

Technical Explanation on Human.exe

2026-03-11 · Updated 2026-03-21

#Botting #Programming #Python #generals.io

⚠ Warning

This article is generated by MiniMax-M2.5.

Human.exe¹ is a **purely algorithmic AI** for generals.io², a multiplayer web game where players compete to capture enemy generals while protecting their own. The bot does not use machine learning but relies on graph theory, combinatorial optimization, and heuristic search.

Key Features:

- Minimum Spanning Tree (MST) based gathering optimization
- Knapsack-based expansion planning
- Enemy general location prediction using BFS into fog of war
- Army interception and defense planning
- Board topology analysis (chokes, zones, central defense points)
- Win condition analysis for strategic decision making

Architecture

Project Structure

text

```
1 generals-bot/
2 |— bot_ek0x45.py      # Main bot class (~16k lines)
3 |— BotHost.py         # Game host / connection handler
4 |— MapMatrix.py       # Efficient 2D tile storage
5 |— Path.py            # Path representation
6 |
7 |— Gather/            # Gather algorithms
8 |   |— GatherMaxIterative.py
9 |   |— GatherPrune.py
10 |  |— GatherSteiner.py
11 |   |— ...
12 |
13 |— Algorithms/
14 |   |— TileIslandBuilder.py
15 |   |— MapSpanningUtils.py
16 |   |— ...
17 |
18 |— Behavior/
19 |   |— ArmyInterceptor.py # Army interception logic
20 |
21 |— Strategy/
22 |   |— OpponentTracker.py
23 |   |— WinConditionAnalyzer.py
```

¹<https://github.com/EklipZgit/generals-bot>

²<https://generals.io>

```

24 |
25 | └─ base/                # Game client library
26 |     └─ client/
27 |         └─ generals.py  # WebSocket client

```

Bot Flow

1. **BotHost.py** receives game state updates from the server
2. Calls `make_move()` with the current map state
3. **EklipZBot.find_move()** is invoked, which calls `select_move()`
4. **select_move()** runs the full decision pipeline:
 - `init_turn()` - Initialize turn state
 - `perform_move_prep()` - Scan map for leaf moves, large tiles
 - `pick_move_after_prep()` - Main decision logic

Core Data Structures

Tile

Represents a single cell on the map:

- `x, y` - Coordinates
- `player` - Owner (-1 = neutral)
- `army` - Army count (positive = owned, negative = neutral)
- `isCity` - Whether this is a city
- `isGeneral` - Whether this is a general
- `discovered` - Fog of war status
- `visible` - Currently visible
- `movable` - Adjacent tiles reachable

MapBase

Represents the game state:

- `turn` - Current turn number
- `cols, rows` - Map dimensions
- `players` - List of Player objects
- `generals` - Dict of `player_index` -> general tile
- `cities` - List of city tiles

Path

Represents a sequence of moves:

- `start` - Starting tile (PathNode)
- `tail` - End tile (PathNode)
- `length` - Number of moves
- `tileList` - List of tiles
- `tileSet` - Set of tiles (fast lookup)
- `value` - Strategic value of the path

Move

Represents a single move:

- `source` - Source tile
- `dest` - Destination tile
- `move_half` - Whether to move only half the army

Main Decision Loop

select_move() - Entry Point (bot_ek0x45.py:1885)

python

```
1 def select_move(self, is_lag_move=False) -> Move | None:
2     self.init_turn()                    # Initialize turn state
3     self.perform_move_prep()           # Scan map for move candidates
4     return self.pick_move_after_prep() # Make decision
```

init_turn() - Turn Initialization (bot_ek0x45.py:894)

1. Clear per-turn state
2. Update board analysis (chokes, zones)
3. Run army tracker scan
4. Recalculate enemy general prediction
5. Update tile islands
6. Run opponent tracker analysis
7. Scan city analyzer
8. Build win condition analysis

perform_move_prep() - Move Preparation

1. Find all **leaf moves** - tiles bordering enemy/neutral territory
2. Find all **capture leaf moves** - leaf moves that can capture
3. Identify **large player tiles** - high army tiles for expansion
4. Update expansion plan if needed

pick_move_after_prep() - Decision Logic (bot_ek0x45.py:1941)

Priority order (top wins):

1. **King Kill Moves** - Can capture enemy general immediately
2. **Defense Moves** - Under immediate threat
3. **Army Bonus Timing** - Maximize army on bonus turns
4. **Quick City Capture** - Fast city captures in early game
5. **First 25 Moves** - Early game expansion
6. **FFA Turtle Move** - Defensive play in multi-player games
7. **Flank Defense** - Protect against flanking attacks
8. **Enemy City Quick Kill** - Take vulnerable enemy cities
9. **Expansion Moves** - Planned expansion paths
10. **Gather Moves** - Default economic play

Gather System

Overview

The gather system builds **Minimum Spanning Trees (MST)** from owned tiles to maximize army value over time. The bot can gather from multiple disconnected regions simultaneously.

Key Components

GatherTreeNode

- tile - The tile being gathered
- value - Army value gathered from this subtree
- turns - Turns to reach this tile
- valuePerTurn - value / turns

- children - Child GatherTreeNodes

GatherPrune.py

Prunes MST to fit within turn constraints using heap-based removal of lowest value-per-turn leaves·python

```
1 def prune_mst_until(root_node, max_turns):
2     while root_node.turns > max_turns:
3         # Remove leaf with lowest valuePerTurn
4         remove_lowest_value_leaf(root_node)
```

Gather Algorithms

GatherMaxIterative.py

Iterative knapsack approach:

1. Find best path to each start tile
2. Use knapsack to select paths within turn limit
3. Extend tree iteratively, pruning low-value branches
4. Handle both positive (friendly) and negative (enemy) tiles

GatherSteiner.py

Prize-Collecting Steiner Tree (PCST) approach:

1. Build graph from map tiles
2. Use `pcst_fast` library³ for Steiner tree optimization
3. Use gradient descent to find optimal prize/cost cutoff
4. Returns connected tile set maximizing gathered value

Key Parameters

- `GATHER_SWITCH_POINT` - Max tiles before switching algorithms
- `gather_include_distance_from_enemy_TILES_as_negatives` - Penalty for gathering near enemies

Expansion System

Overview

Expansion finds the optimal set of attack paths to capture enemy territory. Uses knapsack-style optimization to balance value vs. turn cost.

TileIslandBuilder.py

Groups connected tiles into **islands** for strategic analysis:

- Determines which tiles can be recaptured together
- Used to avoid expanding into corners with no retreat path

IterativeExpansion.py

3-phase expansion approach:

1. **Large tile plans** - Prioritize high-value expansion targets
2. **Small tile expansion** - Fill gaps between large targets
3. **Leaf move inclusion** - Add border tiles that can capture

Key Parameters

- `expansion_single_iteration_time_cap` - Time limit per search

³https://github.com/fraenkel-lab/pcst_fast

- `expansion_enemy_expansion_plan_inbound_penalty` - Penalty for expanding into enemy attack path

Army Analysis & Interception

ArmyAnalyzer.py

Analyzes spatial relationships between two points (army to army, general to enemy):

python

```
1 class ArmyAnalyzer:
2     pathWays: MapMatrix      # All tiles on shortest paths
3     chokeWidths: Dict        # Number of alternate routes per tile
4     interceptTurns: int      # Turns to intercept (best case)
5     interceptDistances: int  # Worst-case intercept turns
```

Key Algorithm:

1. BFS from both points simultaneously
2. Find all tiles where $\text{dist}(A) + \text{dist}(B) == \text{shortest_distance}$
3. These form the “pathway” - all tiles that could be on a shortest path

Choke Points

- **Inner chokes** - Tiles with only 1 path in/out from general
- **Outer chokes** - Tiles where the enemy must pass through to reach general
- Choke width indicates defensibility (width=1 is most defensible)

ArmyInterceptor.py

Plans interceptions against enemy armies:

python

```
1 class ThreatBlockInfo:
2     tile: Tile                # Blocking tile
3     army_needed: int          # Army required to block
4
5 class InterceptionOptionInfo:
6     intercept_point: Tile     # Where to intercept
7     damage_blocked: int       # Army + tiles + recapture cost
8     intercept_turns: int      # Turns to reach intercept
```

Algorithm:

1. For each enemy army, find intercept points using ArmyAnalyzer
2. Calculate timing windows (best vs worst case based on chokes)
3. Compute damage blocked = enemy army + tiles + recapture cost
4. Rank by econ value per turn

Board Analysis

BoardAnalyzer.py

Analyzes overall board topology between generals:

python

```
1 class BoardAnalyzer:
2     inner_chokes: List[Tile]  # Chokes near our general
3     outer_chokes: List[Tile]  # Chokes near enemy
4     central_defense_point: Tile # Best defense position
5     intergeneral_analysis: ArmyAnalyzer
```

```
6 zones: MapMatrix # Core vs extended vs flank zones
```

Zone Classification

- **Core tiles** - Behind inner chokes, very safe
- **Extended tiles** - Past inner chokes but before front
- **Flank-danger tiles** - At edges, vulnerable to flanking

Central Defense Point

Tile that minimizes sum of distances to all friendly cities and general. Used as a strategic launch point.

Enemy General Prediction

The bot uses a sophisticated system to predict where enemy generals are located. The prediction is based on **analyzing where enemy armies emerge from fog of war** - the logic being that armies must originate from somewhere, and that “somewhere” is likely near the general.

Overview

The prediction works by:

1. **Tracking emergences** - Detecting when enemy armies appear from fog
2. **Back-propagating probability** - Scoring fog tiles based on their distance from emergence points
3. **Accumulating evidence** - Multiple emergences build up a “heat map” of likely general locations
4. **Constraining possibilities** - Eliminating tiles that couldn’t possibly be the general

Step 1: Detection of Army Emergence

When an enemy army tile changes in a way that indicates an army moved from fog into visible territory, the bot detects this as an **emergence**.

How Emergence is Detected

In bot_ek0x45.py, the `init_turn()` method processes army emergences:

python

```
1 for tile, emergenceTuple in self._map.army_emergences.items():
2     emergedAmount, emergingPlayer = emergenceTuple
3     if emergedAmount > 0 or not
4         self._map.is_player_on_team_with(tile.delta.oldOwner, tile.player):
5         self.opponent_tracker.notify_emerged_army(tile, emergingPlayer,
6             emergedAmount)
```

This triggers `OpponentTracker.notify_emerged_army()` (`Strategy/OpponentTracker.py:506`) which queues the emergence for processing.

Step 2: Processing the Emergence

When an army can’t be traced to a known source in fog, the `ArmyTracker.new_army_emerged()` method (`ArmyTracker.py:960`) is called.

Key Algorithm (`new_army_emerged`):

python

```
1 def new_army_emerged(self, emergedTile: Tile, armyEmergenceValue: float,
2     emergingPlayer: int = -1, distance: int | None = None):
3     # Distance defaults based on game turn
4     if distance is None:
5         distance = max(10, self.min_spawn_distance) + 1
```

```

5         if self.map.turn < 300:
6             distance = max(distance, int(self.map.turn / 4))
7         else:
8             distance = 75
9
10        # Scale the emergence value (capped at 10)
11        armyEmergenceValue = 2 + (armyEmergenceValue * 0.75)
12        if armyEmergenceValue > 10:
13            armyEmergenceValue = 10
14
15        armyEmergenceScaledToTurns = 5 * armyEmergenceValue / (5 + self.map.turn //
25)

```

The method then performs a **BFS from the emergence point** into fog of war:

python

```

1 def foreachFunc(tile, dist):
2     if not tile.discovered:
3         distCapped = max(distancePlateau, dist)
4         emergeValue = distancePlateau * armyEmergenceScaledToTurns // distCapped
5         # Add to the emergence heat map
6         self.emergenceLocationMap[emergedTile.player][tile] += max(1,
emergeValue)
7
8 breadth_first_foreach_dist(self.map, [emergedTile], distance, foreachFunc,
bypassDefaultSkip=True)

```

Key insight: The value added to each fog tile is inversely proportional to distance from the emergence point. Closer tiles get higher scores.

Step 3: The Emergence Heat Map

The `emergenceLocationMap` is a 2D matrix (one per player) that stores the accumulated prediction score for each tile.

Structure:

python

```

1 self.emergenceLocationMap[player_index] # MapMatrix[float] - scores per tile

```

How it Builds Up:

- Each emergence event adds value to nearby fog tiles
- Multiple emergences compound - tiles that are near multiple emergence points get higher scores
- Over time, the heat map converges on the most likely general location

Additional Emergence Handling:

The `notify_concrete_emergence()` method (`ArmyTracker.py:1051`) handles more certain emergences (when we know the path from fog):

python

```

1 def notify_concrete_emergence(self, maxDist: int, emergingTile: Tile,
confidentFromGeneral: bool):
2     player = emergingTile.player
3     if confidentFromGeneral:
4         # Reduce all existing scores (we found the general!)
5         for tile in self.map.get_all_tiles():
6             self.emergenceLocationMap[player][tile] /= 5.0
7

```

```

8     # Add strong evidence around the emergence
9     incAmount = 500 / len(incTiles) # Higher weight for known paths
10    for incTile, dist in incTiles:
11        self.emergenceLocationMap[player][incTile] += incAmount

```

Step 4: Determining the Predicted Location

The method `get_predicted_target_player_general_location()` (`bot_ek0x45.py:6189`) returns the highest-scoring tile.

Algorithm:

python

```

1  def get_predicted_target_player_general_location(self,
    skipDiscoveredAsNeutralFilter: bool = False) -> Tile:
2      maxTile = self.general
3      maxAmount = 0
4
5      for tile in self._map.pathable_tiles:
6          if tile.discovered or tile.isMountain or tile.isNotPathable or
tile.isCity:
7              continue
8
9          # Skip tiles that can't be the general
10         if not
self.armyTracker.valid_general_positions_by_player[self.targetPlayer][tile]:
11             continue
12
13         foundValue = 0
14
15         # Get the accumulated emergence score
16         if self.armyTracker.emergenceLocationMap[self.targetPlayer][tile] > 0:
17             foundValue +=
self.armyTracker.emergenceLocationMap[self.targetPlayer][tile] * 10
18
19         if foundValue > maxAmount:
20             maxTile = tile
21             maxAmount = foundValue
22
23     return maxTile

```

The predicted general location is simply the undiscovered fog tile with the highest accumulated emergence score.

Step 5: Constraining Valid General Positions

The `valid_general_positions_by_player` matrix tracks which tiles **could possibly** be the general. This gets updated throughout the game.

How tiles become invalid:

1. **Visible tiles** - Once we see a tile, it can't be the general
2. **Obstacles** - Mountains, swamps, etc. can't be generals
3. **Known player tiles** - Tiles owned by the player (after capture)
4. **Distance constraints** - Based on launch timing and tile count

The `limit_gen_position_from_emergence` method (`ArmyTracker.py:2787`):

This method uses game theory to further constrain where the general can be:

python

```

1 def limit_gen_position_from_emergence(self, p: Player, tile: Tile,
  emergenceAmount: int = -1):
2     # Based on when the player launched their first attack, we can limit
  distance
3     pLaunchTiming = self.player_launch_timings[tile.player]
4     maxDist = self.map.turn - pLaunchTiming
5     maxDist = min(maxDist, p.tileCount - 1)
6
7     # Additional constraints based on standing army and timing...

```

This uses the insight that:

- A player can only move `turn - launch_timing` tiles away from their starting position
- They can't have more territory than they've had time to capture

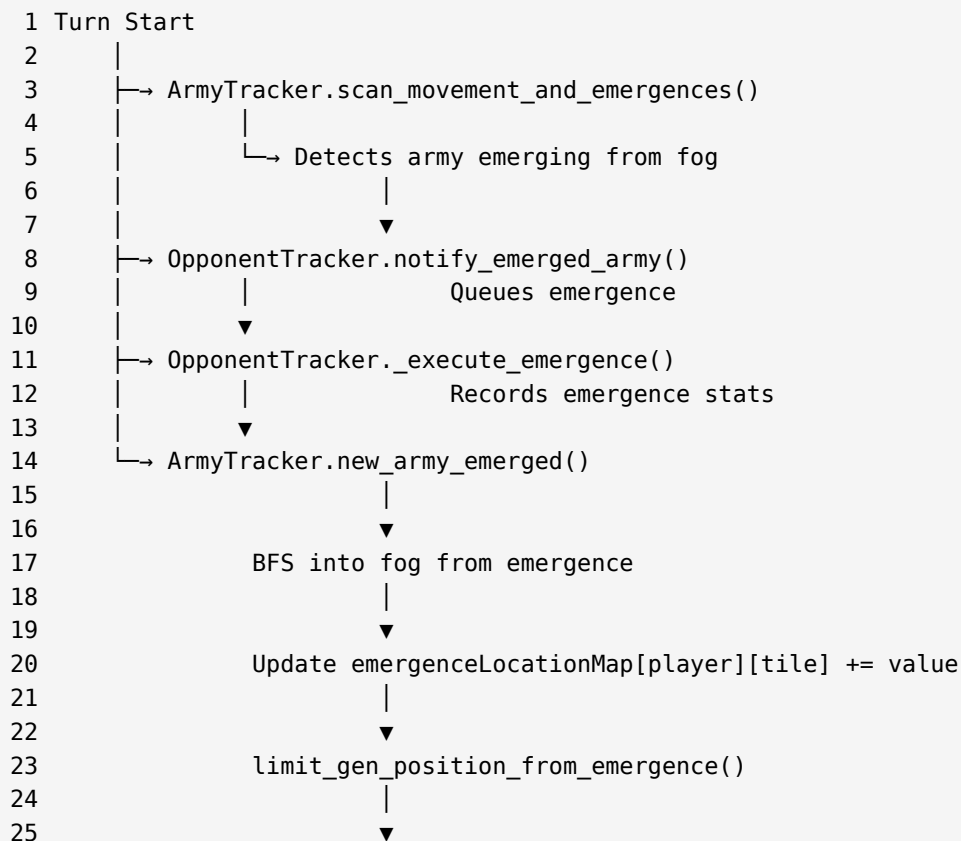
Step 6: Fallback Mechanisms

If the emergence-based prediction fails, there are fallbacks:

1. `get_max_explorable_undiscovered_tile()`
Finds the fog tile furthest from our general that is still reachable (used when enemy general is not yet visible and no emergences have been recorded)
2. `find_hacky_path_to_find_target_player_spawn_approx()`
Uses path-finding to estimate where the enemy general might spawn based on their visible tiles
3. **Edge case: FFA mode**
In free-for-all games with no target player, falls back to a more exploratory strategy

Complete Flow Diagram

raw



```

26         Update valid_general_positions
27
28         ▼
29     ┌───────────┴───────────
30     │
31     ▼
32 bot.get_predicted_target_player_general_location()
33     │
34     └── For each undiscovered tile:
35         │       score = emergenceLocationMap[player][tile] * 10
36         │
37         └── Return tile with highest score

```

Key Parameters

Parameter	Value	Description
min_spawn_distance	8-10	Minimum distance from our general
distance (emergence)	10-75	How far into fog to propagate emergence
armyEmergenceValue	capped at 10	Scaled based on army size
emergence multiplier	10x	Weight when calculating final prediction

Why It's Effective

1. **Compound evidence:** Multiple small emergences build up a clear picture
2. **Distance weighting:** Closer to emergence = higher probability
3. **Game theory constraints:** Uses timing and territory to eliminate impossible positions
4. **Adaptive:** Works early game (few emergences) and late game (many emergences)

The result is “insanely accurate” - even general-hiding strategies typically fail to hide the general location for long, as the bot can infer general location from the trail of armies that must have originated from near it.

Defense System

DangerAnalyzer.py

Identifies threats to the general:

python

```

1 class ThreatObj:
2     turns: int           # Turns until threat arrives
3     threatValue: int     # Army size of threat
4     path: Path           # Path threat will take
5     threatType: ThreatType # Kill, Vision, etc.

```

Defense Priority

1. **Kill threats** - Direct attacks on general
2. **Vision threats** - Enemy armies that could reveal general
3. **City threats** - Attacks on friendly cities

Defense Moves

- **Block** - Move army to block enemy path
- **Gather defense** - Gather army before enemy arrives
- **Interception** - Meet enemy army before reaching general

Strategy & Timings

Timings Class

Controls when to switch between gather and expansion phases:

python

```
1 class Timings:
2     cycleTurns: int        # Length of cycle (50 or 100)
3     splitTurns: int        # When to switch gather -> expand
4     launchTiming: int      # When to launch attack
```

Timing Calculation

Based on:

- **Map distance** to enemy general
- **Tile count** - more tiles = longer gather
- **Path composition** - more enemy/neutral tiles = later launch
- **Tile differential** - if behind, gather longer

Strategy Modes

All-In

Triggers when significantly behind on economy:

- Abandon defense
- Maximize army gather
- Launch aggressive attack

Flanking

Uses alternate enemy general positions:

- Attack from unexpected direction
- Launch earlier than normal

FFA Mode

Different strategy when multiple players remain:

- Prioritize map edges and city taking
- Don't prioritize middle until 3 players remain

2v2 Team Play

Communication

- **TeammateCommunicator** - Sends/receives team messages
- **TileCompressor** - Compresses tile data for efficient messaging

Coordinated Play

- Share enemy positions
- Coordinate launch timing
- Defend each other's generals
- Capture cities together

Special Handling

- Track teammate general separately
- Consider teammate tiles in defense planning
- Send launch timing recommendations

Performance Considerations

Time Management

- **Turn time limit** - 300ms to make move
- **Lag detection** - Skip expensive calculations if behind
- **Per-phase time limits** - Expansion, gather each have limits

Key Optimizations

1. **MapMatrix** - Fast O(1) tile access vs dict
2. **Caching** - Distance matrices, path calculations
3. **Incremental updates** - Only recalculate what changed
4. **Early termination** - Cutoff searches that take too long

Performance Timer

Tracks time spent in each phase for debugging and optimization:

python

```
1 with self.perf_timer.begin_move_event('Expansion phase'):
2     # code here
```

Key Configuration Parameters

Gather

- `gather_use_max_set` - Use max-set vs max-iterative
- `gather_include_distance_from_enemy_TILES_as_negatives` - 3

Expansion

- `expansion_single_iteration_time_cap` - 0.03 seconds
- `expansion_use_legacy` - Use legacy algorithm

Behavior

- `behavior_launch_timing_offset` - 3 (later launch)
- `behavior_flank_launch_timing_offset` - -4 (earlier flank)
- `behavior_out_of_play_distance_over_shortest_ratio` - 0.45

Engine (MCTS)

- `engine_use_mcts` - Enable Monte Carlo tree search
- `engine_mcts_scrim_armies_per_player_limit` - 2

Known Limitations

1. **Many bugs** - Off-by-one errors, edge cases
2. **No unit testing** - Evolved without proper test infrastructure
3. **General protection** - Doesn't hide its general well
4. **Strategic play** - No advanced tactics like disguise routes
5. **Late game** - Suboptimal at managing cities vs gather
6. **1v1 optimized** - Not good at FFA strategy