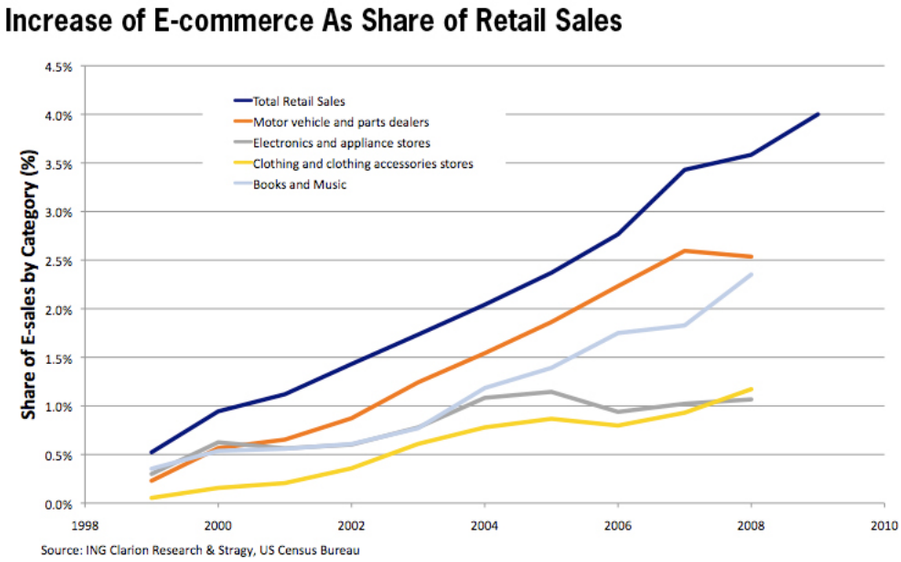
CS182 Final Report

Sami Ghoche, Andrew Malek, Devvret Rishi

December 2014

1 Introduction

We propose "Tailr" a novel service that recommends clothing sizes to online shoppers. The E-commerce industry in general has exploded in recent years, and the apparel and accessories E-commerce industry has been no exception. However, the apparel industry has not reached the heights that other industries have and that many predicted it would.



One major cause for this is that people are still reluctant to purchase clothing online mainly because clothing sizes are not always reliable and vary depending on manufacturer, brand, model, etc. Ideally, people would want to try out their clothes before they buy them, just like they would at a normal retail store. This is precisely what Tailr tries to do. While we obviously can't let people virtually try out their clothes, we can leverage the opinions of other people who have already purchased the same clothes in the past, in order to generate predictions tailored for each specific user.
In order to achieve this goal, we will ask people who buy clothing items to rate them in terms of fit, with ratings ranging from 0 to 100, with 50 meaning a perfect fit, lower values meaning increasingly small fit, and higher values meaning increasingly large fit. Then we will build a graph where nodes represent clothing items with a specific size (so there are many nodes for each clothing item, one for each size). Two nodes will be connected when at least one person has purchased and rated both nodes.

Consider the following canonical example of how this system will work (Here we will assume for simplicity that clothing items have only one size):

Alice has purchased clothing item A and given it a rating of 40, and clothing item B and given it a rating of 60.
Bob is looking at purchasing clothing item A, he has already purchased clothing item B and given it a rating of 50.
We already know that A and B are linked with a difference going from B to A of -20, so we can predict that Bob will give A a rating of 50-20 = 30

This kind of prediction is useful because when there are many sizes of the same item, we can try to find the size of that item with the predicted rating closest to 50 (perfect fit) and return that size.

When building our graph, we will place a confidence value on each link between two nodes, where the confidence value represents how confident we are of the difference of ratings on the link. This confidence value will be used as a form of cost when traversing our graph in order to generate predictions (where higher confidence means lower cost). Intuitively, this confidence value for a link will depend on how many people have purchased and rated both nodes, as well as the standard deviation of the rating differences.

Thus, when a user is asking for a prediction for item A, we can reduce the problem to traversing the graph starting at A in search of a path to some node that this user has already purchased and rated.

In this paper, we will compare five different graph traversal algorithms for generating our size prediction:

1. Uniform Cost Search
2. A-Star Search with a non-admissible heuristic
3. K-directional Uniform Cost Search
4. K-Perimeter Search with a non-admissible heuristic
5. Beam Search

We will compare these five algorithms with respect to accuracy of the generated prediction, as well as computational efficiency in terms of nodes expanded.

2 Problem Modeling

As discussed above, we are building a graph of clothing items, where two nodes are connected when at least one person has purchased and rated both. In our representation, a node will represent a specific size of a specific item.

2.1 Nodes and Edges

Each node will have a unique identifier, a brand it belongs to, and a size (out of a 100).
Each edge will have a number of ratings (number of people who have bought both nodes), a mean of differences (of ratings each person has given them), a standard deviation of these differences, a confidence value, a cost value, and a flag indicating whether these two nodes are different items or different sizes of the same item.

2.2 Edge Confidence and Cost

When computing confidence values from number of ratings n and standard deviation σ, we thought of the following function:

$$confidence = max(min(0.50 * \sqrt{n}, 0.999999) - (\sigma/3.0) * 0.01, 0.01)$$

We thought this function made intuitive sense because as the number of ratings increases, we should become more confident of our difference of ratings, but our increase in confidence should slow down (which explains the square root). On the other hand, when the standard deviation of the differences of ratings increases, we should become less confident of our difference of rating because we cannot predict which difference of ratings would apply nicely to this user. The weights in this function are arbitrary and help generate predictions that match people's intuitions and remain within a range from 0 to 1.

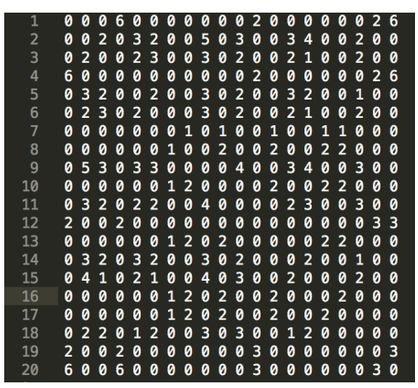
Confidence values are useful because we can aggregate all of them along a path by multiplying them and thus obtaining the confidence of an entire path. However, multiplication is not supported by traditional search algorithms, so we thought of storing a cost for each edge where cost = -log(confidence). The rationale is that $log(a) + log(b) = log(a * b)$ so we can use addition (which is supported by search algorithms) in order to simulate our confidence multiplication. The negative is just to avoid negative numbers (which lead to cycles and infinite minimization) and to turn the problem into cost minimization (thus confidence maximization).

2.3 Simulation of Purchases

Before we could start running and testing our prediction algorithms, we had to generate a realistic clothing graph. Here are the steps we took and the assumptions we made:

- We have four variables that we fix before building our graph: Number of users, number of items, number of purchases, number of brands, and number of sizes per item.
- We generate brands, then we generate items. For each item, we generate "number of sizes per item" nodes (one for each size), and randomly assign each to them increasing sizes.
- We generate users, and randomly assign to each user a true size, then randomly select which items each user should purchase, while making sure of four things:
    1. A user never buys the same node or different sizes of the same item twice.
    2. A user only buys nodes whose sizes are within 20 points of the user's true size.
    3. When a user buys a node, we introduce a small element of randomness in the rating that the user gives the item, so as to reflect the real world where people might be impulsive and not rate things extremely accurately.
    4. A user disproportionately prefers to purchase items from a few select brands. (We believe this fact accurately models real world behavior, and will be very useful for our non-admissible heuristic later on)

    Here is a dump of the number of ratings on the edge between each two nodes of a small graph (15 users, 20 items, 70 purchases, 3 brands and 1 size per item)



2.4 "Fake" Edges

In the real world, when a user rates size "small" of an item, this gives us useful information about how this user would have rated size "medium" of the same item, because we know that "medium" is some number of rating points larger than "small". Right now, our graph does not incorporate this information in any way. In order to extract as much information from every rating as possible and to ensure that our graph traversals don't fail too often in finding paths from a start node to a goal node, we created the notion of "fake" edges: edges between different sizes of the same item. In our implementation, these fake edges never get updated, and have a difference of ratings equal to their difference in sizes, and an arbitrary confidence of 0.7.

3 Algorithms

3.1 Uniform Cost Search

This algorithm takes in two arguments, a user and the node he is asking for a prediction for. UCS starts from this node and runs normally. The distance in the algorithm is represented by the costs of edges (which are in turn based on the confidences). Once we reach a goal node, (a node that the user has already purchased), we stop pushing nodes on the priority queue. At this stage, we have an optimal path from the start node to the closest goal node. However, if we aggregate the differences along the path and add them to the user's rating of the goal, we might get a prediction that the start node had a bad size. Since the purpose of the algorithm is to return the best size item, we now look at taking a fake edge from the start node to another size of the same item. We compare taking all fake edges and pick the one that yields the best predicted rating (closest to 50). If we do end up taking a fake edge (if the original start node was not the best), we have to update the confidence by multiplying it by the fake edge confidence (0.7). We return the best node along with the predicted rating and the confidence.
We repeat this entire process starting from all other sizes of the same item. At the end, we return the result with the highest confidence.
In order to avoid unnecessary computation, we make sure not to push different sizes of the same item as the start node on the queue (since we are going to start from them eventually anyway). In fact, this repetition of the algorithm with every size of the start item as start node is something we do in all five of our algorithms.
This algorithm is complete and optimal, but it might expand many nodes especially for large graphs, trying to find the optimal path to a goal whereas another algorithm might find a confident enough path in a fraction of the nodes expanded.

3.2 A-Star Search With Non-Admissible Heuristic

This algorithm runs identically to UCS except that the cost function is now comprised of not only the costs of edges, but also the costs as determined by our non-admissible heuristic. First of all, the reason why we cannot build an admissible heuristic is because of the nature of the problem itself. When searching for a path from start to a goal, we do not know if we are getting closer to a goal, and we do not know the real cost ahead of time. Therefore, optimal heuristics are impossible in this model. However, we do have a non-admissible heuristic that is based on two features:

1. The number of outgoing edges from the node.
   We prefer to expand nodes that have many neighbors because they have a higher chance of leading us to a goal node in fewer expansions.

2. A measure of the correlation between brands.
   Whenever an edge between two nodes of different brands is created, we update (increase) the correlation between this pair of brands. We can therefore access this correlation in constant time when we need to. We prefer nodes that have a higher average correlation with the brands of all goal nodes. The idea here is that if the current node's brand has a high correlation to a goal node's brand, then it is more likely to lead to a node that has the same brand as the goal node. In turn, two nodes of the same brand

are likely to lead to each other since they have a high correlation (we keep track of correlations even between a brand and itself) because while building the graph, we made sure that users disproportionately prefer nodes from the same brands (so nodes from the same brand will mor e likely be connected). Therefore, a high average correlation is more likely to either lead to a node that can lead to a goal, or even lead directly to a goal.

This algorithm is complete but not optimal. We expect this algorithm to perform well on smaller graphs in which the start node is already close to a goal node and can find it without too many expansions. Otherwise on large graphs, especially since our heuristic is not rock solid, the algorithm might wander for a long time and expand many more nodes than UCS, but we suspect that its returned confidence will be high nevertheless.

3.3 K-Directional Uniform Cost Search

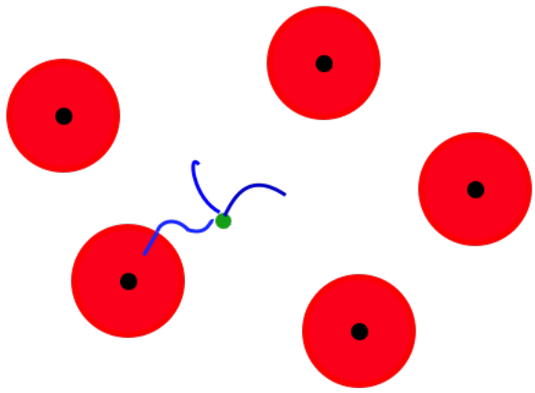
This algorithm is similar to UCS, except that we are expanding nodes from each goal node as well as from the start node. Each time we pop a node from the start, we check if it has already been expanded by a goal node. Each time we pop a node from a goal, we check if it has already been expanded by the start node. If this condition is satisfied, we have found an intersection. However, we cannot yet return this path as an optimal path because there might still be a better path that we will encounter in the future.

To illustrate this, consider the bidirectional UCS (k=2) where both the start node and the goal node have expanded all nodes of costs up to 5, and now find an intersection node with cost=5 from both sides. This path has a total cost of 10, but if we continue the algorithm, the start node might pop a node with cost 6 that has already been expanded by the goal node with cost 1. This path is more optimal, because it has a total cost of 7 as opposed to 10.

Therefore, we cannot stop at the first intersection node we find, we must keep expanding nodes from both sides until the sum of the lowest cost nodes on each priority queue is greater than the best found path cost so far. Aside from these distinctions, the rest of the algorithm is identical to UCS.
This algorithm is complete and optimal, and should expand fewer nodes than normal UCS, especially for larger graphs (where its overhead is compensated by the gains of k-directionality).

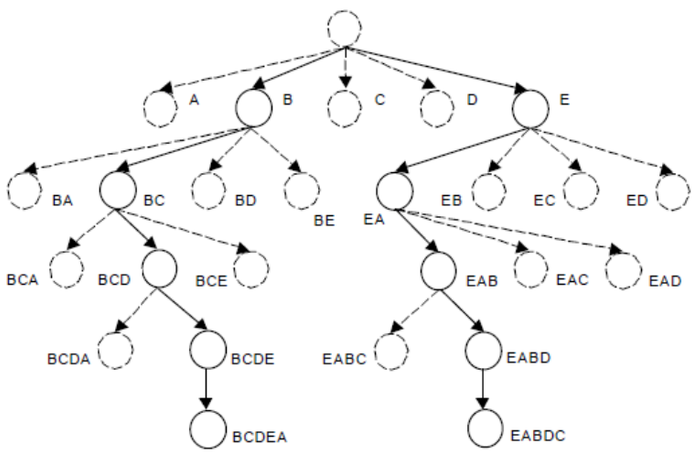
3.4 K-Perimeter Search With Non-Admissible Heuristic

We read a paper (Dillenburg and Nelson) that discussed a new algorithm they called perimeter search. In this algorithm, there is one start node and one goal node. They start running UCS from the goal node for costs lower than some threshold value, at the same time, they run A-star search with an admissible heuristic until an intersection is reached, and the path is returned.
For our purposes, we have many goal nodes as opposed to one, and we have a non-admissible heuristic as discussed above.
We run K-Perimeter search just like K-directional UCS, with UCS expansions from each goal node. At the same time, we run A-Star search from the start node with our non-admissible heuristic. When we encounter the first intersection, we return that path as the optimal one (we make sure to add a fake edge if needed just like we do in all other algorithms as discussed in UCS).
This implementation is complete but not optimal. We suspect that this algorithm will yield confidences close to optimal because of its k-directional UCS component, but it should also expand fewer nodes than k-directional UCS and normal UCS because of its heuristic that should guide it towards goal nodes quickly, as well as because of its greedy nature (taking the first intersection it finds).



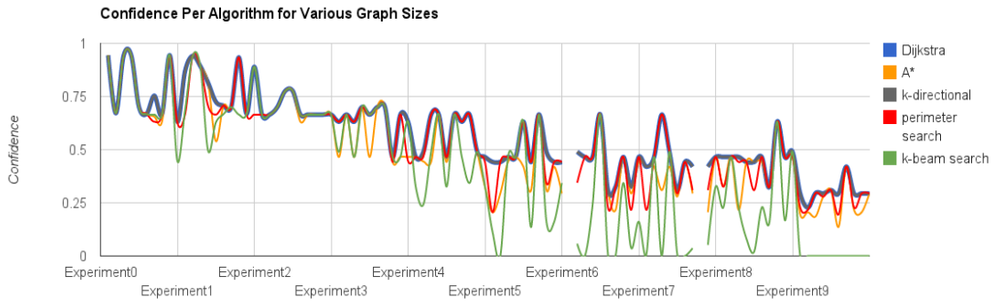
3.5 Beam Search

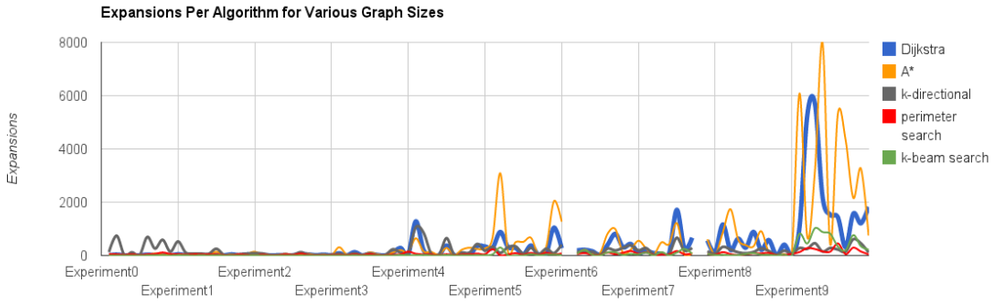
In this algorithm, we start from the start node, and expand the K lowest-cost nodes making sure to mark expanded nodes. We repeat this process until we find the first goal node, or until our queue is empty. If we find a goal node, we check whether we have to take a fake edge (just as discussed in UCS).
This implementation is neither optimal nor complete, but it promises to find goal nodes and return paths quickly. Furthermore, if a path we find is not optimal, we suspect that if the optimal confidence is high, that this algorithm likely won't return a much lower confidence, and will be good enough for our purposes here which are not to necessarily find the optimal confidence (especially since confidence is a contrived measure that might not be perfectly accurate), but mainly to find a good prediction with a high enough confidence.
We will experiment with different graph sizes and different values of K and examine our results closely.

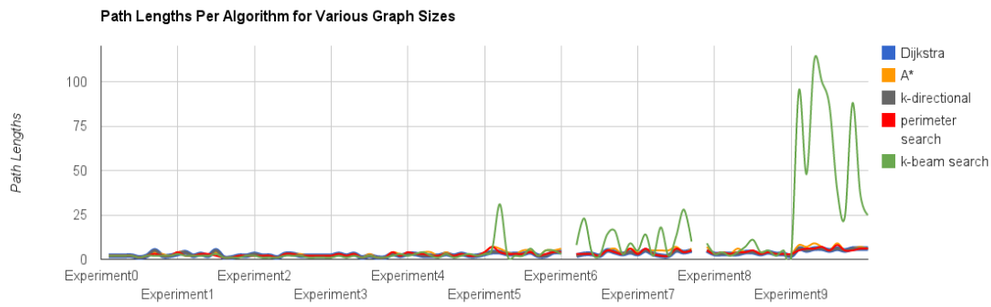


4 Testing Results

Experiments	1	2	3	4	5	6	7	8	9	10
num_users	20	40	80	150	200	400	400	500	1500	3000
num_items	60	90	200	500	700	1000	1100	1100	1200	3000
num_purchases	400	800	1400	2500	3000	4000	3000	3000	8000	10000
num_brands	3	5	8	8	10	12	8	12	8	10
nodes_per_item	2	3	4	5	5	5	3	5	4	5







4.1 Description

We ran 10 experiments, each consisting of building a graph and generating 10 predictions, each for a random user wanting to purchase a random item that they do not currently own. We see that all algorithms tend to return a lower confidence as our experiments progressed, and that is primarily because we are increasing the number of users and items in each experiment, which means our purchases are spread out over more users, and each user is less likely to buy an item that another user has bought. This means that the graph's edges have a lower number of ratings on average (since the number of ratings on an edge between two nodes represents the number of people who have bought both). A lower number of ratings causes a lower confidence value, thus explaining the monotonically decreasing confidence values. Furthermore, there are a few breaks in the chart lines, those mean that because the graph was so sparse, there was actually no path from the start node to any of the goal nodes, which means no confidence and no prediction.
As for the number of expansions, we notice that it also increases (for all algorithms) as our experiments progressed. Similarly, this is because now that our graph is very large, it is more unlikely for a path to just stumble upon the goal node, which means having to expand more and more nodes.
Finally, path length is generally constant for all of them (except beam search) with a slight increase again due to sparser graphs.

4.2 Uniform Cost Search

As expected, UCS returns the highest confidence (since it is optimal).
However, we notice that the number of expanded nodes spikes for the later experiments. As explained in the general comments this is to be expected for all algorithms. However, UCS's number of expansions rises a lot because it also has to make sure it finds the optimal

path to a goal node, which means examining all other paths on the way until it makes sure that the cost was optimal. This means expanding a lot of nodes, much more than all other algorithms (except A-Star as we will see).

4.3 A-Star Search With Non-Admissible Heuristic

In general, A-Star performs well with confidences since it returns values close to UCS's. This is because its heuristic component is not too big and does not affect A-Star's optimality too much, even though it is non-admissible. However, A-Star performs really badly with nodes expanded, especially for the larger graphs but even for small graphs, it performs slightly worse than UCS on average. This means that the heuristic is not good enough at guiding the start node toward the goal node if left on its own. Its behavior is very similar to UCS, but less optimal and less efficient in terms of expansions. There is therefore no reason to use this algorithm under any circumstances unless we can come up with a better heuristic or maybe generate the graph more realistically (with brands mattering a lot).

4.4 K-Directional Uniform Cost Search

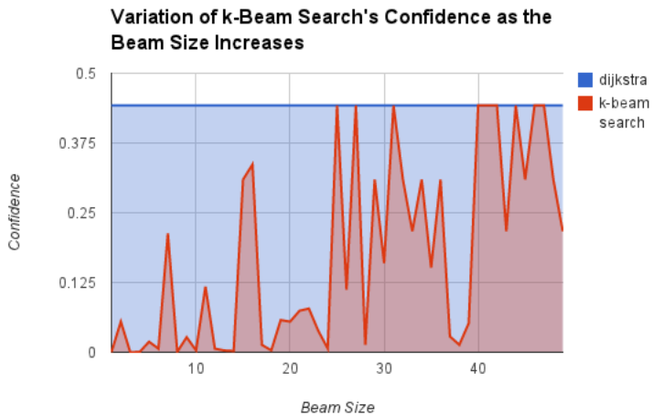
As expected, K-Directional UCS is optimal and performs just as well as UCS in terms of confidence (and better than all other algorithms). In terms of node expansions, we notice that K-Directional UCS starts off expanding more than all other algorithms. This is because it has to expand from K directions, which for small graphs represents too much of an overhead. For larger graphs however, this algorithm is very good in terms of nodes expanded, second only behind Perimeter Search. We thought it might not be as efficient for larger graphs because in a larger graph, each user has more nodes he has purchased, and therefore the algorithm has more goal nodes to start from, which we thought might have a substantial effect on the number of expansions but apparently not.

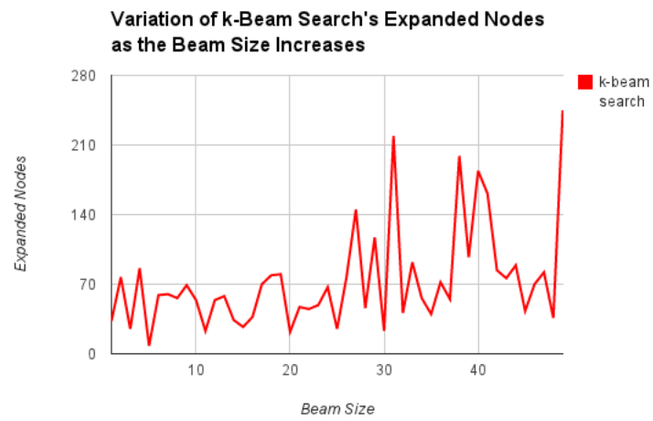
4.5 K-Perimeter Search With Non-Admissible Heuristic

Perimeter Search performs very well in terms of confidence for all graph sizes. We thought that this would be the case only for small to medium-sized graphs because of the K-directional UCS component, but it is also true for larger graphs. As for nodes expanded, Perimeter Search performs better than K-directional UCS. This can be attributed to do reasons: First, the heuristic might be accuraetly directing the algorithm toward a goal node (which is possible for small graphs as we saw with A-star). Second and more importantly, Perimeter Search takes a greedy approach in that it stops expanding nodes once it encounters its first intersection and returns the path immediately.

4.6 Beam Search

Beam Search performs well in terms of confidence for small graphs, and that is because its k-window (which was fixed at 10 in our experiments) is big enough to find a nearly optimal path. That is not true however for larger graphs where Beam Search can wander off on long paths to the goal and thus return diluted confidences. This also translates into much larger path lengths for large graphs than any other algorithm (as we can see). As for the number of expanded nodes, beam search is the most efficient algorithm for small graphs (although the scale of the graph might not highlight this fact clearly). Additionally, beam search is still relatively good but not great when it comes to larger graphs, and that is because in sparser graphs, since confidences are lower, the optimal path will have a low confidence on average, but beam search will miss the path because it will go for higher confidence edges and its window will miss out on the optimal edges. This means that it will expand many more nodes in search for a goal.





The 2 graphs above study the effect of varying the beam size on both the confidence of Beam Search as well as the number of nodes it expands. We conclude that the smaller the beam size is, the fewer nodes are expanded at each time (as we are popping less elements off the beam because it is smaller). If the beam size is small, then the confidence of our path becomes very low. This is due to the sub-optimality of k-beam search as opposed to Dijkstra's algorithm which is always optimal and thus has the highest confidence in this case.

The first graph shows the optimal confidence for a particular graph and a particular "Tailr Me" request (fixed start and goal nodes).

We vary the beam size and notice that the confidence grows with it. In fact, a big window give beam search a better opportunity to scan a bigger array of nodes, and thus increases the search's probability of hitting the optimal one.

We also notice that the number of expanded nodes increases as the window size increases. This is due to the fact that more items are popped off the beam because the latter is bigger, which results in more expansions but better results.

These 2 graphs clearly identify the trade off between confidence and the number of expanded nodes. A good beam size would balance the two, thus maintaining the main advantage of Beam Search which is a very low node expansion, as well as getting reasonnaly closer to search optimality.

5 Conclusion

We have learned that depending on the size of the graph and some characteristics (how dense the edges are etc.), some algorithms may be better than others. Normal UCS provides a very good balance of optimality and efficiency for small graphs, but becomes inefficient for medium graphs and very much so for large graphs. K-directional UCS is optimal just like UCS, but it carries a large overhead with it (associated with expanding from k+1 directions) that makes it inefficient for small graphs. However, it performs very well for large graphs.

A-Star search does well enough in terms of optimality, be it for small or large graphs. It is also efficient for small graphs but generally not more efficient than UCS, because it is hard to find a good heuristic and generate a good realistic graph. For large graphs though, it is highly inefficient, and expands more nodes than all other algorithms do. A-Star is therefore not a good algorithm because normal UCS generally surpasses it in both optimality and efficiency for all graph sizes (at least with the current heuristic). Beam Search is extremely good for small to medium sized graphs. It finds close to optimal values, and expands very few nodes. For large graphs however, because the optimal confidences are not high, beam search wanders off on long paths because it fails to take the appropriate edges, and finds the goal after expanding many more nodes than UCS, and aggregates confidences over very long paths and thus returns extremely low confidence values. Last but not least, K-Perimeter Search with a non admissible heuristic performs well enough on small graphs, very similarly to k-directional UCS in terms of confidence and node expansion. However, it performs extremely well on medium to large graphs, with very good confidence values and relatively very few nodes expanded. K-Perimeter Search is thus the clear winner for this application, at least when it comes to medium to large graphs.

6 Future Works

In the future, Tailr could be downloaded as a cross-browser extension. Once downloaded, whenever users browse to a recognized E-commerce page (such as an Amazon clothing item), Tailr will replace the "sizing-info" button with a "Tailr Me!" button, that opens a pop up showing our size recommendation for the item the user is currently looking at, as well as our confidence. Some time after users purchase an item (enough time for the item to ship), they will be prompted to provide a rating for its fit, which will help them as well as other people to get better predictions in the future.

References

[1] Perimeter Search (John F. Dillenburg and Peter C. Nelson)
http://tigger.uic.edu/ dillenbu/papers/Perimeter

7 Appendix I

The program will generate a graph according to the parameters that are given to it, and will then pick a random user and a random node, and will run all five algorithms on them and print their output. (We personally recommend looking at Perimeter Search's prediction).
Here is the output when the graph is built with the following parameters:
num-users = 400
num-items = 1100
num-purchases = 3000
num-brands = 8
nodes-peritem = 3

```
('best_node: ', <infrastructure.Node object at 0x100795d50>, ' predicted_rating: ', 62, ' confidence: ', 0.30955749999999993, ' length_of_path: ', 5,
' expanded nodes: ', 629)
('best_node: ', <infrastructure.Node object at 0x100795d50>, ' predicted_rating: ', 62, ' confidence: ', 0.2940796249999999, ' length_of_path: ', 6, '
expanded nodes: ', 366)
('best_node: ', <infrastructure.Node object at 0x100795d50>, ' predicted_rating: ', 65, ' confidence: ', 0.30955749999999993, ' length_of_path: ', 5,
' expanded nodes: ', 300)
('best_node: ', <infrastructure.Node object at 0x100795d50>, ' predicted_rating: ', 65, ' confidence: ', 0.30955749999999993, ' length_of_path: ', 5,
' expanded nodes: ', 114)
('best_node: ', <infrastructure.Node object at 0x100795d50>, ' predicted_rating: ', 70, ' confidence: ', 0.022981699111233333, ' length_of_path: ',
11, ' expanded nodes: ', 82)
```

8 Appendix II

Simply go to test.py, choose the graph parameters you would like to experiment with (at the top of the file), then execute: python test.py

The maximum allowed value for the nodes_per_item variable is 5, which is a plausible assumption given that common brands usually have around that number of size variations per item.

For some parameters configuration of the graph, the graph construction might run into an infinite loop. If this happens, it means the make_purchases function is stuck because the four conditions for a user to buy a node are unsatisfiable. Changing the graph parameters when this happens will avoid that loop.

9 Appendix III

All three of us worked on the infrastructure and the modeling.
Sami focused on UCS, A-Star, K-directional UCS, and Perimeter Search.
Andrew focused on Beam Search, providing a testing framework, collecting data and plotting them.
Devvret focused on UCS and K-directional UCS as well as Bidirectional UCS which we ended up not using.