QuickShop – A faster way to shop

Kunal Malhotra & Sam Britt

1. Motivation & Objectives

In our day-to-day lives, we all need to go for grocery shopping at least once a week, and we don't want to waste a lot of time in the store. Existing shopping list apps help by categorizing the items we want into predefined categories, or by displaying available coupons and saving cards, which can be of great help in saving money. Unfortunately, there is not much work being done towards optimizing the user's path through the store, that can help him save time and effort. Different grocery store locations, even within a particular company store chain, can have very different layouts and floor plans [Fig 1]. It is possible for the user to manually sort the list if the grocery store is familiar otherwise even if the grocery list is organized by category, it can be frustrating if categories close together on the list are in wildly different parts of the store; a section might be overlooked, and the user must backtrack to pick up the items left behind. We propose to build an Android app which would not just help the user in making a grocery list, but also dynamically sort the list by category based on the location of the items in the grocery store. We record the store layout information in the database which plays a primary role in sorting the user's list which would act as a virtual guide in picking up items without ever backtracking or returning to a section again.

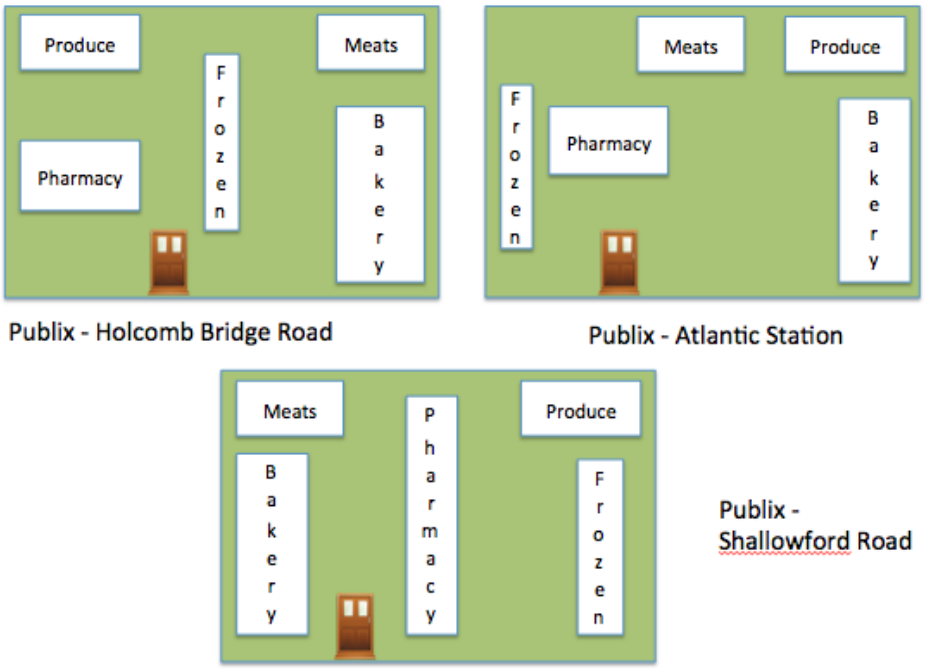

Fig 1. Various Publix stores across Atlanta, showing very different layouts.

2. Related Work

There is no shortage of grocery list apps for the iOS and Android platforms [1-12]. These apps primarily help users make a grocery list for shopping. Some offer free- form item input, while others have large databases of items from which the users can choose. Some apps add frequently purchased items to new lists, some can search or filter favorites to build lists quickly, and most also keep track of what items have been checked out. Many of them can recommend coupons, which the user can use at particular stores, and some offer the ability to sync lists with friends or a spouse.

While many of these apps offer the ability to categorize items according to sections in the grocery store and manually sort the list based on the user's preference, none, to our knowledge, have focused on saving a user's time and effort by dynamically sorting these categories according to the layout of a specific store. The sorted list helps the user pick items in the right order and avoiding multiple trips to the same section. This is of greatest help when entering an unfamiliar store, allowing the user to navigate as efficiently as he / she would in a store frequently visited.

3. This Work

3.1 User Interface Design

The QuickShop interface guides the user through the act of creating a shopping list and selecting a supported store (i.e., a store with layout information in the database). On making the store selection the list would automatically get sorted based on the location of items in the store in such a way that the user makes only one pass through a section.

The initial screen for QuickShop is the grocery list itself [Fig. 2]. The user is provided with a free-form input to add items to his list. He/ she is required to select a category for the item being added; the list of category choices is provided by the app. At any point of time the user can review the list by using a dropdown button for each category which displays the items added to that particular category. When creating a new item, there are no restrictions on the name of the item (that is, we do not make the user choose from a pre-populated list of items). The user can delete any item at any point of time while creating his list. The user can long-press on a category and choose an option to force that category to the bottom of the list for any sort (a so-called "anchor point," see below).

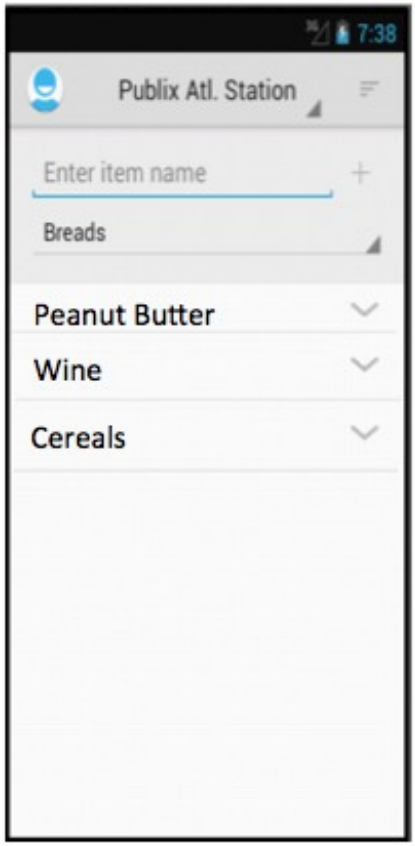

Fig 2. Grocery List

3.2 Database Design

We designed a database to record the store information which includes store names and location coordinates. The location coordinates of the store would be the coordinates of the entrance. Initially we had decided to directly access the store layout and floor plan information from the stores itself but it is seemed to be hard to get access to such information due to administrative issues. We visited the stores personally and manually created a store layout and recorded the location of items. Due to time constraints we focussed on two stores Publix Atlantic Station and Kroger Howel Mill Road, chosen specifically because they have very different layouts and item organization. There were two main questions to consider: exactly what information to store, and how best to collect these data.

Before QuickShop can sort the user's shopping list, it needs to determine which store location the user is visiting. It also stores information about location of items within the store. Each category of items has start and end coordinates i.e each category is located at one or more segments (see Fig. 3). The design is very comprehensive since it plays a primary role in sorting the user's grocery list in a way that can optimize the time he takes to finish shopping. (Relative) spatial coordinates of store sections are not enough — there is a topological component that is critical for finding the proper path. The categories should be specific enough to make the sorting useful and be applicable across stores, but general enough to make the app friendly to the user. For example, the category "dry goods" is likely too general: it will span several aisles within a store and negate the usefulness of the sorting. The category "medicine" might also be too general, because, although one store may stock vitamins and supplements in the same section as pharmacy and first aid, another store may have them in completely different locations, so they should get separate categories. On the other hand, having separate categories for "fruits" and "vegetables" is likely too specific — most stores group them together in a single produce section, and providing both categories will likely just frustrate the user.

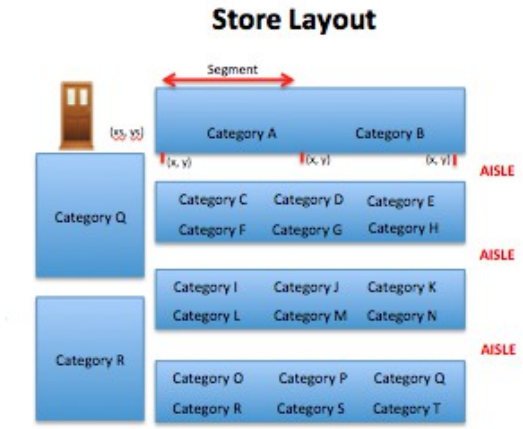

Fig 3. Store Layout

4. Working of the app

Here we present an outline of how a user might use our app. First, the user creates his grocery list, choosing categories for each item. He's ready to start shopping.

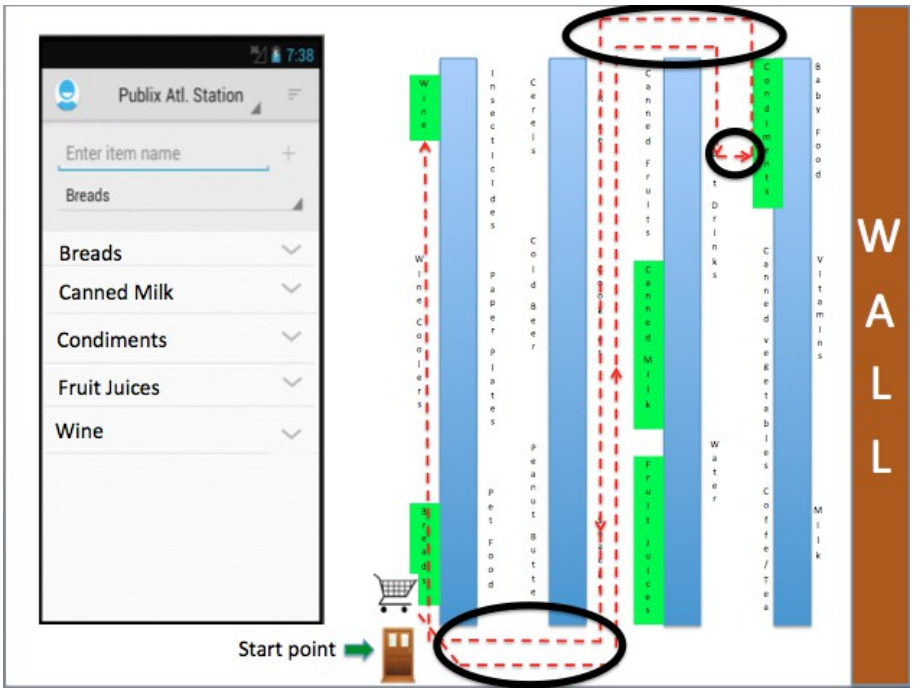

Fig 4. Grocery list created by the user

Here [Fig. 4] the user buys items in the order that he entered the items while creating the list. This makes him take a longer route i.e getting the items under the breads category first and then turning back in the aisle and then getting the other items. The user makes a lot of U-turns within an aisle and also visits a section more than once multiple times. On choosing the proper store (e.g., Publix Atlantic Station) and clicking the sort button, we get a sorted list which optimizes the path in way which can save a lot of time for the user [Fig 5.]

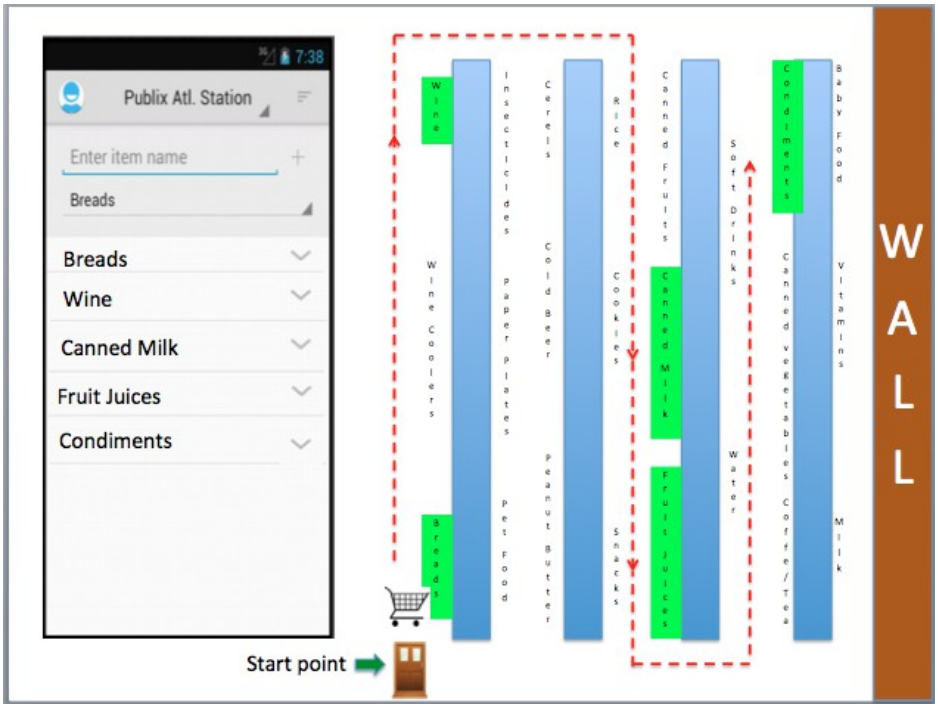

Fig 5. Sorted List

If he uses the same list for another store, then this might not be the best order since layout of different stores differ from each other markedly. Suppose the user takes the same list to Kroger on Howell Mill Road. Fig. 6 shows that the path the he needs to take based on the list above is not optimal, since there are areas where the user needs to make U- turns and backtracks to a section more than once. He then changes the store to 'Kroger Howell Mill Road' and refreshes the sort: the list is dynamically sorted again based on the layout of the new store [Fig. 7].

Also implemented are are so-called "anchor points." These are user-specified categories that should be visited *last* during the shopping trip, regardless of the path that might result from that decision. For example, a user may want to pick up her frozen foods last to avoid them melting while shopping for other items, even if that means she must take a longer or more convoluted path through the store. Our app doesn't just push anchor points to the bottom of the list, but rather it maintains the relative order of all chosen anchor points. For example, suppose categories F1 and F2 are both anchor points (e.g., frozen desserts and frozen meats). Then, depending on the path taken to gather everything else on the list, our app will choose to go to, say, F1 next, because F1 is the closest to the user's location. In another store with a different layout, the app may choose F2 before F1.

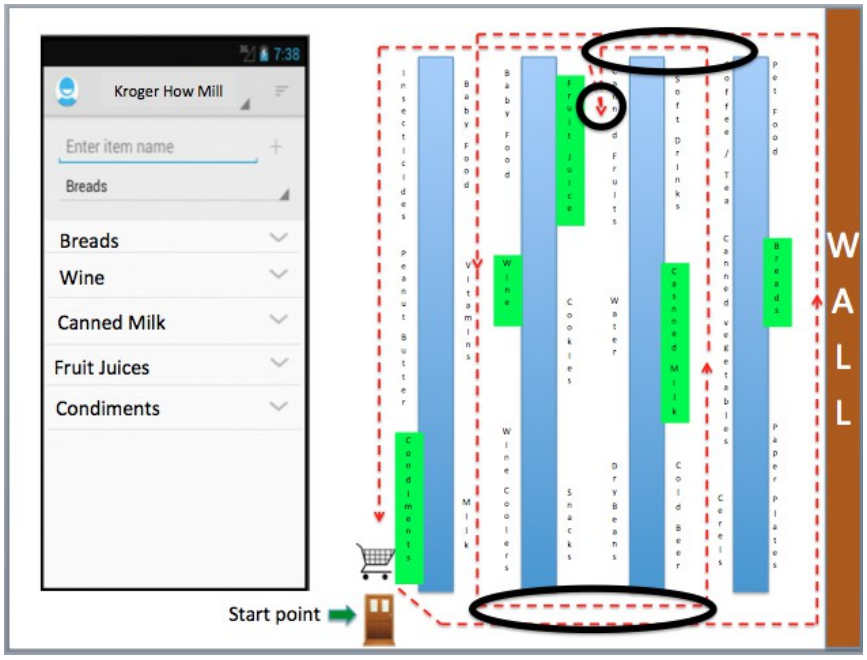

Fig 6. Sorted list of Publix not optimal for Kroger

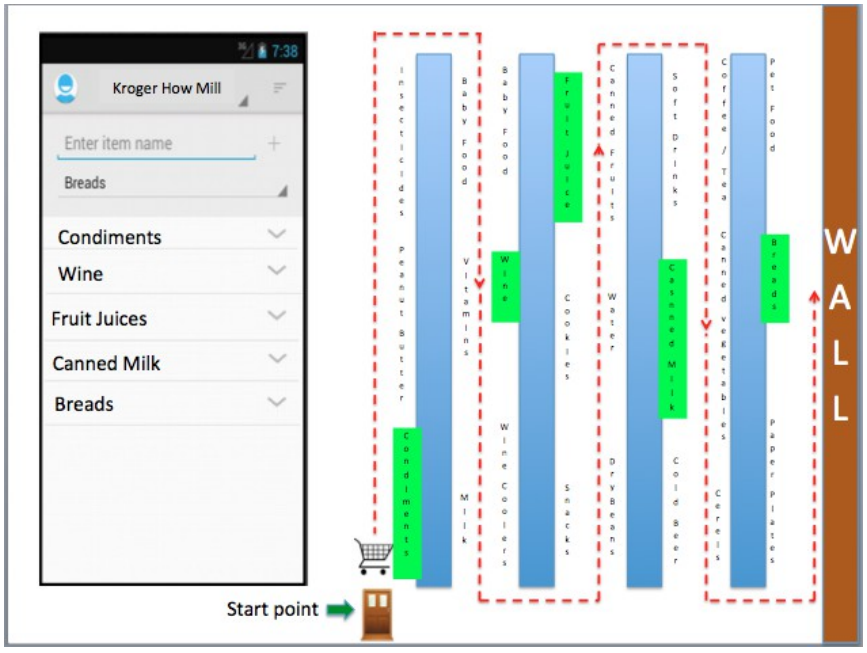

Fig 7. List sorted based on layout of Kroger

5. Category Sorting Algorithm

Although on the surface, finding the most optimal path through a grocery store seems like a simple shortest path problem, there are some subtleties characteristic of grocery store layouts and paths that must considered. For example, consider Figure 8. On the left is shown a

standard path through the store for a user with items A, B, C, and D on her list. Notice that the "closest" node to A, at least based on proximity, is not B but rather D. But a path from A to D would involve taking a U-turn in the aisle after picking up the item in A, which is generally undesirable in a crowded store with a cart. So the sorting algorithm should somehow penalize U-turns when finding the optimal path. However, U-turns may still be necessary. The right-most image in Figure 8 shows the same store layout, but this time item B is not on the list. It may be more desirable to perform a U-turn at A, rather than walk all the way down the aisle before picking up item C. This also illustrates the need for dynamic sorting based on the user's current grocery list; a static ordering of the categories would not output the proper list.

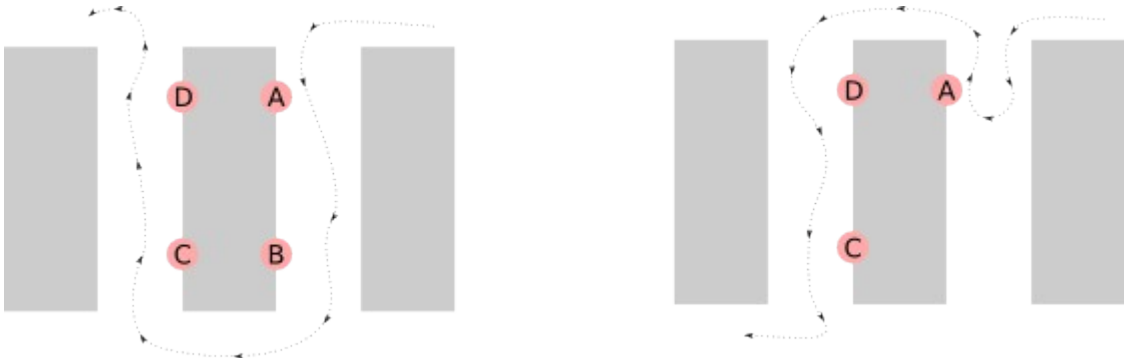

Fig 8. Optimal path when the list contains categories A, B, C, and D (left image) and A, C, D (right image). U-turns should not be favored, but still allowed, and the category order should change dynamically based on the list contents.

To better model the turns a user takes through the store, we follow the work of Winter and Grunbacher [13] introduce the notions of the *primal* and *dual* directed graphs. See Figure 9. On the left is a simple directed graph with 4 nodes and 3 edges; this is the primal graph *P*. Most path-finding algorithms would use the weights of the edges in *P* to determine shortest paths. However, for our application, we require less information about the edge between two nodes, but more about pairs of related edges, which are actually "turns" in *P*. For example, the path A > B > D represents a shallow turn at B, whereas the path A > B > C represents a sharp turn. (In terms of a grocery store, a "turn" could mean turning down an aisle, or a U-turn within an aisle). To capture information about the turns in *P*, we create the dual graph *D* associated with primal graph *P*, shown in red in Fig. 9. The dual graph is formed in two steps:

1. For every edge (u_i, u_j) in *P*, draw a node v_{ij} in *D*.
2. For every pair of adjacent edges ((u_i, u_j), (u_j, u_k)) in *P,* draw an edge (v_{ij}, v_{jk}) in *D*.

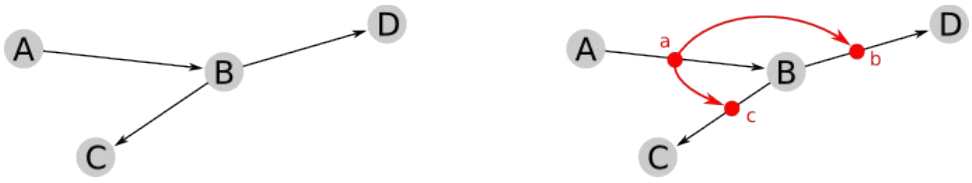

Fig 9. Primal graph (left) and associated dual graph (right, in red).

Once *D* is constructed, each edge in *D* represents a turn in *P*, and can be weighted according to its severity. For example, the edge (*a, b*) in D can be assigned a lower cost than edge (*a, c*),

representing the fact that the path A > B > D in *P* is a shallower turn than the path A > B > C. The dual graph also models U-turns quite naturally. Figure 10 (left) shows a primal graph with two potential U-turns, A > B > A and B > A > B. On the right, the dual graph is created (in red) and each U-turn is now represented by a single edge; for example the edge (*a, b*) represents the U-turn A > B > A. U-turns are undesirable for our application but it is difficult to model that in the primal graph; i.e., the edge (A, B) is not necessarily a "bad" edge to take on its own, but only when it was preceded by the traversal (B, A) should it be penalized. In the dual graph, however, we can heavily weight edges like (*a, b*), and then simple shortest path algorithms on the dual graph will handle these cases correctly.

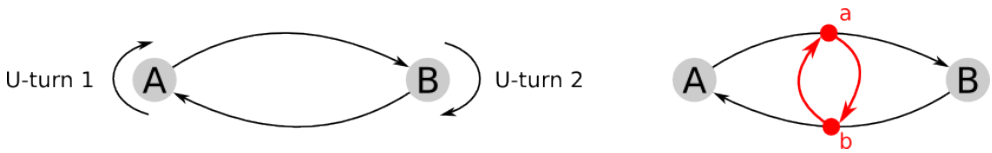

Fig 10. Primal graph containing U-turns (left) and the associated dual graph (right, in red).

In adapting these graphs to our grocery store layout, we must first create the primal graph. Fig. 11 shows the primal graph for a 3-aisle grocery store. Each aisle is divided into three segments; categories on the aisle lie on one or more segments. Because the user can walk in either direction down an aisle, each segment consists of a pair of directional edges, one in each direction. Thus, each category lies on at least two directional edges in the primal graph.

Fig. 12 shows the associated dual graph in red. We see that, by construction, each primal edge has become a dual node, and therefore each category is now associated with two or more dual nodes. If we can somehow sort a subset of nodes in the primal graph, then we have sorted the associated category list.

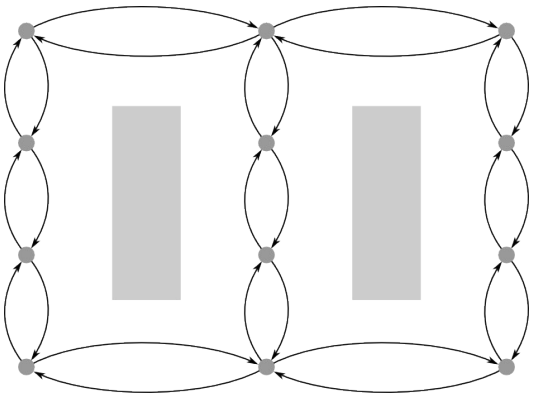

Fig 11. Primal graph for a grocery store.

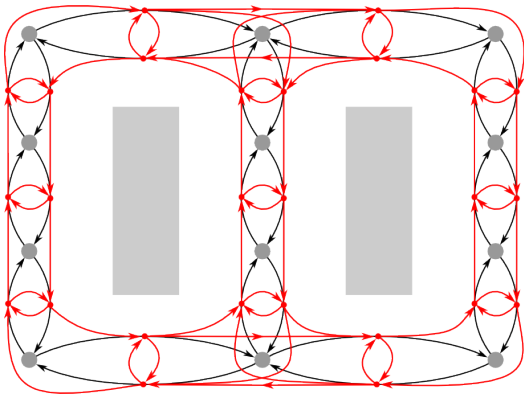

Fig 12. Dual graph (in red) for a grocery store

Figure 13 shows a detail of the same dual graph shown in Fig. 12. We see that, by constructing the dual graph, we now have a representation for every possible turn a user could make in the grocery store. For example, there is an edge in the dual graph representing the turn a user makes when going down an aisle, an edge for U-turns both within an aisle and on an end, an edge representing the user continuing straight down an aisle, etc. By adjusting the relative

weights between these types of edges, we can use standard shortest path algorithms on the dual graph, but still control the types of paths that our app produces. For example, weighting U-turns high makes them avoided in the shortest path, but if they are weighted less than full traversal down the aisle, then they may be sometimes favored. By weighting the right-angle turn to start down an aisle very low, we encourage the kind of zig-zag motion of up and down adjacent aisles, typical of grocery store shopping.

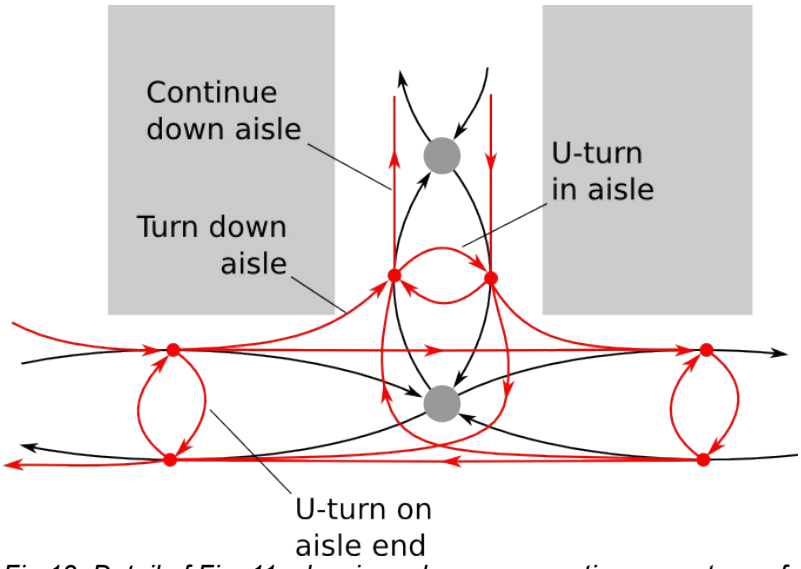

Fig 13. Detail of Fig. 11, showing edges representing every type of turn a user could make.

The full algorithm follows. Before the user presents a list of categories to sort:

1. Start by constructing the primal and dual graphs using the store layout from the database. Weights in the dual graph are chosen with the above considerations for turns, but also a component of the weight is proximity.
2. Use an all-pairs-shortest-path finder to find the shortest path between every node. We used Dijkstra's algorithm *N* times, where *N* is the cardinality of the nodes in the dual graph, starting from each node.

Once the user presents his category list:

1. Each category on the list is associated with a set of primal edges, and therefore a set of dual nodes. Consider all nodes *v* in category *C* as potential nodes on the the shortest final path.
2. Consider the distance from a dual node *u* to a category *C* as the minimum distance from *u* to all the nodes in *C,* that is
$$\text{dist}(u, C) = \min_{v \in C}\{\text{dist}(u, v)\}$$
3. If the category *C* is an anchor point, a large constant distance is added to every node in *C*. This ensures that all nodes in *C* will come after all other non-anchor point nodes, and also that two anchor-point categories C_1 and C_2 will maintain their relative order.
4. Finally, use a greedy algorithm to sort the nodes. Starting from a list of nodes and an initial position (chosen to be the front door of the store), find the closest node, say node *v*. Let the category associated with *v* be the first on the list, and delete all other nodes

associated with that category from the list of nodes to sort. Set the new starting point to *v*, and repeat.

6. Conclusions and Future Work

We have successfully implemented a shopping list app that can sort based on the location of categories in the store. We feel it could be of great benefit to users, especially if we had access to more store layout data in the app. We have identified several ways to extend this app:

- Incorporate geo-location data to help decide at which store the user is currently shopping, and to help the user identify stores in our database.
- Sync the app with a central server to get more up-to-date store data into the app.
- Incorporate functionality to allow the user to map out a store herself, no longer relying on our manual data gathering.
- Identify/enhance aspects of our app that might be of value to commercial partners such as Publix and Kroger. If provide value to store chains (e.g., by providing customer data or displaying store promotions to customers), then they might be willing to provide us with store layout data. Data gathering/entry is the weakest part of our design, and company-provided data would greatly enhance our app and the user experience.
- Incorporate some heuristics to learn about the user preferences. For example, suppose our app decides that the “Bakery” is the first category on the list. But we notice that, at least for a particular user at this store, items in “Produce” are always deleted from the list first (meaning she picked up items in that section), even if Bakery items are present. Perhaps we could capture this, learning that there is something about this user or this store that warrants going to the Produce section first, regardless of our algorithm. We could then tweak the edge weights for that section in the internal graph representation, to help the Produce section float to the top of the sorted list.

7. References

1. Pocket Labs. "Grocery King Shopping List" [Mobile Application Software]. Retrieved September 20, 2012, from https://play.google.com
2. Easicorp. "Grocery Tracker Shopping List" [Mobile Application Software]. Retrieved September 20, 2012, from https://play.google.com
3. Listonic Sp. Z o.o. "Handy Shopping List" [Mobile Application Software]. Retrieved September 20, 2012, from https://play.google.com
4. Aisle Express, LLC. "Aisle Express" [Mobile Application Software]. Retrieved from https://play.google.com
5. Mighty Pocket. "Mighty Grocery Shopping List" [Mobile Application Software]. Retrieved September 20, 2012, from https://play.google.com
6. HeadCode. "OurGroceries" [Mobile Application Software]. Retrieved September 20, 2012, from https://play.google.com
7. Capigami, Inc. "Out of Milk Shopping List" [Mobile Application Software]. Retrieved September 20, 2012, from https://play.google.com
8. MidCentury Media. "Shopper - Shopping List" [Mobile Application Software]. Retrieved September 20, 2012, from https://play.google.com
9. The App Group. "The Grocery List" [Mobile Application Software]. Retrieved September 20, 2012, from https://play.google.com

10. Twicular, Inc. "Grocery Pal" [Mobile Application Software]. Retrieved September 20, 2012, from https://itunes.apple.com
11. Werner Freytag. "ShoppingList" [Mobile Application Software]. Retrieved September 20, 2012, from https://itunes.apple.com
12. Coupons.com, Inc. "Grocery iQ" [Mobile Application Software]. Retrieved September 20, 2012, from https://itunes.apple.com
13. Winter, S., & Grunbacher, A. (2002). Modeling costs of turns in route planning. *GeoInformatica*, *6*(4), 345–361.