Common symbols and notations

- $\%$: comment
- $\boldsymbol{x}$: node
- p: cardinality of discretized beam angles, e.g. $p = 180$ for $2°$ angular resoultion
- m: beam plan, dimensionality of a node's beam set, e.g. $m = 5$
- $v^{\star}$: value (optimal FMO objective)
- ζ: profile map (from column generation)
- `col_net`: column generation prior network
- `CLASS`.**`function`**(`parameter`): call to a class' subroutine
- `User.`N: Number of leafs to add to tree before averaging Q-values
- Tree edge attributes in lower-case; Uppercase for all edges
  - $\{\Upsilon, \upsilon\}$: patient's geometry + profile map; network input
  - $\{\Gamma, \gamma\}$: tree policy
  - $\{\Theta, \theta\}$: beam angles

Algorithm 1 Deep BOO Search

```
function run_search(col_net)
    agent = search(col_net)
    search_data = agent.extract_search_data()
    return search_data
end function

function search(col_net)
    depth ← Initialize tree depth
    agent ← AGENT(col_net) % agent controls search
    x ← agent.root.select_leaf() % x is a node
    ϒ = agent.get_input() % agent samples ϒ from storage
    ζ = network.predict(ϒ)
    x.propagate_results(ζ, x, v = Null) % value null for now
    while True do
        start ← current_time()
        current_depth = x.N    % x.N: node's visit count
        while x.root.N < current_depth + depth do
            % continually add children to x.root
            agent.tree_search(x)
        end while
        move = x.pick_move() % sample from all explored moves
        x.play_move(move) % back up value of move
        if agent.done() then
            break
        end if
    end while
    return agent
end function

function extract_search_data()
    for {υ, γ, θ} in {AGENT.ϒ, AGENT.Γ, AGENT.Θ}
    return agent.extract_search_data()
end function
```

Algorithm 2 Tree Search Agent

```
% procedure may be implemented as a class
% network is prior from column generation
procedure AGENT(network, beam_position=NULL)
    beam_position=BOO_POSITION() if NULL
    AGENT.root = NULL
    AGENT.network = network
    AGENT.initialize_search(beam_position)

    function initialize_search(beam_position)
        AGENT.root = MCTS_NODE(beam_position)
        %Intiailize buffers for storing policy parameters
        AGENT.Υ=AGENT.Γ=AGENT.Θ=EMPTY_ARRAY
    end function

    function tree_search(depth)
        leaves←EMPTY_ARRAY
        depth_monitor = 0
        while depth_monitor < depth do
            depth_monitor←depth_monitor+1
            leaf = AGENT.root.choose_leaf()
            if leaf.done() then
                value = leaf.position.fmo()
                leaf.backup(value, AGENT.root)
            end if
            leaves←add(leaf)
        end while
        if leaves then
            profile = AGENT.network.predict(leaves)
            for {leaf, move, val} ∈ (leaves, profile, vals) do
                leaf.propagate_results(move, val, AGENT.root)
            end for
        end if
        return leaves
    end function

    function pick_move()
        move ← min_move( cdf(AGENT.root.children))
        return move
    end function

    function play_move(move)
        AGENT.Γ.add( AGENT.root.children_policy())
        AGENT.Υ.add(AGENT.root.υ)
        AGENT.root ← AGENT.root.try_add_child(move)
        AGENT.position = AGENT.root.beam_position
    end function
end procedure
```

Algorithm 3 BOO Position

```
procedure Position(beamlets, patient_cases)
    Position.beamlets = beamlets
    Position.patient = patient  % patient for these beams
    Position.n = 0
    Position.patient_cases = patient_cases

    function get_input()
        return Υ
    end function

    function play_move(move, mutate=False)
        if mutate then
            Position.patient ← random_select(patient_cases)
            Position.beamlets.reset()
        end if
        add_beamlets(Position.beamlets, move)
        Position.n += 1
    end function

    function fmo()
        v* = FMO_COST(Position.beamlets, Position.patient)
        return v*
    end function
end procedure
```

```
function add_beamlets(beamlets, move)
    if move ∈ beamlets or beamlets.size=m then
        Return NULL
    end if
    beamlets.insert(move)
end function
```

Algorithm 4 Tree Search Node

```
procedure PARENT_NODE()
    % procedure may be e.g. a class
    PARENT_NODE.parent = NULL
    % child attribs. ops allow vectorized operations
    PARENT_NODE.Child_N = 0^{180×1}
    PARENT_NODE.Child_FMO = 0^{180×1}
end procedure
procedure NODE(position, ζ, parent)
    % "." implies a procedural attribute
    position = BOO_POSITION() if position is NULL
    NODE.parent = parent %null if root
    NODE.position = position %beamlets
    NODE.ζ = ζ
    NODE.Child_N = 0^{180×1}
    NODE.Child_FMO= 0^{180×1}
    NODE.ϒ = position.get_input()
    NODE.prior = 0^{180×1}
    NODE.Child_ζ = 0^{180×1}
    NODE.children = SET()

    function CHILD_Q_VALUE()
        return (NODE.Child_Q + NODE.Child_U)
    end function

    function Child_Q()
        return NODE.Child_FMO / (1+ NODE.Child_N)
    end function

    function Child_U()
        return c√(NODE.N) · NODE.Child_ζ / (1 + NODE.Child_N)
    end function

    function Q()
        return NODE.FMO / 1 + NODE.N
    end function

    function FMO()
        return NODE.parent.child_FMO[ζ]
    end function

    function N()
        return NODE.parent.child_N[ζ]
    end function
    continued on next page
end procedure
```

Algorithm 5 Tree Search Node *cont'd*

```
function choose_leaf()
    new_node = THIS_NODE()
    while True do
        if new_node.expanded then
            break
        end if
        best_move = random(new_node.child_Q_value())
        new_node.add_child(move,mutate)
    end while
    return new_node
end function

function add_child(move)
    % mutate controls the selection of new patients
    if move not in NODE.children then
        mutate=True
    end if
    new_pose = NODE.position.play_move(move, mutate)
    NODE.children[move] = NODE(new_pose, move, THIS_NODE)
    return NODE.children[move]
end function
```

Date	**Model Description**	**Time to Finish Training**	GPUs/CPU Stats
Jan. 27. 2019	Retrained Column Generation	22h4.8m (≈1,324m52.285s)	1 K80 GPU
Jan. 29. 2019	Monte Carlo Tree Search	1100:54	74.2% of cpu mem

- **Time to Retrain with Tree Search Data on 2X GeForce 1080 Titans**
  - real 595m29.466s
  - user 452m59.608s
  - sys 115m16.404s

1 Adapted from Bertsekas' Approximate Policy Iteration Review Paper

Policy iteration is a major alternative to value iteration – it generates a set of policies and associated cost functions through iterations that have two phases: *policy evaluation* and *policy improvement*. Policy evaluation is where the costs function of a policy is evaluated and policy improvement is where a new policy is generated.

2 Monte Carlo retrainig notes

Here, we take the data that was generated by our tree search, turn that data to the column generation trained network by Dan and Azar, and then run the backprop algorithm using the adam optimizer. Below are some of the parameters we tuned to obtain the stated results in this section

All of these was with 70% of data for training and 30% for validation

- tried adding dropout of $.4$ probability to all layers and decreased learning rate to 10^{-1}
  - tr. and val. loss's magnitude very, but noticed steady decrease in both erros per epoch
- tried increasing learning rate $10^{-5} \to 10^{-3}$
  - training loss increases per epoch (in 1dp range) then stays constant at 3.1227, validation error constant at 3.1227
- dropout 0.1 and lr $1e-4$
  - training reducing by 0.001 margin per iteration
  - validation constant at .2222 after starting to reduce from .32xx at start of training
- dropout 0.1 and lr $1e-4$ — tf.train.AdamOptimizer
  - training error fluctuates highly e.g. 0.0322 to 102 and back
  - validation error going from 0.3222 to 3.117 and all over
- dropout 0.0 and lr $1e-6$ — tf.train.AdamOptimizer
  - train/val ration is 65-45
  - training error reducing monotonically $0.27 \leftarrow .005$
  - validation error decreasing monotonically. $0.1807 \leftarrow .0056$
  - I seem to have found the ideal parameters so far at 20th epoch
  - note that we do not shuffle validation data

- 2019-02-16 Data and Neural Network
  - Re-training network on swmed on gpu 0
  - Training/validation split: 65:45
  - real 922m40.636s
  - user 482m9.931s
  - sys 134m10.487s
- 2019-02-17 Data and Neural Network

  - Re-training network on swmed on gpu 1
  - Training/validation split: 65:45
  - rdid not measure time
- searching on rok80 with changes
  - we compute the cdf of all children of the root player
  - we pick the child that has the min move in the cdf
  - that's what we use to compute the fmo that we back up to the root