Random Graph Generation

> Properties:
>
> - All vertices have an equal chance to be connected to any other vertex
> - Any possible graph can be achieved through the algorithm
> - Has to be connected

Graph generation consists of two main parts, I will examine each in a separate section. First challenge of random graph generation is the ability to create a random spanning tree which ensures that all of the vertices are connected. Second challenge is adding elements to the spanning tree to the desired density.

Random Spanning Tree Generation

> Inductive building

To ensure that the graph stays connected, we start with a connected graph of size 1 and each subsequent vertex is connected to any part of the graph. We can show by induction that this results in a connected graph.
An easy way of doing this is to start adding elements in a sequence. Any vertices which have been already added can be chosen as the opposite vertex to form an edge.
This method is great to ensure that all vertices are connected. However there are issues with distribution of edges. Since vertices which are added sooner have a higher chance of being connected to, the first 2 properties are violated. Further I will expand on this algorithm.

> Chain Building

This is a very efficient way of building a spanning tree which ensures that any vertex has nearly the same probability to be connected to any other vertex. The method is to create an array of all the vertices, shuffle the array and the sequence of the vertices in the array now form a random path through the list which traces out a spanning tree.
This method is however violating the second property, because all graphs where any vertex has a degree > 2 is impossible to achieve.

> Inductive building with equal probabilities

This method is very similar to the inductive building, except it favors the last item added to the graph in order to spread out the number of chances each vertex gets to get picked. This is the algorithm I ended up implementing and using to build a spanning tree.

Below is a visualization of my reasoning to calculate the probability using `random.randint()` linear distribution. Suppose all elements in the array except a last one had an equal chance to be picked. Since the first element added needs to have a chance to be picked every iteration. The chance each element needs to have is the same as the first element. So when adding nth element, pick from sequence $1, 2, \ldots (n-2), (n-1), (n-1), \ldots, (n-1)$ where $(n-1)$ is repeated $n-1$ times a random element.

```
ADD 2
1 <- pick random element from this sequence

ADD 3
1
122 <- pick random element from this sequence

ADD 4
12
12333 <- pick random element from this sequence

ADD 5
123
123
1234444 <- pick random element from this sequence

ADD 6
1234
1234
1234
123455555 <- pick random element from this sequence
```

This can be implemented in this function:

```
def random_vertex_inductive(n):
        x = random.randint(1, 2*n-1)
        if x < n:
                return x
        return n
```

Adding Edges

> Random vertex pairs

We begin by having a list of all vertices. After our spanning graph was created we can start adding edges to the graph. This is done by picking 2 random indexes which correspond to 2 random vertices from the vertex list. We try to add them to the graph, which might fail if the

vertices are the same or the graph already has that edge added to it. This initially is not an issue as we can just pick another set of indexes and most likely we would encounter a valid edge to add in expected constant time. This approach starts to be slowed downed significantly when we start reaching dense graphs. The probability becomes astronomical for finding the last vertex to add, completing a fully connected graph would be nearly impossible, since the chance to get the last edge is $n^2/2$, we would need to try roughly $n^2/4$ combinations.
This would be unusable, so if we would want to improve on the time it takes to find a correct vertex we could implement linear probing, were we simply advance to the next index.

```
def rollIndexes(indexes, n):
        i1, i2 = indexes
        i1 = (i1+1)%n
        if i1==0:
                i2 = (i2+1)%n
        return i1, i2
```

However this might result in clusters which would mean that the first property is violated.

Aside: this method could be used by first creating a fully connected graph and then removing edges, while keeping the original spanning tree intact

> Generating a list of all possible edges

We first generate a list of tuples of 2 vertices, where each tuple represents an edge. We have to make sure not to allow for duplicate edges since order in tuples matters but we are trying to build an undirected graph.

```
def generate_full_connections(n):
        conns = []
        for i in range(n-1):
                for j in range(i+1, n):
                        conns.append((i, j))
        random.shuffle(conns)
        return conns
```

I store this list and also keep an index which advances through the list and I store that one as well in the Graph Generator object. This allows to add more edges if needed or remove edges if needed. Of course I need to keep checking if the edge I am about to remove was not in the spanning tree which I keep as a set. Or in case of adding edges I need to check if the edge is not already in the graph from the spanning tree. These are okay checks as the take constant time per edge and they appear overall only $n-1$ times at max.

I chose this technique to generate all graphs. Even thought the random graph generation is faster when it comes to dense extremes (very dense for random removal or very spare for random addition) But the random removal is very close in time complexity to this method.

Both methods would generate a fully connected graph which is already $O(n^2$ which would be the main complexity regardless of how many edges we add or remove later. And generating sparse graphs might take $O(n^2)$ for this technique however since it is a sparse graph the creation is quite fast either way.

Weights

Weight is the third random factor in graph creation. It might seems insignificant at first but further analysis of the prim's algorithm with a binary heap shows that the range of values we allow the weight to take on is going to determine the efficiency of our algorithm to the point where we might get $O(m + C * \log_2 n)$ where $C \in \mathbb{R}$. C might end up being dependent on the weight range we allow.

Prim's Implementation

I implement the prim's algorithm using pseudocode from the lecture notes. Both adaptable priority queues are being used by identical code. Polymorphism is used here to achieve similar testing conditions

```python
    def prim(self, APQ_type):
        pq = APQUnsortedList()
        if APQ_type == "heap":
            pq = APQBinaryHeap()
        locs = {}
        for vertex in self.vertices():
            element = pq.add(float("inf"), (vertex, None))
            locs[vertex] = element
        tree = []
        while pq.length() > 0:
            value = pq.remove_min()[1]
            vertex, entry_edge = value
            locs.pop(vertex)
            if entry_edge:
                tree.append(entry_edge)
            for edge in self.get_edges(vertex):
                opposite_vertex = edge.opposite(vertex)
                if locs.get(opposite_vertex):
                    cost = edge.weight
                    element = locs[opposite_vertex]
                    if cost < pq.get_key(element):
                        element.value = (opposite_vertex, edge)
                        pq.update_key(element, cost)
        return tree
```

> Tests

Quick test to see if the algorithm failed is the following

```
assert len(tree)==(n-1)
```

However this test does not prove that our algorithm is working properly. To check the working of the algorithm I ran it on some smaller samples and then on larger graphs, pasted the results into https://edotor.net/ to check if the graph was indeed a spanning tree and preferred lower value edges.
For this testing there is a function in `graph_test.py` file which prints out the formatted graph to be checked together with the spanning tree.

Unsorted List

Implementing the APQ using an unsorted list is quite straightforward. We are aiming for `update_key()` and `remove()` to be $O(1)$ and `remove_min()` to be $O(n)$.
We achieve the constant access and update time by not keeping any kind of structure inside of the underlying array and storing the index in the element objects themselves (These need to updated anytime they would change position). The user of this class has to manage the element objects themselves, since they need to passed into method like `update_key()` and `remove()` .
Linear time for removing the minimum element is done by searching the whole list and removing the element which had the highest priority.

This implementation is efficient for algorithms where updating and removing from the APQ is preferred or more frequent than removing the minimum.

The complexity of this algorithm is $O(n^2 + m + n)$, and since for a fully connected graph $m = \frac{n*(n-1)}{2}$ we can see that the complexity is $O(n^2)$.

Binary Heap

Similarly the binary heap is storing elements which keep their own index inside the array which represents the binary heap. However this time, when we update or remove we are forced to rebalance the heap, which takes $O(\log n)$. Same complexity applies to removing a minimum element.

When applied to the prim's algorithm, we should try and avoid performing any actions on the APQ as all of them have the disadvantage of rebalancing the heap. It is also worth nothing that for smaller graphs the cost of rebalancing is proportionally higher then for larger ones where the logarithmic growth creates an advantage over for example linear growth.

The time complexity of prim's when using this data structure is $O(n\log n + n\log n + m\log n)$ and since generally $m > n$ the complexity becomes $O(m\log n)$. This means that the performance of this algorithm is heavily dependent on the density of the graph.

Analysis

I will analyse the runtime of the Prim's Algorithm with basic runtime complexities and check the predictions practically. After collection of data I will analyse the trends and improve the theoretical knowledge based on the data.

Predictions

From the analysis of runtime complexities we can notice that heap should perform better on sparse graphs while the unsorted list should overtake it the denser the graph is.
More specifically at $m = \frac{n^2}{\log 2}$.

We can use 3d graphing tools like Desmos to help us plot the the correlation between vertices, edges and the runtime.

When interrogating further, we can take the limit of the ratio of edges where Heap would perform better to the total number of vertices.

$$\lim_{n\to\infty}(\frac{\frac{n^2}{\log n}-(n-1)}{\frac{(n-1)n}{2}-(n-1)})$$

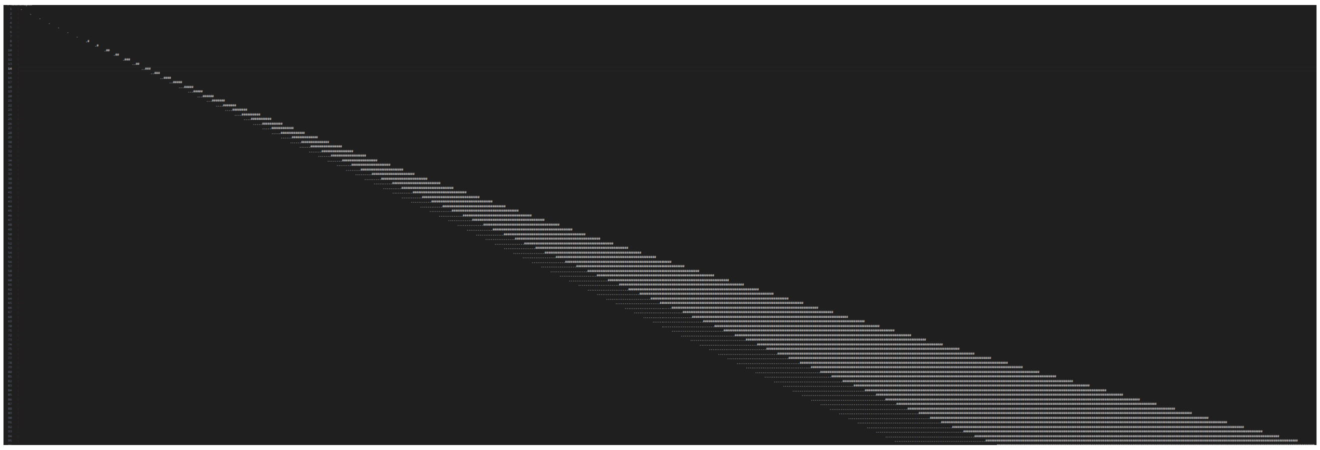
This can be simplified down to:

$$\lim_{n\to\infty}(\frac{2n^2}{n^2 * \log n}) = \lim_{n\to\infty}(\frac{2}{\log n}) = 0$$

We can see that the ratio of possible edges on which the unsorted list should be better is growing faster than heap. However we will never reach a point where the unsorted list would perform better on a spanning tree.

Suppose:
$\exists N \in \mathbb{N} s.t. \forall n > N \wedge (n)\log n > n^2$
Then follows:
$\log n > n$
Which is a contradiction, thus heap should always be better on sparse graphs, however we know that unsorted list should start taking over larger portions of the list.

To visualize this, I have created a simple graphing function which plots different graphs as coordinates in a text file and prints a specified character depending on which one is better for that graph



Practical Testing

I run simple tests on a range of vertices. For one set of vertices I create a given number of slices from a range of values $(n-1; \frac{n*(n-1)}{2})$ endpoints inclusive. These slices give me a rough idea of how the algorithms behave across a spectrum of vertices and densities. Each slice gives me one graph which I measure the performance of Heap Prim and Unsorted List Prim.

results

```
Vertices=   256 Edges=   255    Heap=0.00000000 Unsorted List=0.00591731
Vertices=   256 Edges=  3494    Heap=0.00904083 Unsorted List=0.00690007
Vertices=   256 Edges=  6733    Heap=0.00675368 Unsorted List=0.00215816
Vertices=   256 Edges=  9972    Heap=0.00769448 Unsorted List=0.01296353
Vertices=   256 Edges= 13211    Heap=0.01459694 Unsorted List=0.01482916
Vertices=   256 Edges= 16450    Heap=0.01349425 Unsorted List=0.02640629
Vertices=   256 Edges= 19688    Heap=0.01801300 Unsorted List=0.01171160
Vertices=   256 Edges= 22926    Heap=0.02000141 Unsorted List=0.01867127
Vertices=   256 Edges= 26164    Heap=0.01984072 Unsorted List=0.01929164
Vertices=   256 Edges= 29402    Heap=0.01780105 Unsorted List=0.02702188
Vertices=   256 Edges= 32640    Heap=0.02942586 Unsorted List=0.02210474
Vertices=   512 Edges=   511    Heap=0.00000000 Unsorted List=0.01873016
Vertices=   512 Edges= 13542    Heap=0.02129960 Unsorted List=0.02161527
Vertices=   512 Edges= 26573    Heap=0.02410030 Unsorted List=0.02698421
Vertices=   512 Edges= 39604    Heap=0.03124380 Unsorted List=0.04017973
Vertices=   512 Edges= 52635    Heap=0.04850054 Unsorted List=0.04719377
Vertices=   512 Edges= 65666    Heap=0.05140662 Unsorted List=0.05749321
Vertices=   512 Edges= 78696    Heap=0.05997229 Unsorted List=0.07684660
Vertices=   512 Edges= 91726    Heap=0.07141519 Unsorted List=0.09285998
Vertices=   512 Edges=104756    Heap=0.07698655 Unsorted List=0.10003662
Vertices=   512 Edges=117786    Heap=0.11284113 Unsorted List=0.11713958
Vertices=   512 Edges=130816    Heap=0.11037731 Unsorted List=0.11667442
```

```
Vertices=  1024 Edges=  1023    Heap=0.01284027 Unsorted List=0.05599833
Vertices=  1024 Edges= 53299    Heap=0.05762053 Unsorted List=0.09098458
Vertices=  1024 Edges=105575    Heap=0.09191561 Unsorted List=0.13834524
Vertices=  1024 Edges=157851    Heap=0.14905000 Unsorted List=0.20028639
Vertices=  1024 Edges=210126    Heap=0.21337247 Unsorted List=0.26537108
Vertices=  1024 Edges=262401    Heap=0.26798677 Unsorted List=0.33139110
Vertices=  1024 Edges=314676    Heap=0.33599973 Unsorted List=0.37092638
Vertices=  1024 Edges=366951    Heap=0.46236229 Unsorted List=0.47556877
Vertices=  1024 Edges=419226    Heap=0.51710606 Unsorted List=0.56461382
Vertices=  1024 Edges=471501    Heap=0.56991053 Unsorted List=0.60084701
Vertices=  1024 Edges=523776    Heap=0.62815285 Unsorted List=0.68886137
Vertices=  2048 Edges=  2047    Heap=0.02011585 Unsorted List=0.21645069
Vertices=  2048 Edges=211456    Heap=0.22594357 Unsorted List=0.43698263
Vertices=  2048 Edges=420864    Heap=0.58498549 Unsorted List=0.70193315
Vertices=  2048 Edges=630272    Heap=0.74917006 Unsorted List=0.93990946
Vertices=  2048 Edges=839680    Heap=1.06798840 Unsorted List=1.19983196
Vertices=  2048 Edges=1049088   Heap=1.38133645 Unsorted List=1.57970238
Vertices=  2048 Edges=1258496   Heap=1.54128933 Unsorted List=1.71995401
Vertices=  2048 Edges=1467904   Heap=1.88867259 Unsorted List=2.02714539
Vertices=  2048 Edges=1677312   Heap=2.04961419 Unsorted List=2.21526217
Vertices=  2048 Edges=1886720   Heap=2.31984258 Unsorted List=2.47129464
Vertices=  2048 Edges=2096128   Heap=2.81836367 Unsorted List=2.72852230
Vertices=  4096 Edges=  4095    Heap=0.03884435 Unsorted List=0.89964724
Vertices=  4096 Edges=842342    Heap=1.07939148 Unsorted List=1.84855604
Vertices=  4096 Edges=1680589   Heap=2.28786206 Unsorted List=3.15204215
Vertices=  4096 Edges=2518836   Heap=3.19852924 Unsorted List=3.97246814
Vertices=  4096 Edges=3357083   Heap=4.39886260 Unsorted List=5.04938507
Vertices=  4096 Edges=4195330   Heap=5.38085079 Unsorted List=6.03921032
Vertices=  4096 Edges=5033576   Heap=8.17561269 Unsorted List=8.53456688
Vertices=  4096 Edges=5871822   Heap=8.00880814 Unsorted List=8.33761835
Vertices=  4096 Edges=6710068   Heap=22.72403622        Unsorted
List=21.23875761
Vertices=  4096 Edges=7548314   Heap=14.27042317        Unsorted
List=25.72153616
Vertices=  4096 Edges=8386560   Heap=13.26482844        Unsorted
List=12.08679676
```

Observations

- We can notice that the for sparse graphs, especially the minimum spanning tree, the heap implementation performs much better, this is something we would expect from the complexity analysis.

- We can also see that the jump between vertices with the same density follows $O(n^2)$ growth. Each step in my testing is two times the previous vertices count and we can see that the runtime jumps by roughly 4 times.
- However the main surprise is that our predictions have not been confirmed. It seems that heap is outperforming the unsorted list even on very dense graphs, sometimes even on fully connected graphs.
    - This should be the direction of our enquiry as deciding which APQ to use is crucial for optimal runtimes.

Further Analysis

Firstly we need to find an explanation for heap's improved performance when it comes to dense graphs. There has to be some form of optimization in this pseudocode which went unnoticed.

```
prim():
        create an APQ pq, to contain costs and (vertex,edge) pairs
        create an empty dictionary locs for locations of vertices in pq
        for each v in G
                add (¥, (v,None)) into pq and store location in locs[v]
        create an empty list tree, which will be the output
        while pq is not empty
                remove c:(v,e), the minimum element, from pq
                remove v from locs
                if e is not None, append e to tree
                for each edge d incident on v
                        w = d's opposite vertex from v
                        if w is in locs # and so it is not yet in the tree
                                cost = d's cost
                                if cost is cheaper than w's entry in pq
                                        replace ?:(w,?) in pq with cost: (w,
d)
        return tree
```

We can see that the section of the code which is responsible from the runtime $O(m \log n)$ is:

```
while pq is not empty
        remove c:(v,e), the minimum element, from pq
        remove v from locs
        if e is not None, append e to tree
        for each edge d incident on v
                w = d's opposite vertex from v
        if w is in locs # and so it is not yet in the tree
                cost = d's cost
                if cost is cheaper than w's entry in pq
                        replace ?:(w,?) in pq with cost: (w, d)
```

Note that the sum of degrees of all vertices is $2 * m$, only m make it through `if w is in locs`

Here the while loop runs precisely n times while the inner loop runs for every edge on the vertex v.
We only update the APQ `if cost is cheaper than w's entry in pq`. This means that the actual runtime is $O(m(1 + c \log n))$. This is interesting because if c is a not a constant but a function of m, specifically $\frac{1}{m}$ in some form, the overall runtime might be smaller.

So what is c? c is the probability that the current examined cost is cheaper than the cost in the entry in the APQ. Since it is a probability, it helps to formulate it as a number of possibilities we want: overall number of updates: u over the overall number of edges m

> $$c = \frac{u}{m}$$

Defining u is tricky since it relies on probability, but I find it helpful to formulate in terms of a single vertex. Each vertex has a degree, this average degree of vertices is $2 * m/n$, but only half of the edges are examined, so m/n. The reason I define the average degree here is because we want to create a set of all possible unique elements the vertex has access to. This is helpful because over the course of the algorithm, as long as it is random, we will keep inspecting elements from this set. We only update the APQ if the element we inspect is lower than the previous one.

If we assume that, on average, the set is split into two by a random item being put inside the APQ, then that creates two equal subsets, where only one will result in the APQ being updated. This splitting of the set continues. So if we assume that a vertex will have m/n inspections, then the first inspection has a 100% chance to update APQ, second one 50% and so on these probabilities cascade like in a Pascal's triangle where let's say the most left means all were successful updates and the further right you go, the less updates to the APQ have been made.

> If we allow for rough estimates as I was not able to find a proper probability: The overall probability for degree m/n is $\log n$ roughly.

So now we know that the set can split, now the question is what is size of the set that is being split. Well it is a set of all possible unique elements which can be found on average. First of all there are two limiting factors, if I have 20 weight options, then I can only update a vertex 20 times at max ,at best 1 time and on average $log_2 20$ times. Similarly I cannot update a vertex more time than it has connections, so $\log_2 m/n$ or $\log_2 w$ per vertex updates. Where

w is a weight. We need to pick the smaller one of these because whichever one is smaller will determine what new item cannot be selected anymore.

> $\min\{\log(m/n), log_2 w\}$

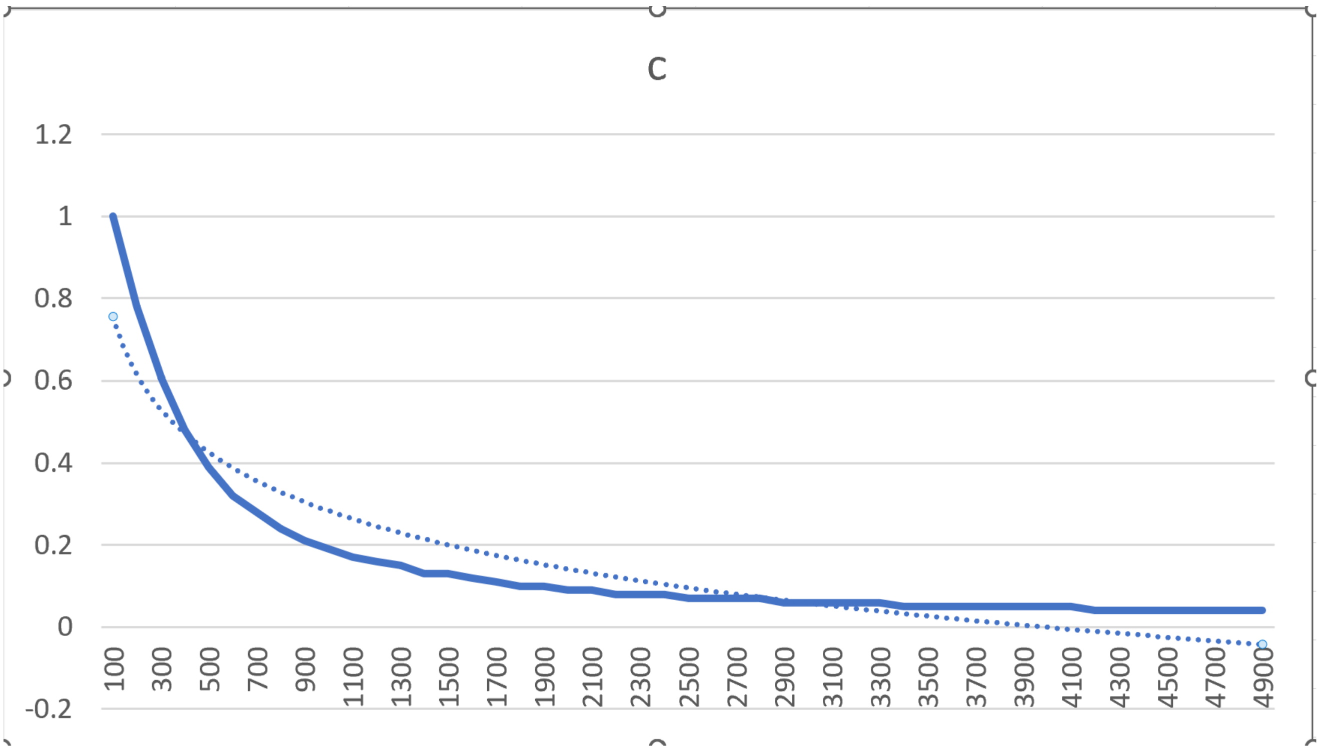
Putting all of this together;

> $c = \frac{n * \min\{\log(m/n), log_2 w\}}{m}$
> $m(1 + c\log n) = m + m * c\log n = m + n * \min\{\log(m/n), log_2 w\} * \log n$
> $m + n * \log n * \min\{\log(m/n), log_2 w\}$

So if we have the weight set to a small number, like in our test cases where $w = 20$, we can view it as a constant and get $O(m + n\log n)$ as the time complexity for the heap. This would explain why the run times come so close to each other when it comes to Dense Graphs, they are both technically $O(n^2)$.

Further Testing

To ensure my reasoning is correct and the math is giving a good approximation, I measured the total number of times the line: `if cost is cheaper than w's entry in pq`. It was m and than the total number when the if statement was True has been tested. I have plotted the data with 100 vertices and variable number of edges. We can see that it most likely follows a logarithmic curve.


C
1.2
1
0.8
0.6
0.4
0.2
0
-0.2
100
300
500
700
900
1100
1300
1500
1700
1900
2100
2300
2500
2700
2900
3100
3300
3500
3700
3900
4100
4300
4500
4700
4900