

### Common symbols and notations

- $x$ : node
- $v$ : heuristic value
- $v^*$ : value (optimal FMO objective)
- $\mathcal{T}$ : patient's geometry & fitness values
- $\zeta$ : 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  $\leftarrow$  AGENT(network, False)
   $x \leftarrow$  agent.select_leaf()
   $\mathcal{T} =$  agent.get_input()
   $\zeta, v =$  network.predict( $\mathcal{T}$ )
   $x$ .propagate_results( $\zeta, v, x$ )
  while True do
    start  $\leftarrow$  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. $\gamma$ .Buffer=EMPTY_ARRAY
  end function

  function GUESS_MOVE(position)
    depth = AGENT.root.N
    while AGENT.root.N < depth + User.N do
      AGENT.TREE_SEARCH( $\kappa$ )
    end while
    return AGENT.PICK_MOVE()
  end function

  function PICK_MOVE()
    c  $\leftarrow$  cdf(AGENT.root.children)
    select  $\leftarrow$  RANDOM(low=1, high=180)
    move  $\leftarrow$  SORT(c[select])
    return move
  end function

  % “.” implies a procedural attribute
  function PLAY_MOVE(move)
    AGENT.policy.BUFFER.add(AGENT.root.CHILDREN_POLICY())
    AGENT. $\gamma$ .BUFFER.add(AGENT.root. $\gamma$ )
    AGENT.root  $\leftarrow$  AGENT.root.MAYBE_ADD_CHILD(move)
    AGENT.position = AGENT.root.beam_position
  end function

  function TREE_SEARCH(num_threads=NULL)
    % buffer for leaves
    leaves  $\leftarrow$  EMPTY_ARRAY
    num_threads = depth if NULL
    fail = 0
    while fail < num_threads  $\times$  2 do
      fail  $\leftarrow$  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
  end function
  continued on next page
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} ∈ (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
  MCTS_NODE.parent = NULL
  MCTS_NODE.child_N = set()
  MCTS_NODE.child_Ag = set() % agent's child
end procedure

procedure MCTS_NODE(position, profile, parent)
  % "." implies a procedural attribute
  position = BOO_POSITION() if position is NULL
  MCTS_NODE.parent = parent %null if root
  MCTS_NODE.position = position %beamlets
  MCTS_NODE.move_probs = profile
  MCTS_NODE.child_N = 0180×1
  MCTS_NODE.child_Ag = 0180×1
  MCTS_NODE.nonterminal = False %terminal?
  MCTS_NODE.γ = position.get_input()
  MCTS_NODE.prior = 0180×1
  MCTS_NODE.child_prior = 0180×1
  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 × 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  $c\sqrt{\max(1, \text{NODE}.N - 1) \frac{\text{MCTS\_NODE.child\_prior}}{1 + \text{MCTS\_NODE.child\_N}}}$ 
  end function

  @property
  function Q()
    return 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

---

```

N = 360 % BEAM_SPACE
STEP_SIZE = 2 % GRID_RESOLUTION
EMPTY_CONFIGURATION =  $\mathbf{0}_{[N/STEP\_SIZE] \times 1}$ 
RUNNING_CONFIGURATION =  $\mathbf{0}_{5 \times 1}$ 
function PLACE_BEAMS(running_config, global_config, moves)
    global_config[moves] = moves
    n = 0 % times game has been played
    select_idx = random(1, size_of(running_config))
    running_config[select_idx] = global_config[moves]
end function
procedure POSITION(global_config, running_config, to_play=1)
    Position.global_config = global_config
    Position.running_config = running_config
    Position.patient = patient % patient for these beams

    @property
    function GET_INPUT ()
        return  $\gamma$  % retrieved from Column generation prior
    end function

    function PLAY_MOVE(move, player, mutate=False)
        if player is NULL then
            player=Position.to_play
        end if
        if mutate is True then
            Position.patient=random_select(patient_cases)
        end if
        PLACE_BEAMS(Position.global_config,
            Position.running_config), move
        Position.n += 1
    end function
    function SCORE()
        return FMO_COST(Position.running_config)
    end function
end procedure

```

---