META-LEARNING FOR PLANNING: AUTOMATIC SYNTHESIS OF SAMPLE-BASED PLANNERS

Anonymous authors
Paper under double-blind review

ABSTRACT

In this paper, we discuss the challenge of generating domain-specific path planners in a data-driven fashion. Via the multi-objective optimization of Python code, we synthesize new sampling-based path planners that allow robots to adapt to new tasks and environments involving sequential decision-making. In addition to the ability to adapt to new environments, our approach also enables robots to balance their computational needs with improvements in task performance. We show that new computer programs can be generated which represent diverse variants of RRT* optimized to StarCraft maps.

1 INTRODUCTION

Learning and planning are two critical elements of intelligent and adaptive behavior. However, computational approaches for learning and planning are addressed by two largely separate research communities. Despite this challenge, various works have highlighted the strong relationship between these two fields. In particular, techniques that require *sequential* decision-making, e.g., reinforcement learning (RL), often blur any preconceived boundaries. Sutton & Barto (2018), argue that "all state-space planning methods involve computing value functions, either explicitly or implicitly", similar to the methodology followed in RL. Sampling-based planners, such as Rapidly-Exploring Random Tree (RRT) (LaValle & Kuffner Jr, 2001), in particular, bear a strong resemblance to stochastic learning techniques leading to interesting hybrids (Chiang et al., 2019; Shiarlis et al., 2017; Scholz & Stilman, 2010). Given these relationships, it is reasonable to wonder "Can we extend meta-learning to address planning domains, i.e., sequential decision-making?", or "After repeatedly getting lost in a new city, can a robot learn to generate better plans?"

In this paper, we address this question by focusing on the automatic generation of sampling-based planning algorithms. Given a set of environments or maps, our objective is to synthesize specialized path-planners that exploit any domain characteristics to their advantage. More specifically, we are interested in planners that minimize the involved computational costs, i.e., the number of evaluated nodes, while also improving the task performance, i.e., reduction in path length. Previous expert-created solutions for path planning include the works in (Hart et al., 1968; LaValle & Kuffner Jr, 2001; Karaman & Frazzoli, 2011; Gammell et al., 2014; Tahir et al., 2018; Bast et al., 2016; Lai et al., 2019; LaValle, 2006). In contrast to such hand-crafted methods, we propose an algorithmic search process, in the vein of meta-learning, for the discovery of novel planning algorithms. Our objective is to be able to a.) synthesize a large, diverse set of planners which b.) can be refined to adapt to the (multi-objective) needs of the current application domain.

We search for path planners by representing them explicitly in Python code, similar in spirit to genetic programming (GP) techniques (Banzhaf et al., 1998; Poli et al., 2008; Van Rossum et al., 2007). However, in conventional GP, a tree-based domain-specific program representation is used. Instead, we opt for general-purpose Python code to allow for the evolution of human-understandable and interfaceable source code. In contrast to traditional function approximation, program synthesis guarantees Turing-completeness but comes at the cost of large, discontinuous search spaces (Peng et al., 2021; Forstenlechner et al., 2017; Whigham et al., 1995; Gulwani et al., 2017; Chen et al., 2017; Lake et al., 2015).

2 METHODS

Developing novel path planners typically requires a careful balancing act between the optimality of the found paths and the number of nodes considered. Since these objectives are often at odds with each other, we employ multi-objective optimization to identify solution candidates. In particular, we use the non-dominated sorting genetic algorithm (NSGA2), henceforth referred to as *Pareto evolution* (Deb et al., 2002; Srinivas & Deb, 1994). The multi-objective search process repeatedly generates a population of Python programs that are evaluated on a set of (task-specific) maps. These programs are evaluated on three primary factors: the number of nodes they consider, the length of the paths they find, and their code length (which is only considered halfway through optimization). To represent these multiple objectives, Pareto evolution uses vectorized fitnesses and a *dominance* operator: one program *dominates* another if and only if every objective metric o_i is superior. If two fitness vectors are not strictly better than one another, then these vectors are said to be *Pareto equivalent*. In each generation, dominance-based tournament selection creates the initial offspring. These offspring are mutated and concatenated to the parent population[1]. Then, this larger population is sorted into Pareto fronts, i.e. Pareto equivalent programs, and the initial programs are selected for the next generation.

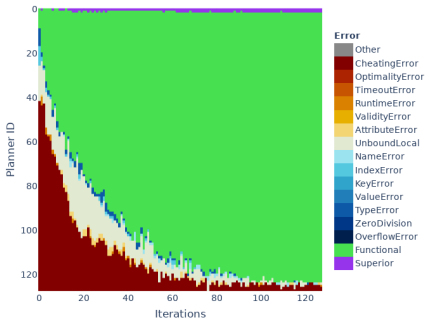

Figure 1: Iterative mutation of RRT* templates with selection for validity only. Shown at 128x128 scale for ease of interpretation.

All generated path planners follow a generic template that consists of the three functions *build*, *sample*, and *run*. The *run* function provides the entry point for executing a given program, while the *build* and *sample* functions address the sub-tasks of generating a graph and sampling new nodes. This template-based approach is inspired by Real et al. (2020) and aims at providing an interpretable structure for learned programs. Insertion of new code elements and modifications of existing code occurs within either of the three structural functions. All generated code is parsed using Python's generic parser but is then converted to a custom, mutable tree representation. We also allow for the addition of "dead" code to templates, to increase the evolvable code surface. This idea takes inspiration from biology, where large sections of the DNA string are non-coding. Our mutable syntax tree model uses three primary components: discrete uniform distributions, composites, and lists. For example, constants, variables, operators, and function names are all modeled as discrete uniform distributions. A potential composite type is an assignment statement, which is represented as a target (name) and an expression. In turn, the expression is also a composite that is defined recursively. Lastly, a list is a function or if-statement body, i.e. a list of statements, where a statement could be an assignment or return expression. Fig. 2 depicts a subset of our grammar and the full grammar is reproduced in the Appendix. The *build, sample,* and *run* functions allow for large portions of Python code to be modified in a generic and reusable fashion. Our grammar also provides the important ability to *mask* individual functions, lines of code, and entire grammar components. For instance, it is possible to disable the *init* function and hold the starting data structures constant. Unlike genetic programming systems (Banzhaf et al., 1998), we do not define type or arity constraints on our grammar, but rather apply strict rejection sampling. The optimization process begins with a working (or slightly broken) path planner represented in pure Python and, through evolutionary search, repeatedly changes this initial seed to improve performance on multiple objectives. As a benchmark environment, we use city graphs from major international cities and obstacle maps from the commercial game StarCraft. Importantly, randomness from sampling is controlled for by seeding the number generator. All experiments use the same seed: 2021.

```
constant  = .. | -1 | 0 | 1 | 2 | ..
atom      = name | constant
binopexpr = [atom binop expr]
compexpr  = [atom compops atoms]
expr      = atom | binopexpr | ...
assign    = target expr
```

Figure 2: Subset of Python Grammar

[1]Potentially, crossover can occur here. While crossover is extremely important in genetic programming, we do not study it in this paper

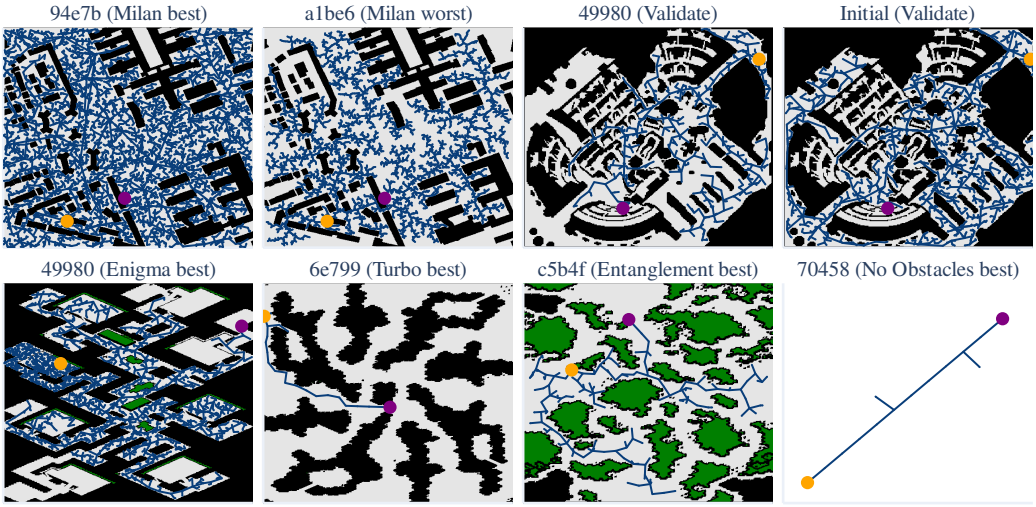

Figure 3: A diverse population, comparing unique planners and map-specialized behavior (Table 1).

3 Results and Discussion

First, we explore the density of our search space. In summary, the probability of a functional mutation is 93.9%, and the probability of a superior mutation is only 0.84%[2]. For reference, in the RRT* template, there are roughly 74 valid mutation locations, 52 in the *build* function and 22 in the *sample* function. Figure 1 shows frequencies of different program outcomes after repeated mutations to 128 program templates over 128 iterations. At each iteration, non-functional planners are removed via rejection sampling. The repeated application of mutations and rejection sampling leads to a larger fraction of functional or superior path-planner source code. Also, certain types of errors are more likely than others. For this RRT* template, unbound local variables are common because the code has many local variables. Also, the rejection process filters synthesized planners which attempt to cheat the optimization process by generating pathological trees or dishonestly altering path length.

To aid in reporting results, we identify unique planners by the MD5 hash of their Python source code, e.g. $94e7b$[3]. For interpretability, we also report important parameters of RRT* variants. Changes to these parameters happen in addition to structural changes that add and remove new code segments. These parameters are: α, r_0, and r_1. α is the inverse probability of sampling the goal node to move towards, r_0 is the maximum length for a tree segment, and r_1 is the rewiring radius. For example, a planner with $\alpha = 0.2$, $r_0 = 50$, $r_1 = 100$ biases towards the goal 80% of the time, takes steps of maximum length 50 pixels, and rewires nodes in a 100 pixel radius. This level of interpretability is a unique feature of program learning.

Figure 3 visualizes selected planners on various maps. Detailed statistics from many of these planners are also in Table 1. This Table describes best-in-class planners, i.e. those that consider the fewest nodes or generate the shortest path on a particular set of problems. This data comes from five experiments on different obstacle maps [4]. Two maps act as controls: an obstacle-free map and a section of Milan, Italy. The other three experiments use different StarCraft maps: Enigma, Turbo, and Entanglement. These experiments explore if planners will specialize to each map or alternatively perform well across many maps. Each experiment uses a population of 256 RRT* templates, which are mutated once and then optimized over 1024 iterations using Pareto evolution. While Table 1 uses problem sets with multiple (start, end) pairs, Figure 3 shows the behavior on a single pair.

Table 1 also describes the depth (edit distance) of a particular solution. Depth represents the difficulty of finding each solution. This value is non-linear, in the sense that finding a depth-n planner

[2]These probabilities are from $10,000$ planner mutations

[3]This hash is a *genotypic* hash, but we also use *phenotypic* hashes internally for caching and comparison

[4]Full experiment configurations are in the Appendix.

with random search would potentially require exponentially more samples (b^n where b is the number of possible mutations). Planners in Table 1 find a reduction in nodes between 64% and 89%. Reductions in path length are between 5% and 36%. Table 2 shows validation comparisons between interesting planners in Table 1 and planner $49980/c5b4f$. We select planner 49980 because it was evolved identically on multiple maps and appears to be generally effective. Each planner ran on the validation map shown in Figure 3 [5].

We selected two interesting planners from the Milan map: $94e7b$ and $a1be6$, shown in the upper left of Figure 3. Here, planner $94e7b$ considered the fewest nodes, while $a1be6$ considered the most. These planners differ in maximum step size (r_0) and goal-bias (α). Planner $94e7b$ can add edges up to $50,000$ pixels in length, which allows it to use fewer nodes to cover large open spaces. Alternatively, planner $a1be6$ takes steps of only 8 pixels, which allows it to create a fractal-like pattern with more uniform exploration, but causes it to consider significantly more nodes. Planner $94e7b$ has a 0% chance of biasing towards the goal, and $a1be6$ has a 12.5% chance of biasing. Despite these differences, planner $94e7b$ returns a 149 pixel long path, while $a1be6$ returns a 151 pixel long path. In validation, planner $94e7b$ fails to return a complete path and returns ∞ on both metrics. Planner $a1be6$ considers many more nodes ($+178.48\%$) due to a small step size of 8 pixels. However, $a1be6$ finds a path only $+11.13\%$ longer planner 49980's path.

The best planner on the obstacle-free map shows the largest reduction in the number of nodes considered (-89%) and path length (-36.23%). This reduction is due to the planner's (70458) preference to move towards the goal with a probability of 89%. Also, variant 70458 is shallow, at depth 1, while the solutions for non-trivial maps are typically between depth 2 and depth 4. This depth is possible because it only takes a single edit to change α and bias heavily towards the goal. In Table 2, this planner fails to generalize to the validation map because the map is quite cluttered, and biasing leads to frequent collisions.

In the bottom left of Figure 3 are the best-in-class planners on the StarCraft Engima, Turbo, and Entanglement maps. The planners $c5b4f$ (Entanglement) and 49980 (Enigma) are best-in-class for nodes considered. Phenotypically, these planners are identical: they have $\alpha = 7/8$ and $r_0 = 100$, i.e. a 12.5% chance of biasing and a maximum edge length of 100 pixels. Table 2 also shows several comparisons between other planners and planner 49980, and Figure 3 shows an example run of 49980 on the validation map (row 0, column 2). When validated, many specialized planners, such as those specialized to the Milan map or no-obstacles map, are not competitive with planner 49980. However, planner $6e799$ makes an interesting trade-off: It considers far fewer nodes at the cost of a significantly longer path, even in validation. These inter-metric trade-offs are at the heart of Pareto optimization and demonstrate the central goal of this paper.

Table 1: Best-in-class RRT* Statistics vs Unbiased RRT* (D = Depth)

Map	**$-\Delta\%$ Nodes**	**D**	**Hash**	r_0	α	**$-\Delta\%$ Path**	**D**	**Hash**	r_0	α
Milan	82.35%	4	$94e7b$	$50,000$	$7/4$	5.37%	2	$cd01e$	50	$7/8$
Enigma	74.87%	3	49980	100	$7/8$	9.70%	1	$1aefd$	50	$1/8$
Turbo	64.29%	3	$6e799$	50	$4/9$	12.20%	2	$6e799$	50	$4/9$
Entanglement	85.16%	3	$c5b4f$	100	$7/8$	21.77%	4	$ec420$	100	$6/8$
No Obstacles	89.47%	1	70458	50	$1/9$	36.23%	1	70458	50	$1/9$
Baseline	0.0%	0	Initial	50	1.0	0.0%	0	Initial	50	1.0

Table 2: Validation Comparisons (Planner 49980 as baseline, ∞ indicates failure)

Planner	$94e7b$	$a1be6$	70458	$6e799$	$c5b4f$	49980	Initial
$\Delta\%$ Nodes	∞	$+178.48\%$	$+113.86\%$	-73.37%	0%	0%	$+110.09\%$
$\Delta\%$ Path	∞	$+11.13\%$	$+13.25\%$	$+61.51\%$	0%	0%	$+3.83\%$

In conclusion, this work provides a proof-of-concept of our approach to generate new and potentially improved variants of path-planners for real-time adaptation. In the context of learning to learn, we would like to extend this technique to enable online adaptation in tasks that involve sequential decision-making, e.g., locomotion, manipulation planning, etc. In such tasks, a myopic action generation strategy may not be sufficient. Instead, we need to learn an algorithm that adapts through a action optimization over a time horizon, i.e., learning-to-plan.

[5] The Appendix includes further validations on other maps.

REFERENCES

Wolfgang Banzhaf, Peter Nordin, Robert E Keller, and Frank D Francone. *Genetic programming: an introduction*, volume 1. Morgan Kaufmann Publishers San Francisco, 1998.

Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F Werneck. Route planning in transportation networks. In *Algorithm engineering*, pp. 19–80. Springer, 2016.

Xinyun Chen, Chang Liu, and Dawn Song. Towards synthesizing complex programs from input-output examples. *arXiv preprint arXiv:1706.01284*, 2017.

Hao-Tien Lewis Chiang, Jasmine Hsu, Marek Fiser, Lydia Tapia, and Aleksandra Faust. Rl-rrt: Kinodynamic motion planning via learning reachability estimators from rl policies. *IEEE Robotics and Automation Letters*, 4(4):4298–4305, 2019.

Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2): 182–197, 2002.

Stefan Forstenlechner, David Fagan, Miguel Nicolau, and Michael O'Neill. A grammar design pattern for arbitrary program synthesis problems in genetic programming. In *European Conference on Genetic Programming*, pp. 262–277. Springer, 2017.

Jonathan D Gammell, Siddhartha S Srinivasa, and Timothy D Barfoot. Informed rrt*: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2997–3004. IEEE, 2014.

Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *The international journal of robotics research*, 30(7):846–894, 2011.

Tin Lai, Fabio Ramos, and Gilad Francis. Balancing global exploration and local-connectivity exploitation with rapidly-exploring random disjointed-trees. In *2019 International Conference on Robotics and Automation (ICRA)*, pp. 5537–5543. IEEE, 2019.

Brenden M Lake, Ruslan Salakhutdinov, and Joshua B Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.

Steven M LaValle. *Planning algorithms*. Cambridge university press, 2006.

Steven M LaValle and James J Kuffner Jr. Randomized kinodynamic planning. *The international journal of robotics research*, 20(5):378–400, 2001.

OpenStreetMap contributors. Planet dump retrieved from https://planet.osm.org . `https://www.openstreetmap.org`, 2017.

Daiyi Peng, Xuanyi Dong, Esteban Real, Mingxing Tan, Yifeng Lu, Hanxiao Liu, Gabriel Bender, Adam Kraft, Chen Liang, and Quoc V Le. Pyglove: Symbolic programming for automated machine learning. *arXiv preprint arXiv:2101.08809*, 2021.

Riccardo Poli, William B Langdon, Nicholas F McPhee, and John R Koza. *A field guide to genetic programming*. Lulu. com, 2008.

Esteban Real, Chen Liang, David So, and Quoc Le. Automl-zero: evolving machine learning algorithms from scratch. In *International Conference on Machine Learning*, pp. 8007–8019. PMLR, 2020.

Jonathan Scholz and Mike Stilman. Combining motion planning and optimization for flexible robot manipulation. In *2010 10th IEEE-RAS International Conference on Humanoid Robots*, pp. 80–85. IEEE, 2010.

Kyriacos Shiarlis, Joao Messias, and Shimon Whiteson. Rapidly exploring learning trees. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1541–1548. IEEE, 2017.

Nidamarthi Srinivas and Kalyanmoy Deb. Muiltiobjective optimization using nondominated sorting in genetic algorithms. *Evolutionary computation*, 2(3):221–248, 1994.

N. Sturtevant. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games*, 4(2):144 – 148, 2012. URL `http://web.cs.du.edu/~sturtevant/papers/benchmarks.pdf`.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. URL `http://incompleteideas.net/book/the-book-2nd.html`.

Zaid Tahir, Ahmed H Qureshi, Yasar Ayaz, and Raheel Nawaz. Potentially guided bidirectionalized rrt* for fast optimal path planning in cluttered environments. *Robotics and Autonomous Systems*, 108:13–27, 2018.

Guido Van Rossum et al. Python programming language. In *USENIX annual technical conference*, volume 41, pp. 36, 2007.

Peter A Whigham et al. Grammatically-based genetic programming. In *Proceedings of the workshop on genetic programming: from theory to real-world applications*, volume 16, pp. 33–41, 1995.

A Appendix

For the classical node-based path planning problems, this paper uses entire city maps from OpenStreetMap (OpenStreetMap contributors, 2017). This dataset includes 1.3 terabytes of graphs for the entire planet, and uses both walking and driving maps. For sampling based problems, we use the benchmarks defined in (Sturtevant, 2012)

A.1 Experiment Parameters and Scripts

Table 3 describes experiments, and 4 describes important scripts for replicating results.

Table 3: Experiment Setups

Experiment	Template	Optimizer	Pop. Size	Iterations	Maps
0	A* Geodesic	Pareto Ev.	1024	1024	Seattle Washington Drive
1	A* Inverse	Pareto Ev.	1024	1024	Seattle Washington Drive
2	Depth First	Pareto Ev.	1024	1024	Seattle Washington Drive
3	Breadth First	Pareto Ev.	1024	1024	Seattle Washington Drive
4	RRT*	Pareto Ev.	1024	1024	StarCraft Across The Cape
5	RRT*	Reg.Ev.	256	2048	StarCraft Across The Cape
6	RRT*	Pareto Ev.	256	2048	StarCraft Across The Cape
7	RRT*	Pareto Ev.	256	2048	Baldurs Gate Two Ar0205Sr
8	RRT*	Pareto Ev.	256	2048	Maze Maze512-32-9
9	RRT*	Pareto Ev.	256	2048	StarCraft Turbo
10	RRT*	Pareto Ev.	256	2048	StarCraft Enigma
11	RRT*	Pareto Ev.	256	2048	StarCraft Entanglement
12	RRT*	Pareto Ev.	256	2048	Blank
13	RRT*	Pareto Ev.	256	2048	Streets Milan 2 512

Table 4: Scripts

ID	Name	Purpose
0	density_of_program_space.py	Quantify Probability of Superior Mutation
1	mutation_selection_heatmap.py	Create Fig 1

A.2 GRAMMAR REPRESENTATION

```
name        = x | y | .. # any available variable
stored_name = x | y | ..
funcname    = sin | cos | ..
local       = a | b | .. # locals only
constant    = -10 .. | -1 | 0 | 1 | .. 10 # fixed range, customizable
stored      = stored_name | stored_local # Python store/load context aware
atom        = name | constant # Simple abstract types
binopexpr   = [atom binop expr]
unaryopexpr = [unaryop expr]
compexpr    = [atom compops atoms]
compexpr    = [boolop 2-atoms]
expr        = atom | binopexpr | unaryopexpr | boolexpr | compexpr
assign      = [targets expr]
return      = [expr]
stmt        = assign | return | logic_stmt | call
subscript   = [expr index]
index       = [expr]
if          = [expr body body]
body        = [stmt]
exprs       = [expr]
compops     = [compop]
names       = [name]
atoms       = [atom]
targets     = [stored]
call        = [funcname args kwargs]
binop       = Add | Sub | Mult | Div | Mod | Pow # bit operations disabled
unaryop     = UAdd | USub | Not | Invert
boolop      = And | Or
compop      = Eq | NotEq | Lt | LtE | Gt | GtE | Is | IsNot | In | NotIn
logic_stmt  = Pass # break and continue disabled: decreased sparsity
```

Example planner code (experiment 10, planner 49980).

```
Member fingerprint: 499806b2df0d88cce364a2e39f0bc660
Metrics: 48.78125 166.07939950634213, 0.13947296887636185, 485.625, None
def build_rrt_star(parent):
    n_nodes = 10 * 10 * 10 * 10
    n_nodes = n_nodes * 1
    K = n_nodes * 1
    rewire_rad = 100
    rewire_rad = rewire_rad * 1
    rewire_rad = rewire_rad * 1
    rad = 50
    rad = rad * 1
    rad = rewire_rad * 1
    cost = {space.start: 0}
    tree = {space.start: set()}
    parents = {space.start: None}
    for k in range(K):
```

```
        searching = True
        while searching:
            x, y = sample(parent)
            bx, by = brute_force_nearest_neighbor(parent,
                                                  tree.keys(),
                                                  (x, y))
            x, y = towards(parent, bx, by, x, y, rad)
            searching = space.intersects(((bx, by), (x, y))
                )
        if (x, y) not in tree:
            parents[x, y] = bx, by
            tree[bx, by].add((x, y))
            tree[x, y] = set()
            cost[x, y] = cost[bx, by] + distance(parent, bx, by, x, y)
            parent.considered.append((x, y))
        else:
            pass
        neighborhood = list(brute_force_neighbors(parent, tree.keys(),
                                                  (x, y), rewire_rad))
        for nx, ny in neighborhood:
            dist = distance(parent, x, y, nx, ny)
            if (cost[nx, ny] + dist < cost[nx, ny] and
                not space.intersects(((nx, ny), (x, y)))):
                tree[x, y].add((nx, ny))
                cost[nx, ny] = cost[x, y] + dist
                p = parents[nx, ny]
                tree[p].remove((nx, ny))
                p[nx, ny] = x, y
        if (distance(parent, x, y, *space.end) < rad and
            not space.intersects(((x, y), space.end))):
            tree[x, y].add(space.end)
            tree[space.end] = set()
            parents[space.end] = x, y
            return tree, parents, True
    return tree, parents, False

def sample(parent):
    alpha = 7
    alpha = alpha / 8
    x_max, y_max = space.shape
    r = random()
    if r < alpha:
        sampling = 4
        while sampling:
            x = randint(0, x_max - 1)
            y = randint(0, y_max - 1)
            sampling = space.check(x, y)
    else:
        r = -1
        x, y = space.end
    return x, y
```

This is the sampling function learned on the no-obstacles map (experiment 12). Notice that it biases towards the goal very heavily.

```
Member fingerprint: 70458588df13321e46189afefefb7908
Metrics: 1.84375 88.45203463004563, 0.0002014562487602234, 480.0, None

def sample(parent):
```

```
alpha = 1.0
alpha = alpha / 9
x_max, y_max = space.shape
r = random()
if r < alpha:
    sampling = True
    pass
    while sampling:
        x = randint(0, x_max - 1)
        y = randint(0, y_max - 1)
        sampling = space.check(x, y)
else:
    pass
    x, y = space.end
return x, y
```

This is the sampling function learned on the Milan map (experiment 13). Notice that is does no biasing towards the goal.

```
Member fingerprint: 94e7beac9994dd651987aec432d4420c
Metrics: 6.34375 149.27493716087838, 0.005812779068946838, 481.75, None

def sample(parent):
    alpha = 7
    alpha = alpha / 4
    x_max, y_max = space.shape
    r = random()
    if r < alpha:
        sampling = True
        while sampling:
            x = randint(0, x_max - 1)
            y = randint(0, y_max - 1)
            sampling = space.check(x, y)
    else:
        r = -1
        parent = -1
        x, y = space.end
    return x, y
```

For comparison, defaults are:

```
def sample(parent):
    alpha = 1.0
    alpha = alpha / 1
    x_max, y_max = space.shape
    r = random()
    if r < alpha: # Biased for potentially returning end node
        sampling = True
        while sampling:
            x = randint(0, x_max - 1)
            y = randint(0, y_max - 1)
            sampling = space.check(x, y)
    else:
        x, y = space.end
    return x, y
```

A.3 Additional Figures

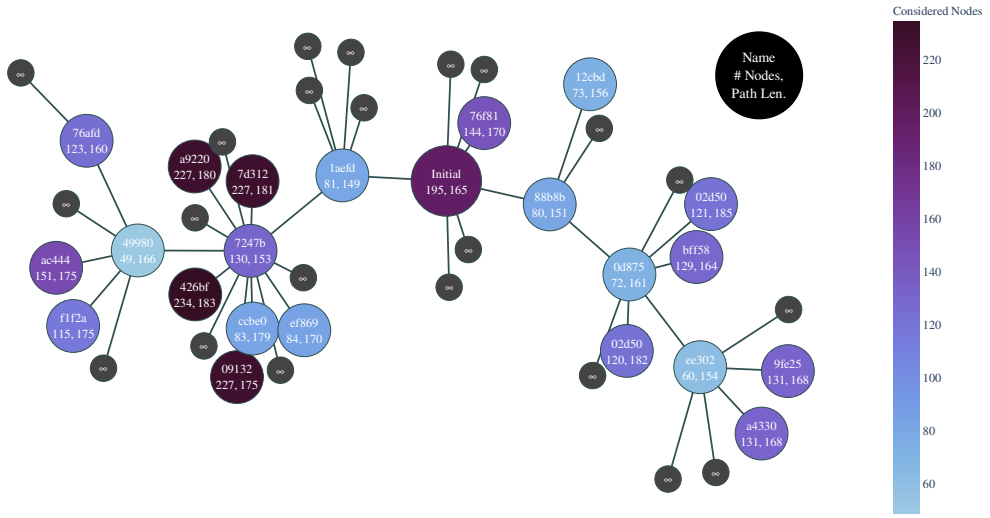

Figure 4: Tree representing phenotypically distinct RRT* mutations on the StarCraft Engima map. Labeled (fingerprint, nodes, path length) and colored by considered nodes

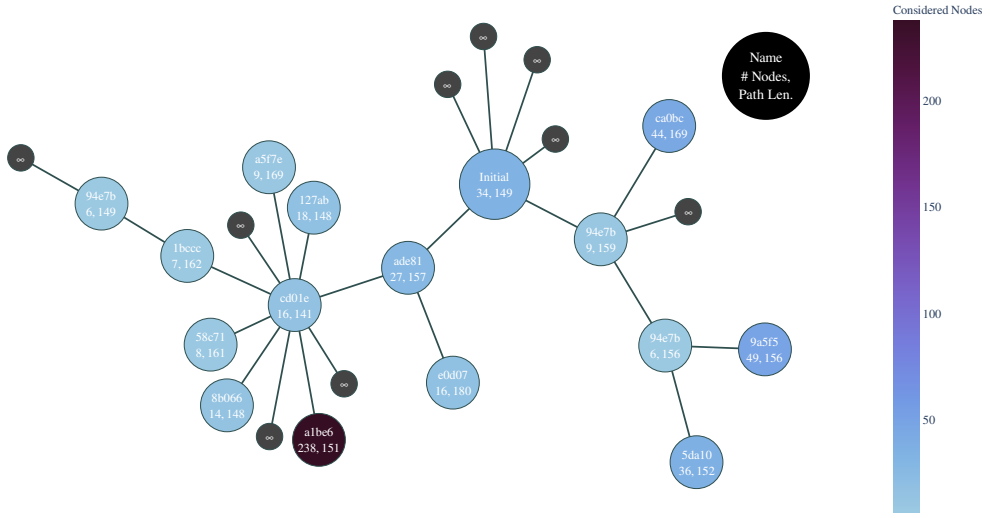

Figure 5: Tree representing phenotypically distinct RRT* mutations on the Milan map. Labeled (fingerprint, nodes, path length) and colored by considered nodes

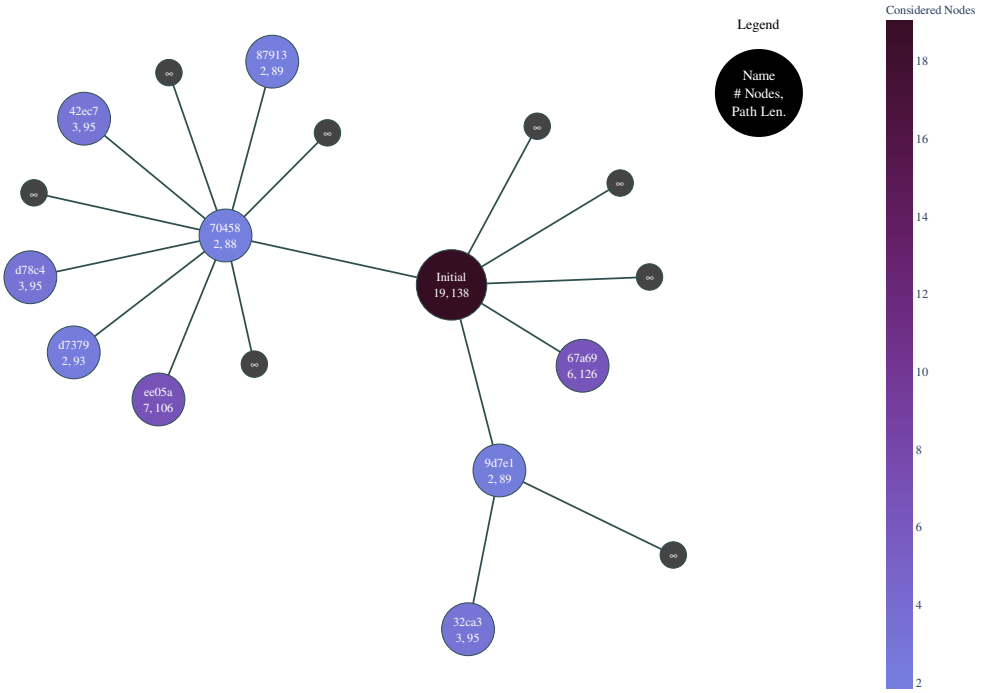

Figure 6: Tree representing phenotypically distinct RRT* mutations from optimization over the no-obstacles map (Used as a control). Labeled (fingerprint, nodes, path length). Colored by considered nodes.

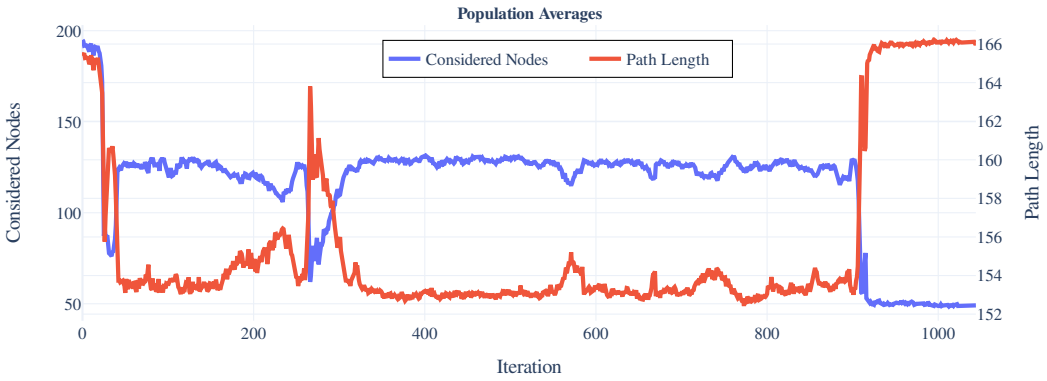

Figure 7: Average population metrics. While both metrics cannot be optimized completely, optimization only ever makes trade-offs. In this case, it makes a large trade-off between Path Length and Considered Nodes

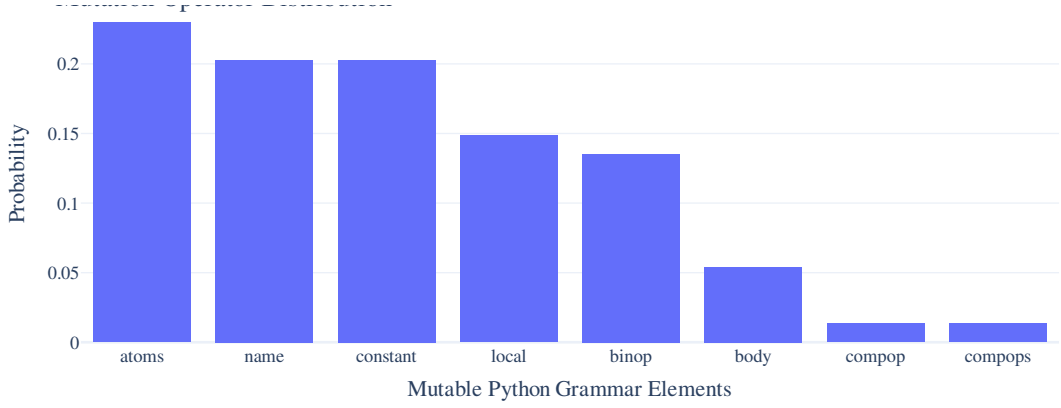

Figure 8: Probability Distribution Considered by the Mutation Operator

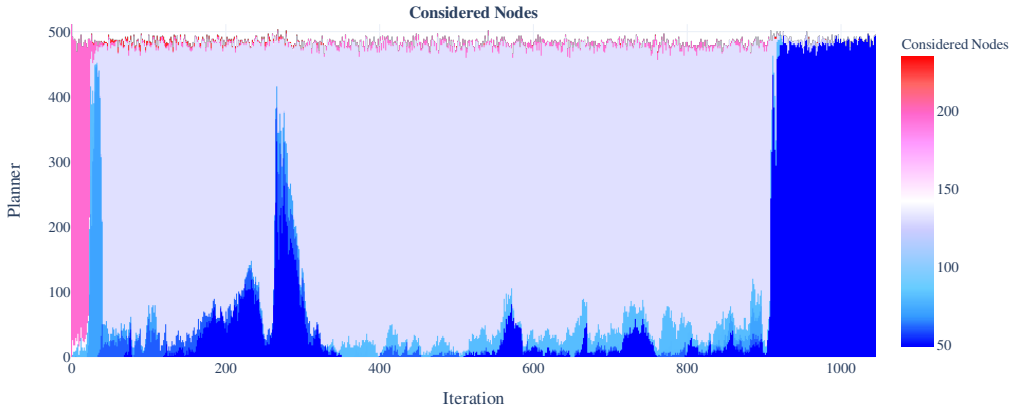

Figure 9: Population dynamics of optimized RRT* on the StarCraft Engima map. Planner *49980* eventually takes over the population.

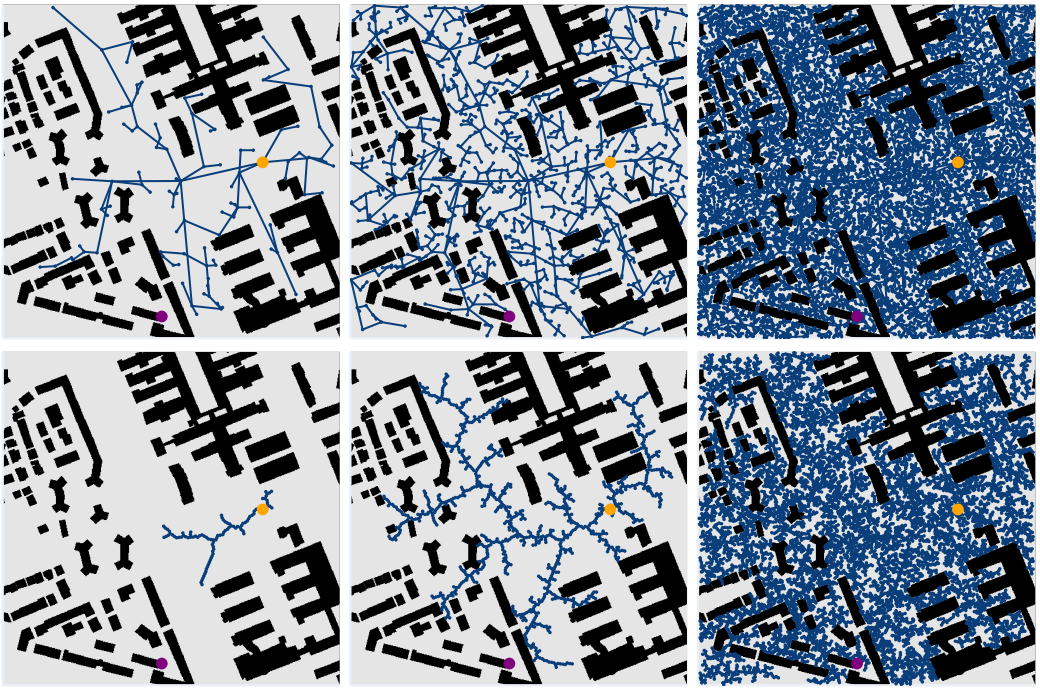

Figure 10: A comparison of the best/worst evolved planners on the Milan city map. Top planner (94e7b) has $r_0 = 50000$ and $\alpha = 7/4$. Bottom planner (aeb16) has $r_0 = 8$, $\alpha = 7/8$.

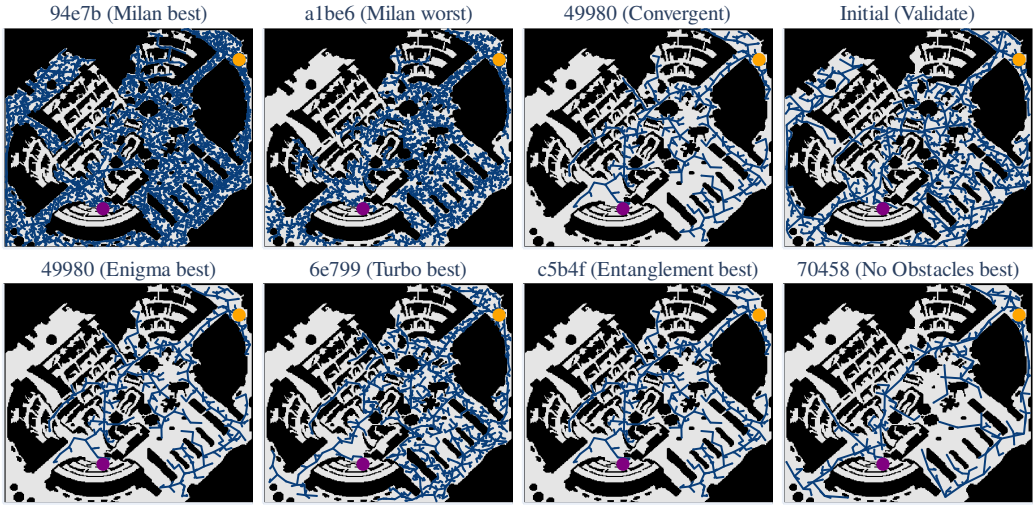

Figure 11: Validation

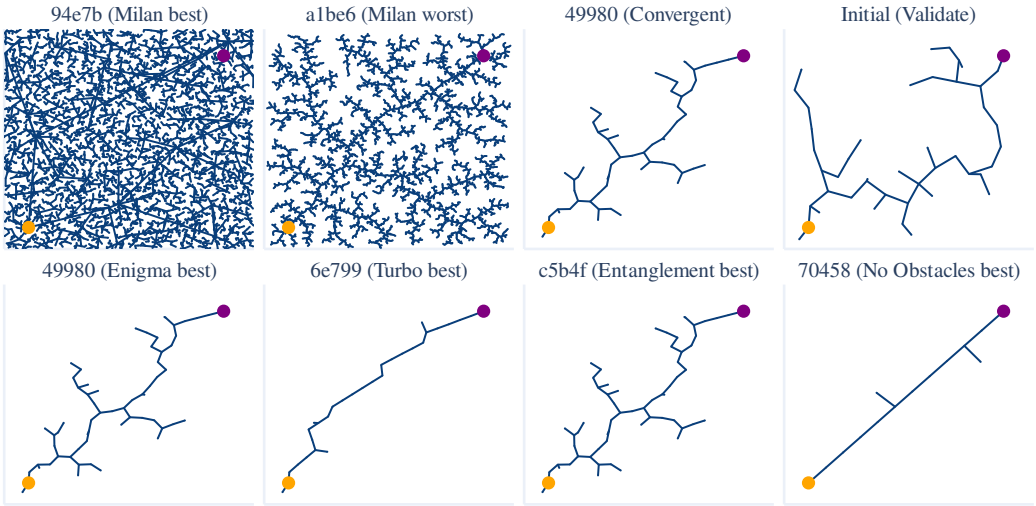

Figure 12: No Obstacles map validation

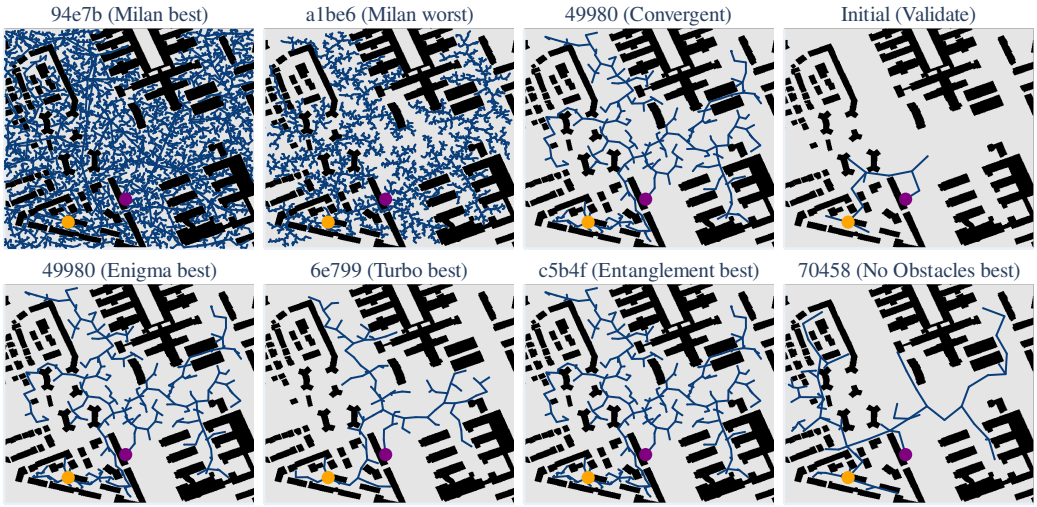

Figure 13: Milan map validation