Technical Report TR06-023

Department of Computer Science
Univ. of North Carolina at Chapel Hill

Unsupervised Task Extraction from a Stream of Window Focus Events

Karl Gyllstrom and David Stotts

Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175

{karl, stotts}@cs.unc.edu

Unsupervised Task Extraction from a Stream of Window Focus Events

Karl Gyllstrom David Stotts

1 INTRODUCTION

We describe the process of collection data and analyzing the data for task extraction. The task extraction method is composed of two parts. First, we map window-to-window relationships to a directed graph, where nodes represent individual windows and weighted links represent the relative importance of the linked node to the linking node. Once we have a graph, a graph partitioning method is used to find clusters of high mutual link value. These clusters are used to represent conceptual tasks.

2 DATA COLLECTION

Data collection revolved around the collection of window focus events through a passive, non-intrusive event monitoring system that could be deployed as users went about their system work. We developed an application for capturing window focus events and maintaining a log. Window focus events are instances in time when a user driven event changes the window that is currently focused. Intuitively, a window focus event indicates that a user intends to interact with the contents of the newly focused window, although in practice, window focus events can sometimes be the result of spurious clicks or other user errors.

Our monitoring application polls the system for the foremost window by accessing the system's depth-ordered list of on-screen windows. The polling rate is on the millisecond granularity to optimize precision without requiring substantial system resources. When the foremost window is different from the foremost window of the previous cycle, an entry is added to the log

featuring the application name, the window number - which is unique within the system, and the time stamp of the event. Furthermore, an image of the window is stored to assist future recollection.

3 GRAPH CONSTRUCTION

The theory motivating this work is that windows that are consistently focused at close points in time are more likely to be related to each other with respect to a user task. To find the relatedness of window W_B to window W_A, then, we need to find how likely W_A is to appear soon after or before W_B appears.

A *focus event* is an instant in time when a window is brought forward by the user to the focused position, depicted by a two-tuple (w_i, t_i) where w_i is the window number and t_i is the moment when the window is focused. A *window interval* for window w_i consists of

- Boundary focus events $w_{i\times t1}$ and $w_{i\times t2}$, occurring at times t_1 and t_2, respectively. No focus event for w_i can occur between these two focus events.
- The set of all focus events transpiring between t_1 and t_2.

A window interval is depicted as a 4-tuple (w_i, t_i, t_j, S), where w_i is the window number, t_i is the time at which the interval began, t_j is the time at which the interval ended, and S is the set of focus events occurring between t_i and t_j, exclusively. In other words, a window interval is the set of focus events for other windows occurring between a time that window is brought forward and the next time it is brought forward. The duration of the window interval is the time elapsed since the initial window event ended and the last event starts, and measures the time elapsed within the window interval where window w_i was not the focused window. Henderson and Card[1] provide a similar treatment to the notion window interval.

If a window is used frequently, it will have many intervals within the total time it has been used. Other windows that tend to occur often within these intervals are likely to be task related windows. This metric rates windows by their likelihood of occuring within a task interval. The function $WSV(w_j, w_i)$ represents this value. To compute $WSV(w_j, w_i)$, we find the number of window intervals for w_i in which w_j is brought forward, and divide this by the total number of intervals within some time interval T.

Using raw probability for link values makes it difficult to view relationships between different windows along a common currency. For example, window A might be %20 likely to switch to window B, just as window C is %20 likely to switch to B. We cannot infer from this data alone that B is equally related to window C as it is to A, as the behavior of C and A skew the significance. This issue will be addressed in more detail in the discussion section.

Thus, we introduce the function $Rank(w_j, w_i)$ to capture this information. To compute $Rank(w_j, w_i)$, we order the list of $Link$ values for w_j and order them from greatest to least, returning the level of w_i on the list (Figure 1). Where ties occur, each link is given the same value, and the next non-tie value is given a higher rank according to the number of ties. We then construct a graph from our data, where windows are mapped to nodes and the links between all w_i and w_j are given the value $Rank(w_i, w_j)$.

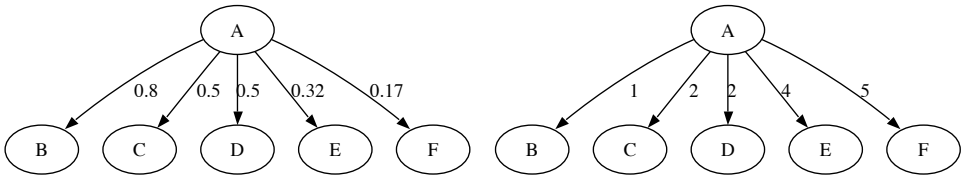


Figure 1: *Link values to rank values* Outgoing link values are sorted and ranked, with ties given equal rank.

4 GRAPH CLUSTERING

While much literature on graph clustering is available, we chose not to apply an existing implementation for a number of reasons. First, we found that a user-level understanding of window relationships could help inform the clustering algorithm to create more accurate results over an algorithm that operates on generic data. Traditional clustering implementations emphasize performance and scalability to large data sets; in our case, there is a natural limitation to the amount of data that a user could produce within a time interval, and it was thus unnecessary to incorporate the same attention to performance within our work.

We define a cluster as two sets of nodes: a primary node set and a secondary node set. The primary set consists of nodes that are fully connected

and each node in the set satisfies the fitness metric described below. The fitness metric for the inclusion of a node within a node set relies on the notion of *average outgoing rank* and *average incoming rank*. The *average outgoing rank* for node n_i with respect to node set N_i is defined as the sum of the ranks of outgoing values from n_i to the nodes in N_i divided by the number of nodes in N_i. The *average incoming rank* is the sum of the incoming ranks of n_i from each node in the set, divided by the number of nodes in set N_i. For a node n_i to satisfy the fitness requirement for primary set N_i, its *average incoming rank* and *average outgoing rank* must both be less than or equal to the number of nodes currently in N_i. Intuitively, this constraint means that for node n_i to become part of node set N_i, where the size of set N_i is L, it must be, on average, at least the L^{th} most relevant node to the set.

$$AIR(n : Node, N : NodeSet) = \frac{\sum_{i=1}^{|N|} linkValue(n_i, n)}{|N|} \tag{1}$$

$$AOR(n : Node, N : NodeSet) = \frac{\sum_{i=1}^{|N|} linkValue(n, n_i)}{|N|} \tag{2}$$

The secondary set consists of nodes that satisfy only a single part of the fitness metric; namely, that one of AIR and AOR with respect to the primary set are satisfied but not both. Note that the node set used in AIR and AOR is the primary set; the secondary set is not used to test for new cluster candidates.

The separation of a cluster into two sets allows a cluster to expand without raising the exclusivity of the task. Because primary set candidacy requires a node to satisfy a condition that depends on each node within the primary set, adding a node to the primary set can make it more difficult for another node to be added later, as that node must satisfy more conditions. Figure 2 depicts such an example.

The secondary set thus allows nodes to be added to a cluster without affecting future candidates. Nodes in the secondary set represent windows that are considered related to the task but not as strongly as nodes within the primary task.

The algorithm for building clusters works as follows. We first generate a list of 3-tuples (w_a, w_b, r), representing the rank value r from window w_a to window w_b. We then sort this list by rank value, such that tuples appearing earlier in the list have lower values for r. We then iterate upon the list while maintaining a set of clusters. For each tuple, we check if either of the

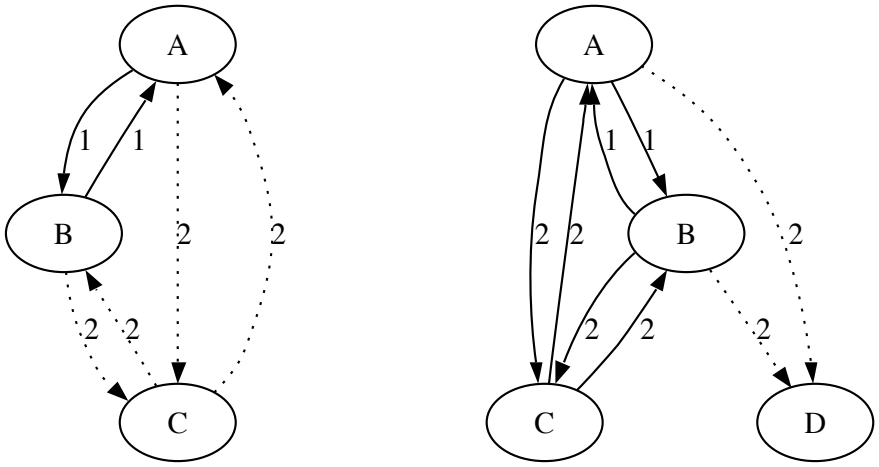


Figure 2: Adding nodes to cluster affect future candidates. Node D would have qualified before C was added. After C was added, the lack of a connection between C and D prevents D from qualifying.

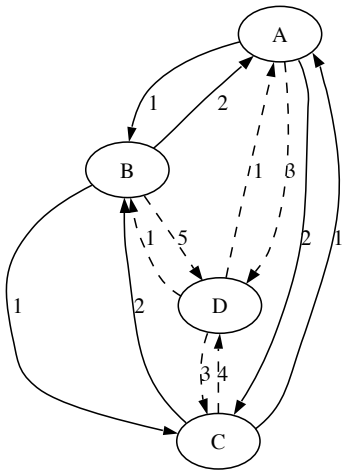


Figure 3: Cluster of primary and secondary set windows. D is the only member of the secondary set, as indicated by the dashed line. Notice the lower link values.

windows are part of the primary node set of an existing cluster. For each matching cluster, we evaluate the window with respect to functions AIR and AOR on the primary nodes set of the cluster. If both pass, and the node is

fully, bidirectionally connected to the nodes of the primary set, it is added to the cluster. If only AOR passes, the window is added to the secondary set. If no task contains either node, a new task is created containing both nodes in the primary set, and this task is added to the list of tasks.

Once the list of 3-tuples is fully processed, we execute some post processing on the set of tasks. First, each task is pruned of orphan windows, where orphan windows are windows that have no incoming links from the primary set of that task. This pruning is necessary because some windows that occur infrequently on the boundaries of the examination interval will have links pointing to most of the windows focused during the interval.

The second phase of the post processing involves modifying tasks whose initial task construction violates the rules of cluster construction. The reasons for this happening are subtle. Note that the first two nodes of a new cluster don't necessarily satisfy the AIR/AOR rules. Thus, if nodes or links are not added to the cluster such that both these two nodes belong properly to the primary set, they are demoted to the secondary set.

The next phase in post processing involves merging redundant and partially redundant tasks. Because all tasks begin with a "seed" link between windows, there may be multiple clusters that involve the same nodes, as the clusters grew together. Currently, any cluster C_A is considered a superset of C_B if the union of the primary and secondary set of C_A is a superset of the primary set of C_B. For all i, j such that $C_i \supset C_j$, we remove C_i and C_j from the cluster set and add a merged cluster C_m such that the primary set of C_m is the union of the primary sets of C_i and C_j, and likewise for the secondary set.

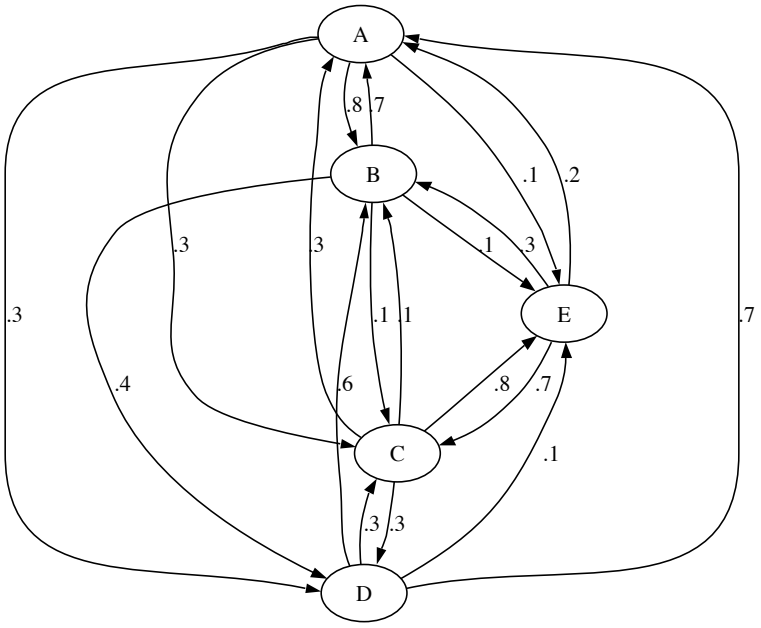
5 DISCUSSION

It is important to note that the conversion of window switch values to rank enumeration adds a much needed normalization. Before the conversion, we can only use likelihood of interval precense as a metric, and these values can vary a lot from window to window. For example, assume the WSV from W_A to W_B is 0.7, while the WSV from W_B to W_A is 0.5. W_B could still have fewer windows it is more likely to switch to than W_A does, although a direct comparison of the WSVs is not sufficient to understand this relationship. We thus say that the conversion normalizes the values and allows a more direct and even comparison.

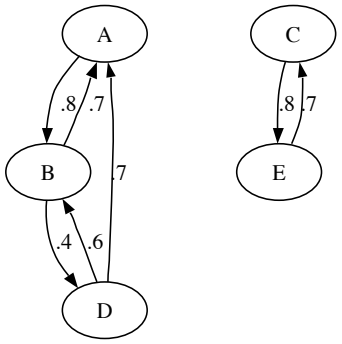
A further advantage of the conversion is that it avoids the needs to remove low WSV values from the set. Using a generic graph clustering method based on connectivity metrics on such a set might produce clusters that are larger than desired since low WSV values will make the graph more connected. In such a case, we would be forced to remove WSV values below a certain threshhold to maintain a meaningfully connected graph. Such a threshhold would be heuristic, parameterizing the algorithm and possibly forcing supervision. Figure 4 depicts this scenario.

References

[1] Jr. D. Austin Henderson and Stuart Card. Rooms: the use of multiple virtual workspaces to reduce space contention in a window-based graphical user interface. *ACM Trans. Graph.*, 5(3):211–243, 1986.



(a) With no WSV threshhold



(b) Threshhold of ≤ 0.3

Figure 4: (a) With no WSV threshhold, the graph is fully connected and cannot be partitioned by connectivity alone. (b) A threshhold of ≤ 0.3 allows for some clustering