6.6 (R) Reach Set Approximation

Motivation: *Reach set* is strong tool for *Obstacle Avoidance* because it contains all possible *avoidance maneuvers* in set. The current implementation have following flaws:

1. *Realistic approximation* - *nonlinear systems* or *heavily constrained systems* can not be approximated well by *continuous-time Reach Sets*.

2. *Non Deterministic calculations* - continuous-time *Reach Set* contains infinite possibilities for *avoidance maneuvers*, the SAA system demands conflict resolution in finite time.

3. *Property binding* - binding related properties seems problematic, because *continuous- time reach sets* does not have unique identifier of maneuver, trajectory nor segment.

Proposed Solution Features: Our Reach set Estimation method will provide following features:

1. *System Control Interface* - implemented via *Movement Automaton*, requiring only *discrete command chain* to approximate system behaviour.

2. *Deterministic Calculation* - finite number of elements in *Reach set* will enable *scalable* calculation.

3. *Property binding* - approximation of Reach set as a set of trajectories, each trajectory can be split into finite number of segments. Each element will have unique identifier enabling both-side property binding.

4. *Behaviour encoding* - some specific behaviour, like horizontal/vertical separation, or maneuver shape can be encoded into *Reach Set*.

Discretization of Reach set: There is a need for a discrete finite *Reach Set approximation* to enable *Avoidance Strategy Evaluation* in finite time. Replacing *Continuous Control Set Inputs(t)* by *Movement Automaton* is feasible:

Definition 1 (Reach set Approximation by Movement Automaton)**.** *A trajectory (def.* **??***) for system $\dot{state} = f(time, state, input)$ under control of the movement automaton $\mathcal{MA}$ is given as execution of movement buffer (def.* **??***) with initial state of system $state_0$. Therefore notation $Trajectory(state_0, buffer)$ is used.*

The Complete Reach Set *(6.1) for system with initial state $state_0$ with existing* control strategy $control(time) \in Controls(time)$. *for time $\tau > time_0$.*

$$ReachSet(\tau, time_0, state_0) = \bigcup \{state(s) : control(s) \in Controls(s), s \in (time_0, \tau]\} \tag{6.1}$$

The Reach Set Approximation by Movement Automaton *(6.2) of the system under the control of the movement automation $\mathcal{MA}$ consist from the set of trajectories $Trajectory\ (state_0,\ Buffer)$, which are executed in constrained time $\tau > time_0$.*

$$ReachSet(\tau, time_0, state_0) = \left\{ Trajectory(state_0, buffer) : \begin{matrix} duration(buffer) \\ \leq \\ (time_0 - \tau) \end{matrix} \right\} \quad (6.2)$$

Note. Reach Set Approximation (def. 1) is subset of *Full Reach Set* (def. **??**) in continuous space $\mathbb{R}^n$ it inherits all important properties, like *Invariance* [1].

Discretization of *Reach Set* have been achieved leaving us with *finite count* of *Trajectories*, instead of *Infinite subspace or* $\mathbb{R}^N$

Approximated Reach Set Containment: The *Approximated Reach Set* introduced in (def. 1) is constrained only by *future expansion time* τ. UAS makes space assessment in *Avoidance Grid*. There is no point to consider Trajectories outside of *Avoidance Grid*

Definition 2 (Contained Aproximated Reach Set)**.** *For pair ($state_0$, $AvoidanceGrid_0$) at time $time_0$ and* prediction horizon $\tau = \infty$ *there is* Contained Reduced Reach Set:

$$ReachSet \begin{pmatrix} time_0, \\ state_0, \\ AvoidanceGrid_0 \end{pmatrix} = \left\{ \begin{matrix} Trajectory(\dots) \\ \in \\ ReachSet(6.2) \end{matrix} : \begin{matrix} \forall segment \in AvoidanceGrid_0, \\ segment \in Trajectory(\dots) \end{matrix} \right\} \quad (6.3)$$

Properties: Container Aproximated Reach Set *contains only trajectories where all segments belongs to* Avoidance Grid*, there are following functions:*

1. Membership function *for any* Trajectory *in* Constrained Reduced Reach set *returns* Ordered Set *of* Passing Cells.

2. Cost function *for any* Trajectory Portion *in* Constrained Reduced Reach Set *return* Cost of Execution

Passing cell: Cell *of* Avoidance Grid *which has some intersection with Trajectory.*

Note. Contained Reduced Reach Set (eq. 6.3) which is contained in *Avoidance Grid* and have an *Membership Function* enable Property transition between Reach set and *Avoidance grid.*

Example: Visibility from cells along *Trajectory* can be gathered to calculate *Trajectory's* feasibility.

Reach Set Pruning: There is a need to implement *Set Difference* between *Reach Set* and *Constraint Set*. Constraint Set cam be *Obstacle Set* from *Known World* (sec. **??**) and other different constraints.

Reach Set Trajectory Tree: (6.4) *Any Reach Set* where *Control Strategy Constraint* is implemented as *Movement Automaton*, with defined *Movements* set and for singe initial $state_0$. The *Reach Set* is given as discrete tree with root $Trajectory(state_0, \varnothing)$.

$$ReachSet(state_0, \dots) = \left\{ Trajectory(state_0, buffer) : \begin{matrix} buffer \in Movements^i, \\ i \in \{1, \dots, k\} \end{matrix} \right\} \quad (6.4)$$

For each *Trajectory Segment*, there exists *intersection function* which evaluates as true if there exists at least one point in *Segment* which belongs to *Constraint Set*. Formally:

$$intersection(segment, Set) : \begin{cases} \exists point \in segment, & : true \\ \ point \in Set & \\ Otherwise & : false \end{cases} \quad (6.5)$$

Definition 3 (Pruned Reach Set)**.** *For* Reach set *represented as* Trajectory Tree *(eq. 6.4) and some constraint set (Set) where exist* intersection function *(eq. 6.5). The* Pruned Reach set *is given as follows:*

$$Prune(ReachSet, Set) = \left\{ Trajectory(\dots) : \begin{matrix} \forall segment \in Trajectory, \\ \neg intersection(segment, Set) \end{matrix} \right\} \quad (6.6)$$

Note. Pruning(def. 3) [2] is applicable multiple times for various *Constraints Set.*

Example of *Approximated Reach set Calculation* (def. 1), *Reach Set Containment* (def. 2), and, *Pruning* is given in [3].

6.6.1 (R) Reach Set Performance Criteria

Motivation: The need to Make *Reach Set* scalable approach. This may be a problem due the *Expansion rate*. *Reach set* represented as a *Trajectory Tree* (eq. 6.4) for Avoidance Grid with *layerCount* and Movement automaton with *movementCount*, the *Node count* is given as:

$$1 + \left(\sum_{i \in \{1 \dots layerCount\}} (movementCount)^i \right) \quad (6.7)$$

This scaling is not feasible for *Avoidance Grid* with many layers (< 10) or *Movement Set* with many movements (< 9). There is need for *Reduced Reach set calculation.*

Core Performance Criteria: The scaling factor (eq. 6.7) shows that there are going to be many trajectories. The main point is that not every trajectory in *Reach Set* are giving us *maneuverability advantage*. Our expectations lies in following *Performance Requirements*:

1. *Reach set* must *Cover* maximum of the *possible unique maneuvers* in *Avoidance Grid.*

2. *Trajectories* in *Reach Set* should be smoothest possible to prevent cargo damage / UAS wear.

Trajectory footprint: Discrete space of *Avoidance Grid* is organized in cells. *Cell* is minimal space portion accessible by *property binding.* There is need to know if two trajectories contribution to *Maneuverability* in this environment.

Each trajectory passes trough space in *Avoidance Grid.* If there exists a method to extract unique identifier for each *trajectory passed cells*, we can compare two trajectories *Coverage* in *Avoidance Grid.*

Definition 4 (Trajectory footprint)**.** *For* Trajectory *from* Reach set *(def. 2) defined for* Avoidance Grid *has membership function.* Membership Function *returns* ordered set of passing cells*:*

$$footprint\begin{pmatrix} Trajectory, \\ AvoidanceGrid \end{pmatrix} = \begin{Bmatrix} cell \in AvoidanceGrid : \\ isMember(trajectory, cell) \end{Bmatrix} \tag{6.8}$$

Then we can define equality function for $Trajectory_1$ and $Trajectory_2$, as comparison of their footprints in common Avoidance Grid *as follow:*

$$isEqual\begin{pmatrix} Trajectory_1, \\ Trajectory_2, \\ AvoidanceGrid \end{pmatrix} : \begin{cases} \begin{pmatrix} footprint(Trajectory_1, \dots) \\ = \\ footprint(Trajectory_2, \dots) \end{pmatrix} & : true \\ Otherwise & : false \end{cases} \tag{6.9}$$

Note. Depending on *Movement Automaton‘s* movement set and *Avoidance Grid* parameters, there can be multiple *trajectories* which are equal.

Coverage set: Now it is possible to create set of unique *trajectory footprints* due to *footprint function* (eq. 6.8). Similarly there is a possibility to create *Reach set skeleton* containing unique trajectories, by using *equality function* (eq. 6.9). *Coverage set* is sufficient for now.

Definition 5 (Coverage Set)**.** Coverage set *(6.10) is defined for* Avoidance Grid *and* Reach Set *pair as set of unique* Trajectory footprints*:*

$$CoverageSet\begin{pmatrix} AvoidanceGrid, \\ ReachSet \end{pmatrix} = \left\{ footprint\begin{pmatrix} Trajectory, \\ AvoidanceGrid \end{pmatrix} : \begin{matrix} \forall Trajectory \\ \in ReachSet \end{matrix} \right\} \tag{6.10}$$

Coverage set properties: Trajectory footprint (eq. 6.8)is not *bijection*, neither *injection* for $ReachSet \rightarrow CoverageSet$. This implies following properties:

1. Equal *Reach Sets* in same *Avoidance Grid* have equal *Coverage Sets.*

2. Equal *Coverage Sets* does not imply *Reach Set* equality.

3. For two Coverage Sets there is a possibility to compare their member count to create coverage ratio.

The second *Property* gives us a preposition that there is a possibility of *Reach Set Reduction* without loosing *Coverage.*

Definition 6 (Coverage Ratio)**.** Coverage Ratio *is a ratio of* Coverage Set Member Count *between two* Reach Sets*. Reach set with* lesser count of unique Trajectories *is considered as* Reduced Reach Set*. Reach set with* greater Count of unique Trajectories *is considered as* Reference Reach Set*.*

$$\begin{aligned} referenceCoverage &= |CoverageSet(ReferenceReachSet, AvoidanceGrid)| \\ reducedCoverage &= |CoverageSet(ReducedReachSet, AvoidanceGrid)| \\ CoverageRatio &= \frac{reducedCoverage}{referenceCoverage} \in [0,1] \end{aligned} \tag{6.11}$$

Note. Reference Reach Set is usually *Full Reach Set* containing all possible trajectories in space contained by *Avoidance Grid*.

In case *Full Reach Set* can not be computed, Avoidance Grid is too large, most complex *Reach Set* is used as *Reference Reach Set*.

Trajectory smoothness: Trajectory other than straight line have some changes in *UAS* heading.

The goal is to minimize *Maneuvering* of UAS, because:

1. *Every Heading Change* needs to be reported to *UTM*.
2. *Sharp Maneuvering* can damage cargo/wear UAS.
3. *Often course changes* makes *Intruder prediction* harder for other Civil General Aviation.

For this purpose *Smoothness Metric* needs to be applied for *Reach Set* or *Trajectory*. In case of *Movement Automaton Control* two distinguish *Movement Sets* can be introduced:*Smooth* nad *Chaotic* movements set with following properties:

$$\begin{gathered} MovementSet = SmoothMovements \cup ChaoticMovements \\ SmoothMovements \cap ChaoticMovements = \varnothing \\ |SmoothMovements| > 0, \quad |ChaoticMovements| > 0 \end{gathered} \tag{6.12}$$

Then *Smoothnes clasificator* for $Trajectory(initialState, buffer)$ can be defined as *isSmooth* and *Smooth Movement Counter* function as *smoothCount* like follow:

$$\begin{aligned} isSmooth(movement) &= \begin{cases} movement \in SmoothMovements & : 1 \\ movement \in ChaoticMovements & : 0 \end{cases} \\ smoothCount(Trajectory(\ldots, buffer)) &= \sum isSmooth(movement), \\ &\qquad \forall movement \in Buffer \end{aligned} \tag{6.13}$$

Definition 7 (Smoothness Rating for Trajectory)**.** Smoothness *for trajectory generated by* Movement Automaton *for some* Initial State *with some* Movement Buffer*, under assumption of* Smooth and Chaotic Movement Set *split (eq. 6.12), with existing* classification *and* counter *functionals (eq. 6.13) is given as follows:*

$$Smoothness(Trajectory(\ldots, buffer)) = \frac{isSmooth(Trajectory)}{movementCount(Trajectory)} \in [0,1] \tag{6.14}$$

For Trajectory *with* $buffer = \varnothing$ Smoothness *is given as 1.*

6.6.2 (R) Constrained Trajectory Expansion

Motivation: *Purpose* of *Navigation* is to move forward to *Goal Waypoint* in *Mission*. *Structure* of *Avoidance Grid* is designed to enable *forward* and *turning* maneuvers. The *Avoidance Grid* is organized in *Layers* characteristic by same distance from *Avoidance Grid Origin*.

Survey of motion planning algorithm was given in [4]. The ideal candidate for propagation algorithm is *Wave-front* algorithm propagating *Trajectory tree* trough Layers. Due the *Avoidance Grid* onion like layers, there is possibility to implement turn maneuver trough layers iterative and effectively .

Rapid Exploration Tree (fig. 6.1) was selected, because it enables *Movement Automaton Utilization* and *Property Binding*. Similar approach was used in rapid exploration tree for space exploration [5].

./Pics/RS001RapidExplorationTreeExampleShrink.png

Figure 6.1: *Rapid Exploration tree as result of* Constrained trajectory expansion.

The example (fig. 6.1) shows a *Rapid Exploration Tree* in *Free Space* containing *Waypoint Navigation Path* and *Turn Away Path*. Both paths are starting in same *Root Node* (red circle) which was expanded with simple *Movement Automaton* (bunch of nodes originating from one node are showing way of expansion). The connection (blue line) between two nodes (red circles) represents *Trajectory portion* for *Executed Movement*.

Rapid Exploration Tree Node will contain following information:

1. *Initial state* - root entry point, used in state evolution calculation.

2. *Trajectory (state evolution)* - trajectory passing trough *state space* in local coordinate frame of *Avoidance Grid*.

3. *Buffer* (applied movements) - ordered list of *executed movements* applied on *initial state* to obtain *state evolution*.

4. *Cost* - calculated for *state evolution* based on *predefined cost function.*

5. *Footprint* - ordered set of *passing cells* in *Avoidance Grid.*

6. *Parent Node Reference* - tree reference for parent node, not in case of *root node.*

7. *Other Bounded Properties* - value list of other properties, depending on *Expansion Constraints* and *Reachibility* evaluation algorithm.

Wave-front propagation of Rapid Exploration Tree is given in (alg. 6.1).

The *Avoidance Grid* have UAS with *position* $\in$ *Initial State* at the *origin.* The *Grid Layer* is a column ordered set of cells with same *Mean distance* from origin. *Grid Layers* are indexed from origin starting with 1, there is maximum of $i \geq 1$ layers.

Step: Initialization contains base structure preparation like follows:

1. *Avoidance Grid* - Space containing *Reach set* (def. 2).

2. *Movement Automaton* - Used as *Predictor*, consuming *buffer* containing *Movements* to generate $Trajectory(initialState, buffer)$.

3. *Reach Set* - tree consisting from *Wave-frontNodes* representing the end point of $Trajectory(initialState, buffer)$ where each *Edge* represents *one Movement application.* The root is set as node containing *Initial State.*

Function *initializeReachSet(root,stack,grid,automaton)* will take the root and enforces *full wavefornt propagation* to *First Layer.*

Step: Wave-front Propagation is forced propagation of trajectories from layer i to layer $i + 1$. The process goes as follows:

1. *Selection of Feasible candidates* - function *[candidates,leftovers] = ExpansionConstraints.select(stack)* for working layer, row and cell selects *feasible trajectory nodes* ordered by *Cost function.* The *Example of Cost Function* can be *Trajectory Smoothness* (def. 7).

2. *Expansion of Candidates* - for each *candidate* function *candidate.expandNode (automaton)* is invoked. This function will expand *Candidate Node structure* by appending *Full Trajectory Tree Evolution* until each *Leaf Trajectory* reaches *Next Layer.* Simply put *Par rent Node Node(initialState, buffer, cost, footprint)* buffer is appended by movements until the next layer is reached.

3. *Leftovers purge* - function *reachSet.purge(leftovers)* removes unexpanded *Nodes* leading to cell, effectively removing trajectories which does not lead to *next layer.*

4. *Append Reach Set* - function *reachSet.append(leafs)* puts newly created *Nodes (Trees)* into *Reach Set* structure. The *Wave-front Propagation* for one cell is finished.

Step: After Layer Propagation Purge is covered by function *reachSet.purgeSame-Footprint()* which takes trajectories with same footprint and keeps some of them based on *Selection criteria*, more in (sec. 6.6.3, 6.6.4). *Pruning methods* over *Large Decision Trees* are *fast* and *viable* [6].

Algorithm 6.1: *Wave-front propagation* of *Rapid Exploration Tree* to form *Reach Set.*

```
Input  : Node(initialState,buffer=∅,cost=0,footprint=∅), AvoidanceGrid,
         ExpansionConstraints, MovementAutomaton(movementSet)
Output: ReachSet(AvoidanceGrid)

# Initialization Sequence;
grid=AvoidanceGrid, automaton=MovementAutomaton, root = Node;
reachSet = initializeReachSet(root,stack,grid,automaton);

# Main Expansion trough, layers (i), rows (j), cells(k);
for layer(1...i) in grid do
    for row(1...j in layer) do
        for cell(1...k) in row do
            # apply selection criteria ;
            [candidates,leftovers] = ExpansionConstraints.select(stack);

            # collect expansions ;
            leafs = [];
            for candidate in Candidates do
                leafs= [leafs, candidate.expandNode(automaton)];
            end
            reachSet.purge(leftovers);
            reachSet.append(leafs);
        end
    end
    reachSet.purgeSameFootprint();
end
```

Note. Reach Set is usually computed *Prior the Flight* for *some Initial State* in *Local Coordinate Frame* in *right had coordinate frame* with X^+ used as *main axis.*

6.6.3 (R) Chaotic Reach set

Motivation: Design of calculation method for *Reach Set Approximation* guarantying high *Maneuverability.*

Background: There is *Coverage Ratio* property of *Reach Set* (def. 6). It has been shown that creating *Reach Set* via *greedy approach* is not feasible due the *Scaling Factor. Contracted Expansion* (sec. 6.6.2) is enabling to apply selection criteria while building *Reach Set* in given *Cell.*

The *Cell* $cell_{i,j,k}$ has a center and walls from UAS viewpoint: front wall , back wall (for *layer* > 1), top wall, left wall, right wall, bottom wall. It is expected that trajectory leading close to one cell walls will continue to different cell, increasing chance to obtain more *Unique Footprints.*

Expansion Constraint Function Implementation (alg. 6.2) is based on simple principle: *Select candidate Nodes which are closest to outer walls of Cell, with unique footprint.*

Tuning Parameters : *Proximity to Cell outer wall* gives good chances to break into other rows or columns in *Avoidance Grid*. *Unique footprint* guarantees future *Unique Footprint* after appending Trajectory by *Movement application*.

1. *Considered Footprint Length* - how much last cells in footprint should be considered in unique path track, minimal value 1, default value 3, maximal value ∞. If you want to generate non redundant trajectories use ∞, it will consider full footprint.

2. *Spread Limit* - upper limit of candidates which are going to be select for further expansion, minimal value 1, default value *Count of unique Moves in Movement set*, maximal value ∞. If more than default values is selected the algorithm will generate *redundant trajectories*. If less is selected then some trajectories are omitted and *Coverage Rate* decreases sharply.

Step: Initialization initialization of *candidate* array (return value), *leftovers* array (return Value). Node array *passing* is populated with *Nodes* which represents *end node of Trajectory* and the tip of *trajectory is constrained in* $\text{cell}_{i,j,k}$.

Step: Evaluate best trajectories with unique Footprints following steps are executed:

1. *Best Performance Map* is created with *footprint* as key set element to ensure footprint uniqueness.

2. *Wall distance* for *test node* is calculated as a closest trajectory portion distance to *top, bottom, left, right* wall of cell $cell_{i,j,k}$

3. *Footprint* for *test node* is created with maximal length given by *Footprint Length* tuning parameter.

4. *Existence and Performance Test* is executed to ensure that best performing node is selected. If there is not key entry in the *Best Performance Map*, then new entry for *Test Node* is created. If there is key entry, the performance of *Old Node* and *Test Node* is compared and better is stored.

Step: Select candidates is executed on *Best Performance Map* records using *Wall distance* as pivot parameter, ordering by closest proximity and limited by *Search Limit* tuning parameter. The *Leftovers* are difference set between *Passing Nodes* and *Candidate Nodes.*

Algorithm 6.2: Expansion Constraint function for *Chaotic Reach Set Approximation*

```
Input                : Node[] stack, Cell cell_{i,j,k}
Tuning Parameters: int+ footprintLength, int+ spreadLimit
Output               : Node[] candidates, Node[] leftovers

# Initialize structures;
Node[] candidates = [], Node[] leftovers=[];
Node[] passing = cell_{i,j,k}.getFinishingTrajectories(stack);

# Select best performing trajectories with unique footprint;
Map<Footprint,Node> bestPerformanceMap;
for Node test ∈ passing do
    wallDistance= test.minimalDistanceToWall(cell_{i,j,k})];
    footPrint = test.getFootprint(lastCells = footprintLength);
    if bestPerformanceMap.contains(footPrint) then
        old = bestPerformanceMap.getByKey(footprint);
        oldPerformance= old.minimalDistanceToWall(cell_{i,j,k});
        if oldPerformance > wallDistance then
            bestPerformanceMap.setByKey(footprint,test);
        end
    else
        bestPerformanceMap.setByKey(footprint,test);
    end
end

# Select best performing nodes up to spreadLimit count;
candidates = bestPerformanceMap.select(count =
  spreadLimit).orderBy('wallDistance','Ascending');
leftovers = passing - candidates;
return [candidates,leftovers]
```

Example: for *Avoidance Grid* with *Distance 10 m*, *Layer count 10*, *Horizontal range* $[-45°, +45°]$, *Horizontal Cell Count 7*, *Vertical range* $[-30°, +30°]$, and *Vertical Cell Count 5*. Is given in (fig. 6.2). The UAS is at *Back-side* of *Figure* (initial state is at all *Trajectory Origins*). The *black dashed line* marks *Avoidance Grid* space boundary. Each trajectory has its own color and ends at *Front-side* of *Avoidance Grid Boundary.*

./Pics/RS002ChaoticReachSetEstimationMethod.png

Figure 6.2: *Chaotic* reach set *approximation*.

Pros and Cons: It can be seen from example (fig. 6.2) that *Chaotic Reach Set Approximation Method* (alg. 6.2) generates a lot of *turning* and *shaky trajectories*.

High Coverage Ratio (~ 0.9) is provided, while keeping *medium node count*. The calculation complexity scales linearly with grid size. The *upper limit of trajectories* is given as follow:

$$countTrajectories(ReachSet) \leq layerCellCount \times spreadLimit \times size(Movements) \tag{6.15}$$

The *upper limit of nodes* is given as follow:

$$countNodes(ReachSet) \leq layerCount \times layerCellCount \times size(Movements) \times spreadLimit \tag{6.16}$$

Absence of *Smooth Trajectories* disqualifies *Chaotic Reach Set Approximation* to be used for *Navigation*. This type of reach set is feasible for *Avoidance*, because it contains variety of maneuvers.

6.6.4 (R) Harmonic Reach set

Motivation: Imagine having an *Avoidance Grid* like (fig. **??**). There is a need of *Reach Set Approximation* which will have *Smooth Trajectories* (def. 7) going nearby *cell* centers.

Background: The *Smoothness Rating for Trajectory* (def. 7) uses two distinct sets *Smooth Movements* and *Chaotic Movements* (eq. 6.12) which are defined for our *Movement Automaton* (sec. **??**) like follow:

$$
\begin{aligned}
SmoothMovements &= \{Straight\} \\
ChaoticMovements &= Movements - SmoothMovements
\end{aligned}
\tag{6.17}
$$

Smooth Movements contains only *Straight* movement, because others are considered as extreme turning movements. *Smooth Movements* should contain only direct flight movements or slight heading correction. *Chaotic Movements* set is supplement of *Movement Automaton's Movement Set*.

The *Avoidance Grid* (fig. **??**) cell centers for fixed indexes j_{fix}, k_{fix} are linearly aligned with *initial state*. That means that cell centers of cells $cell_{1,j_{fix},k_{fix}}, \ldots, cell_{i,j_{fix},k_{fix}}$, where i is count of *layers* lies on one line. If the trajectory can achieve *cell center* on some *layer* only minor trajectory corrections are required to stay on given line. This type of trajectory gives us following advantages:

1. *Minimal steering at beginning* - the minimal steering is advantageous in *Controlled Airspace* because is diminishing the amount of communication to *UTM Service*.

2. *Additional safe space in Linear segment* - once the *center of cell* is reached, *Trajectory* sticks to line between cell centers. Each point on this line has *maximal distance* to outer walls of cell. This gives us extra space given as minimum of distance between *UAS position* and *Outer cell walls*.

Expansion Constraint Function Implementation (alg. 6.3) is based on simple principle: *Select candidate Nodes which are closest to Cell center, with unique footprint.*

Note. Cell center can be closely reached by *smooth movement* from previous cell or *chaotic movement* from neighbouring cell from current or previous layer. These trajectories are usually equivalent in *Smoothness*.

Tuning Parameter: *Proximity to Cell Center* gives a good chance to keep trajectory smooth or *smooth after one correction maneuver*. It has been mentioned that *Cell Center* can be reached by various trajectories. In this method full footprint length is always considered, therefore only one tuning parameter can be offered:

1. *Spread Limit* - upper limit of candidates which are going to be selected for further expansion, minimal value 1, default value *Count of unique Moves in Movement set*, maximal value ∞. If maximal value ∞ is selected, algorithm will generate skeleton of *Reach Set* with full Coverage and with the smoothest *Trajectories*.

Step: Initialization sets candidate *Nodes* as empty set, leftover *Nodes* as empty set. and selects all *Nodes* from *Stack* which represents *Finishing Trajectories* in working cell $cell_{i,j,k}$.

Step: Evaluate smoothest trajectories with unique Footprints is implemented as *multi-criteria filtration*.

First criterion is *distance to Cell Center* which is penalized by trajectory *smoothness rate* implemented in method *Node.getPerformance(Cell* $cell_{i,j,k}$*)* defined as follow.

$$
getPerformance(Node, Cell) = \frac{distance(Node.Trajectory, Cell.Center)}{SmoothnessRate(Node.Trajectory)} \tag{6.18}
$$

Distance of *Trajectory* is *enumerator*, because its considered as *base value* and is defined in interval $[0, maximalWallDistance]$. The *Smoothness Rate* is in denominator, because it is a penalization coefficient defined in interval $[0, 1]$.

Second criterion is *trajectory uniqueness* This is provided by *Best Performance Map*, where best performing *Node* belongs to one unique *trajectory footprint*. The implementation is identical to *chaotic reach set expansion* (alg. 6.2).

Step: Select candidates is executed on *Best Performance Map* records using *Penalized Cell Center Distance* as pivot parameter, ordered in ascending order and limited by *Spread Limit* tuning parameter. The *Leftovers* are difference set between *Passing Nodes* and *Candidate Nodes*.

Algorithm 6.3: Expansion Constraint function for *Harmonic Reach Set Approximation*

```
Input              : Node[] stack, Cell cell_{i,j,k}
Tuning Parameters: int+ spreadLimit
Output             : Node[] candidates, Node[] leftovers

# Initialize structures;
Node[] candidates = [], Node[] leftovers=[];
Node[] passing = cell_{i,j,k}.getFinishingTrajectories(stack);

# Select unique smoothest trajectories;
Map<Buffer,Node> bestPerformanceMap;
for Node test ∈ passing do
    centerDistance= test.getPerformance(cell_{i,j,k})];
    footPrint = test.getFootprint();
    if bestPerformanceMap.contains(footPrint) then
        old = bestPerformanceMap.getByKey(footprint);
        oldPerformance= old.getPerformance(cell_{i,j,k});
        if oldPerformance > centerDistance then
            bestPerformanceMap.setByKey(footprint,test);
        end
    else
        bestPerformanceMap.setByKey(footprint,test);
    end
end

# Select best performing nodes up to spreadLimit count;
candidates = bestPerformanceMap.select(count =
  spreadLimit).orderBy('cellCenterDistance','Ascending');
leftovers = passing - candidates;
return [candidates,leftovers]
```

Example: for *Avoidance Grid* with *Distance 10 m*, *Layer count 10*, *Horizontal range* $[-45°, +45°]$, *Horizontal Cell Count 7*, *Vertical range* $[-30°, +30°]$, and *Vertical Cell Count 5*. Is given in (fig. 6.3). The UAS is at *Back-side* of *Figure* (initial state is at all *Trajectory Origins*). The *black dashed line* marks *Avoidance Grid* space boundary. Each trajectory has its own color and ends at *Front-side* of *Avoidance Grid Boundary*. The *Spread Limit* in this case was set to 9 which is *Size of the Movement Set*.

Note. Please note *Trajectories* are organized in bundles going around *Cell Centers smoothly*. Most of the steering maneuvers are executed at the *beginning* of *Avoidance Grid*.

./Pics/RS003HarmonicReachSetEstimationMethod.png

Figure 6.3: *Harmonic* reach set *approximation*.

Pros and Cons: It can bee seen from example (fig. 6.3) that *Harmonic Reach Set Approximation Method* (alg. 6.3) generates *smooth evenly spread trajectories*.

High smoothness ratio (≥ 0.9) is provided, while keeping low node count for UAS systems. The calculation complexity scales linearly with grid size. The upper limit of trajectories is given as follow:

$$\begin{aligned} countTrajectories(ReachSet) \leq layerCellCount \times spreadLimit \\ \times size(Movements) \end{aligned} \tag{6.19}$$

The *upper limit of nodes* is given as follow:

$$countNodes(ReachSet) \leq layerCount \times layerCellCount \times spreadLimit \tag{6.20}$$

Absence of *High Coverage Ratio* disqualifies *Harmonic Reach Set Approximation* to be used for *Emergency Avoidance*. This type of *Reach Set* is feasible for *Open Space Navigation* or *Controlled Airspace Navigation*. Its low turning rate in contained *Trajectories* are desired for such tasks.

6.6.5 (R) Combined Reach set

Motivation: Harmonic reach set (sec. 6.6.4) is *efficient* for *Navigation in* Controlled Airspace. Chaotic reach set (sec. 6.6.3) is good for *Emergency avoidance*. The need to differentiation between *Navigation* and *Emergency Avoidance* mode is necessary in

Controlled Airspace. but not in *Non-controlled Airspace.* The combination of *Harmonic* and *Chaotic* reach sets is obvious solution.

Automatic mode switch can be provided by combination of *Navigation Reach Set* and *Avoidance Reach Set* with elevated cost function. Overall having a method to merge multiple trees would be beneficial.

Background: If two *Reach Set Approximation* were calculated for same *Avoidance Grid* and *Initial State*, using same *Movement Automaton* and *UAS model* are possible to merge.

The *Reach Set Approximation* is *tree* with *Root Node* in *initail state* with movement buffer $= \varnothing$. The *movement buffer* in each node can be used as *route trace* during merging procedure. The example two reach set merge can be given as follow, where only *latest* applied movement is taken into account.

$$\left[\begin{array}{l}\texttt{First Reach Set}\\ \varnothing \to \left\langle \begin{array}{l} left \to \left\langle\begin{array}{l} left\\ right\end{array}\right.\\ \varnothing \end{array}\right. \\ \texttt{Second Reach Set} \\ \varnothing \to \left\langle\begin{array}{l}\varnothing\\ right \to \left\langle\begin{array}{l}left\\ right\end{array}\right.\end{array}\right.\end{array}\right] \to \left[\begin{array}{l}\texttt{Combined Reach Set}\\ \varnothing\to\left\langle\begin{array}{ll} left & \to \left\langle\begin{array}{l}left\\ right\end{array}\right.\\ right & \to \left\langle\begin{array}{l}left\\ right\end{array}\right.\end{array}\right.\end{array}\right] \tag{6.21}$$

First Reach Set contains two trajectories given by buffers {*left,left*} and {*left,right*}. *Second Reach Set* contains two trajectories given by buffers {*right,left*} and {*right,right*}. The *Combined Reach Set* contains all four trajectories.

Note. The combined tree [7] does not need to have combined amount of original *Reach Sets* trajectories. There can be *Duplicity* which means that any bounded property like *Cost* must be *calculated* again.

Combined Reach Set Calculation Function (alg. 6.4) is implemented as function $NodecombinedReachSet(\dots)$ which takes root Node with *initial State*, *Avoidance Grid* and respective parameters for each calculation method. *Harmonic spread* for *Harmonic Reach set calculation* and *Chaotic Spread*, *Footprint Length* for *Chaotic Reach set calculation.*

Separate Reach Sets are calculated using *Wave-front propagation* (alg. 6.1) using respective *Constrained Expansion* functions for *Harmonic* (alg. 6.3) and *Chaotic* (alg. 6.2) reach sets.

Combined Reach Set is created using *Node mergeTree(...)* function. Because different cost function or *Bounded Parameters Calculation* may be applied on *Original Reach Sets*.

Cost for *each node* needs to be recalculated due to original reach sets disparity. Function *combined.applyCostFunction()* will recalculate the new cost for each node.

The Goal is to have penalization for *Chaotic behaviour*, implementation of *Automatic Mode Switch* can be done like follows:

1. *Calculate Normal Cost* for Node $Cost(Node)$ for associated trajectory: $Cost(Node.Trajectory)$.

2. *Calculate Penalization for* Chaotic Behaviour, calculate *Smoothness Rating for Trajectory* (def. 7) in interval $[0,1]$, introduce penalization with base 100%.

The final $Cost(Node)$ function is applied on each *Combined Reach Set Node* and look like follows:

$$Cost(Node) = Cost(Node.Trajectory) \times \ldots$$
$$\cdots \times (1 + (1 - SmoothnessRate(Node.Trajectory))) \quad (6.22)$$

Merge Tree Function $mergeTree(\ldots)$ implements *Outer Join* operation on two trees. Example was given in (eq. 6.21). Function is applied on *root Node* iterating over *Movements in Movement Set*, because *Movement is pivot.*

Algorithm 6.4: Reach Set Merge Function and Combined Reach Set calculation

```
# Tree merge function;
Node mergeTree(Node firstNode, Node secondNode)
    # Try to copy reference node or return null;
    Node referenceNode = (firstNode?:(secondNode?: return null));
    Node merged = new Node(referenceNode);
    merged.leafs= [];
    # Try to fetch movement nodes if exist in any sub tree;
    for movement ∈ Movements do
        firstLeaf = firstNode.getLeafFor(movement);
        secondLeaf = secondNode.getLeafFor(movement);
        newLeaf = mergeTree(firstLeaf,secondLeaf);
        if newLeaf ∼= null then
            merged.leafs.append(newLeaf);
        end
    end
    return merged
# Combined Reach Set calculation function;
Node combinedReachSet(Node root, AvoidanceGrid grid,int+ chaoticSpread,
 int+ harmonicSpread, int+ footprintLength)
    Node chaotic = chaoticReachSet(root,grid, footprintLength,chaoticSpread);
    Node harmonic = harmonicReachSet(root,grid, harmonicSpread);
    Node combined = mergeTree(chaotic,harmonic);
    combined.applyCostFunction();
    return combined
```

Example: for *Avoidance Grid* with *Distance 10 m*, *Layer count 10*, *Horizontal range* $[-45°, +45°]$, *Horizontal Cell Count 7*, *Vertical range* $[-30°, +30°]$, and *Vertical Cell Count 5*. Is given in (fig. 6.4). The UAS is at *Back-side* of *Figure* (initial state is at all *Trajectory Origins*). The *black dashed line* marks *Avoidance Grid* space boundary. Each trajectory has its own color and ends at *Front-side* of *Avoidance Grid Boundary.* The *Chaotic Spread* was set to 8, *Footprint Length* to 3 and *Harmonic Spread* to 1.

Note. Notice there are typical trajectories from both *Harmonic* (fig. 6.3) and *Chaotic* (fig. 6.2) *Reach Set Approximations.*

./Pics/RS004CombinedReachSetEstimationMethod.png

Figure 6.4: *Combined* reach set *approximation*.

Pros and Cons: It can bee seen from example (fig. 6.4) that *Combined Reach Set Approximation* (alg. 6.4) contains both types of maneuvers. *Cheaper Smooth* for navigation and *More Expensive Chaotic* for *Emergency Avoidance*. The upper limit of trajectories is given as follow:

$$\begin{aligned} countTrajectories(ReachSet) \leq\ & countTrajectories(Chaotic) \\ & + countTrajectories(Harmonic) \end{aligned} \tag{6.23}$$

The *upper limit of nodes* is given as follow:

$$countNodes(ReachSet) \leq countNodes(Chaotic) + countNodes(Harmonic) \tag{6.24}$$

Harmonic Reach Set is ideal for *Non-controlled Airspace* missions, because it contains *Automatic Mode Switch* between *Navigation* and *Emergency Avoidance*.

6.6.6 (R) ACAS-X like Reach set

Motivation: The implementation of *ACAS-Xu* behavior in DAA system will be mandatory for *National Airspace System Integration* in United spaces [8].

Implementation of ACAS-Xu like behaviour increase usability of approach, if it can be achieved without major concept changes.

Background: The *ACAS-Xu* system on operational level has been described in [9]. The *Policy for Collision Avoidance* proposal has been given in [10].

Some behavioural patterns can be encoded into *Reach Set*. ACAS-Xu navigation part is basically *Look-up table of Maneuvers for Allowed Separations*.

The *Evasive Maneuver* selection process in ACAS-Xu is similar to our approach: *Select most energy efficient maneuver in compliance with space-time constraints*. ACAS-Xu intruder model is similar to our *Body Volume Intersection Model* (sec. ??). The *ACAS-Xu* defines following base separations:

1. *Horizontal* - movements on *Horizontal Plane* in *Global Coordinate System.*
2. *Vertical* - movements on *Vertical Plane* in *Global Coordinate System.*

There are allowed custom separations which can be used, for further experimentation:

1. *Slash* - movement on $+45^\circ$ *Tilted Plane to Horizontal Plane* in *Global Coordinate System.*
2. *Backslash* - movement on -45° *Tilted Plane to Horizontal Plane* in *Global Coordinate System.*

For given *Movement Automaton* implementation (sec. ??) the separations are given as follow:

$$\begin{aligned} Horizontal &= \{Straight, Left, Right\} \\ Vertical &= \{Straight, Up, Down\} \\ Slash &= \{Straight, UpLeft, DownRight\} \\ Backslash &= \{Straight, UpRight, DownLeft\} \end{aligned} \tag{6.25}$$

For each $Node(\dots, buffer)$ and each *separation* there is a evaluation function *isSeparation* which decides, if *Trajectory* defined by node buffer is made up only from *Separation* movements. The function *isSeparation*$(\dots)$ is defined like:

$$isSeparation(buffer, separation) = \begin{cases} \forall movement \in buffer, & \\ \quad movement \in separation & : true \\ otherwise & : false \end{cases} \tag{6.26}$$

Following *Separation Modes* can be defined with given *separations*:

1. *Horizontal* (ACAS-X defined mode) containing *horizontal* separation.
2. *Vertical* (ACAS-X defined mode) containing *vertical* separation.
3. *Horizontal-Vertical* (ACAS-X defined mode) containing *horizontal, vertical* separations.
4. *Full* (custom defined mode) containing all *Separation Modes.*

Note. Every separation modes generates 2D trajectories set on *Respective plane.* There is no need for *Tuning parameters* for further *Expansion Constraint.*

Expansion Constraint Function Implementation (alg. 6.5) is based on simple principle: *Select only candidate Nodes which Trajectories have at least one desired Separation Mode.*

Step: Initialization sets candidate *Nodes* as empty set, leftover *Nodes* as empty set, and, select all nodes form *stack* which represents *Finishing Trajectories* in working $cell_{i,j,k}$,

Step: Candidate Selection Process is evaluated for each *test Node* from *passing Node Set*.

For each *applicable separation*, given as input parameter *separations*, The test function *isSeparation* (eq. 6.26) is applied:

1. If *test Node* trajectory belongs to at least one allowed separation it is added to candidates set.
2. Else is added to *Leftovers*.

Note. Separation sets (eq. 6.25) are not *exclusive sets* in *Movement Automaton* domain. One *Trajectory* contained by Node can belong to multiple *Separations*.

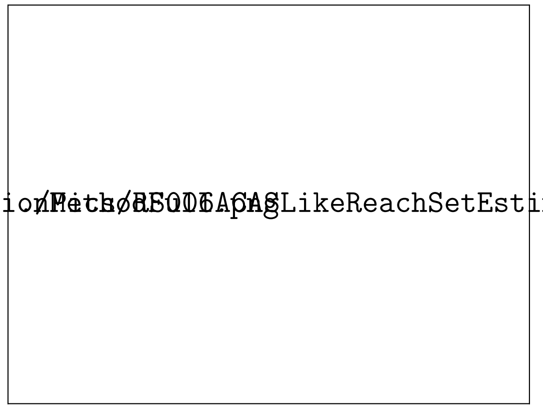
Algorithm 6.5: Expansion Constraint function for *ACAS-like Reach Set Approximation*

```
Input             : Node[] stack, Cell cell_{i,j,k}, Separation[] separations
Tuning Parameters: None : ∅
Output            : Node[] candidates, Node[] leftovers

# Initialize structures;
Node[] candidates = [], Node[] leftovers=[];
Node[] passing = cell_{i,j,k}.getFinishingTrajectories(stack);

# Select nodes containing trajectories with usable separations;
for Node test ∈ passing do
    for separation ∈ separations do
        # Get separations for Node;
        Separations[] nodeSeparations = test.getSeparations();

        # If trajectory given by buffer is on Separation plane;
        if isIn(isSeparation(test.buffer,separation)(6.26) then
            candidates.append(test);
        end
    end

    # If there was no applicable separation, throw Node away;
    if test ∉ candidates then
        leftovers.append(test);
    end
end

# Return results;
return [candidates,leftovers]
```

Example: for *Avoidance Grid* with *Distance 10 m*, *Layer count 10*, *Horizontal range* $[-45°, +45°]$, *Horizontal Cell Count 7*, *Vertical range* $[-30°, +30°]$, and *Vertical Cell Count 5*. Is given in (fig. 6.5). The UAS is at *Back-side* of *Figure* (initial state is at all *Trajectory Origins*). The *black dashed line* marks *Avoidance Grid* space boundary. Each trajectory has its own color and ends at *Front-side* of *Avoidance Grid Boundary.*

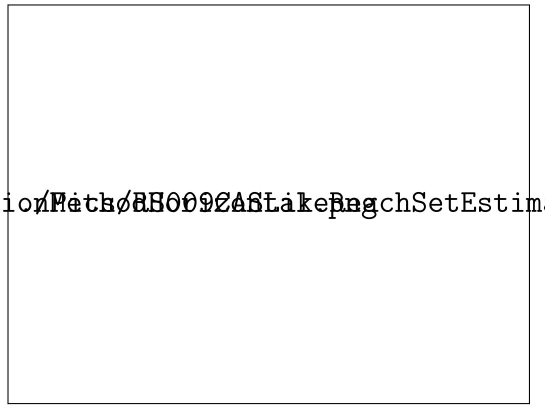
Full separation mode given in (fig. 6.5a). *Horizontal-Vertical* separation mode, used in original *ACAS-Xu* testing [9], given in (fig. 6.5b). *Horizontal* separation mode given in (fig. 6.5c) is usually used by planes. *Vertical* separation mode given in (fig. 6.5d) is usually used by copters.

(a) Full.



(b) Horizontal-Vertical.

(c) Horizontal.



(d) Vertical.

Figure 6.5: ACAS-X imitation *reach set* approximation for various *separation modes*.

Pros and Cons: It can be seen from examples (fig. 6.5) that *ACAS-like Reach Set Approximation Method* (alg. 6.5) generates full reach set for 2D plane located in 3D space.

The *Reach Set* contains trajectories with *high coverage ratio* and *high smoothness rating* for selected 2D separation plane. Overall performance compared to full 3D reach sets (sec. 6.6.3, 6.6.4 6.6.5) is poor.

The *node* and *trajectory* count boundary was not implemented. It is common knowledge that *2D* avoidance sets does not require scaling [9]. Otherwise trajectory footprint mechanism like in *Harmonic Reach Set Approximation* (alg. 6.3) can be introduced.

This reach set implements *Planar-Separation* as native feature, it can be used for both *navigation* and *avoidance* tasks in *Controlled Airspace*. For *Non-controlled Airspace* there are far more superior *Combined Reach Set* (sec. 6.6.5).

Bibliography

[1] Franco Blanchini. Set invariance in control. *Automatica*, 35(11):1747–1767, 1999.

[2] John Birmingham and Peter Kent. Tree-searching and tree-pruning techniques. In *Computer chess compendium*, pages 123–128. Springer, 1988.

[3] Alojz Gomola, João Borges de Sousa, Fernando Lobo Pereira, and Pavel Klang. Obstacle avoidance framework based on reach sets. In *Iberian Robotics conference*, pages 768–779. Springer, 2017.

[4] Chad Goerzen, Zhaodan Kong, and Bernard Mettler. A survey of motion planning algorithms from the perspective of autonomous uav guidance. In *Selected papers from the 2nd International Symposium on UAVs, Reno, Nevada, USA June 8–10, 2009*, pages 65–100. Springer, 2009.

[5] Catherine Plaisant, Jesse Grosjean, and Benjamin B Bederson. Spacetree: Supporting exploration in large node link tree, design evolution and empirical evaluation. In *Information Visualization, 2002. INFOVIS 2002. IEEE Symposium on*, pages 57–64. IEEE, 2002.

[6] John Mingers. An empirical comparison of pruning methods for decision tree induction. *Machine learning*, 4(2):227–243, 1989.

[7] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[8] Jay Shively. Uas integration in the nas: Detect and avoid. 2018.

[9] Mike Marston and Gabe Baca. Acas-xu initial self-separation flight tests, 2015.

[10] Kyle D Julian, Jessica Lopez, Jeffrey S Brush, Michael P Owen, and Mykel J Kochenderfer. Policy compression for aircraft collision avoidance systems. In *Digital Avionics Systems Conference (DASC), 2016 IEEE/AIAA 35th*, pages 1–10. IEEE, 2016.