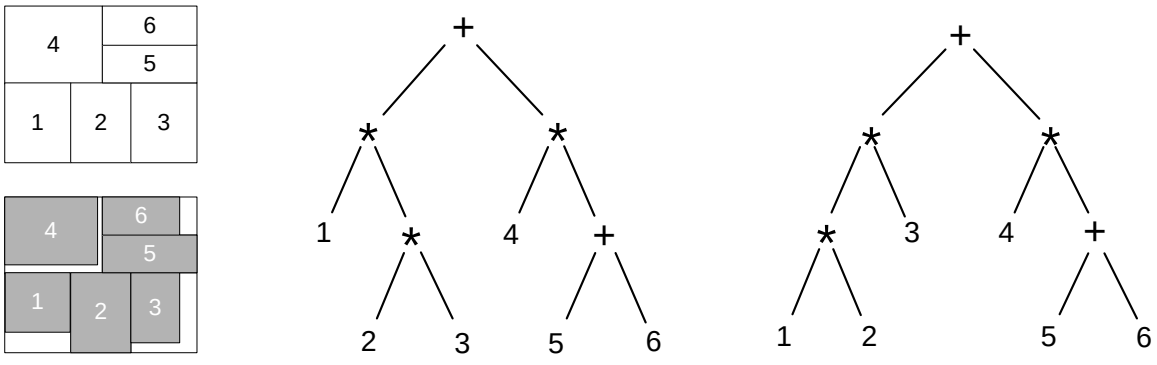
Nachreichung Master's Thesis: Automatische Generierung von Softwarekarten: Entwicklung eines Ansatzes zum Layout deklarativ beschriebener Visualisierungen

Packings (Auszug der Ausarbeitung)

Die unten skizzierten Heuristiken arbeiten auf *Floorplan Trees* eines *Slicing Packings*. *Slicing Packings* haben die Eigenschaft, dass sie sich durch horizontale und vertikale Guillotinenschnitte über die gesamte Breite, bzw. Höhe des Packings sukzessive zerlegen lassen. Die bei einem solchen Guillotinenschnitt entstehenden zwei *Teilpackings* werden als *Supermodules* bezeichnet, die sich ihrerseits wiederum solange zerschneiden lassen, bis sie nur noch ein Rechteck enthalten.

Da sich ein *Supermodule* durch genau einen Schnitt in zwei Teile zerlegen lässt, bietet es sich an, *Slicing Packings* über Binärbaume zu beschreiben. Die Blätter eines solchen *Slicing Trees* bestehen dabei aus den Indizes der zu packenden Rechtecke, während innere Knoten Auskunft über die Schnittoperation (horizontaler oder vertikaler Schnitt) geben. Eine durch einen *Slicing Tree* beschriebene Partitionierung der Ebene in *Supermodules* wird als *Slicing Floorplan* (*Floorplan*) des *Packings* bezeichnet. Ein *Floorplan* lässt sich über einen *Floorplan Tree* beschreiben, der neben den Informationen des *Slicing Trees* auch die Dimensionen und Positionen der *Supermodules* enthält, also anders formuliert Auskunft darüber gibt, bei welchen Koordinaten sich Schnitte befinden. Da sich die Rechtecke in den Blättern eines *Floorplan Trees* befinden, ergibt sich deren Position trivialer Weise aus der Position der entsprechenden *Floorplanzelle*. In einem *Floorplan* verortete Rechtecke werden als *Packing*, bzw. *Teilpacking* bezeichnet.

Slicing Trees lassen sich bequem durch einen *Stack* repräsentieren. Hierzu wird ein *Slicing Tree* in Postfixnotation in diesem *Stack* abgelegt. Offensichtlich ist die Beschreibung eines *Slicing Packings* durch Binärbäume nicht eindeutig.



In der Abbildung ist ein *Slicing Packing* (unten) und der zugehörige *Floorplan* (oben) dargestellt. Auf der rechten Seite finden sich zwei verschiedene *Slicing Trees*, die dem *Floorplan* zugrunde liegen. Vertikale Schnittoperatoren sind mit „*“ und horizontale Schnittoperatoren mit „+“ gekennzeichnet.

Die Berechnung eines *Packings* geschieht über *Branch&Bound*-Enumeration von *Slicing Trees*, wobei der *Deadspace* (nicht verwendete Flächen in *Supermodules*) von Teilpackings

herangezogen wird. Eine detaillierte Beschreibung des *Branch&Bound*-Verfahrens zur Berechnung von *Slicing Packings* findet sich in [CM04].

Ergebnis der *Branch&Bound*-Enumeration ist ein *Slicing Tree*, welcher ein *Packing* mit minimaler, bzw. bei Verwendung der Heuristik von [CM04], näherungsweise minimaler Fläche, beschreibt. Zu diesem Slicing Tree wird ein *Floorplan Tree* berechnet, auf dem die Heuristiken operieren.

Pseudo-Code der Best-Fit-Heuristik

Algorithmus `insertSubTrees`(*tree:FloorPlanTreeNode, subTrees:{FloorPlanTreeNode}, targetWidth:int, targetHeight:int*)

- **while** *subTrees* not empty **do**
- *subTree* := tree of *subTrees* with largest area
- *subTrees* := *subTrees* – {*subTree*}
- [*penalty_right, area_right, deadspace_right*] := `computePenaltyRightInsertion`()
- [*penaly_bottom, area_bottom, deadspace_bottom*] := `computePenaltyBottomInsertion`()
- **if** *penalty_right* > 0 **and** *penalty_bottom* > 0 **and** *subTree* has children **then**
- *newSubTrees* := `cutTree(`*subTree*`)`
- *subTrees* := *subTrees* ∪ *newSubTress*
- **continuewhile**
- **endif**
- [*node, penalty_into*] := `bestFitIntoTree(`*tree, subtree, targetwidth, targetHeight*`)`
- *area_into* := *tree.area* after insertion of *subTree* at *node*
- *deadspace_into* := area_into - $\sum$ area of block in *tree* after insertion of *subTree* at *node*
- **if** ((*penalty_into* < *penalty_right* **and** *penalty_into* < *penalty_bottom*) **or**
- (*penalty_into* = *penalty_right* **and** *area_into* < *area_right*) **or**
- (*penalty_into* = *penalty_bottom and area_into* < *area_bottom*)) **and**
- (*deadspace_into* < *deadspace_right* and *deadspace_into* < *deadspace_bottom*) **then**
- *tree* := `insertTree`(*node, subTree*)
- **else if** (*penalty_right* < *penalty_bottom*) **or**
- *(penalty_right* = *penalty_bottom* **and** *area_right* < *area_bottom*) **then**
- *tree* := `mergeTreesHorizontal`(*tree, subtree*)
- **else**
- *tree* := `mergeTreesVertical`(*tree, subtree*)
- **endif**
- **endwhile**
- **return** *tree*

Anmerkungen zu `insertSubTrees`

- `cutTree` zerlegt den betrachteten Baum in Teilbäume $\{t_1,\ldots,t_n\}$mit, $1 \leq m \leq n$, so dass gilt:
  $t_m.width \leq targetWidth - tree.width \wedge t_m.height \leq \max(targetHeight - tree.height) \vee$
  $t_m.width \leq \max(targetWidth, tree.width) \wedge t_m.height \leq tragetHeight - tree.height \vee$
  t_m hat keine Kinder

- Die Heuristik zerlegt zunächst den umzupackenden *Floorplan Tree* mittels `cutTree`. Der flächenmäßig größte Teilbaum wird in *tree* und die restlichen Teilbäume in *subTrees* übergeben.
- […] in den Zeilen 4 und 5 ist in Anlehnung an die Syntax von MatLab als Vektor der Rückgabewerte zu verstehen.
- {…} steht für eine Menge
- `computePenaltyRightInsertion` berechnet einen Strafwert für das Anhängen des betrachteten Teilbaumes an die rechte Seite des Packings, die Fläche des entstehenden Packings und dessen Deadspace:

  newTableWidth := *tree.width* + *subTree.width*
  newTableHeight := *tree.height* + *subTree.height*
  penalty_right := abs(min(targetWidth – newTableWidth, 0)) +
  abs(min(targetHeight – newTableHeight, 0))

  Analoges gilt für `computePenaltyBottomInsertion`

Algorithmus `bestFitIntoTree`(*tree:FloorPlanTreeNode, subTree:FloorPlanTreeNode, targetWidth:int, targetHeight:int*)

```
    initialization node := null, best_penalty := ∞
1.  if tree is empty then exit
2.  d := abs(min(width of deadspace region of tree – targetWidth, 0)) +
3.       abs(min(height of deadspace region of tree – targetHeight, 0))
4.  if d > 0 then
5.    if tree has children then
6.      width := tree.width after insertion
7.      height := tree.height after insertion
8.      penalty := abs(min(targetWidth – width, 0)) + abs(min(targetHeight – height, 0))
9.      if (penalty < best_penalty) or (penalty = best_penalty and best_area > width * height) then
10.       node := tree , best_peanlty := penalty
11.     endif
12.   else
13.     width := tree.width after horizontal merge
14.     height := tree.height after horizontal merge
15.     penalty := abs(min(targetWidth – width, 0)) + abs(min(targetHeight – height, 0))
16.     if (penalty < best_penalty) or (penalty = best_penalty and best_area > width * height) then
17.       node := tree , best_peanlty := penalty
18.     endif
19.
20.     width := tree.width after vertical merge
21.     height := tree.height after vertical merge
22.     penalty := abs(min(targetWidth – width, 0)) + abs(min(targetHeight – height, 0))
23.     if (penalty < best_penalty) or (penalty = best_penalty and best_area > width * height) then
24.       node := tree , best_peanlty := penalty
25.     endif
26.   endif
27. else
28.   node := tree, best_penalty: = 0
29. endif
30. bestFitIntoTree(tree.leftChild, subTree , targetWidth, targetHeight)
31. bestFitIntoTree(tree.rightChild, subTree, targetWidth, targetHeight)
32. return [node, best_penalty]
```

Pseudo-Code der Heuristik zum Packen von Prozessunterstützungskarten

Algorithmus `compactTable`(*table:Array(FloorPlanTreeNode)*)

```
1.  compute optimization potential for each column of the table
2.  compute optimization potential for each row of the table
3.  while true do
4.     c := the column with the biggest potential
5.     r := the row with the biggest potential
6.     if potential of c = -1 and potential of r = -1 then exit
7.     if potential of c > potential of r then
8.        if compactCol(c) then potential of c := -1
9.     else
10.       if compactRow(r) then potential of r := -1
11.    endif
12. endwhile
```

Anmerkungen zu `compactTable`

- Bevor `compactTable` in der *solve()*-Methode der *ProcessSupportPattern-Occurrence* angestoßen wird, wurden für die Zellen der Prozessunterstützungskarte *Packings* mit der Hierarchischen Packing Heuristik von [CM04] berechnet und in einem zweidimensionalen Array (*table*) in Form von *Floorplan Trees* gespeichert. *column* und *row* indizieren dieses Array.
- Das Optimierungspotential einer Zeile berechnet sich aus der Summe der Differenzen aus der Höhe der einzelnen Packings und der Zeilenhöhe.
- Analoges gilt für das Optimierungspotential einer Spalte.
- `compactCol` und `compactRow` liefern wahr, wenn eine Verbesserung erzielt werden konnte, ansonsten falsch.

Algorithmus `compactRow`(*r:int, table:Array(FloorPlanTreeNode)*)

```
1.  while true do
2.  maxBlockHeight := largest height of bottom most block of the packings in the row
3.  newRowHeight := height of row – maxBlockHeight
4.  if newRowHeight has not changed then breakwhile
5.  for each column do
6.     if height of packing in cell (column, r) > newRowHeight then
7.        repack packing of cell (column, r) to newRowHeight
8.     endif
9.     if width of cell (column, r) > old width of column then
10.       width of column := width of cell (column, r)
11.       for each row do
12.          repack packing in cell (column, row) to width of cell (column, r)
13.       endfor
14.    endif
15. endfor
16. if  new table.area < old table.area then
17.    set best solution to new table
18.    return [true, table]
```

```
19. endif
20. endwhile
21. discard table
22. while true do
23.     maxBlockWidtht := largest height of right most block of the packings in the row
24.     newRowHeight := height of row + maxBlockHeight
25.     for each column do
26.         repack packing in cell (column, r) to newRowHeight
27.         if width of cell (column, r) has increased then
28.             for each row do
29.                 h := height of packing if repacked to new width of cell (column, r)
30.                 if h ≤ height of row then repack packing of cell (column, row) to width of cell (column, r)
31.                 compute height of row
32.             endfor
33.         endif
34.     endfor
35.     if  new table.area < old table.area then
36.         set best solution to new table
37.         return [true, table]
38.     endif
39. endwhile
40. discard table
41. return [false, table]
```

Analoges gilt für `compactCol`

Anmerkungen zu `compactCol` und `compactRow`

- Zum Umpacken von Zellen auf neue Höhen oder Breiten wird auf die Best-Fit-Heuristik zurückgegriffen.
- Wurde eine bessere Lösung gefunden, werden die umgepackten *Packings* übernommen und die Optimierungspotentiale der Spalten und Zeilen neu berechnet. Ansonsten werden die *Packings* verworfen.

Pseudo-Code der Heuristik zum nachträglichen Verbessern von Clusterkartenlayouts

Algorithmus `compactPacking(`*tree: FloorPlanTreeNode*`)`
1. **while** old *tree.area* > *tree.area* **do**
2. `compactPackingToHeight(`*tree, tree.height*)
3. **if** *tree.area* >= old *tree.area* **then** reset *tree*
4. `compactPackingToWidth(`*tree, tree.width*)
5. **if** *tree.area* >= old *tree.area* **then** reset *tree*
6. **endwhile**
7. `return` *tree*

Algorithmus `compactToHeight(`*tree:FloorPlanTreeNode, height:int*)
1. **if** *tree.root* has no children **then** resize *node.box* to *height*
2. **else**
3. **if** vertical merge **then**
4. **if** *tree.height* < *height* **then**
5. equally increase height of left and right child of *tree* to make the sum fit *height*
6. compact the packings of the children of *tree* to the new heights by calling `compactToHeight`
7. **else if** *tree.height* > *height* **then**
8. **if** *tree.leftChild.widht* < *tree.rightChild.width* **then**
9. `compactToHeight(`*tree.LeftChild, height-tree.rightChild.height*`)`
10. `compactToWidth(`*tree.rightChild, max(tree.leftChild.width, tree.rightChild.width*`))`
11. **else**
12. `compactToHeight(`*tree.rightChild, height-tree.lefttChild.height*`)`
13. `compactToWidth(`*tree.leftChild, max(tree.leftChild.width, tree.rightChild.width*`))`
14. **endif**
15. **else**
16. `compactToHeight(`*tree.leftChild, tree.leftChild.height*`);`
17. `compactToHeight(`*tree.rightChild, tree.rightChild.height*`);`
18. **if** *tree.width* has not changed **then**
19. **if** *tree.leftChild.width* < *tree.rightChild.width* **then**
20. `compactToWidth(`*tree.leftChild, tree.rightChild.width*)
21. **else if** *tree.rightChild.width* < *tree.leftChild.width* **then**
22. `compactToWidth(`*tree.rightChild, tree.leftChild.width*)
23. **endif**
24. **endif**
25. **else**
26. `compactToHeight(`*tree.leftChild, height*`);`
27. `compactToHeight(`*tree.rightChild, height*`);`
28. **endif**
29. **endif**
30. **return** *tree*

Analoges gilt für `compactToWidth`:

- Alle `compactToHeight` durch `compactToWidth` Aufrufe ersetzt
- Alle `compactToWidth` durch `compactToHeight` Aufrufe ersetzt
- Alle *width*-Attribute durch *height*-Attribute ersetzt
- Alle *height*-Attribute durch *width*-Attribute ersetzt
- *height* durch *width* ersetzt

Anmerkungen zu `compactToWidth` und `compactToHeight`

- Zum Umpacken von Boxen auf neue Höhen oder Breiten wird auf die Best-Fit-Heuristik zurückgegriffen.
- *Vertical merge* bezieht sich auf die Schnittoperation des betrachteten Knotens. Ein *vertical merge* entspricht einem horizontalen Schnitt.

Anmerkungen zum nachträglichen Verbessern von Clusterkartenlayouts

- Die Heuristik `compactPacking` wird nach der Berechnung des Layouts eines Clustermustervorkommens durch den Algorithmus von [CM04] aufgerufen.
- Die Dimensionen einer Box (Zeile 1 von `compactToHeight/Width`) können durch umpacken mittels der Best-Fit-Heuristik nur verändert werden, wenn sie Abbild eines tiefer *geclusterten* Packings ist.
- Dimensionen von Boxen, die Abbild eines einzigen Kartensymbols sind, werden nicht verändert.

Literatur

[CM04] Chan, H.; Markov, I.: *Practical Slicing and Non-slicing Block-Packing without Simulated Annealing*. ACM Great Lakes Symposium on VLSI, S. 282-287, 2004.