Bioinformatics Summative Assignment

hzwr87

1 This question is about the Build algorithm from [1].

1.1 Explain what the algorithm does and how it works in your own words. Do not use pseudocode [25 marks]

The BUILD algorithm recursively builds a tree using constraints of the form $(i, j) < (k, l)$where $i \neq j$ and$k \neq l$. This notation means that the lowest common ancestor of i and j is a proper descendant of the lowest common ancestor of k and l. The algorithm itself requires a set of such constraints and and a non-empty set of nodes. If the set of nodes has only 1 element, then the algorithm returns a tree consisting of this element alone. Otherwise, the set of nodes is partitioned into blocks based on 3 rules.

1. If $(i, j) < (k, l)$ is a constraint, then both i and j are in the same block
2. If $(i, j) < (k, l)$ is a constraint, and k and l are in the same block, then i, j, k, and l are all in the same block.
3. No two nodes are in the same block unless specified by the above rules.

For each of the blocks created by this partitioning method, we find which of the original constraints are relevant to each block, which means all of i, j, k, and l are in this block. We then call the BUILD algorithm again with this new set of constraints and nodes. This continues until BUILD is called on only 1 node, in which case it returns a tree consisting of only this node, which will be a leaf of the overall tree.

1.2 Expand the partition step in pseudocode: compute $\pi_C = S_1, S_2, ...S_r$; [20 marks]

Algorithm 1 Partition Step of BUILD(S,C)

```
π = S
//Apply rule 1
for c in C:
    i,j,k,l = c
    if i and j are not in the same block:
        merge blocks in π containing i and j
repeat
//Apply rule 2
for c in C:
    i,j,k,l = c
    if k and l are in the same block of π and i and j are in a different block of π :
        merge blocks of π containing i, j and k, l
end repeat if rule 2 had no effect
return π
```

1.3 Write a recurrence that expresses the running time of Build depending on the number of different leaf-labels n and the number of constraints m. Use it to estimate the running time of the algorithm assuming that the partitioning step runs in time $f(n, m)$ for some function monotonically non-decreasing function f. [25 marks]

Here we assume that the partition step from the previous question runs in $f(n, m)$ time. As this is monotonic, it will always be expressed below as the worst case $f(n, m)$.

$T(1, 0) = O(1)$

$T(n, m) < KT(n_i, m_i) + f(n, m)$

Where K is the number of sets in the the partition π_C and i represents the ith block in π_c, where $0 \leq i \leq K$.

$T(n, m) < K(K^{'}T(n_i, m_i) + f(n, m)) + f(n, m)$

$T(n, m) < K(K^{'}(K^{''}T(n_i, m_i) + f(n, m)) + f(n, m)) + f(n, m)$

The number of calls to BUILD referred to by $K \times K^{'} \times K^{''}$... is bounded by $(2n - 1)$, and the number of partioning calls (i.e. calls to $f(n, m)$) is bounded by $(n - 1)$, we can infer:

$T(n, m) < (2n - 1)T(1, 0) + (n - 1)f(n, m)$

$T(n, m) < (2n - 1)O(1) + (n - 1)f(n, m)$

$T(n, m) = O(n + nf(n, m))$

1.4 Run the algorithm on the following set of constraints. You should show the partitioning and the recursive calls at each stage. [25 marks]

1. $(e, f) < (k, d)$
2. $(c, h) < (a, n)$
3. $(j, n) < (j, l)$
4. $(c, a) < (f, h)$
5. $(j, l) < (e, n)$
6. $(n, l) < (a, f)$
7. $(d, i) < (k, n)$
8. $(d, i) < (g, i)$
9. $(c, l) < (g, k)$
10. $(g, b) < (g, i)$
11. $(g, i) < (d, m)$
12. $(c, h) < (c, a)$
13. $(e, f) < (h, l)$
14. $(j, l) < (j, a)$
15. $(k, m) < (e, i)$
16. $(j, n) < (j, f)$

BUILD$((a, b, c, d, e, f, g, h, i, j, k, l, m, n), ((e, f) < (k, d)\ , (c, h) < (a, n)\ , (j, n) < (j, l)\ , (c, a) < (f, h)\ , (j, l) < (e, n)\ , (n, l) < (a, f)\ , (d, i) < (k, n)\ , (d, i) < (g, i)\ , (c, l) < (g, k)\ , (g, b) < (g, i)\ , (g, i) < (d, m)\ , (c, h) < (c, a)\ , (e, f) < (h, l)\ , (j, l) < (j, a)\ , (k, m) < (e, i)\ , (j, n) < (j, f)))$

Applying partition rule 1 gives the blocks (e, f), (a, c, h, j, l, n), (b, d, g, i), (k, m)
Applying partition rule 2 gives the blocks (a, c, e, f, h, j, l, n), (b, d, g, i), (k, m)

- BUILD$((a,c,e,f,h,j,l,n),\ ((c,h) < (a,n)\ ,\ (j,n) < (j,l)\ ,\ (c,a) < (f,h)\ ,\ (j,l) < (e,n)\ ,\ (n,l) < (a,f)\ ,\ (c,h) < (c,a)\ ,\ (e,f) < (h,l)\ ,\ (j,l) < (j,a)\ ,\ (j,n) < (j,f))$

  Applying partition rule 1 gives the blocks (a,c,h), (j,l,n), (e,f)
  Applying partition rule 2 changes nothing

  - BUILD$((a,c,h),\ ((c,h) < (c,a))$

    Applying partition rule 1 gives the blocks (c,h), (a)
    Applying partition rule 2 changes nothing

    * BUILD$((c,h),\ ())$

      By partition rule 3, as there are no constraints, these will then be in separate blocks.

      ⋆ BUILD$((c),\ ())$

        This returns a tree only containing the node c.

      ⋆ BUILD$((h),\ ())$

        This returns a tree only containing the node h.

    * BUILD$((a),\ ())$

      This returns a tree only containing the node a.

  - BUILD$((j,l,n),\ ((j,n) < (j,l)))$

    Applying partition rule 1 gives the blocks (j,n), (l)
    Applying partition rule 2 changes nothing

    * BUILD$((j,n),\ ())$

      By partition rule 3, as there are no constraints, these will then be in separate blocks.

      ⋆ BUILD$((j),\ ())$

        This returns a tree only containing the node j.

      ⋆ BUILD$((n),\ ())$

        This returns a tree only containing the node n.

    * BUILD$((l),\ ())$

      This returns a tree only containing the node l.

  - BUILD$((e,f),\ ())$

    By partition rule 3, as there are no constraints, these will then be in separate blocks.

    * BUILD$((e),\ ())$

      This returns a tree only containing the node e.

    * BUILD$((f),\ ())$

      This returns a tree only containing the node f.

- BUILD$((b,d,g,i),\ ((d,i) < (g,i)\ ,\ (g,b) < (g,i)))$

  Applying partition rule 1 gives the blocks (d,i), (b,g)
  Applying partition rule 2 changes nothing

  - BUILD$((d,i),\ ())$

    By partition rule 3, as there are no constraints, these will then be in separate blocks.

    * BUILD((d), ())

      This returns a tree only containing the node d.
    * BUILD((i), ())

      This returns a tree only containing the node i.
  - BUILD((b, g), ())

    By partition rule 3, as there are no constraints, these will then be in separate blocks.
    * BUILD((b), ())

      This returns a tree only containing the node b.
    * BUILD((g), ())

      This returns a tree only containing the node g.

- BUILD((k, m), ())

  By partition rule 3, as there are no constraints, these will then be in separate blocks.

  - BUILD((k), ())

    This returns a tree only containing the node k.
  - BUILD((m), ())

    This returns a tree only containing the node m.

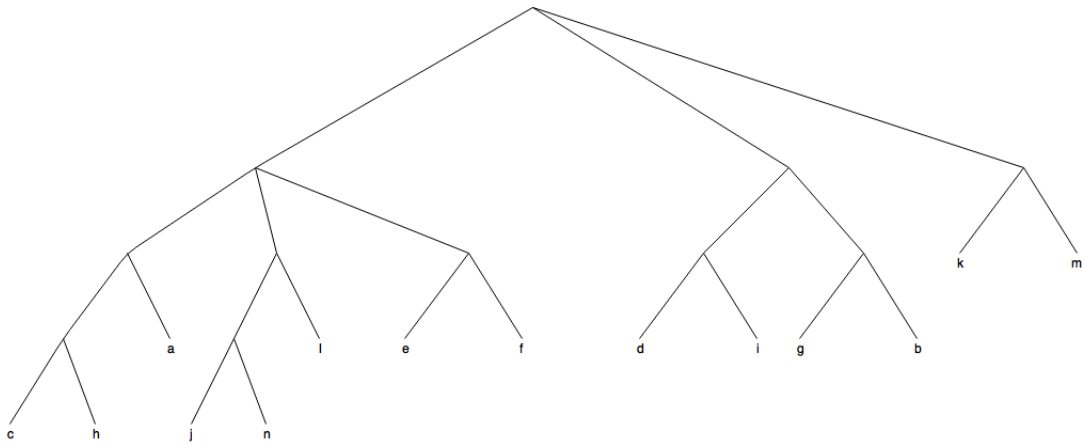

Figure 1: Final tree produced by BUILD on this set of nodes and constraints

1.5 Reverse the Build algorithm, i,e. design an algorithm that takes a tree with labelled leaves as an input, and produces a set of constraints of the form $(i, j) < (k, l)$, such that when Build runs on that set, the result is (an isomorphic copy of) the input tree. Prove the correctness of your algorithm. Also, a smaller output (number of constraints) would give you a better mark. [30 marks]

The reverse of the BUILD algorithm can best be described as a sequence of steps.

1. Label each internal node of the tree with the Lowest Common Ancestor of two of the leaves of its children. For example, the root of the tree from the previous question could be labelled (c, m)

2. Any leaf that is not referred to by any of the internal node constraints should get a label of itself and a leaf descendent of one of its siblings. (This occurs when an internal node has 3 or more leaf children).
3. Remove all leaves that have not been labelled by step 2 from the tree.
4. Create a constraint for each of the remaining nodes of the tree, of the form (node) $<$ (parent). There are 2 sub conditions
   (a) The root is not on the left-hand-side of any constraint
   (b) The leaf constraints from step 2 must be labelled of the form (node) $<$ (parent's parent) to ensure that they are true descendents.

This ensures that all of the leaves of the tree will appear on the left-hand-side of at least one constraint, so will all be appropriately represented in the final graph. If the tree is binary, then all leaves will appear in the label of their parent. Due to the formation of the constraints as (children)$<$(parent), the correct subtree structure will be preserved. In the case of a non binary tree, there will be children that are not represented by their parent's label. The constraint added by step 2 of this reverse algorithm ensures that these nodes are placed in the correct subtree.

2 This question is about the MinCutSupertree algorithm from [2].

2.1 One of its properties is that it preserves nesting and subtrees that are shared by all of the input trees. Point where precisely in the algorithm this property is achieved. [15 marks]

The property of preserving nesting and subtrees that are shared by all of the input trees is achieved in line 5 of the pseudocode when the graph $S_{\mathscr{T}}/E_{\mathscr{T}}^{max}$ is constructed. This graph preserves subtrees that occur in all input trees as the edges between them will always be contracted when forming $S_{\mathscr{T}}/E_{\mathscr{T}}^{max}$, so they will remain together in the output graph. Nesting is also preserved, as if a subtree A is nested within a subtree B, then calling MinCutSupertree on B will end up with the nodes of A being contracted, preserving this subtree and therefore the nesting.

2.2 Argue that the MinCutSupertree algorithm is a generalisation of the Build algorithm, i.e. show how to encode a constraint from the inputs of the later as a tree, which is one of the inputs of the former. [10 marks]

The constraints for the BUILD algorithm can be converted into subtrees by considering 2 cases. Constraints can only be made up of either 3 or 4 unique nodes for example $(i, j) < (k, l)$ or $(i, j) < (k, i)$. By running the BUILD algorithm on each of these constraints individually, we get the 2 possible tree patterns in the Figure below.

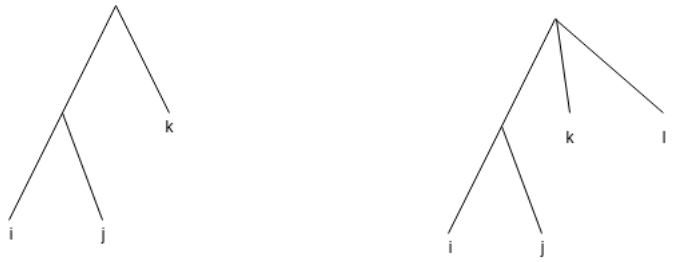

Figure 2: Two possible subtrees

Any constraint that is an input to BUILD can be encoded as one of these 2 trees, and from here the MinCutSupertree algorithm can be run to give us the final tree.