Common symbols and notations

- x: node
- v: heuristic value
- $v^\star$: value (optimal FMO objective)
- Υ: patient's geometry& fitness values
- ζ: profile map (column generation fitness values)
- `col_gen_net`: pretrained network with column generation as prior
- `User.N`: User defined readout. Tells how many search to add to tree before carrying out FMO

Algorithm 1 BOO Search

---

function RUN_SEARCH(col_gen_net)
    agent = **SEARCH**(col_gen_net)
    search_data = agent.**extract_search_data()**
    return search_data
end function

function SEARCH(network)
    depth = User.N
    agent ← **AGENT**(network, False)
    x ← agent.**select_leaf()**
    Υ = agent.**get_input()**
    ζ, v = network.**predict(**Υ**)**
    x.**propagate_results(**ζ, v, x**)**
    while True **do**
        start ← current_time()
        current_depth = $x.N$
        while x.root.N < current_depth + depth **do**
            agent.**tree_search(**x**)**
        end while
        move = x.**pick_move()**
        x.**play_move(move)**
        if agent.**done()** **then**
            break
        end if
    end while
    return agent
end function

Algorithm 2 Tree Search Agent

```
% procedure may be implemented as a class
procedure AGENT(network, two_player_mode=False)
    AGENT.root = NULL
    AGENT.network = network
    AGENT.mode = two_player_mode % no min-max strategy
    AGENT.start_search()

    function START_SEARCH(beam_position)
        AGENT.root = MCTS_NODE(beam_position)
        AGENT.policy.Buffer=EMPTY_ARRAY
        AGENT.Υ.Buffer=EMPTY_ARRAY
    end function

    function GUESS_MOVE(position)
        depth = AGENT.root.N
        while AGENT.root.N < depth + User.N do
            AGENT.tree_search(x)
        end while
        return AGENT.pick_move()
    end function

    function PICK_MOVE()
        c ← cdf(AGENT.root.children)
        select ← random(low=1, high=180)
        move ← sort(c[select])
        return move
    end function

    % "." implies a procedural attribute
    function PLAY_MOVE(move)
        AGENT.policy.BUFFER.add( AGENT.root.c_policy())
        AGENT.Υ.BUFFER.add(AGENT.root.Υ )
        AGENT.root ← AGENT.root.maybe_add_child(move)
        AGENT.position = AGENT.root.beam_position
    end function

    function TREE_SEARCH(num_threads=NULL)
        % buffer for leaves
        leaves←EMPTY_ARRAY
        num_threads=depth if NULL
        fail = 0
        while fail < num_threads×2 do
            fail←fail+1
            leaf = AGENT.root.select_leaf()
            if leaf.done() then
                value = leaf.position.score()
                leaf.backup(value, AGENT.root)
            end if
            leaves.add(leaf)
        end while
        continued on next page
    end function
end procedure
```

Algorithm 3 Agent's Tree Search Agent Continued

---

procedure AGENT(`network, two_player_mode`)
    function `TREE_SEARCH`(`num_threads=NULL`)
       if `leaves` **then**
          `probs, vals = AGENT.network.`**`predict`**`(leaves)`
          for $\{$`leaf, move, val`$\}$ $\in$ `(leaves, probs, vals)` **do**
             `leaf.`**`propagate_results`**`(move, val, AGENT.root)`
          end for
       end if
       return `leaves`
    end function
end procedure

Algorithm 4 Tree Search Node

% procedure may be e.g. a class
procedure PARENT_NODE()
    % essentially, tree's root node placeholder
    `NODE.parent = NULL`
    `NODE.child_N = set()`
    `NODE.child_Ag = set() % agent's child`
end procedure

procedure MCTS_NODE(`position, profile, parent`)
    % "." implies a procedural attribute
    `MCTS_NODE.parent = parent %null if root`
    `MCTS_NODE.position = position %beamlets`
    `MCTS_NODE.move_probs = profile`
    `MCTS_NODE.child_N = ` $\mathbf{0}_{180\times1}$
    `MCTS_NODE.child_Ag = ` $\mathbf{0}_{180\times1}$
    `MCTS_NODE.nonterminal = False %terminal?`
    `MCTS_NODE.`Υ` = position.get_input()`
    `MCTS_NODE.prior = ` $\mathbf{0}_{180\times1}$
    `MCTS_NODE.child_prior = ` $\mathbf{0}_{180\times1}$
    `MCTS_NODE.children = set() % e.g. a python dict`

    % calculates the child action-value score
    `@property`
    function CHILD_ACTION_SCORE()
        % .to_play negates score for oppo. ag. NB: Unused
        return `MCTS_NODE.child_Q` $\times$ `MCTS_NODE.position.to_play +`
`MCTS_NODE.child_U`
    end function

    function CHILD_Q()
        return MCTS_NODE.child_Ag $/(1 + $ **child_N**$)$
    end function

    function CHILD_U()
        %essentially exploration bonus term. Diverges
moves from early play. See alpha go paper
        % broadcast scalars to vectors for vector ops.
        return $c\sqrt{max(1, \text{NODE}.N - 1)}\frac{\texttt{MCTS\_NODE.child\_prior}}{\texttt{1+MCTS\_NODE.child\_N}}$
    end function

    `@property`
    function Q()
        return $\text{NODE}.$**Ag** $/ 1 + $ **Node.N**
    end function

    function AG()
        return `MCTS_NODE.parent.child_Ag[`**probs**`]`
    end function

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

Algorithm 5 MCTS Node Procedure Continuation

```
procedure PARENT_NODE()
    function SELECT_LEAF()
        current = MCTS_NODE()
        mutate=False
        mutate_count=0
        while current.non_terminal do
            previous = current
            if mutate_count % 20 == 0 then
                mutate = True
            end if
            if current.expanded: then
                break
            else
                move = argmax(current.child_action_score)
                current.may_be_add_child(move,mutate)
            end if
        end while
        best_move = argmax(current..child_action_score)
        current.may_be_add_child(move,mutate)
        mutate_count += 1
        return current
    end function

    function MAY_BE_ADD_CHILD(move, mutate=False)
        % mutate controls the selection of new patients
        new_position = MCTS_NODE.position.play_move(move,
mutate)
         MCTS_NODE.children[move] = MCTS_NODE(new_position,
probs=move, MCTS_NODE)
        return MCTS_NODE.children[move]
    end function
end procedure
```

Algorithm 6 BOO_Position

---

procedure BOO_POSITION() % Selects position of beamlets
    function PLACE_BEAMS(running_config, global_config, moves)
        global_config[moves] = moves
        search_data = agent.**extract_search_data()**
        return search_data
    end function

end procedure