Finding big boxes in partition strategy

Andreas Holck Høeg-Petersen

June 22, 2022

1 Algorithm

We start with a decision tree $\mathcal{T}$ representing af complete partitioning of the state space. Let $\mathcal{V}$ be the set of variables in the state space (ie. its dimensions) and let $\mathcal{L}$ be the set of all leaves in the tree, where each leaf L_i has a state $S_i = \{(l_{i,1}, u_{i,1}), \ldots, (l_{i,m}, u_{i,m})\}$ consisting of a list of tuples of lower and upper bounds for each of the m variables in $\mathcal{V}$. This state represents one m-dimensional box or area in the state space.

We then define $\mathcal{C}$ as the set of constraints in the partitioning given by all the upper bounds, ie. $\mathcal{C} = \{u_{1,1}, \ldots, u_{1,m}, u_{2,1}, \ldots, u_{n,m}\}$. Lastly, we define a mutable list $\mathcal{P}$ in which we store all the points from which our searches for mergable partitions begin. If we denote the minimum value of each variable $V_i \in V$ as $v_i^{\min}$ then we initialise $\mathcal{P}$ as $\mathcal{P} = \{(v_1^{\min}, \ldots, v_m^{\min})\}$.

The algorithm works by continually popping the next element of $\mathcal{P}$ and using this point p as a starting point. From p we construct an infinitesmally small box $b = \{(p_1, p_1), \ldots, (p_m, p_m)\}$ and then query $\mathcal{T}$ to find out what action a_p is chosen at this symbolic state (which at this point is the same as querying at point p). Then we pick the constraint $u_{i,j} \in \mathcal{C}$ whose euclidean distance to the upper bound of b in dimension j is the smallest and update b to reflect this new upper bound in dimension i. That is, we replace $(p_j, p_j) \in b$ with $(p_j, u_{i,j})$.

Now we again query $\mathcal{T}$ for the action(s) in b and continue updating b untill we get more than one type of action. We then know, that the latest update of b expanded the box into an area of the state space where a different action is optimal, and thus we should not include it in our current box. We therefore roll back the latest update and mark that particular dimension as 'exhausted'. If there are still unexhausted variables (dimensions) we continue as before, but now discarding any constraints on the exhausted dimensions.

If after we have rolled back our update, we see that all variables are exhausted, we then have a box b that spans the largest possible area (hyperplane?) from p with only one optimal action. We store this and then add to $\mathcal{P}$ the edge points of b. That is, if $b = \{(l_1, u_1), \ldots, (l_m, u_m)\}$ then we let $\mathcal{P} = \mathcal{P} \cup \{(u_1, l_2, \ldots, l_m), (l_1, u_2, \ldots, l_m), \ldots, (l_1, l_2, \ldots, u_m)\}$. After that, we start over with the next point in $\mathcal{P}$ and continue until we have emptied $\mathcal{P}$.

Below is the pseudocode for the algorithm which includes a couple of more checks and edge cases needed for the algorithm to work. However, several details are still left out or relegated to (hopefully) helpfully named functions.

Algorithm 1 Find boxes

```
1: function FindBoxes(𝒯, 𝒞, 𝒱, p_start)
2:     boxes ← {}
3:     𝒫 ← {p_start}
4:     while 𝒫 is not empty do
5:         exhausted ← {}
6:         𝒫.sort()                                              ▷ left to right
7:         p ← 𝒫.pop()
8:         if p has been visited then          ▷ can be checked by maintaining a separate tree
9:             continue
10:        b ← MakeBoxFromPoint(p)
11:        SortAccordingToDistance(𝒞, p)
12:        for u_{i,j} in 𝒞 do
13:            if u_{i,j} <= p_j or V_j ∈ exhausted then                ▷ p_j is value of V_j in p
14:                continue
15:            oldVal ← b_j^u                                  ▷ b_j^u is the upper bound of variable j in b
16:            b_j^u ← u_{i,j}
17:            if Count(𝒯.actionsAtState(b)) > 1 or b is explored or b_j^u = V_j^max  then
18:                exhausted.add(V_j)
19:                if exhausted.size < 𝒱.size then
20:                    continue
21:                if b_j^u < V_j^max then                  ▷ V_j^max is the maximum possible value of V_j
22:                    b_j^u ← oldVal
23:                boxes.add(b)
24:                for p' in EdgePoints(b) do
25:                    𝒫.add(p')
26:    return boxes
```