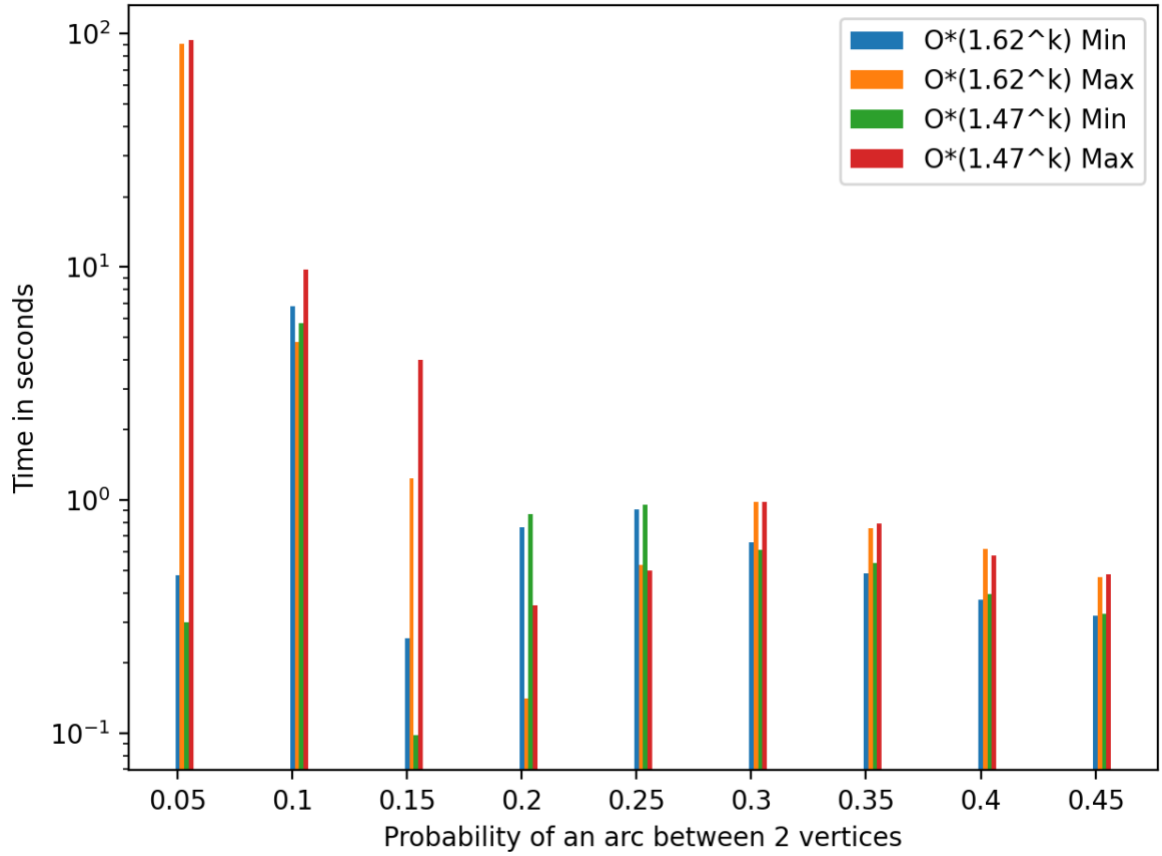
The results:

Plot of the results (Time in seconds of each algorithm for probability of an arc between 2 vertices):
Plot 1:

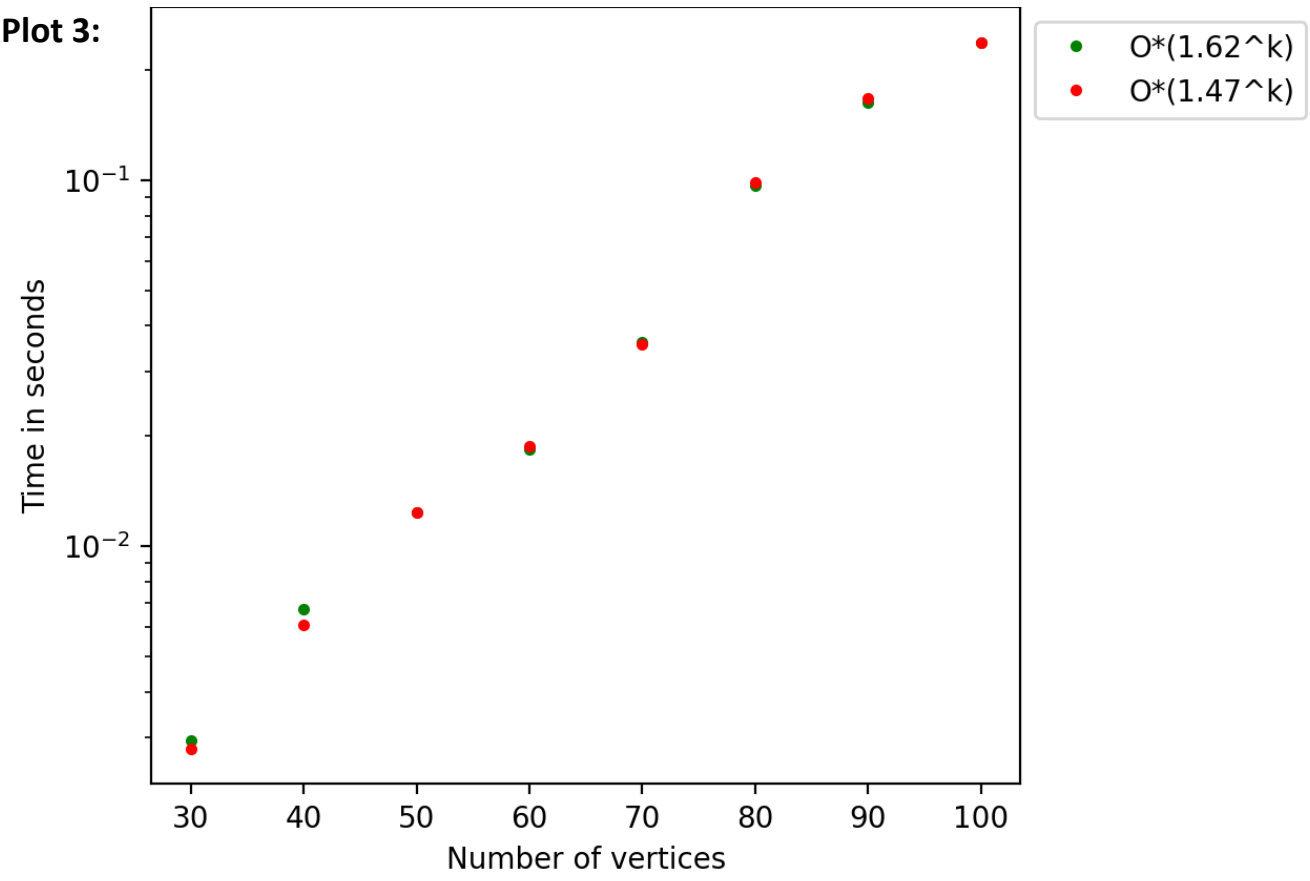


Plot 2:

The average results for each and every p can be seen in the plot. We also looked at the max and min run time of each algorithm for each and every p.



Plot 3:



Conclusions:

- First, from both plots 1 and 2, we can conclude that the $O^*(1.62^k)$ time complexity algorithm, seems to run slightly better in practice than $O^*(1.47^k)$ algorithm for sparse graphs which is surprising, because we initially thought that as the number of edges in the graph is smaller than the probability of the 5th rule (if the maximum degree of a vertex in the graph is 2, then solve the problem in polynomial time) in the $O^*(1.47^k)$ algorithm will be greater, thus will produce good running time.
  We think, the reason it wasn't better in practice is that although the graph was sparse, in each iteration of the algorithm there was at least 1 vertex with at least 3 neighbors. Thus, the polynomial calculation of iterating over the degree of each vertex in the graph and checking if its degree is less than 3 is taking too much time in each iteration where this calculation is redundant.
- From plot3 we conclude that there are still maybe some benefits for running the $O^*(1.47^k)$ algorithm on sparse graphs. Because for 30, 40 and 70 vertices and fixed $p = 0.05$ in average the $O^*(1.47^k)$ had better runtime.
- Also, we notice that as the probability for an arc increase, as we saw in the previous experiment (for p=0.2 to p=0.4), the runtime is better.

In order to repeat this experiment, using our implementations of the 2 algorithms, one should run the following code (each time saving the time results in some list and the use different statistics method to calculate average, min and max time):

```
k = 45
v = 100
for p in numpy.arange(0.05, 0.5, 0.05):
for i in range(1, 51):
graph_1 = nx.erdos_renyi_graph(v, p)
start = timeit.default_timer()
VertexCoverAlg3.run_vertex_cover_alg_3(graph_1, k)  # run algorithm O*(1.47^k)
stop = timeit.default_timer()
start = timeit.default_timer()
VertexCoverAlg2.run_vertex_cover_alg_2(graph_1, k)  # run algorithm O*(1.62^k)
stop = timeit.default_timer()
k = 20
p = 0.05
for v in numpy.arange(30, 110, 10):
for i in range(1, 51):
graph_1 = nx.erdos_renyi_graph(v, p)
start = timeit.default_timer()
VertexCoverAlg3.run_vertex_cover_alg_3(graph_1, k)  # run algorithm O*(1.47^k)
stop = timeit.default_timer()
start = timeit.default_timer()
VertexCoverAlg2.run_vertex_cover_alg_2(graph_1, k)  # run algorithm O*(1.62^k)
stop = timeit.default_timer()
```

in order to plot our results we used matplotlib.pyplot library.

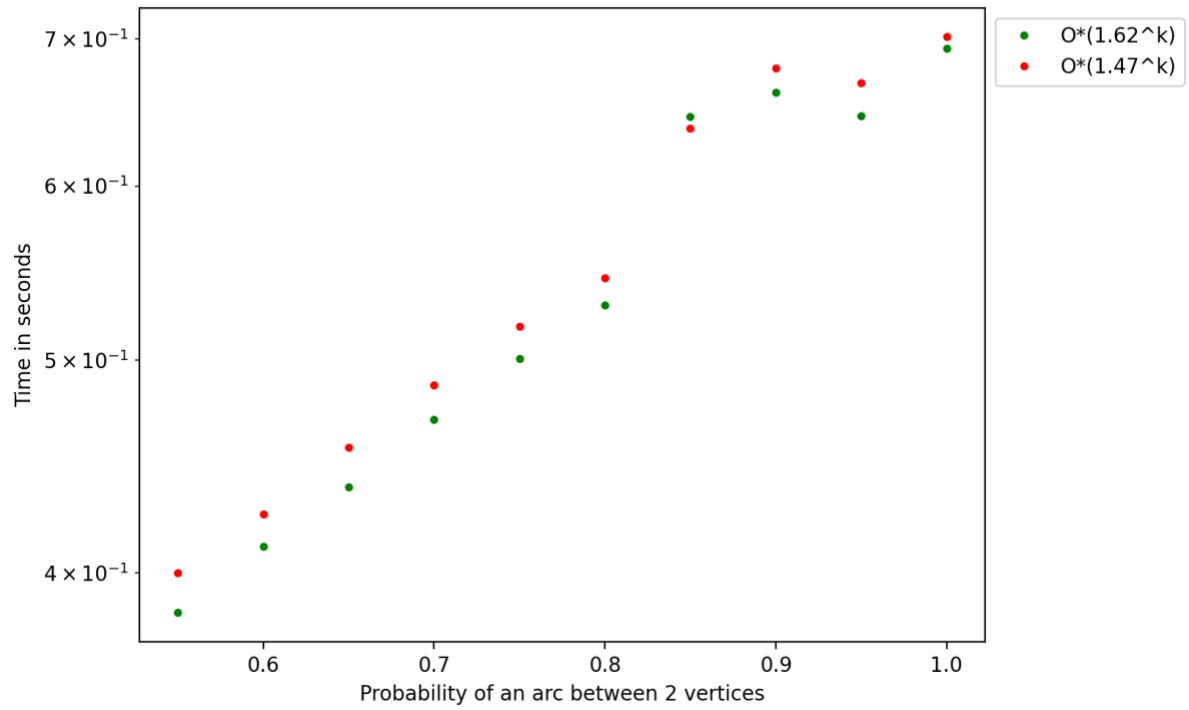
Ilan Naiman & Yoan Shimoni

Experiment 3

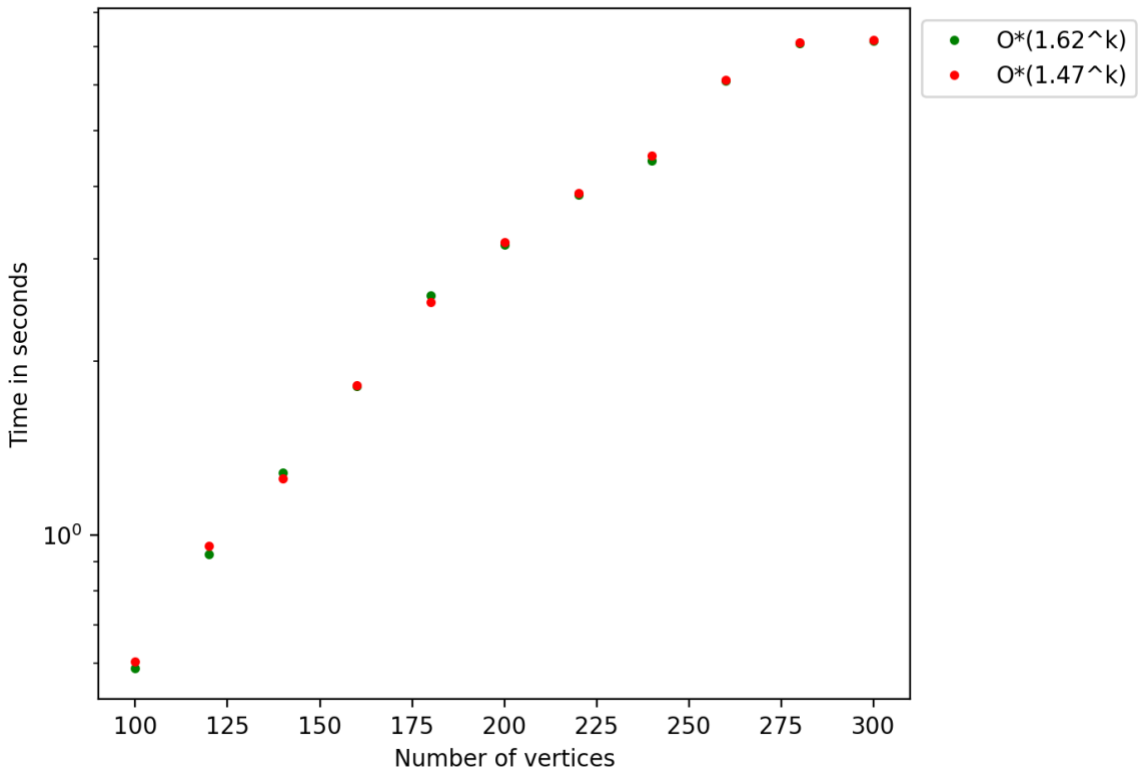
In this experiment we will randomly generate graphs with 100 vertices and the probability of edge between every 2 vertices will be changed in each time we run the algorithms. For each probability, $(p = 0.55, p = 0.6, \dots, p = 1)$. We will run 2 algorithms, the $O^*(1.47^k)$ and the $O^*(1.62^k)$ time complexity algorithms, for 50 iterations on each.
Then we collect the result, such that for every p, we take the mean of all the 50 iterations for each algorithm, plot the results in a graph and then compare them all.
We want to generate dense graphs with squared number of edges and to check whether the $O^*(1.47^k)$ time complexity algorithms is really better in practice on some dense random graph. We run the algorithm with $k = 45$.
Then, we also will try run both algorithms with p ($p = 0.8, p = 0.9$) close to 1 and changing v, number of vertices. (Plots 2 and 3 show the results for this experiment which is also running the algorithms on dense graphs)

The results:

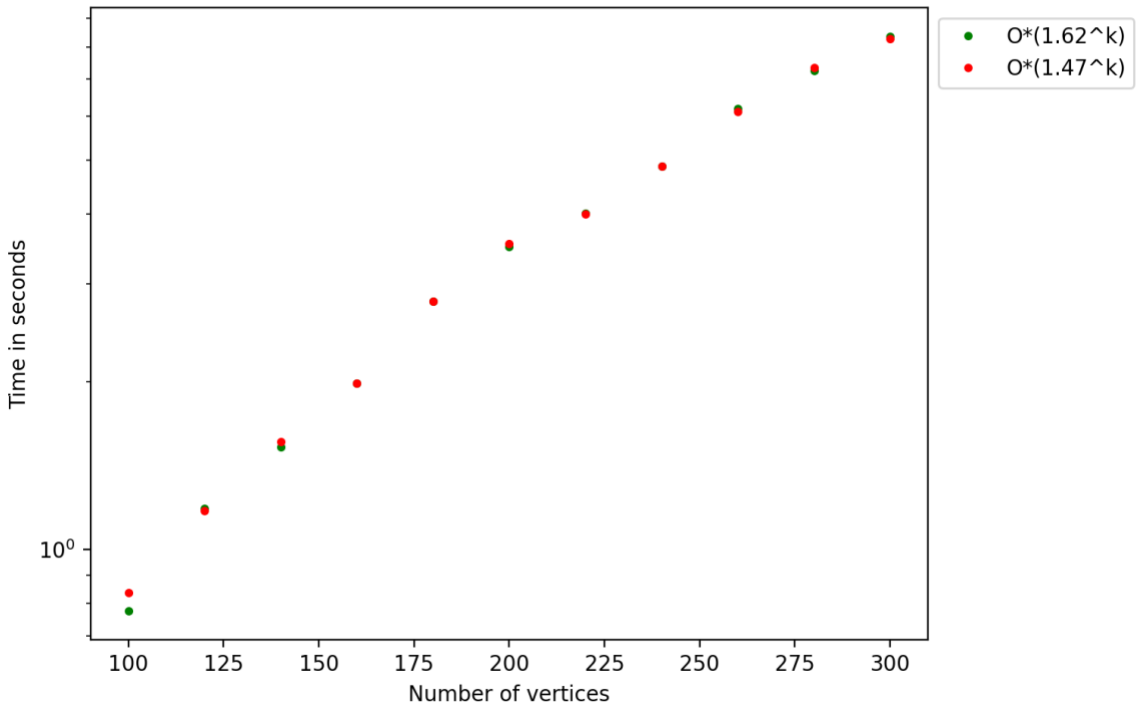
Plot of the results (Time in seconds of each algorithm for probability of an arc between 2 vertices):
Plot 1:



Plot 2 (p=0.8):



Plot 3 (p=0.9):



Conclusions:

- Here in this experiment, we see how for dense graphs the $O^*(1.62^k)$ algorithm runs better in practice then the $O^*(1.47^k)$ algorithm. We assume the reason for that as we said earlier is the 5th rule in the $O^*(1.47^k)$ algorithm. We claim that the overhead of the operation in rule 5 where it is obviously unnecessary (when the number of edges is really big) makes it slower in comparison to the $O^*(1.47^k)$ algorithm.
- Also, we see from plot 1 that the runtime grows as p grows, which means that for dense graphs both algorithms become slower. If we add this to the previous results (when running the algorithms on graphs with linear number of edges) we get similar results to the first experiment showing again that both algorithms are really slow when we use them for the complete graphs or sparse graphs (probably with some vertices of degree higher than 2).
- In plot 2 and 3, we examined how the runtime will change as a function of a fixed p and growing v (number of vertices). We got a reasonable result where the runtime grows as v growing. We assume the reason for that is that the polynomial factor inside O^* gets bigger with v.
- In addition, from plots 2 and 3, we learn that for most of the time the $O^*(1.62^k)$ algorithm runs better in practice, yet it doesn't seem as it's strictly a function of the number of vertices, rather it's a random case. We will explain this claim: in plot 2, where p is 0.8, when we run the algorithms with 180 vertices and 140 vertices, and got that in average, $O^*(1.47^k)$ algorithm had better runtime, but, the same happened when we ran with 260 and 300 vertices and p fixed to 0.9. So, we conclude that as v grows it doesn't mean that one of the algorithms will perform better, and in average it seems that in practice $O^*(1.62^k)$ performs better on dense graphs.
- From all the 3 experiments above we can conclude we reach optimal running time of those algorithms when running on graphs with probability 0.5 for an arc between every 2 vertices in the graph.
- From our last observation, we would also be doing more experiments on another kinds of graphs to see how the algorithms behave on some special kinds of graphs, such as bipartite graphs, and graphs which are composed of components of degree at most 2 where we add some edges to them, etc.

In order to repeat this experiment, using our implementations of the 2 algorithms, one should run the following code (each time saving the time results in some list and the use different statistics method to calculate average, min and max time):

```
k = 45
v = 100
```

```
for p in numpy.arange(0.55, 1.05, 0.05):
for i in range(1, 51):
graph_1 = nx.erdos_renyi_graph(v, p)
start = timeit.default_timer()
VertexCoverAlg3.run_vertex_cover_alg_3(graph_1, k)  # run algorithm O*(1.47^k)
stop = timeit.default_timer()
start = timeit.default_timer()
VertexCoverAlg2.run_vertex_cover_alg_2(graph_1, k)  # run algorithm O*(1.62^k)
stop = timeit.default_timer()

k = 45
p = 0.8
for v in numpy.arange(100, 320, 20):
for i in range(1, 51):
graph_1 = nx.erdos_renyi_graph(v, p)
start = timeit.default_timer()
VertexCoverAlg3.run_vertex_cover_alg_3(graph_1, k)  # run algorithm O*(1.47^k)
stop = timeit.default_timer()
start = timeit.default_timer()
VertexCoverAlg2.run_vertex_cover_alg_2(graph_1, k)  # run algorithm O*(1.62^k)
stop = timeit.default_timer()

k = 45
p = 0.9
for v in numpy.arange(100, 320, 20):
for i in range(1, 51):
graph_1 = nx.erdos_renyi_graph(v, p)
start = timeit.default_timer()
VertexCoverAlg3.run_vertex_cover_alg_3(graph_1, k)  # run algorithm O*(1.47^k)
stop = timeit.default_timer()
start = timeit.default_timer()
VertexCoverAlg2.run_vertex_cover_alg_2(graph_1, k)  # run algorithm O*(1.62^k)
stop = timeit.default_timer()
```

in order to plot our results we used matplotlib.pyplot library.

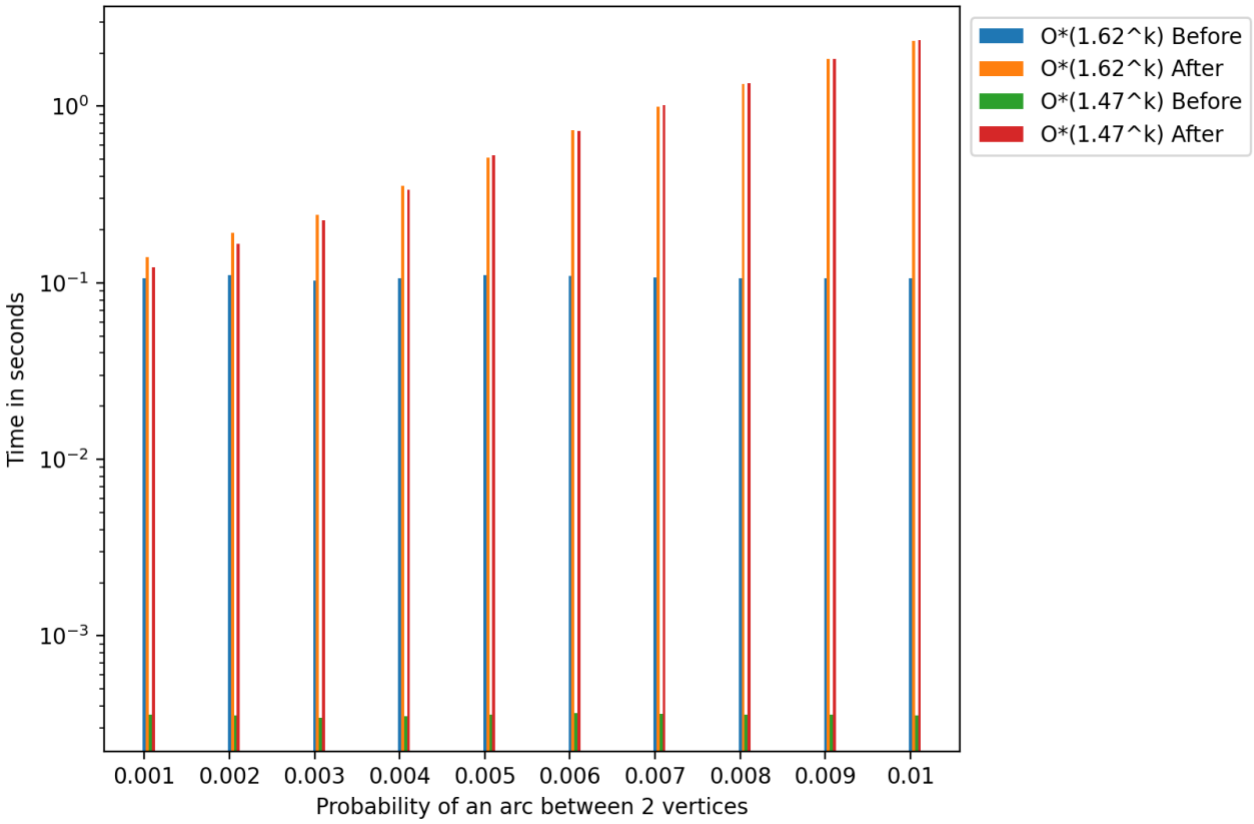
Ilan Naiman & Yoan Shimoni

Experiment 4

In this experiment we will randomly generate graphs in which the degree of each vertex is at most 2. We will test it on each algorithm (the $O^*(1.62^k)$ algorithm and $O^*(1.47^k)$ algorithm as we did previously) and compare how adding some random edges to this graph changes the runtime of the algorithms for each probability of adding an edge, $(p = 0.005, p = 0.01 \dots)$. We run this experiment with v = 100 and k = 45. Then we run similar experiment where we fixed the probability for an edge and we examine the changes while changing the number of vertices $(v = 75, v = 100, \dots, v = 190, v = 200, v = 210)$. We take the mean of all the 50 iterations for each algorithm, plot the results in a graph and then compare them all.

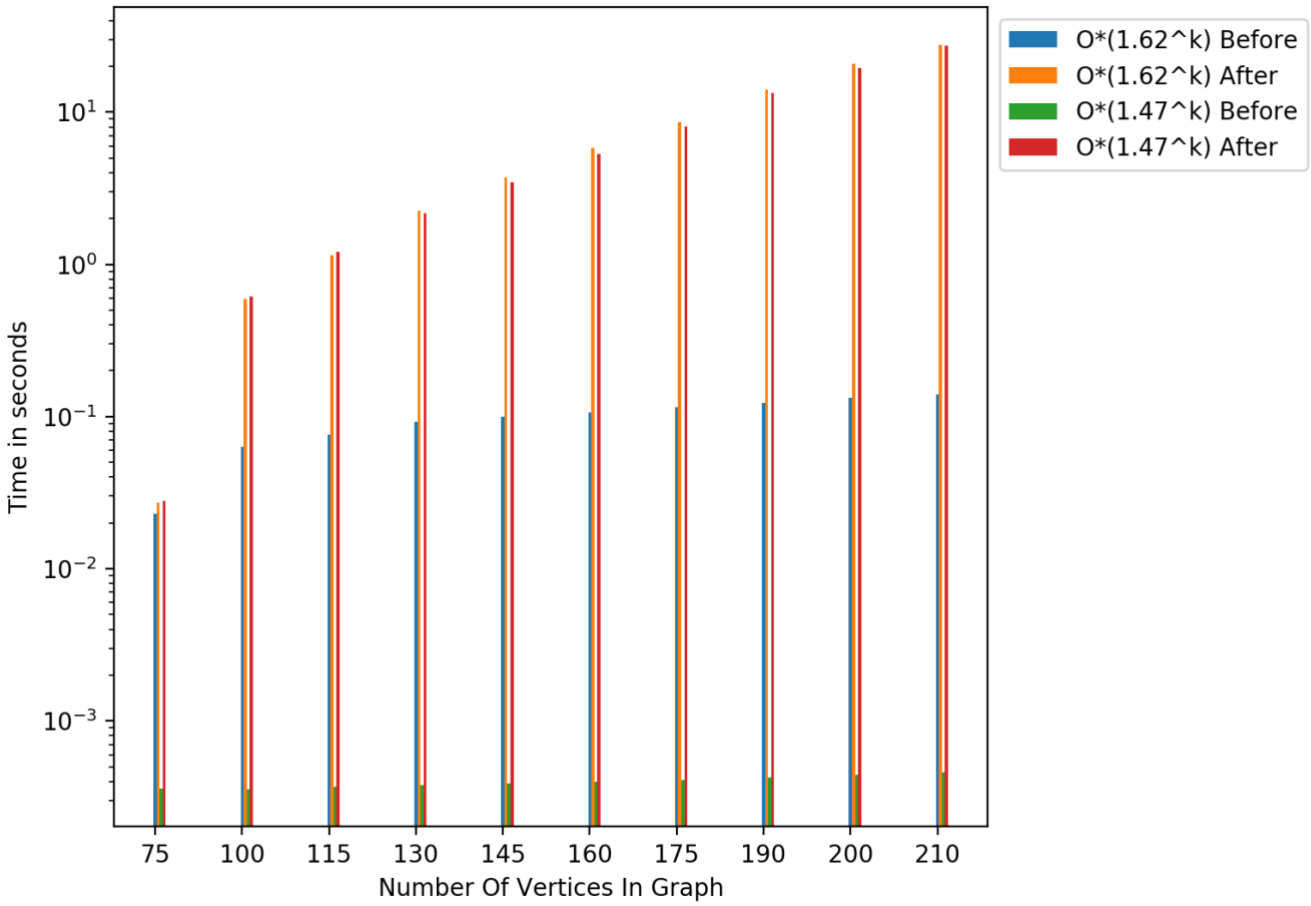
The results:

Plot 1 (Time as a function of probability for adding a random arc to the graph):



Ilan Naiman & Yoan Shimoni

Plot 2 (Time as a function of number of vertices in the graph):



Conclusions:

- First, from plot 1, when we run the algorithm on the circles components without adding any arcs, we see huge difference between the runtimes of the algorithms as a result of rule 5 which is: "If the maximum degree of a vertex in the graph is 2, then solve the problem in polynomial time". (the difference doesn't change as the probability grows as the probability is only playing a role after adding edges to the graph).
- Also, from plot 1 we learn that at first, when the probability for adding an edge is low, there is only small difference between the run time before and after adding the edge for the $O^*(1.62^k)$ algorithm, and as the probability gets higher also the difference in runtime becomes more significant, as oppose to the $O^*(1.47^k)$ algorithm which has big difference in runtime for any probability of adding edge. We assume we got this result as there were smaller chances of the $O^*(1.47^k)$ algorithm to 'satisfy' the conditions of the 5th rule. Thus, it behaves similar to the other algorithm in practice.
- In addition, from plot 1 we conclude that for small probabilities $(p < 0.005)$ for adding a random edge to the graph, we observe that the $O^*(1.47^k)$ time complexity has better runtime in practice. For $p \geq 0.005$, both algorithms behave similar and doesn't have significant difference in runtime. We assume the reason for this kind of result is the same as we mentioned in the bullet above.
- In plot 2, we look at results from running the same experiment where p is fixed to 0.005 and we change the number of vertices. As we saw previously, also here there is big difference in runtime when running both algorithms on the circle's components graph.
- We notice that as the number of vertices grows, also the $O^*(1.47^k)$ complexity algorithm runs faster in practice, which may imply that as the graph is bigger and sparser, the chances that the 5th rule will help to improve the runtime in practice becomes greater.
- Finally, from this experiment we learn that as we add more edges to the graph (e.g., run the experiment with higher probability to add an edge), the runtime of the $O^*(1.47^k)$ complexity algorithm becomes similar to the $O^*(1.62^k)$ algorithm or worse in practice. Also, it seems that as the graphs are bigger (has more vertices) and we add small number of edges to a graph composed of circles, e.g. the graph is also sparse, we get better runtime in practice of the $O^*(1.47^k)$ complexity algorithm.

In order to repeat this experiment, using our implementations of the 2 algorithms, one should run the following code (each time saving the time results in some list and the use different statistics method to calculate average time):

```
for v in vertices:
    for i in range(1, 51):
        vertices = [i for i in range(v)]
        h = 0
        circles_vertices = []
        while v - h > 3:
            rand_point = random.randint(3, v - h)
            circles_vertices.append(vertices[h:h+rand_point])
            h += rand_point
        graph_1 = nx.Graph()
        for circle in circles_vertices:
            graph_1 = nx.compose(graph_1, nx.cycle_graph(circle))
        alg147_runner(f, graph_1, i, k, results, alg147_res_bef)
        alg162_runner(f, graph_1, k, alg162_res_bef)
        print("\nRun The algorithms after adding some edges")
        nonedges = list(nx.non_edges(graph_1))
        for nonedge in nonedges:
            if random.random() <= p:
                graph_1.add_edge(nonedge[0], nonedge[1])
        print("Num of edges after adding some: " + str(len(graph_1.edges())))
        alg147_runner(f, graph_1, i, k, results, alg147_res_aft)
        alg162_runner(f, graph_1, k, alg162_res_aft)
```

once we run the experiment with p changing, then we fix p and change the number of vertices.

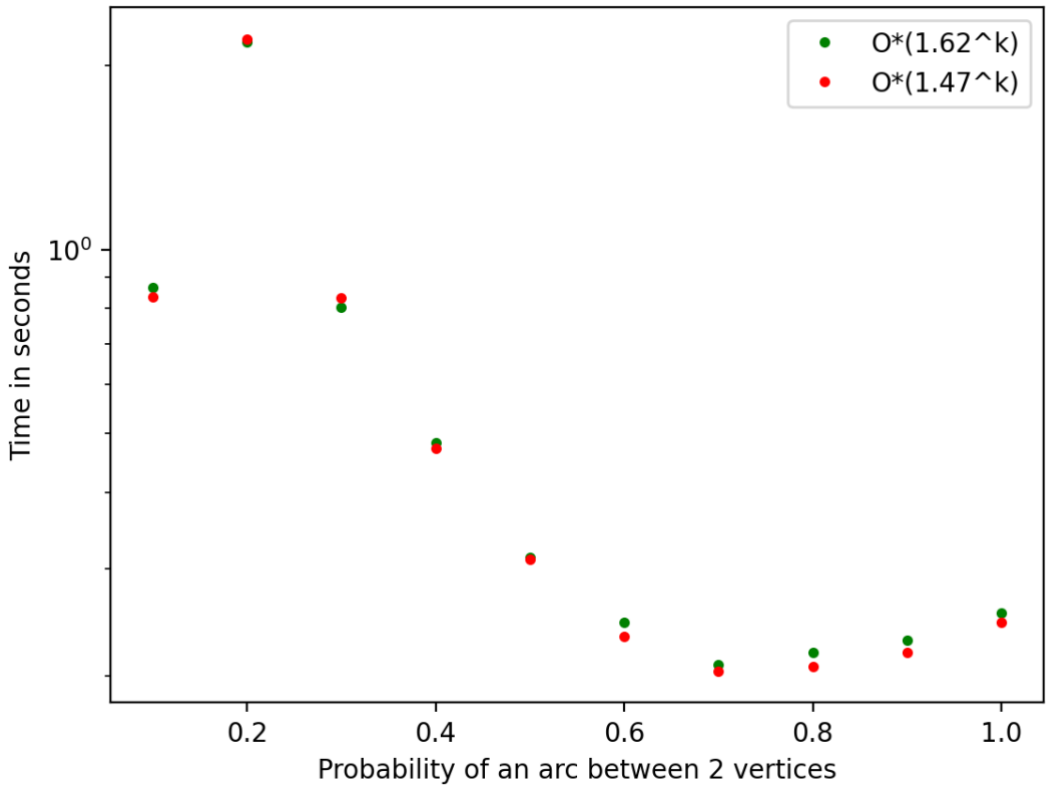
Ilan Naiman & Yoan Shimoni

Experiment 5

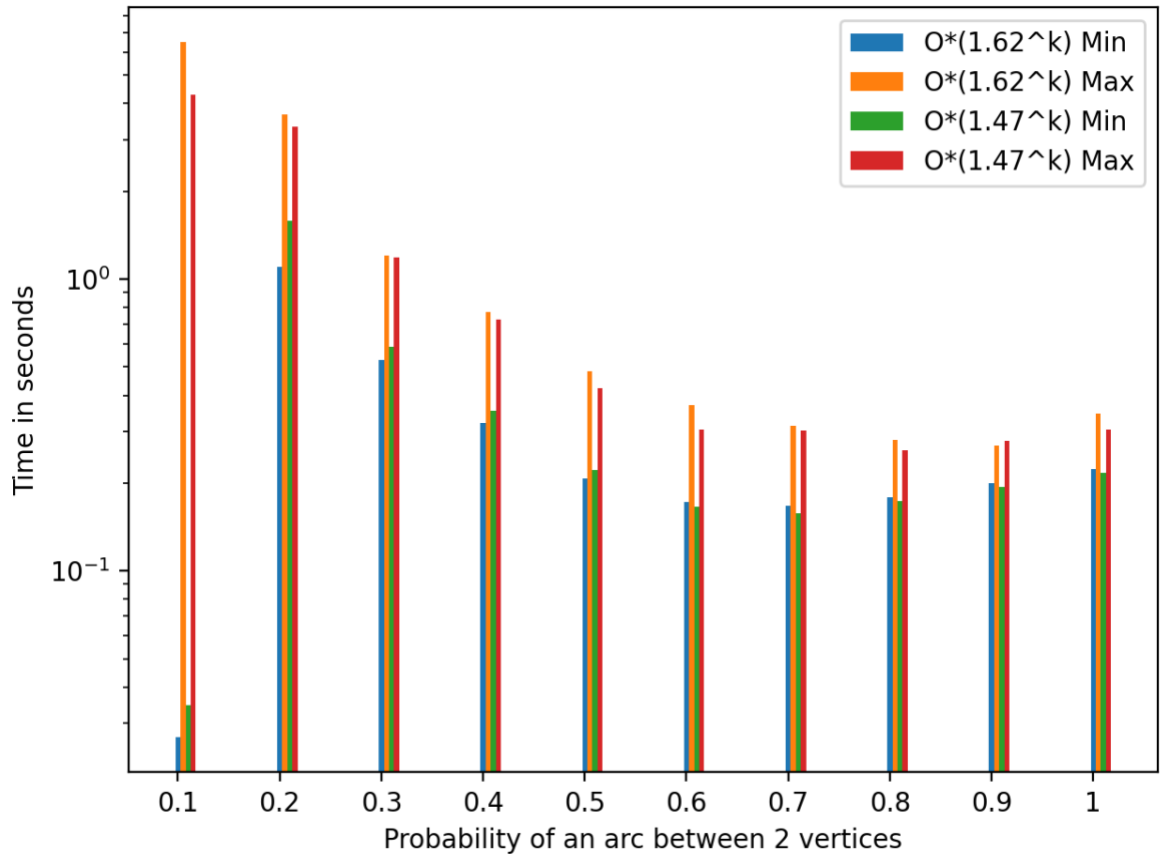
In this experiment we interested in examining the behavior of the algorithms on bipartite graphs (although we are aware of polynomial algorithm to find vertex cover in bipartite graphs with flow network). In this experiment we will randomly generate graphs $G = (A, B, E)$ with $A = 50, B = 40$ and the probability of edge between every 2 vertices will be changed in each time we run the algorithms. For each probability, $(p = 0.1, p = 0.2, \dots, p = 1)$ and $k = 30$. We will run both $O^*(1.47^k)$ and $O^*(1.62^k)$ algorithms for 50 iterations on each. Then we collect the result, such that for every p, we take the mean of all the 50 iterations for each algorithm, plot the results in a graph and then compare them all. Then, we repeat the same experiment but this time we will fix $p = 0.5$ and change the vertices number such that $\big((A = 40, B = 50), (A = 50, B = 60), \dots, (A = 140, B = 150)\big)$, $k = 40$.
First, we want to examine what happens to the run time of each algorithm as the probability for edge changes.
Second, we want to look at the runtime change as we increase the number of vertices in the graph with fixed probability for and edge for every 2 vertices in the graph.
Also, we want to see if there is difference in run time not only in theory but also in practice.

The results:

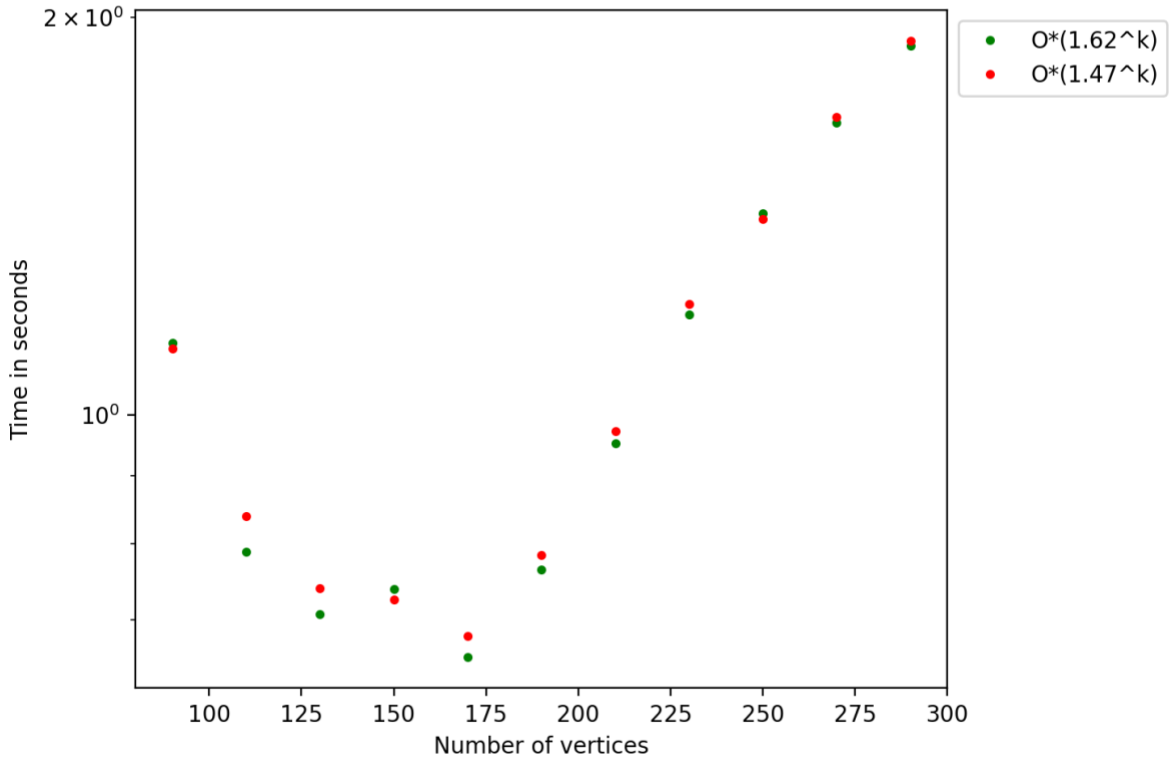
Plot 1 (average run time for all iterations, changing p):



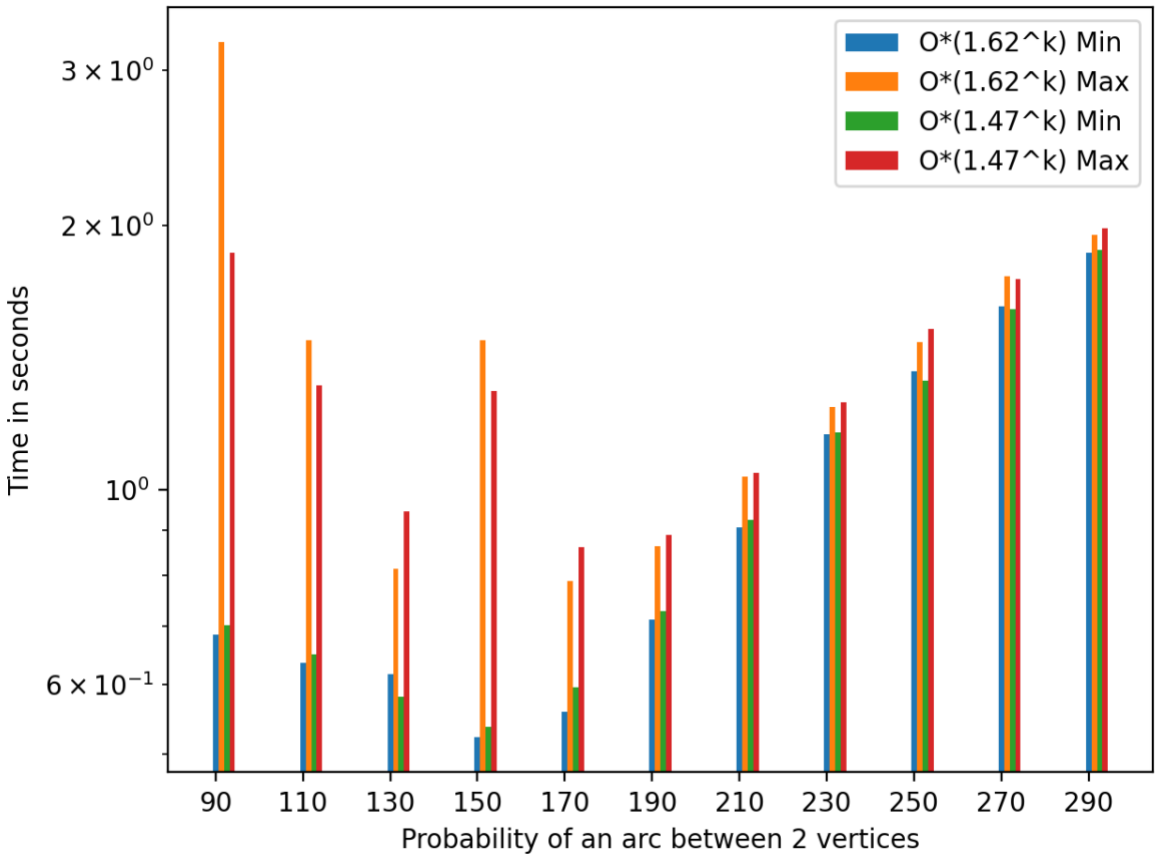
Plot 2 (min-max run time of all iterations, changing p):



Plot 3 (avg run time of all iterations, changing v):



Plot 4 (max min run time of all iterations, changing v):



Conclusions:

- First, looking at plot 1, the runtime of both algorithms gets better as the graph becomes denser which correlates previous results.
- Also, from plot 1 we conclude that in general, the $O^*(1.47^k)$ time complexity algorithm runs faster than the $O^*(1.62^k)$ time complexity algorithm.
  The reason is maybe due to the fact that that we have at least $|A| + |B|$ independent vertices in the graph, which increase the probability that after less iterations we will be left with a graph G' in which the maximum degree of a vertex in the graph is 2, and then the $O^*(1.47^k)$ algorithm will solve the problem in polynomial time in contrast to the other algorithm which will call the algorithm again recursively.
- In addition, we learn from Plot 2, as said in previously, that for each probability for an edge, the orange bar (represents the maximum runtime of $O^*(1.62^k)$ algorithm) is always higher than the red bar (represents the maximum runtime of $O^*(1.47^k)$ algorithm). Which means that if we examine the worst case of each algorithm in the experiment, $O^*(1.47^k)$ algorithm ran better in practice.

- Looking at the results of plot 3, we get another interesting result. We see that as the number of vertices grow, in most of the times, the runtime of $O^*(1.62^k)$ algorithm is better in practice which again maybe due to the 5th rule in the $O^*(1.47^k)$ algorithm, which has small overhead in calculating the maximum degree of a vertex, while as the graph has more vertices, it has more edges (with $p = 0.5$) which decrease the probability for such case.
- In conclusion, we learn from this experiment, that for bipartite graphs, when the graph is relatively small, then $O^*(1.47^k)$ algorithm will be better also in practice (for sparse and dense graphs), but as the graphs get bigger with $p = 0.5$, in most of the cases, the other algorithm runtime will be better in practice.

```
k = 45
v = (40, 50)
for p in numpy.arange(0.1, 1.1, 0.1):
    for i in range(1, 51):
        graph_1 = nx.algorithms.bipartite.generators.random_graph(v[0], v[1], p)
        start = timeit.default_timer()
        VertexCoverAlg3.run_vertex_cover_alg_3(graph_1, k)  # run algorithm O*(1.47^k)
        stop = timeit.default_timer()
        start = timeit.default_timer()
        VertexCoverAlg2.run_vertex_cover_alg_2(graph_1, k)  # run algorithm O*(1.62^k)
        stop = timeit.default_timer()
```

```
k = 40
vertices = [(40, 50), (50, 60), (60, 70), (70, 80), (80, 90), (90, 100), (100, 110), (110,
120), (120, 130), (130, 140), (140, 150)]]
p = 0.5
for v in vertices:
    for i in range(1, 51):
        graph_1 = nx.algorithms.bipartite.generators.random_graph(v[0], v[1], p)
        start = timeit.default_timer()
        VertexCoverAlg3.run_vertex_cover_alg_3(graph_1, k)  # run algorithm O*(1.47^k)
        stop = timeit.default_timer()
        start = timeit.default_timer()
        VertexCoverAlg2.run_vertex_cover_alg_2(graph_1, k)  # run algorithm O*(1.62^k)
        stop = timeit.default_timer()
```

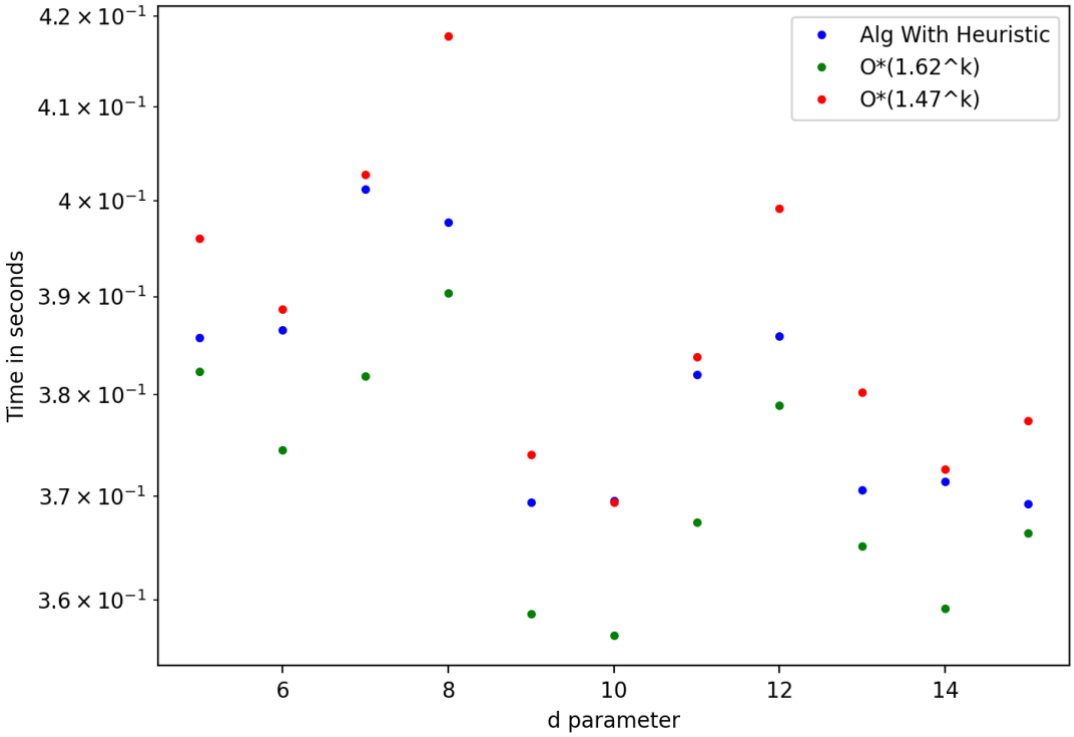
Experiment 6

In the end, we try to add a new rule. We try to exploit the tradeoff between both algorithms. As we have more edges, the probability to have a maximum degree 2 of a vertex in the graph is lower, thus we want to reduce the number of the time we try to apply the rule. Therefore, we add the next heuristic to the $O^*(1.47^k)$ algorithm before we perform rule 5:

If the number of edges in the graph is more than $\frac{|V|*(|V|-1)}{d}$, then run rule 6 (which is just call the algorithm recursively) else, run rule 5 and then if we need rule 6. We could have chosen d randomly, but we first will try to investigate if we can find the best d parameter such that will optimize the runtime in practice between 5 and 15. The way we will do it is by conducting an experiment where we choose $p = 0.5$, $v = 100$ and $k = 45$. Next we run all algorithms for 50 iterations, in each time we changed from $d = 5$ to $d = 15$.

The results:

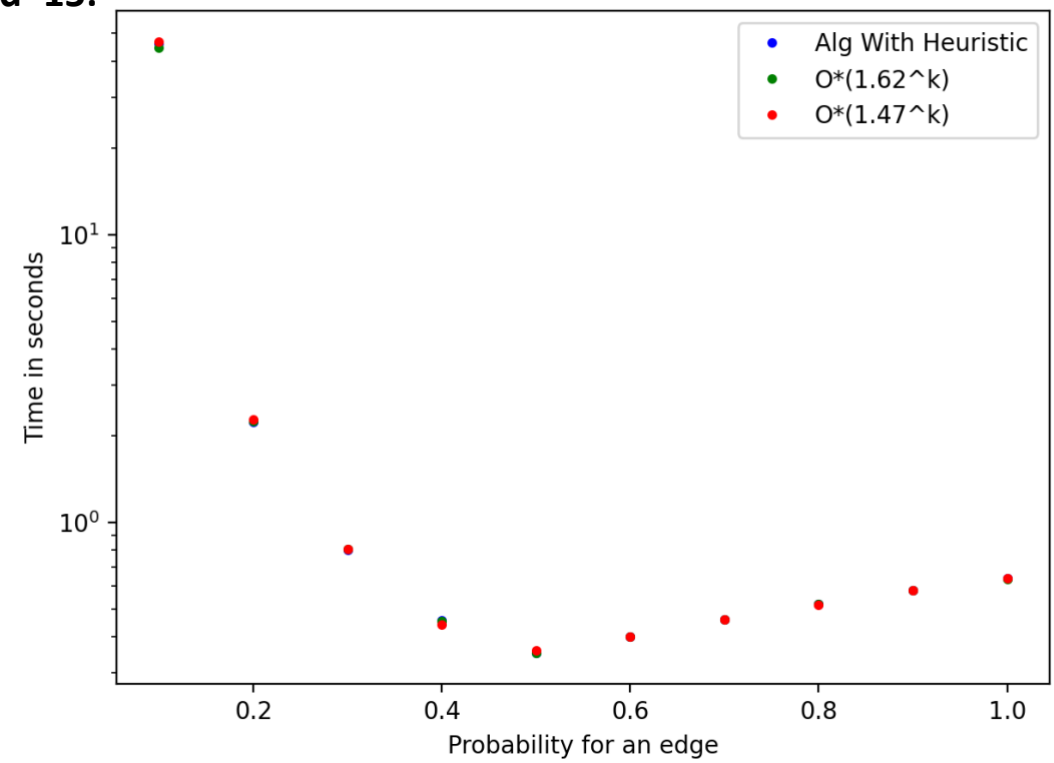
Plot 1: Running all algorithms when we changed d.



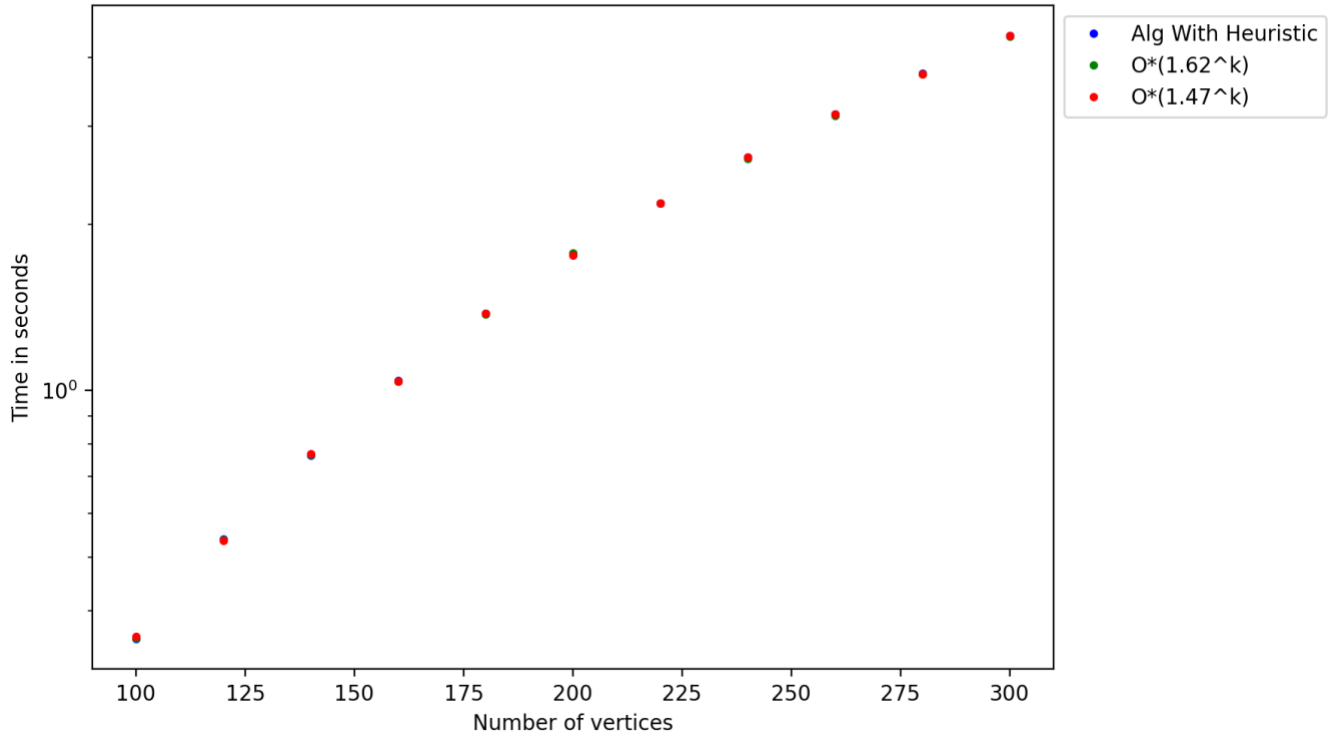
From the plot we learn that we got some interesting results. For any d we picked, the algorithm with the heuristic runtime was better in practice than the $O^*(1.47^k)$ algorithm.

Also, its runtime was somewhere between both other algorithms. So, we pick d = 15 where it seems to be optimized in the range we tested.
Due to previous results, we will run a general experiment which will examine how the algorithm with the heuristic behaves as oppose to the 2 other algorithms, $O^*(1.47^k)$ and $O^*(1.62^k)$ when we choose d = 15 and we change p. Also, we would be interested in the results when we change v.

Plot 2: Running all algorithms when we changed p with constant d=15:



Plot 3: Running all algorithms when we changed v with constant d=15.



Conclusions:

- First, from plot 2 we learn that the algorithm with heuristic runs almost the same time as both algorithms, it can hardly be seen through the plot (although the y scale is logarithmic) but for instance, when p = 0.6 we got the following results:
  - $O^*(1.47^k)$ algorithm – 0.40133
  - $O^*(1.62^k)$ algorithm – 0.40123
  - algorithm with heuristic – 0.40118
- We see that sometimes the heuristic slightly helped to improve the runtime and sometimes not, in general, there is no significant influence by the heuristic we added to the algorithm.
- We assume it runs almost in the same time because the probability for rule 5 to be applied is low as we learnt from previous results, we get that the heuristic unable to improve significantly the runtime in practice.
- In addition, from plot 3 we infer that as the number of vertices grows, the results of the runtime stay the same (it gets slower but the relatively to the previous result), all algorithms runtime is almost the same.
- For future work, we would like to try and find better heuristic. A heuristic which is maybe some mixture of the algorithms or rather be only on the $O^*(1.62^k)$ which yielded us better results most of the time.

```
k = 45
v = 100
p = 0.5
for d in range(5, 16):
   for i in range(1, 51):
   graph_1 = nx.erdos_renyi_graph(v, p).
   start = timeit.default_timer()
   VertexCoverAlg3.run_vertex_cover_alg_3(graph_1, k) # run algorithm O*(1.47^k)
   stop = timeit.default_timer()
   start = timeit.default_timer()
   VertexCoverAlg2.run_vertex_cover_alg_2(graph_1, k)  # run algorithm O*(1.62^k)
   stop = timeit.default_timer()
   num_edges = len(list(graph_1.edges()))
  start = timeit.default_timer()
   VertexCoverAlgHeuristic.run_vertex_cover_heuristic(graph_1, k, d, num_edges)
    stop = timeit.default_timer()
```

To repeat results from 2nd and 3rd plots, one should follow code in previous experiments.

Ilan Naiman & Yoan Shimoni

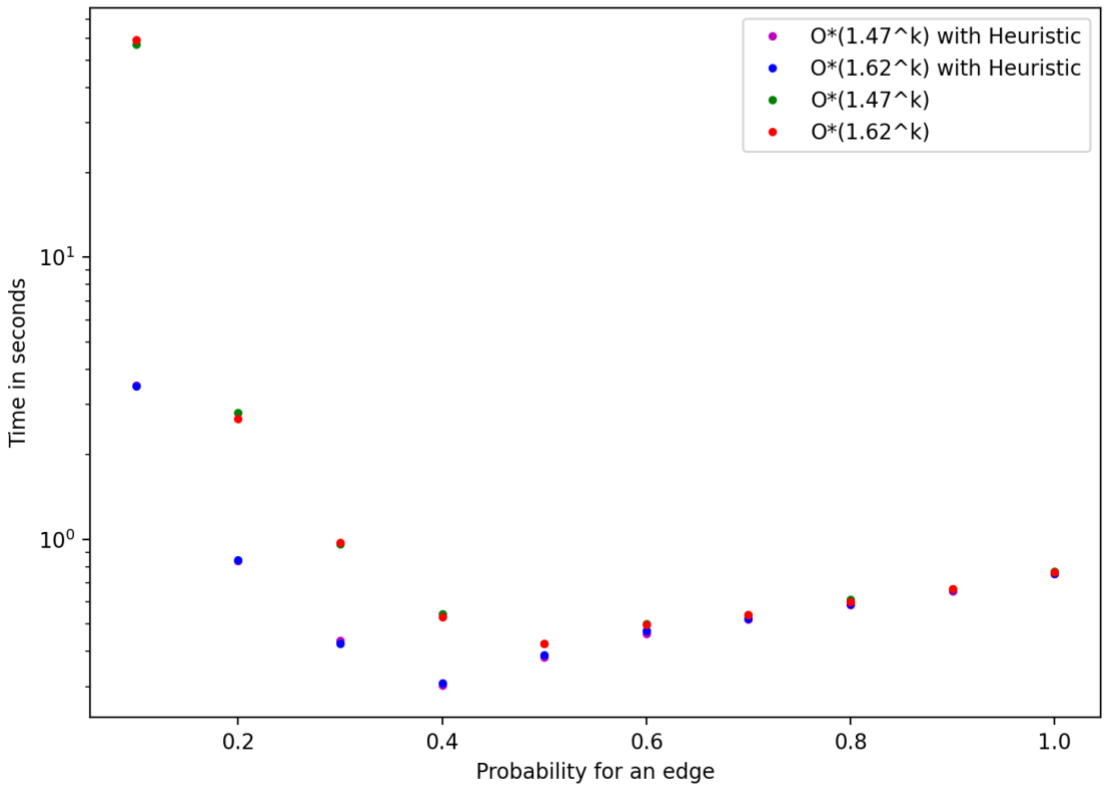
Experiment 7

In this experiment we will try another heuristic. Here we try to change the last rule in each algorithm. Instead of randomly choosing a vertex from the graph, we will choose the vertex with maximum degree. In theory, it doesn't change the runtime, but we think using this technique we will reduce the graph faster in practice which will yield us better results.
We will run similar experiment to the first one, where we randomly generate graphs and run for each probability from 0.1 to 1 all the algorithms.
The algorithms we run here will be the 2 algorithms from the book, $O^*(1.47^k)$ and $O^*(1.62^k)$ time complexity.
And 2 new algorithms with heuristics, $O^*(1.47^k)$ where in the last rule we choose maximum degree and $O^*(1.62^k)$ where in the last rule we choose maximum degree.

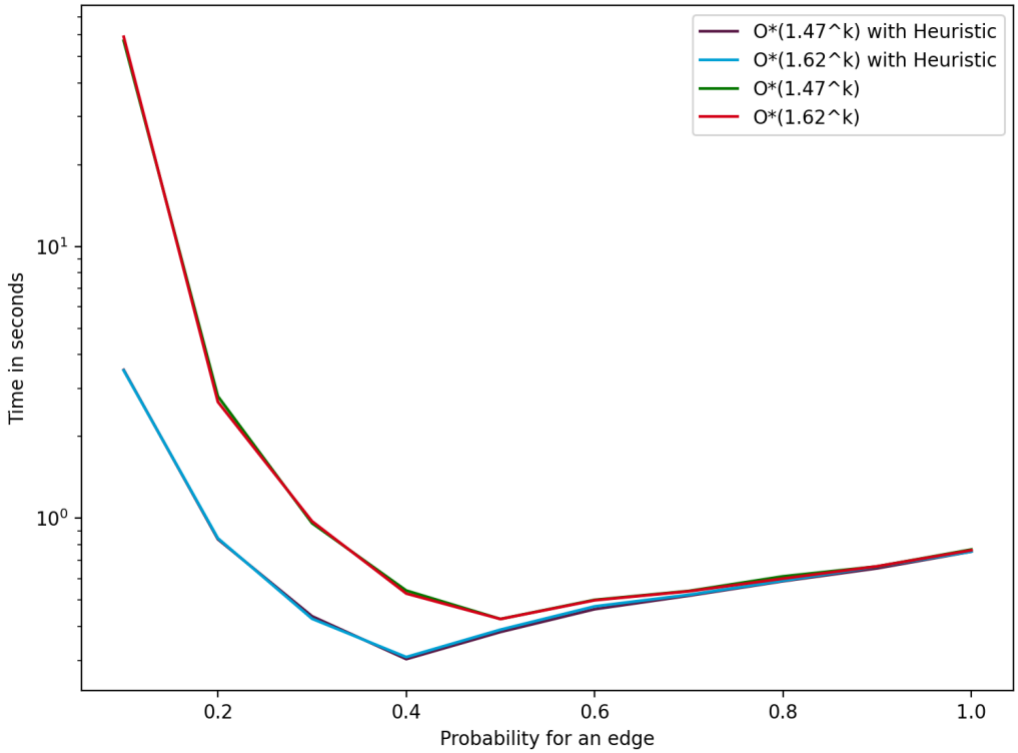
Here we take different approach from previous heuristic where we don't try to combine both algorithms to yield better results, instead we look at each algorithm independently and try to improve it in practice.

The results:

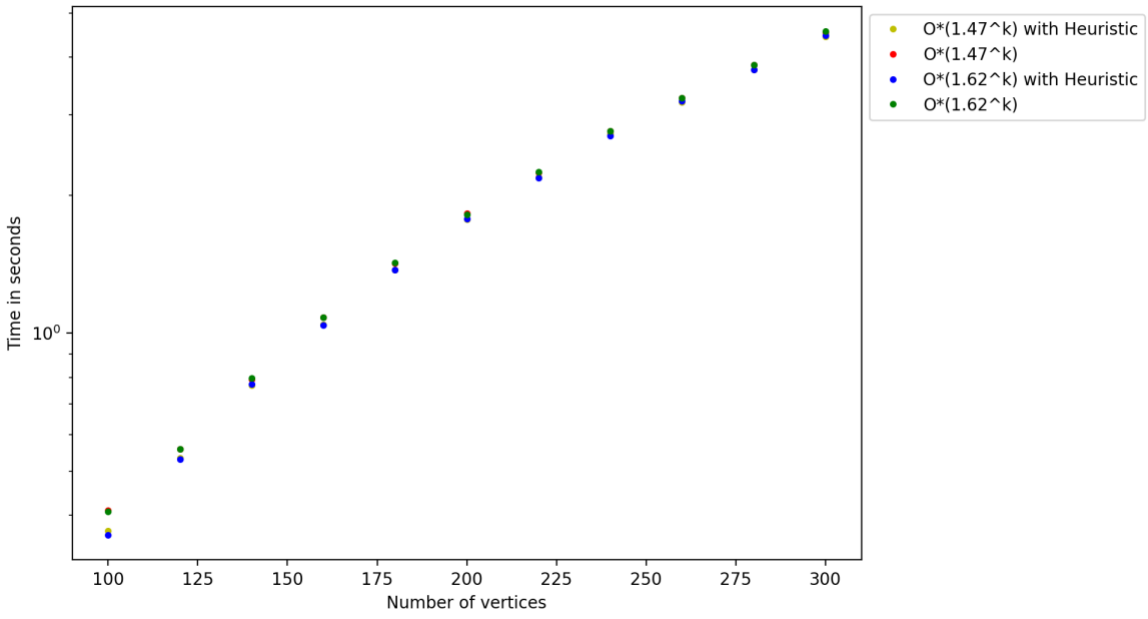
Plot 1: Running all algorithms where changing p.



Plot 2:



Plot 3: Running all algorithms with fixed p=0.5, where we change the number of vertices in each time.



Conclusions:

- First, although the graphs are printed in logarithmic scale, it is hard to see the difference between the $O^*(1.47^k)$ algorithm with the heuristic to the $O^*(1.62^k)$ algorithm with the heuristic. The yellow and the blue dots are almost the same. For example, if we look at p=0.5 in the first plot, the average results for 50 iterations were:
  - $O^*(1.47^k)$ time avg for p = 0.5 is 0.4258892083400497
  - $O^*(1.62^k)$ time avg for p = 0.5 is 0.4256775086799462
  - $O^*(1.47^k)$ time with heuristic avg for p = 0.5 is 0.38153727662001985
  - $O^*(1.62^k)$ time with heuristic avg for p = 0.5 is 0.3890873697200368

    We see how close the results. That is for almost every p we take.
- From plots 1, 2, we learn that as the graph is sparse, we get very good results with the heuristic, for both algorithms. When the graph becomes dense (i.e. for $p \geq 0.5$) the results are similar to the regular algorithms and there is no significant improvement by the heuristic. Thus, we would recommend this kind of heuristic for sparse graphs with liner number of edges.
- From plot 3, we observe that now for p=0.5 and number of vertices is growing we get that run time is getting similar, the heuristic improves the runtime in practice only slightly. We get same results as the experiment for changing p, that is to say, it is better be used for sparse graphs.
- As we learnt in previous experiments, there is no big difference between the runtime in practice of both algorithms ($O^*(1.47^k)$, $O^*(1.62^k)$), and here we witness to same results when we use the same heuristic for both algorithms. We learn that using the heuristic on each algorithm yields us same results.
- For further work on this topic, we would like to try and come up with better heuristic with this approach of choosing a vertex and find out maybe eventually we could say that one algorithm is better than the other also in practice.

To repeat this experiment run:

```
probability = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]
k = 45
v = 100
for p in probability:
    for i in range(1, 51):
        graph_1 = nx.erdos_renyi_graph(v, p)
        start = timeit.default_timer()
        VertexCoverAlg3.run_vertex_cover_alg_3(graph_1, k) #run algorithm O*(1.47^k)
        stop = timeit.default_timer()
```

```
        start = timeit.default_timer()
        VertexCoverAlg2.run_vertex_cover_alg_2(graph_1, k) # run algorithm O*(1.62^k)
        stop = timeit.default_timer()
        start = timeit.default_timer()
        VertexCoverAlg3Heuristic.run_vertex_cover_alg3_heuristic(graph_1, k)
        stop = timeit.default_timer()
        start = timeit.default_timer()
        VertexCoverAlg2Heuristic.run_vertex_cover_alg_2_heuristic(graph_1, k)
        stop = timeit.default_timer()
```

```
vertices = numpy.arange(100, 320, 20)
k = 45
p = 0.5
for v in vertices:
    for i in range(1, 51):
        graph_1 = nx.erdos_renyi_graph(v, p)
        start = timeit.default_timer()
        VertexCoverAlg3.run_vertex_cover_alg_3(graph_1, k) #run algorithm O*(1.47^k)
        stop = timeit.default_timer()
        start = timeit.default_timer()
        VertexCoverAlg2.run_vertex_cover_alg_2(graph_1, k) # run algorithm O*(1.62^k)
        stop = timeit.default_timer()
        start = timeit.default_timer()
        VertexCoverAlg3Heuristic.run_vertex_cover_alg3_heuristic(graph_1, k)
        stop = timeit.default_timer()
        start = timeit.default_timer()
        VertexCoverAlg2Heuristic.run_vertex_cover_alg_2_heuristic(graph_1, k)
        stop = timeit.default_timer()
```