This manuscript consists of the following contents

Content Title	Page Index
Web Page Functional Parsing Report	2
How Web Page Functional Parsing works	16
Future Works	19

Web Page Functional Parsing Report

Problem: Given a web page, our goal is to re-generate the same web page in Json format with minimal number of HTML elements and label the elements such as navigation bars (static/dynamic), image carousels, search box, vertically aligned elements and so on. The steps towards this problem are depicted in the following figure.

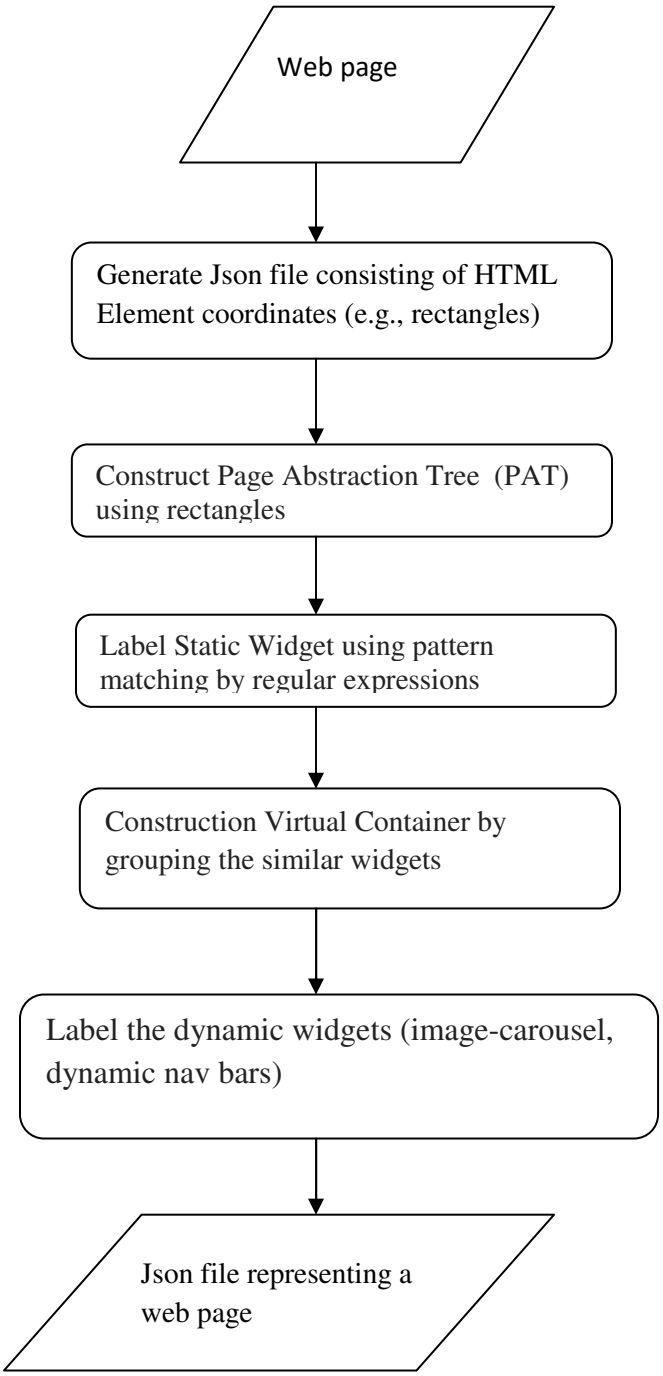

Figure: Web page re-generation steps

Step 1: Generate Json File consists of HTML-Element coordinates for a webpage and corresponding HTML file with CustomNodeIds

input: (i) unitedway.ca.wholepage.html (saved HTML page of unitedway.ca)

output: (i) unitedway.ca.wholepage.json containing rectangles with customNodeId for each HTML element.
(ii) unitedway.ca.wholepage.html containing additional attribute customNodeId for each HTML element. customNodeId will be used to match the HTML element in unitedway.ca.wholepage.html with corresponding Rectangle in unitedway.ca.wholepage.json

We are proposing a Rectangular based web page re-regeneration approach because each element in a page can be represented as a rectangle having coordinates such as left, top, bottom and right. When we see a web page we mainly see the inner most html element instead of seeing their top level wrapper elements. For example, we see an input box, but we do not see its top level parent container or wrapper such as div. Our approach is a bottom-up approach, that means we start from the leaf or inner most elements and from there we build the **Page Abstraction Tree** (described in step 2) that consists of minimal number of elements to represent a web page. The objective of considering the leaf rectangles is to depict a web page in a way that, what are the elements that are actually visualized to the audience. A portion of a web page and it's skeleton using only leaf rectangles is shown in the Fig. 1 and Fig. 2 in that order.

Fig. 1. Footer part of unitedway.ca.wholepage.html

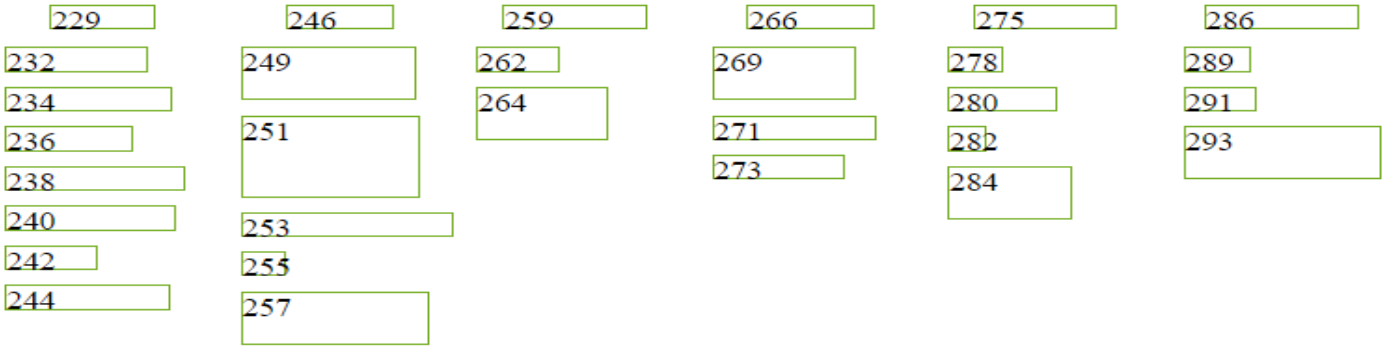

Fig. 2 Leaf rectangles for the footer part of unitedway.ca.wholepage.html

We generate a Json file for a web page (unitedway.ca.wholepage.html) consisting of the HTML Element Coordinates (we represent all the coordinates of an element as an unit called rectangle) along with customNodeIds where a customNodeId is an ID assigned to each HTML Element in order to distinguish it from other elements; Also that customNodeId is set as an attribute for a HTML Element in the *.html file so that we can map the HTML element with corresponding Json element in the *.json file.

The Json and HTML file name formats are <web page>.wholepage.json and <web page>. wholepage.html respectively which in turn will be converted into unitedway.ca.wholepage.json and unitedway.ca.wholepage.html correspondingly. Sample fragments of unitedway.ca.wholepage.json file and unitedway.ca.wholepage.html are illustrated in Fig.3 and Fig.4 respectively.


```
  {
     "nodeName":"A",
     "customNodeId":"customNodeId_229",
     "nodeProperties":"href=http://www.unitedway.ca/about-us",
     "rectangle":{
       "top":1435,
       "left":232,
       "bottom":1449,
       "right":282,
       "height":14,
       "width":50
     }
   },

   {
     "nodeName":"A",
     "customNodeId":"customNodeId_232",
     "nodeProperties":"href=http://www.unitedway.ca/about-us/our-promise",
     "rectangle":{
       "top":1462,
       "left":210,
       "bottom":1476,
       "right":278,
       "height":14,
       "width":68
     }
   }
```


Fig.3: HTML elements represented by rectangles using coordinates in unitedway.ca.wholepage.json

```
<body class="home blog" customNodeId ="customNodeId_0">
        .....
    <a href="http://www.unitedway.ca/about-us" customNodeId ="customNodeId_229">About Us</a>
        ....
     <a href="http://www.unitedway.ca/about-us/our-promise" customNodeId ="customNodeId_232">Our Promise</a>

... </body>
```

Fig.4 HTML elements with additional custom attribute "customNodeId" in unitedway.ca.wholepage.html

Step 2: Construct Page Abstraction Tree (PAT)

input: (i) unitedway.ca.wholepage.json containing the rectangles for HTML elements
(ii) unitedway.ca.wholepage.html (saved HTML page of unitedway.ca) consists of HTML elements with customNodeIds

output: (i)unitedway.ca.tree.json consists of the minimal number of rectangles to represent a web page along with their relationships (e.g., which rectangle is the parent of whom). The relationship is obtained from the relationship among the HTML elements in the DOM tree of unitedway.ca.wholepage.html.

In order to construct **PAT**,

-At first we load the unitedway.ca.wholepage.json and unitedway.ca.wholepage.html files in memory,

-After that find the innermost HTML elements (e.g., leaves) by traversing the DOM tree of unitedway.ca. wholepage.html; and using the 'customNodeId' extract the corresponding rectangle from unitedway.ca.wholepage.json

-Finally construct the Page Abstraction Tree using the leaf rectangles described in ***Algorithm: Page Abstraction Tree***; and save it in unitedway.ca.tree.json file.

Before going into the detail of the **PAT** algorithm the terminologies used in **PAT** are described below and illustrated in Fig. 5. A section of the unitedway.ca.wholepage.html is shown at the left most side of Fig.5, then a skeleton comprises the leaf rectangles respective to leaf HTML elements of that section is drawn. After that containers built from the rectangles by following the hierarchy among them. The hierarchy is obtained from the DOM-hierarchy among the corresponding HTML elements of those rectangles. For example, container-1 is built from the leaf rectangles 202 and 203 demonstrated as a red rectangle. Container-1 along with a leaf rectangle 200 are inside another container, container-2. Container-3 is constructed using the container-1, container-2 and leaf rectangles 196, 197, 198.

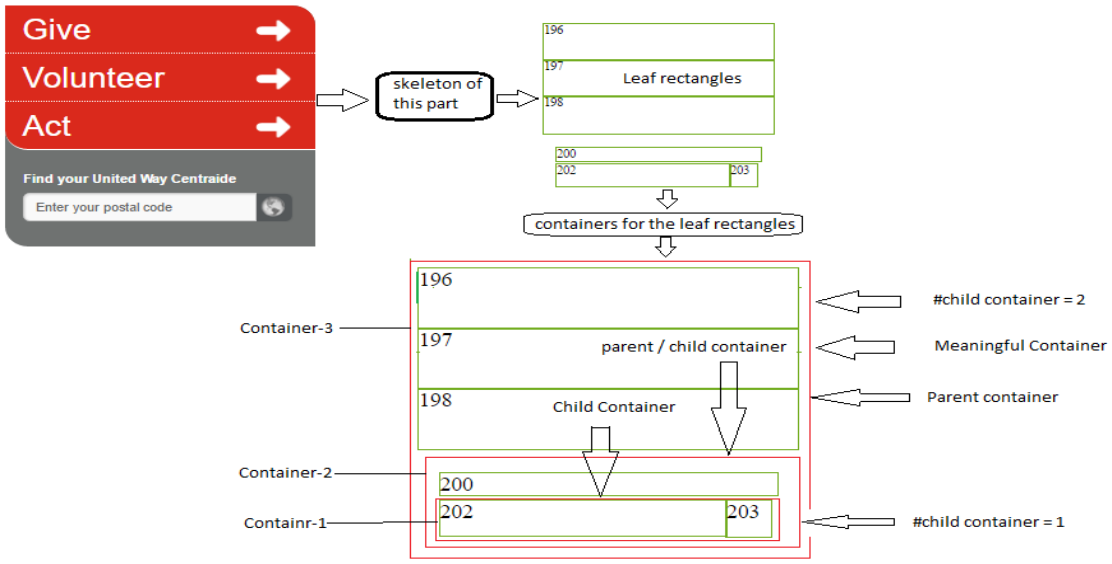

Fig. 5 Section of unitedway.ca.wholepage.html and corresponding skeleton using leaf rectangles, Container Relationships. Red rectangles are containers. Greens are the leaves.

Page Abstraction tree (PAT) is a pruned tree consisting of minimal number of meaningful containers, containers and leafs along with their relationships, that represents a web page in the original form.
Container is a non-leaf rectangle corresponding to its non-leaf html element
Meaningful container is a non-leaf rectangle that contains reliable number of child containers and leaf rectangles.
Child Container is a non-leaf rectangle which is within its parent container.
Number of Child container: Number of child containers under a parent container.

Example : Container-1 in Fig.5 all leaf rectangles, so the number of child containers of Container-1 is zero. If a non-leaf rectangle (e.g., container) has all leaf rectangles and no other descendent child containers, then the value of its child containers is zero. The container-3 has two child containers.

Non-meaningful container: It has many children (e.g., 7) comparing to the number of other containers' (container-1, container-2) child containers (e.g., 1, 2). A picture of non-meaningful container is depicted in Fig. 6. How the non-meaningful containers are pruned illustrated in **PAT** algorithm.

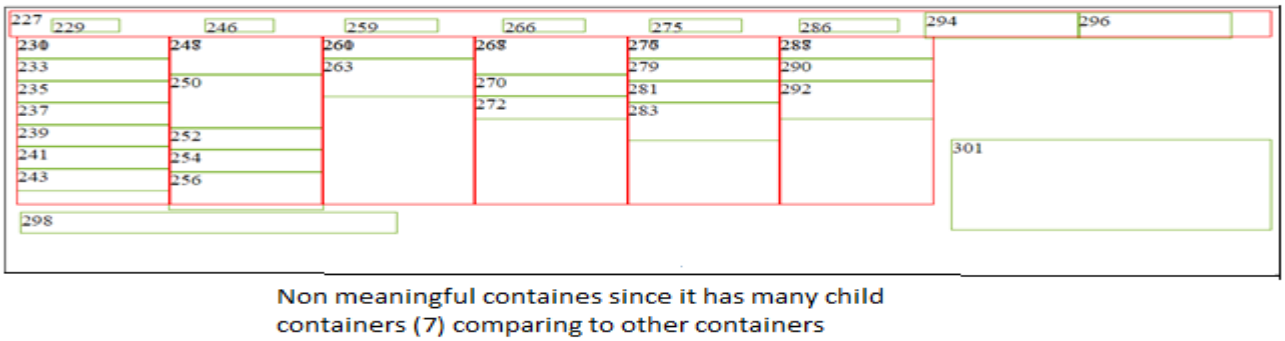

Fig. 6 A non-meaningful container.

Algorithm: Page Abstraction Tree
***input*:** leaf rectangles corresponding to the leaf HTML elements in unitedway.ca.wholepage.html
***output*:** A tree in Json format comprising meaningful containers, containers and leaf rectangles

converge = false;
LEVEL=1 // child level in tree
PAT = null; //page abstraction tree
ROOT_CONTAINER = Rectangle for HTML body element

while (!converge){

1. Find the corresponding HTML elements for the leaf rectangles by mapping the rectangles in unitedway.ca.wholepage.json with the HTML elements in unitedway.ca.wholepage.html using customNodeIds.

2. Find the immediate parent HTML and it's corresponding Rectangle (e.g., container) for each HTML element up to level LEVEL. e.g., Find parent of an HTML element, then parent of its parent, and so on up to level LEVEL. The parent HTMLs are extracted from the DOM tree of unitedway.ca.wholepage.html.

3. Construct hierarchy among the rectangles by following the hierarchy among their corresponding HTML elements in the DOM tree of unitedway.ca.wholepage.html.

4. Calculate the number of child containers of each container.
5. Compute the mean and standard deviation from the non-zero number of child containers for each container.
6. Rule out the containers that have the number of child container > mean + standard deviation.
7. The rest of the containers will be the meaningful containers.
8. Find the Containers that do not have any parent containers; attach those containers to ROOT_CONTAINER; Now we have the ultimate Tree.
9. if(LEVEL >1 && the trees of two successive LEVELs are identical){
    //In order to get two trees, the LEVEL must be greater than 1
    PAT = tree at level LEVEL
    converge = true;
}

LEVEL++;
}

Intuition of Mean (u) and Standard Deviation (sd): We presume that a container should not contain too many child containers, and a container comprises a large number child containers is taken into account as an **outlier**. For instance, if we consider '*body*' as a container, it will contain all the elements, however we are not interested about this kind of container. Instead of that, we are looking for the containers where all of them have the number of child containers close to each other. Hence, we apply the *u* and *sd* on the number of child containers, and it is known to us that in normal distribution most of the numbers close to each other, lay within the *u* and *sd*. Therefore, by applying this mechanism, we can keep the containers in which we are interested in and rule out those which are out of interest (e.g., outliers). If we consider the containers in Fig. 5 and 6, the number of child containers are 1,2,7; u=3.3, sd=3.2, so the container in Fig.6 will be ruled out since it has too many child containers (7) with respect to the number of child containers (1, 2) of other containers.

How two trees are identical: In order to measure the identicality between two trees, we only extract the first level child containers and their corresponding HTML elements of two trees. When the first level HTML tag names (e.g., DIV, IMG, A) and 'customNodeIds' of two trees are same, then they are identical. The customNodeId and tag names are extracted from the HTML element as shown in Fig. 4.

Tree Construction Example: For simplicity, we consider only a part of the unitedway.ca.wholepage.html as depicted in Fig. 7(a). The skeleton obtained from the leaf elements (LEVEL=0 elements) of this part is also drawn at the beginning of Fig. 7(b). At each LEVEL>=1 , the hierarchy among the containers are constructed by following the steps 1, 2 and 3 in PAT algorithm. At the same time pruning is also performed at each LEVEL as illustrated in Fig. 7(c). When the trees at two conjugative LEVELs are same, then we stop and consider the tree at level LEVEL as the ultimate PAT, shown in Fig.7(d). How the pruning is being carried out, exemplified in the following.

LEVEL=1: Fig. 7(b) Only one container has the # child container (CC) = 1. So the *u*=1, *sd* =NAN (e.g., 0). Fig.7(c) no pruning
LEVEL=2: Fig. 7(b) *u*=1.75, *sd* =0.95 from the #CC of their parent containers. Fig.7(c) Parent container with #CC=3 (>1.75 +0.95) is pruned .

LEVEL=3: Fig. 7(b) u=3.4, sd =3.78 from the #CC of their parent containers. Fig.7(c) Parent container with #CC=10 (>3.4+3.78) is pruned .
LEVEL=4: Fig. 7(b) u=4.66, sd =4.58 from the #CC of their parent containers. Fig.7(c) Parent container with #CC=10, 11 (>4.66+4.58) is pruned .

At each LEVEL>1, two successive or conjugative trees are compared by the step 9 of PAT algorithm. Finally, the algorithm stops at LEVEL 4 since the two trees at LEVEL 3 and 4 are equivalent according to step 9 of PAT algorithm.

Fig. 7. (a) A part of unitedway.ca.wholepage.html (b) Containers along with relationship at each LEVEL, built by applying steps 1-3 of **PAT algorithm**. (c) Pruned containers along with relationship at each LEVEL, built by applying steps 4-7 of **PAT algorithm**. (d) The ultimate tree (saved in unitedway.ca.tree.json) obtained at LEVEL=4 by the steps 8-9 of **PAT algorithm**

Container naming in unitedway.ca.tree.json: Generally Each rectangle container is named by the ID-HTMLtagname (e.g., 202-input). ID is elicited as an Integer from the customNodeId (customNodeId_202) where the customNodeId is obtained from the rectangle-container's corresponding HTML element as shown in Fig. 4. HTMLtagname (e.g., input) is the tag name obtained from the rectangle-container's corresponding HTML element's tag name.

Vertical Alignment: In unitedway.ca.tree.json, vertical alignment information is stored. The vertical alignment is detected based on the rectangle coordinates. Vertical alignment implies that, the element is vertically aligned with its siblings, e.g., the elements are on top of each other. Vertical alignment is encoded by "_" at the front of rectangle name. For example, the rectangles 196, 197, 198 of corresponding HTML Anchor elements are vertically aligned in Fig. 7(c), therefore, "_" added in front of their names (e.g., _196-A, _197-A, _198-A) in Fig. 7(d).

Step 3: Static Widget Labelling

input: (i) widget.regx consists of the regular expressions to label the names of container rectangles (e.g., 195-DIV) in unitedway.ca.tree.json

(ii) unitedway.ca.tree.json contains PAT tree in JSON format. A node in that tree implies a rectangle. unitedway.ca.tree.json consists of the minimal number of rectangles to represent a web page along with their relationships (e.g., which rectangle is the parent of whom).

output: (i) unitedway.ca.widget.json be made of nodes symbolizing rectangles; nodes are labelled based on pattern matching using regular expressions in file widget.regx.

Regular expressions in file widget.regx.

```
nv=^_?ul(_li|_a)+$                          //navbarVertical
nh=^_?ul(li|a)+$                            //navbarHorizontal
nn=^(_?ul(_?li|_?a)+){2,}$                  //nestedNavBar
sb=^_?(form|div)inputinput{2}$              //searchBox
bv=^_?div(_[a-z]+)+$                        //varticalBox; elements are vertically aligned within a div
cd1=^_?divimgh[0-9]divdiv{2}$                                   //cardTyp1
nc1=^\b_?div(divimgh[0-9]divdiv{2}){2,}.*$                      //nestedCardTyp1
cd2=^_?h[0-9]_?a_?small$                                        //cardTyp2
nc2=^\b_?div(_?h[0-9]_?a_?small){2,}.*$                         //nestedCardTyp2
cl=^_?div(_?ul(li|a)+|(div)+|(img)+|(span)+|(li|a)+)+$          //image-carousel
```

In order to label the nodes of PAT tree in unitedway.ca.tree.json

(i). At first we load the regular expressions in file widget.regx.

(ii). Following that Load the unitedway.ca.tree.json

(iii). Then, traverse the PAT tree in BOTTOM-UP fashion. The reason to traverse PAT in BOTTOM-UP fashion is that we construct PAT using the leaf element rectangles and go up to find the pertinent containers for the leaves. That is why we adopt BOTTOM-UP approach to label a container by a widget.

(iv). At the time of traversing, parse each non-leaf node (e.g., "_201-FORM") in Fig.7 (d) into string and match the string with a regular expression except (image-carousel) *cl* since image carousel will be identified and labelled in step 5 (e..g, Dynamic widget labelling).

"_201-FORM" will be parsed into "_FORMINPUTINPUT" ignoring node IDs. The "_" at the front indicates that, "_201-FORM" is aligned vertically with its siblings. "_FORMINPUTINPUT" will be matched with regular expression "sb" (e.g., search box). Another node "_199-DIV" in Fig.7 (d) will be parsed into "_DIV_SPAN_FORMINPUTINPUT"

(v). Label "_201-FORM" as "_201-FORM-h-sb" where 201 is the node id, FORM means input form, "*h*" implies that this node is homogenous since all the children are same (e.g., INPUT) and vice versa. Heterogeneous nodes are presented as "...-...-e-..." (e.g., 195-DIV-e-bv).

(vi). Save the labelled widgets in unitedway.ca.widget.json file as shown in Fig. 9.

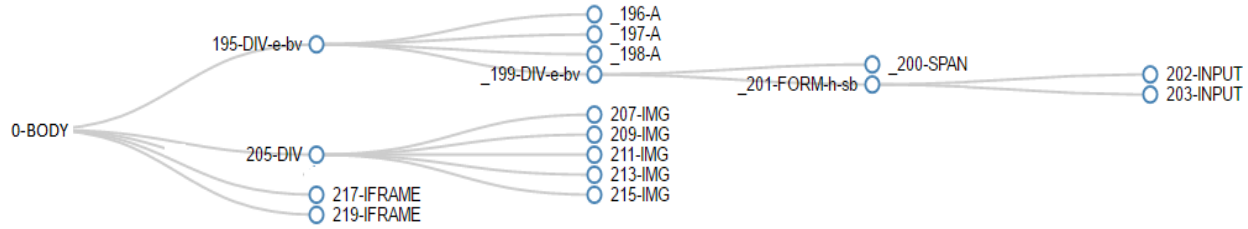

Fig. 9: Labelled nodes in unitedway.ca.widget.json using regular expressions in file widget.regx.

Step 4: Virtual Container Construction

input: (i) unitedway.ca.widget.json contains labelled nodes using regular expressions
output: (ii) unitedway.ca.virtual.json be made of similar widgets to be grouped and set as the children of a Virtual Container.

In order to construct virtual container,

(i). At first, Load the unitedway.ca.widget.json file

(ii). Group the widgets having identical widget labels (e.g., 158-DIV-e-cd1, 164-DIV-e-cd1, e.t.c) (ignoring node Ids) under a container (DIV-104) in Fig. 10(a); and create a virtual div (e.g., 1041-DIV-h-VIRTUAL). 1041=104*10+1 (104= DIV Id, 10=constant, 1=first virtual container under DIV-104)

Remove the group of widgets from the container (DIV-104) and set them as the children of the virtual container (1041-DIV-h-VIRTUAL), demonstrated in Fig. 10(b).

(iii). Save the tree in the unitedway.ca.virtual.json as shown in Fig. 10 (b).

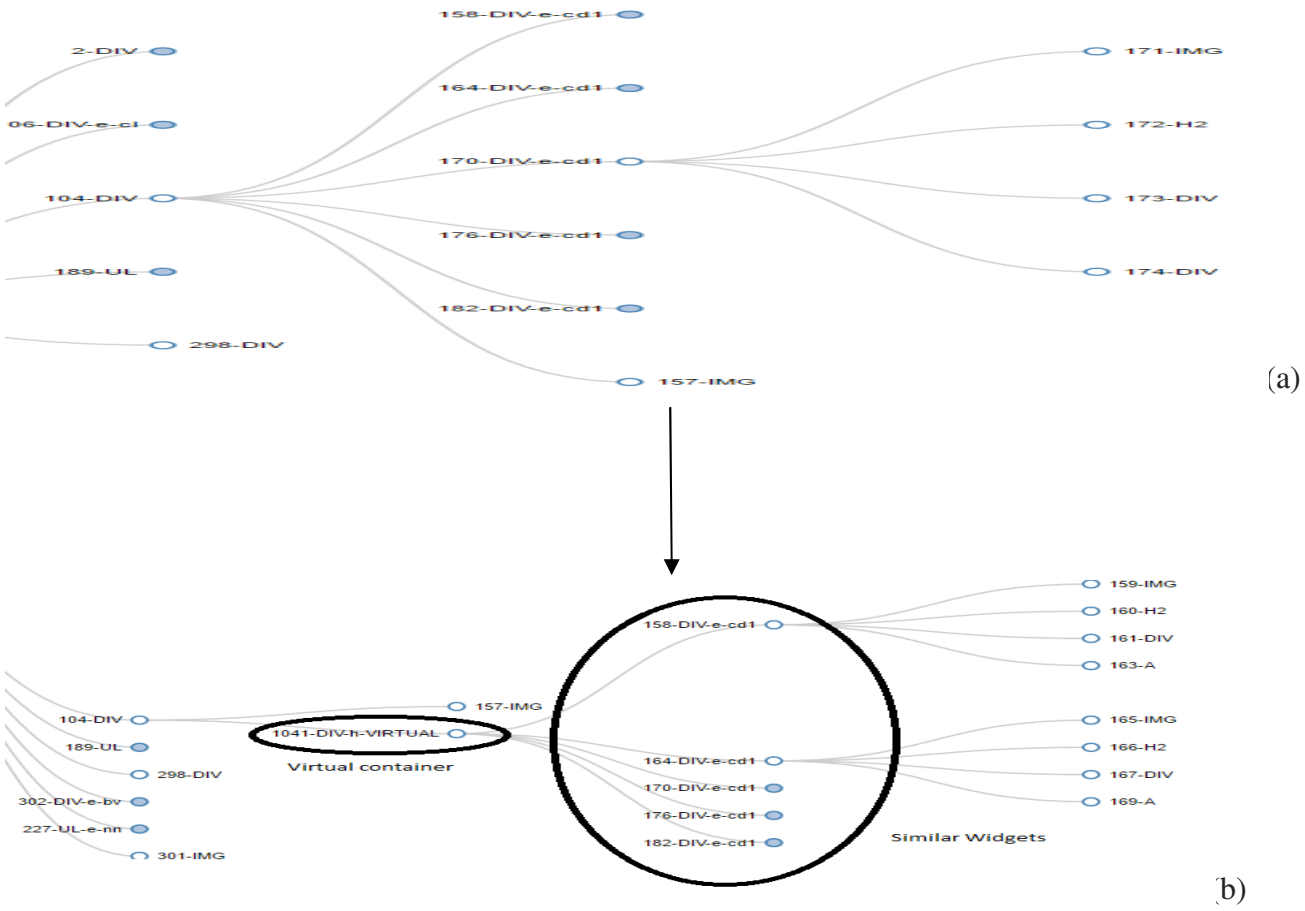

Fig10: (a) Similar widgets under 104-DIV without grouping in unitedway.ca.widget.json (b) Similar widgets under 104-DIV are grouped and set as the children of a Virtual Container (1041-DIV-h-VIRTUAL) in unitedway.ca.virtual.json

Step 5: Dynamic Widget Labelling

input: (i) unitedway.ca.virtual.json contains grouped widgets under virtual container.
output: (ii) unitedway.ca.virtual.json containing Labelled Dynamic Widgets

In order to find dynamic widgets and label them, Load unitedway.ca.virtual.json file

Step 5.1 *Find the dynamic Image Carousels*

- Traverse the tree in unitedway.ca.virtual.json in BOTTOM-UP order and Find a non-labelled container that match the regular expression of the image carousel (cl) in widget.regx. After that we find out whether the container (e.g. 106-DIV in Fig.11) contains repetitive child patterns (encircled in Fig.11).

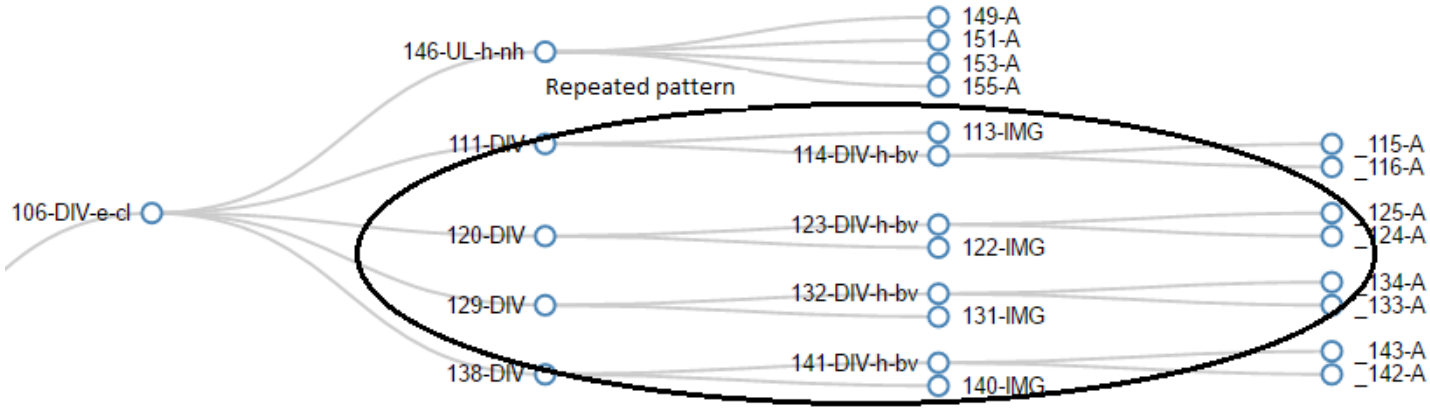

Fig. 11. Sample Image Carousel with 4 recurring child patterns in 111-DIV, 120-DIV, 129-DIV, 138-DIV.

In Fig. 11, A container (106-DIV) has 4 recurring children (111-DIV, 120-DIV, 129-DIV, 138-DIV) and the string representations of these 4 children are portrayed in Fig.12, where the rectangle coordinates in a single pattern are non-zero (e.g., ii) and zero in other patterns (e.g., i, iii, iv).

(i) div:img:div:_a:_a
(ii) div:img:div:_a:_a -> HTML element rectangle coordinates are non-zero for each one
(iii) div:img:div:_a:_a
(iv) div:img:div:_a:_a

Fig.12: String representation of recurring child Pattern of 111-DIV, 120-DIV, 129-DIV, 138-DIV.

The non-zero rectangles implies the HTML elements that are rendered in the browser and vice versa; since image carousel is a dynamic container, some of its elements are not rendered and therefore the coordinates for those elements become zero.

Therefore, the container (106-DIV) fulfills the constraints (e.g., i. image-carousel (*cl*) regular expression matching, ii. repetitive pattern matching iii. one pattern has non zero rectangles and others not) to be considered it as an image carousel. Now, 106-DIV will be labelled as "106-DIV-e-cl" where 106 is the customNodeId, DIV is the container, e signifies heterogeneous container, cl implies image carousel.

Step 5.2 Find the dynamic Navigation bars

In order to find the dynamic navigation bar,

(i). At first, find the navigation bar containers (e.g., 29-UL-e-nh) in unitedway.ca.virtual.json file, portrayed in Fig.13. We also the find their corresponding HTML elements in unitedway.ca.wholepage.html

(ii). Find List of Anchor (A) elements and their corresponding Rectangles, under each child anchor (e.g., 31-A, 48-A, e.t.c) from DOM tree and unitedway.ca.virtual.json in that order.

(iii). If the Rectangles of Anchor (A) elements under the child containers (e.g., 31-A, 48-A) have zero coordinates, then sub-menus under the child containers (e.g., 31-A, 48-A) are dynamic.

(iv). Attach the sub-menu child rectangles under the child containers (e.g., 31-A, 48-A)

(v). Repeat the steps (ii)-(iv), until all the sub-menu traversing is finished.

(vi). Label the static container as Dynamic menu (29-UL-e-nh-DYNAMIC), as depicted in Fig. 14.

(vii). Save the tree in unitedway.ca.virtual.json file which is the ultimate Json file to representing the web page unitedway.ca.wholepage.html

Fig. 13. Static Navigation bar (29-UL-e-nh), already labelled in Step 3 (Static widget labelling)

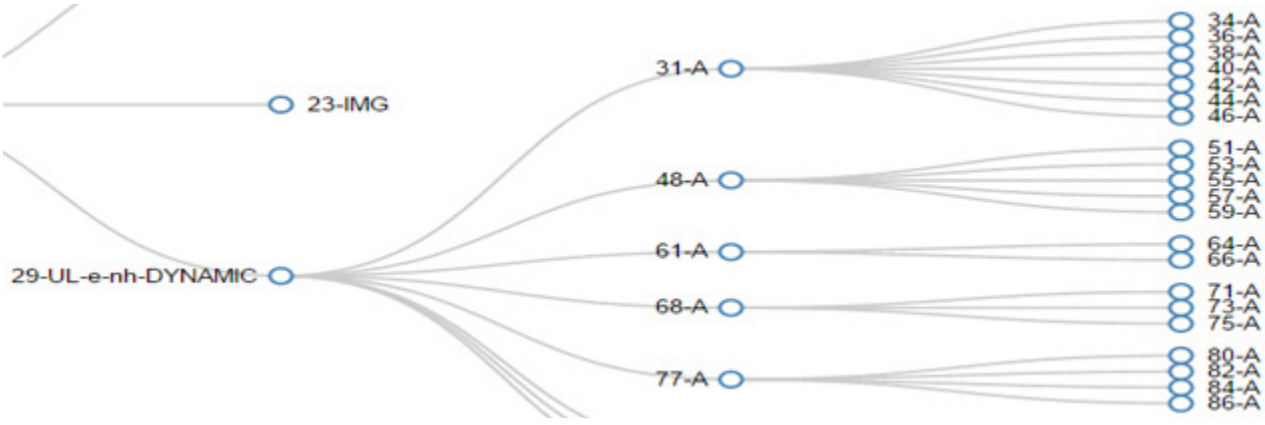

Fig. 14: Dynamic navigation bar (29-UL-e-nh-DYNAMIC) with dynamic sub menus (34-A, 36-A, e.t.c) under each first level menu (31-A)

How Web Page Functional Parsing works

The coding steps to re-generate web page are given below:

1. To generate Json File (unitedway.ca.wholepage.json) consists of HTML Element coordinates for a webpage and corresponding HTML file (unitedway.ca.wholepage.html) with CustomNodeIds, we run the following javscript:

```
var pageModel = {
   "pageUrl": document.URL,
   "elementModels": []
};

function isValidInt(str) {
   if (str == undefined || str == null) return false;
   str = str.toString();
   var numberPattern = /^-?\d+\.?\d*$/;
   return str.match(numberPattern) ? true : false;
}
allDescendants(document.body);

var customNodeId=-1;
function allDescendants(node) {
   var str1 = "";
   for (var j = 0; j < node.attributes.length; j++) {
      if (j == 0) {
         str1 = node.attributes[j].name + "=" + node.attributes[j].value;
      } else {
         str1 = str1 + "|" + node.attributes[j].name + "=" + node.attributes[j].value;
      }
   }
   if (!isValidInt(node.offsetTop) || !isValidInt(node.offsetLeft) || !isValidInt(node.offsetHeight) ||
!isValidInt(node.offsetWidth)) return;
      customNodeId=customNodeId+1;
   pageModel.elementModels.push({
      "nodeName": node.nodeName,
   "customNodeId": "customNodeId_" + customNodeId,
      "nodeProperties": str1,
      "rectangle": {
         "top": Math.round(node.getBoundingClientRect().top),
         "left": Math.round(node.getBoundingClientRect().left),
         "bottom": Math.round(node.getBoundingClientRect().bottom),
         "right": Math.round(node.getBoundingClientRect().right),
         "height": Math.round(node.getBoundingClientRect().height),
         "width": Math.round(node.getBoundingClientRect().width)
      }
   });
   for (var i = 0; i < node.childNodes.length; i++) {
```

```
                var child = node.childNodes[i];
                if (child.nodeType != 1) continue;
                allDescendants(child);
            }
        }
        console.log(JSON.stringify(pageModel));
                        allDescendantsHtml(document.body);
                        var customNodeId = -1;

                        function allDescendantsHtml (node) {
                            if (!isValidInt(node.offsetTop) || !isValidInt(node.offsetLeft) ||
!isValidInt(node.offsetHeight) || !isValidInt(node.offsetWidth)) return;
                            customNodeId = customNodeId + 1;
                            node.setAttribute("customNodeId", "customNodeId_" + customNodeId);
                            for (var i = 0; i < node.childNodes.length; i++) {
                                var child = node.childNodes[i];
                                if (child.nodeType != 1) continue;
                                allDescendantsHtml (child);
                            }
                        }
                        console.log(document);
```

2. The rest of the programs are developed in Java, Spring MVC, Jsp to generate PAT tree (unitedway.ca.tree.json) and labelling widgets (static/dynamic) (unitedway.ca.widget.json), creating virtual nodes (unitedway.ca.virtual.json). We have used Jsoup, an extraneous java library to manipulate DOM tree of HTML file.

2.1 Since we are dealing with different files, we use a central configuration file (dirInfo.txt) saved in the application directory (WEB-INF/classes/dirInfo.txt). This file stores all the necessary directory paths, as shown below. There is a directory key (wholepagejsondir) for each directory.

```
wholepagejsondir=C:\\phantomjs\\wholepagejson\\              //store unitedway.ca.wholepage.json
wholepagehtmldir=C:\\phantomjs\\wholepagehtml\\              // store unitedway.ca.wholepage.json.html
treejsondir=C:\\phantomjs\\treejson\\                        // store unitedway.ca.tree.json
nonTerminalwidgetDir=C:\\phantomjs\\widget\\nonterminal\\    //store www.unitedway.ca.widget.json
nonTerminalVirtualwidgetDir=C:\\phantomjs\\widget\\virtual\\ //store www.unitedway.ca.virtual.json
widgetRegxDir=C:\\phantomjs\\widget\\           //store widget.regx
```

Fig. 15. dirInfo.txt comprises the directory locations used in our application

2.2 To generate PAT tree and store it in unitedway.ca.tree.json we are sending the following request to a controller, *SegmentationController.java*. The <domainname> will be replaced by a domain name (e.g. unitedway). This controller contains a method `invokeBuildPATTreeNodeJson`() that produces the PAT tree and save it to unitedway.ca.tree.json

http://localhost:8080/visualpersing/segmentation/wholepage/<domainname>

2.3 To generate labelled widgets and virtual nodes we send the following request to a controller, *NonTerminalAnnotatingController.java* which has a method, `invokeBuildNonTerminalWidgetTreeAndVirtualNodes()` that produces the labelled widgets (static/dynamic), virtual nodes and saves them in unitedway.ca.widget.json, unitedway.ca.virtual.json file respectively. In addition, it displays the ultimate PAT tree (saved in unitedway.ca.virtual.json) in browser as portrayed in Fig. 16.

`http://localhost:8080/visualpersing/widget/nonterminal/tree/`<domainname>

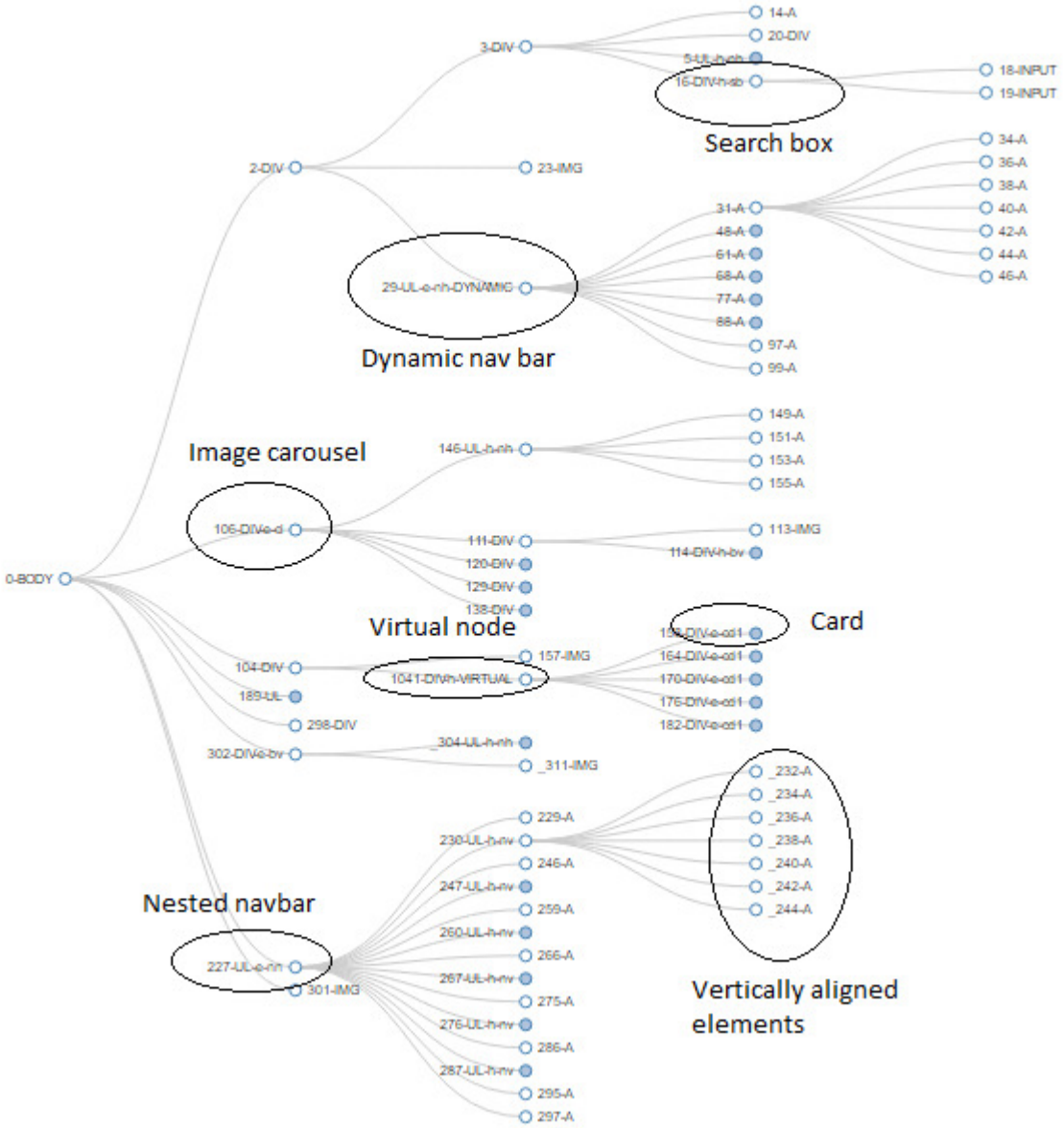

Fig. 16. Ultimate PAT tree (saved in unitedway.ca.virtual.json) with labelled navbars (static/dynamic), image carousel, card, nested navbar, vertically aligned elements, virtual node

Future works

1. We have employed mean (*u*) and standard deviation (*sd*) to prune the non meaningful containers which perhaps not be successful for all PATs since different web page follow different DOM structures. In the future we plan to observe that whether 1.5*sd,* 2*sd,* 3*sd* and so on works better or not in the extent of pruning.

2. We are using regular expressions to detect pattern and label the containers in PAT, which is a greedy approach. The problem of this approach is that, if it fails to label a node, that node will be unlabelled finally. In the future, we plan to use backtracking approach (e.g., http://camlp5.gforge.inria.fr/doc/htmlc/bparsers.html) so that the unlabelled nodes can be labelled later.

3. While the nodes are being labelled, it just takes into account the node string (discussed in step 3: Static Widget Labelling) and match that string against a regular expression. While matching, it does not consider the pattern or label of its child node. Perhaps, it may cause ambiguity problem (e.g., one node string can be matched with different regular expressions), predominantly detecting the nested widgets such as nested cards, nested navigation bars. In the future, we need to tackle this issue, perhaps labelling the nodes using grammar.