Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling (ICAPS 2020)

Learning Neural Search Policies for Classical Planning

Paweł Gomoluch,[1] **Dalal Alrajeh,**[1] **Alessandra Russo,**[1] **Antonio Bucchiarone**[2]
[1]Department of Computing, Imperial College London, [2]Fondazione Bruno Kessler, Trento, Italy
{pawel.gomoluch14, dalal.alrajeh, a.russo}@imperial.ac.uk, bucchiarone@fbk.eu

Abstract

Heuristic forward search is currently the dominant paradigm in classical planning. Forward search algorithms typically rely on a single, relatively simple variation of best-first search and remain fixed throughout the process of solving a planning problem. Existing work combining multiple search techniques usually aims at supporting best-first search with an additional exploratory mechanism, triggered using a hand-crafted criterion. A notable exception is very recent work which combines various search techniques using a trainable policy. That approach, however, is confined to a discrete action space comprising several fixed subroutines.

In this paper, we introduce a parametrized search algorithm template which combines various search techniques within a single routine. The template's parameter space defines an infinite space of search algorithms, including, among others, BFS, local and random search. We then propose a neural architecture for designating the values of the search parameters given the state of the search. This enables expressing neural search policies that change the values of the parameters as the search progresses. The policies can be learned automatically, with the objective of maximizing the planner's performance on a given distribution of planning problems. We consider a training setting based on a stochastic optimization algorithm known as the *cross-entropy method* (CEM). Experimental evaluation of our approach shows that it is capable of finding effective distribution-specific search policies, outperforming the relevant baselines.

1 Introduction

Modern classical planners usually rely on heuristic forward search. Much research effort in recent years has been devoted to the development of advanced domain-independent heuristic functions (e.g. (Hoffmann and Nebel 2001; Helmert 2006; Richter and Westphal 2010; Domshlak, Hoffmann, and Katz 2015)). In contrast, the search algorithms at the core of many successful planners have largely remained simple variations of best-first search, such as greedy best-first search (Helmert 2006) or weighted A* (Richter and Westphal 2010). In recent years, some work has sought to combine best-first search with additional exploratory mechanisms, such as randomized order of node expansion (Valenzano et al. 2014; Asai and Fukunaga 2017), random walks and local search (Xie, Müller, and Holte 2014) or novelty search (Lipovetzky and Geffner 2017). The motivation behind these approaches is to enable the search to escape local minima and plateaus of the heuristic function.

Introducing even a single auxiliary exploration mechanism necessarily comes with a number of nontrivial design choices, such as when to switch between the main and auxiliary search approach. In addition, many of the exploration mechanisms come with a number of parameters of their own, such as the length and number of random walks to perform. The values of the parameters are typically selected by human experts. Furthermore, they stay fixed throughout the process of solving the problem.

The work introduced in this paper aims at automating the design of multi-technique search algorithms. To achieve it, we first construct a parametrized search algorithm which combines multiple search techniques in a flexible manner. Depending on the values of the parameters, the search algorithm can take form of BFS, iterated local search or random search, among others. Rather than choosing fixed values of the parameters, we introduce a trainable model mapping the current state of the search to an assignment over the parameters. We then train the model using the cross-entropy method (CEM) to obtain search policies tailored to specific problem distributions. We also consider a simpler setting, in which CEM is used to find values for the search parameters directly, without the state-dependent policy.

To the best of our knowledge, no existing literature has focused on a similar problem, except for our earlier work (Gomoluch, Alrajeh, and Russo 2019). That approach, however, is limited to selecting from a discrete set of several fixed search routines.

We implement our approach within the Fast Downward planning system (Helmert 2006) and evaluate it using five domains from the *International Planning Competition* (IPC). The learned search policies outperform baselines built around each the component techniques on their own, as well as a fixed hand-crafted combination of all the techniques.

2 Related Work

By using learning to improve the performance of a planner, our work joins the ranks of numerous learning approaches in classical planning. Over the years, learning has been used

to acquire macro operators (Fikes, Hart, and Nilsson 1972; Coles and Smith 2007; Gerevini, Saetti, and Vallati 2009), heuristic functions (Yoon, Fern, and Givan 2008; Virseda, Borrajo, and Alcazar 2013; Garrett, Kaelbling, and Lozano-Perez 2016) and search control rules (Leckie and Zukerman 1998; Yoon, Fern, and Givan 2008).

Little work has sought to learn directly in the space of possible search algorithms until our recent approach, in which policy gradient was used to learn search strategies (Gomoluch, Alrajeh, and Russo 2019). That work is closely related to this paper in that it combines various search techniques in a single search run, and it learns how to best do so, given a particular distribution of planning problems. The most important difference is the space of possible search policies. In our previous work, a policy is trained to choose a search subroutine out of a set of five. While shown to be effective, this approach imposes somewhat arbitrary restrictions on the action and policy spaces. For example, the planner can choose to perform ϵ-greedy search, but not the value of epsilon. It can choose to perform random walks but not set their length. The span of local search is also effectively determined by one of the hyperparameters. Further, local and ϵ-greedy search are made mutually exclusive, although in some domains it may be beneficial to combine the two, i.e., randomize the order of node expansion in local search. In contrast, in this paper, various search techniques are merged into a single parametrized search routine that subsumes each of the subroutines on its own for some specific assignment of the parameters. In reinforcement learning terms, the previous work deals with a discrete action space while this one is close in spirit to continuous action spaces.[1]

Another important difference is the representation of the search policies. Our previous work (Gomoluch, Alrajeh, and Russo 2019) adopts a tabular approach with four states generated by two binary features. In this work, we use numeric features and a function approximator mapping them into the values of search parameters.

A different approach to planning with multiple search techniques is to divide the available time and perform a number of separate search runs. The original Fast Forward planner (Hoffmann and Nebel 2001) first attempts to solve the problem using *enforced hill climbing*. If this fails, it falls back to best-first search. The LAMA planner (Richter and Westphal 2010) follows greedy search with a number of weighted A* runs with decreasing weights in order to find the first solution as quickly as possible and then keep improving on it while the time permits. The idea is taken even further in the planner portfolios which run a number of potentially unrelated solvers independently (e.g. (Helmert, Röger, and Karpas 2011; Cenamor, De La Rosa, and Fernández 2016)). Our approach differs from the portfolios by dealing with a single search run and learning how to best proceed with the current search frontier. In doing so, it is complementary to portfolio approaches. For example, future work could consider building portfolios composed of parametrized planners optimized for various problem distributions.

[1]The action space is not strictly continuous as some of the search parameters take integer values. However, the presence of real-valued parameters makes the space infinite.

3 Background

Classical Planning Planning is the problem of finding sequences of actions leading from an given initial state to state in which a given goal is satisfied. Classical planning, in particular, relies on a perfect and deterministic world model known to the planner. Formally, the classical planning task is given by a tuple $\langle V, O, s_0, g\rangle$, where V is a set of finite-domain variables, O is the set of operators, s_0 is the initial state and g is the goal. The initial state s_0 is an assignment over the variables of V. The goal g is a partial assignment over V. The operators $o \in O$ are themselves defined by specifying their preconditions $\mathrm{pre}(o)$ and effects $\mathrm{eff}(o)$, both of which are partial assignments over V. An operator o can be applied in state s if and only if its preconditions are satisfied in the state, $\mathrm{pre}(o) \subseteq s$. The state resulting from applying an operator o to state s, $o(s)$ is determined by setting the values of variables covered by $\mathrm{eff}(o)$ to the corresponding values, and keeping all the other variables unchanged. The task is to find a sequence of operators o_0, o_1, o_n, such that applying them in order, starting from the initial state, leads to satisfaction of the goal g: $g \subseteq o_n(o_{n-1}(...o_0(s_0)...))$.

The most common approach to classical planning is forward search. Forward search starts by inserting a node containing the initial state s_0 in the *open list*. It proceeds by iteratively removing nodes from the list and expanding them by applying all of the operators applicable in the given state and adding the resulting nodes to the open list. This is repeated until a state satisfying the goal g is reached or the open list becomes empty. Best-first search (BFS) always expands the node n with the lowest value of some evaluation function $f(n)$. If $f(n)$ depends solely on the heuristic estimate of the distance to the goal $f(n) = h(n)$ the search becomes *greedy* best-first search (GBFS).

Cross-Entropy Method The cross-entropy method is a gradient-free stochastic optimization technique originating from rare event simulation (Rubinstein 1999; de Boer et al. 2005). It belongs to a wider family of population-based optimization techniques known as Evolution Strategies (ES), which also include approaches such as Covariance Matrix Adaptation (Hansen and Ostermeier 2001) and Natural Evolution Strategies (Wierstra et al. 2014). In recent years, various forms of evolution strategies have been successfully applied as policy search methods in a number of challenging reinforcement learning domains (Mannor, Rubinstein, and Gat 2003; Salimans et al. 2017; Chrabaszcz, Loshchilov, and Hutter 2018; Conti et al. 2018). The common idea underlying ES is to maintain a population of candidate solutions and iteratively update it towards solutions of higher quality.

Given a (possibly stochastic) function $s(\mathbf{x}), \mathbf{x} \in \mathcal{X}$, the cross-entropy method introduces an auxiliary distribution $f_v(\mathbf{x})$ over the possible solutions. The distribution is itself parametrized by a vector $\mathbf{v}$. For example, if the distribution is a multivariate Gaussian, $\mathbf{v}$ can contain its mean and flattened covariance matrix. At each iteration t, n candidate solutions $\mathbf{x}_t^{(1)}, \mathbf{x}_t^{(2)} \ldots \mathbf{x}_t^{(n)}$ are sampled from the distribution

and evaluated. The parameters of the distribution are then updated to maximize the likelihood of the m best candidate solutions ($m < n$). To select the best solutions, a performance threshold γ_t is introduced, equal to the $\frac{m}{n} \cdot 100$-th percentile of the candidate scores $s(\mathbf{x}_t^{(1)}), s(\mathbf{x}_t^{(2)}) \dots s(\mathbf{x}_t^{(n)})$. The value of $\mathbf{v}$ for the next iteration is designated by solving:

$$\tilde{\mathbf{v}}_t = \underset{\mathbf{v} \in \mathcal{V}}{\operatorname{argmax}} \frac{1}{N} \sum_{i=1}^{N} \mathbf{I}_{s(\mathbf{x}_t^{(i)}) > \gamma_t} \log f(\mathbf{x}_t^{(i)}, \mathbf{v}) \qquad (1)$$

where $\mathbf{I}_{s(\mathbf{x}_t^{(i)}) > \gamma_t}$ is the indicator variable of the event $s(\mathbf{x}_t^{(i)}) > \gamma_t$, i.e., it is 1 if the score of sample i is higher than the threshold γ_t and 0 otherwise.

In the particular case when the auxiliary distribution is a multivariate Gaussian, solving Equation 1 amounts to setting $\tilde{\mathbf{v}}_t$ to the mean and covariance of the m samples above the threshold γ_t.

Instead of using the solution of Equation 1 directly for the next iteration's distribution ($\mathbf{v}_{t+1} = \tilde{\mathbf{v}}_\mathbf{t}$), it is also possible to perform a *smoothed* update:

$$\mathbf{v_{t+1}} = \alpha \tilde{\mathbf{v}}_\mathbf{t} + (1 - \alpha)\mathbf{v_t} \qquad (2)$$

The process can be repeated for a fixed number of iterations or until a specific stopping criterion is met. After the final iteration t the result can be obtained by extracting the mean of the final distribution from $\mathbf{v_{t+1}}$.

4 Parametrized Search

The key idea underlying this work is that multiple forward search techniques can be combined in a single forward search algorithm. For instance, the algorithm can interleave between global and local best first search by performing a number of standard node expansions and then a number of local expansions. Both local and global expansions can optionally be followed by a number of random walks started from the expanded node. Additionally, both global and local search can randomize the order of node expansion, by choosing a random node from the open list instead of the one with lowest h, with the probability of ϵ.

Pseudocode implementing this idea is presented in Algorithm 1. Typically for a forward approach, the search starts by initializing the open list to contain the initial state s_0 (line 2). The main loop of the algorithm (lines 3-10) performs a number of search steps using the global open list (lines 5-6). Then it initializes a new local open list with a single node removed from the global list (line 7). The local list is used to perform a number of local search steps (lines 8-9). After the local search, all the nodes from the local list are merged into the global one (line 10). The number of global and local steps in a single iteration of the main loop is controlled by two parameters: C is the total number of steps and c is the proportion of local steps. One iteration of the main loop consists of $(1 - c) \cdot C$ global and $c \cdot C$ local steps.

A search step (lines 12-31) involves a single node expansion (lines 21-26), optionally followed by N random walks of length L, starting at the expanded node (lines 27-30). One of the details omitted from Algorithm 1 is keeping track of the lowest known heuristic value $h_{\min}$. The value is initialized to the heuristic evaluation of the initial state and then updated whenever a state with lower evaluation is encountered. Similarly, the algorithm needs to keep track of the number of node expansions performed since the last update of $h_{\min}$. This value is used to decide whether to follow the current node expansion with random walks (line 27). The walks are triggered on condition that no decrease of $h_{\min}$ has been observed during the last S expansions. The S parameter plays a role analogous to the *STALL_SIZE* parameter used by (Xie, Müller, and Holte 2014) to decide when to switch from GBFS to the auxiliary exploration technique.

If the step is using the local list and it turns out to be empty, another node is moved from the global list to the local one (line 20). If the global list becomes empty without reaching the goal, the search fails (lines 15 and 18).

Overall, the search parameters include:

- ϵ – the probability of selecting a random node from the open list;
- S – the number of expansions without progress necessary to trigger a random walk;
- R – the number of random walks following a single node expansion;
- L – the length of a random walk;
- C – the number of node expansions in the global-local cycle;
- c – the proportion of local search in the global-local cycle.

Assigning the values of the parameters positions the resulting algorithm in the space between various search approaches. For example, with $\epsilon = 0$, $R = 0$ and $c = 0$ the algorithm becomes greedy BFS, independently of the remaining parameters. With $\epsilon = 0$, $R = 0$, $c = 1$ and $C = 100$, the algorithm performs iterated local search with a span of 100. Intermediate values of c result in interleaving of global and local search, with random walks optionally added in both phases, as determined by remaining parameters.

The values of the search parameters can be changed during execution of the algorithm. It is convenient to update them at the beginning of every iteration of the main loop (line 4) and keep them fixed throughout the iteration. In the following sections we introduce the task of learning search policies, whose purpose is to set the values of the parameters, given the current state of the search.

5 Search State Representation

To facilitate learning of state-dependent search policies, we introduce a high-level representation of the state of the planner. In particular, we consider the following features of the planner's state:

- the heuristic value of the initial state $h(s_0)$;
- the lowest heuristic value encountered within the search $h_{\min}$;
- the time elapsed since the search started;
- the number of node expansions performed since the last change in the value of $h_{\min}$;

Algorithm 1 Parametrized planner

```
1: function PLANNER(s_0, g, O)
2:     global_open ← [s_0]
3:     while true do
4:         ε, S, R, L, C, c ← set_search_parameters()
5:         for i = 1 ... (1 − c) · C do
6:             STEP(global_open, O, g)
7:         local_open ← [pop(global_open)]
8:         for i = 1 ... c · C do
9:             STEP(local_open, O, g)
10:        merge local_open into global_open
11:
12: function STEP(open, O, g)
13:     if open is empty then
14:         if open is global_open then
15:             return failure          ▷ return from PLANNER
16:         else
17:             if global_open is empty then
18:                 return failure      ▷ return from PLANNER
19:             else
20:                 local_open ← [pop(global_open)]
21:     s ← pop(open, ε)
22:     if s ∈ g then
23:         plan ← extract_plan(s)
24:         return plan                 ▷ return from PLANNER
25:     successor_states ← expand(s, O)
26:     add(open, successor_states)
27:     if expansions_without_progress > S then
28:         for i = 1 ... R do
29:             walk_states ← random_walk(s, L)
30:             add(open, walk_states)
31:     return in_progress              ▷ return to the PLANNER loop
```

- the number of states the search has generated;
- the number of unique states the search has generated;
- the total number of nodes the search has expanded.

Intuitively, the features capture important information about the state of the search. For example, a large number of node expansions performed since the last decrease in $h_{\min}$ indicates that the planner is facing a local minimum or a plateau of the heuristic function. In such a situation, increasing the amount of exploratory behavior is likely to be beneficial. Comparing the current value of the heuristic with $h(s_0)$ enables estimating the relative progress towards the goal. If the values are close, the search has likely made little progress towards the goal. On the other hand, a value of $h_{\min}$ close to 0 can suggest that the search frontier is close to the goal. Naturally, the reliability of such estimates will depend on how well the heuristic function correlates with actual distance to the goal in the current domain. This information can be particularly useful in conjunction with the elapsed time t. If sufficient progress is being made at early stage of the search, there may be no incentive to deploy techniques making rapid progress towards the goal at the cost of sacrificing the plan quality (for example random walks in logistics-style domains). On the other hand, if time is running out while significant distance remains to be covered, more aggressive approach may be desirable in order to avoid timeout.

In the following section, these features are used as the state representation for learning search policies.

6 Learning Search Policies

We consider a search policy to be a vector valued function mapping the state of the search to the values of the search parameters. Formally, $Y = \pi(\Phi)$, where Y is a vector of the search parameters $\langle \epsilon, S, R, L, C, c \rangle$, introduced in Section 4, and Φ is the vector of state features listed in Section 5.

Policy model

We represent the search policies using a parametric function approximator $\pi_\theta(\Phi)$, with trainable parameters θ. Specifically, we use a feed-forward neural network. The inputs of the network are the planner's state features and its outputs are the search parameters. The network contains one hidden layer of 7 units with sigmoid activation function $f(z) = \frac{1}{1+e^{-z}}$. The treatment of the network's output depends on the search parameter it represents. For real-valued search parameters (ϵ and c) taking values from range $(0, 1)$, the network outputs are passed through a sigmoid function. For search parameters taking nonnegative integer values (S, R, L and C) the outputs are restricted to nonnegative values $y_r = \max(y, 0)$ and truncated to the integer part.

Because the planner's state features differ in order of magnitude, we scale them before passing them through the network. We choose the scaling factors on a per-domain basis, by running our parametrized planner with fixed parameter values on a separate batch of problems from the domain and recording the maximum values $\phi_{\max}$ of all the features observed in the process. The maximum values are then used to scale each of the features $\phi_{\text{scaled}} = \frac{\phi}{\phi_{\max}}$, so that value 1 of the scaled feature represents the highest value recorded on the batch of problems. The scales remain unchanged throughout training and testing of the system.

Similarly, we scale the outputs of the network. With the exception of ϵ and c, whose values are put in the range of $(0, 1)$ by the sigmoid, we scale the outputs so that the value of 1 results in moderate use of the corresponding search technique. Concretely, we multiply the number of random walks R by 5, the length of the walks L by 10, the stall condition S by 10 and the length of the global-local cycle C by 100.

Policy evaluation

To learn search policies in an evolutionary process, it is necessary to establish a way of evaluating candidate policies. In this work, we adopt an approach based on the scoring function used in the satisficing track of the *International Planning Competition* (IPC) since the 2008 edition[2]. For a failed problem, a planner receives a score of 0. For a solved one the score is defined as:

$$g = \frac{c_{\min}}{c}$$

[2] http://icaps-conference.org/ipc2008/deterministic/

where c is the cost of the found plan, and c_{min} is the lowest known cost of a plan solving the problem. In practice, c_{min} is often the lowest cost of a plan returned by any of the competing planners. The score obtained on a set of problems is the sum of scores for each of the problems.

$$G = \sum_p g_p$$

In our training setting, detailed in the next section, the candidate policies are put in competition against each other: $c_{\min}$ is the lowest solution cost found by any of the candidate policies. This is convenient since the problems are generated randomly for each iteration, and so no reference costs are known a priori. We remark that when training on a fixed set of problems, the lowest plan cost could be retained between iterations or designated using a set of reference planners.

Policy learning

To learn search policies best suited to specific problem distributions, we employ the cross-entropy method (Rubinstein 1999; Mannor, Rubinstein, and Gat 2003). We introduce a multivariate Gaussian distribution over the policy parameters θ, which is itself parametrized by the mean vector μ and covariance matrix Σ.

$$\theta \sim \mathcal{N}(\mu, \Sigma)$$

We initialize μ with a zero vector and Σ with an identity matrix.

At each iteration t, n samples $\theta_1 ... \theta_n$ are drawn from $\mathcal{N}(\mu_t, \Sigma_t)$. The resulting policies are evaluated on the set of training problems. The mean and covariance of the parameter distribution are then updated towards the mean and covariance of the m candidate solutions with the best performance:

$$\nu_t = \frac{1}{m} \sum_{i=1}^{m} \theta_i^t \tag{3}$$

$$\mu_{t+1} = (1 - \alpha)\mu_t + \alpha\nu_t \tag{4}$$

$$\Sigma_{t+1} = (1 - \alpha)\Sigma_t + \alpha \frac{1}{m-1} \sum_{i=1}^{m} (\theta_i^t - \nu_t)(\theta_i^t - \nu_t)^\top \tag{5}$$

where α is a smoothing factor and $\theta_1^t ... \theta_m^t$ are the m best scoring candidates of iteration t.

The pseudocode of the learning approach is presented in Algorithm 2. The inputs of the algorithm include a distribution of planning problems ($\mathcal{P}$), the number of iterations (u), the number of planning problems sampled at each iteration (r), the number of policies sampled at each iteration (n) and the number of elite policies used to update the policy distribution (m).

The main loop of the algorithm starts by randomly generating a set of problems following the target distribution (line 4). Generally, the problems $p_1 ... p_r$ do not have to be independently and identically distributed. As detailed in Section 7, in our experiments we use problem distributions resembling the IPC problem sets, which contain problems of varying difficulty. In the absence of problem generators, this step could be replaced by sampling from a fixed set of training problems.

Further, n policies are sampled from the current policy distribution (line 5). Each of the policies is used to tackle each of the generated problems (lines 6-8). The results are used to compute IPC score for each policy (line 9). The policies are then sorted according to their scores in order to select the m best performing ones. The new mean of the policy distribution is obtained by averaging the m samples and performing a weighted sum with the old mean (line 11). The algorithm returns the final mean of the policy distribution as the resulting policy (line 12).

Algorithm 2 Policy learning

```
1:  function TRAIN(P, u, r, n, m)
2:      initialize μ and Σ
3:      for i = 1...u do
4:          p_1...p_r ← P                          ▷ sample r problems
5:          θ_1...θ_n ← N(μ, Σ)                    ▷ sample n policies
6:          for j = 1...n do
7:              for k = 1...r do
8:                  run policy θ_j on p_k, record plan cost c_{j,k}
9:          G_1...G_n ← compute IPC score for θ_1...θ_n
10:         sort θ_1...θ_n by scores G_1...G_n (highest first)
11:         update μ and Σ according to Equations 4 and 5
12:     return μ
```

CEM can also be used directly to find the values of the search parameters best suited to a particular distribution of planning problems. In this setting, the parameter vectors θ simply store each of the search parameters $\theta = \langle \epsilon, S, R, L, C, c \rangle$. There is no state-dependent search policy and the values of the search parameters remain fixed throughout the process of solving the planning problem. In Section 7, we evaluate this approach along with the state-dependent search policies.

7 Experiments

We have implemented the parametrized planner as a component of the Fast Downward (Helmert 2006) planning system. The source code is available online[3].

We evaluate our approach on five IPC domains: *Elevators*, *Floortile*, *No-mystery*, *Parking* and *Transport*. These are all the domains of the learning track of IPC 2014, with the exception of the *Spanner* domain. The latter was designed specifically not to work well with delete-relaxation heuristics, such as the Fast Forward heuristic (Hoffmann and Nebel 2001), which we use throughout our experiments. For problem generation, we use the problem generators published by the organizers of the learning track[4]. In terms of problem size, we use problem distributions of the satisficing track of IPC 2011, which is the last edition in which the domains occurred together. Note that by a problem distribution we mean the values of the parameters passed to the

[3] https://github.com/pgomoluch/fd-learn

[4] http://www.cs.colostate.edu/~ipc2014/

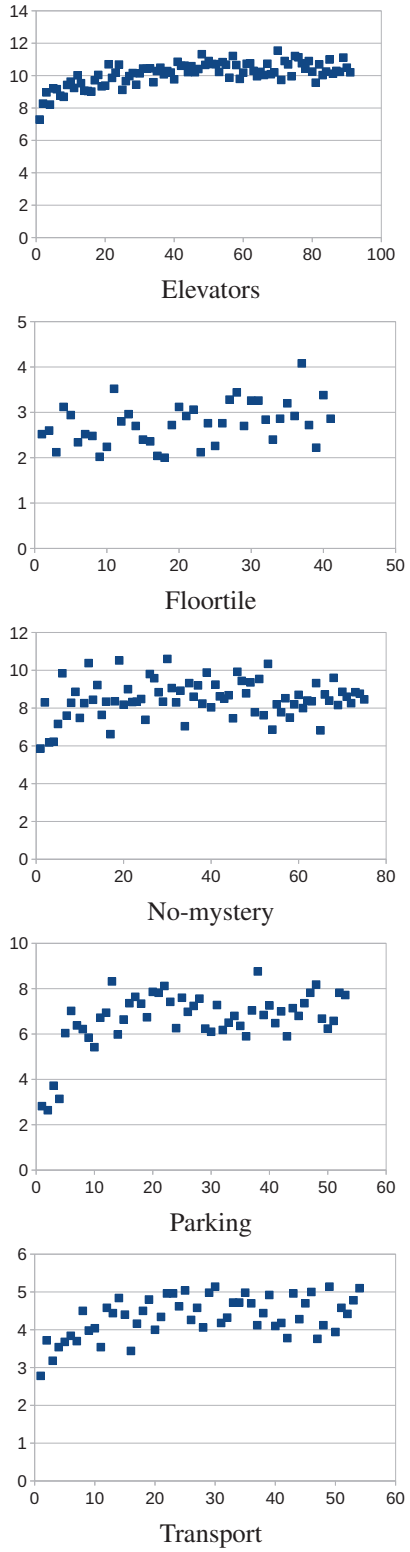

Figure 1: The average IPC score (y-axis) obtained on the training batch by the $n = 50$ policies sampled at each training iteration (x-axis).

problem generators, which do not necessarily give full account of the difficulty of the actual IPC 2011 problem sets. This is because the selection of the competition problems may have involved additional criteria not captured by the generator parameters, for example manual rejection of problems that have empirically proven too easy or too hard for some set of reference planners. For both training and evaluation, we use a time limit of 3 minutes per problem, which is lower than the 30 minutes traditionally used in IPC. The primary motivation for this is the training setting, which requires multiple planner runs at every iteration. We remark that training with a larger time limit is equally possible, although significantly more expensive, as it requires up to 10 times more computation time.

Training

We train a separate model for each of the domains. At every iteration, we randomly generate 20 problems of difficulty roughly corresponding to the problem sets of the satisficing track of IPC. Every batch of randomly generated problems preserves the generator parameters used for the competition set. For example, if the *Parking* set of the 2011 competition contains 20 problems including two problems with 22, three problems with 24 cars, three problems with 26 cars and so on, then the same holds for every of the training batches.

The training setup follows Algorithm 2, with the number of policies sampled at each iteration $n = 50$ and the the number of elite policies $m = 10$. The smoothing factor is $\alpha = 0.7$, as suggested by (Mannor, Rubinstein, and Gat 2003). For each of the domains, we train the model for 48 hours using 32 CPUs. Depending on the domain, this allows the training process to perform between 41 (*Floortile*) and 91 (*Elevators*) iterations. However, most of the policy improvement seems to happen in the first several iterations of the training process. This can be seen in Figure 1. The plots show the average IPC score obtained by the policies sampled at each iteration. Note that the policies themselves are not the only source of variability: the training problems are randomly sampled for each iteration and some of them turn out to be harder than others despite using the same generator parameters. Even with this caveat, the plots suggest that most of the progress is made in the early iterations. In the extreme case of *Floortile*, it is not clear from the plot whether the learner makes any progress at all. However, the results discussed in the next section suggest that a reasonably effective policy is still learned.

For the case of learning the search parameters directly, without the state-dependent policy, we use the same training setup. The only difference is that we limit the training time to 24 hours, which, in this simpler setting, has proven enough for the parameter distribution to converge.

Quantitative evaluation

We perform two separate evaluations of the trained policies. In the first evaluation, we use unseen problems generated in the same manner as the training ones. The motivation for testing on randomly generated problems is twofold. First, it allows for performing each of the 10 test runs on a different set of problems and, in doing so, allows us to assess

the planners over a total of 200 rather than just 20 problems per domain. Secondly, by removing any manual filtering of the problems, it allows us to observe the planners performance on problems unseen in training, but still exactly following the training distribution. In the second evaluation, we use the actual problem sets from the satisficing track of IPC 2011. In this case, for every domain and planner configuration, we perform 10 test runs on the problem set and average the scores.

We compare the neural search policies (denoted *NSP*) and the search parameters optimized directly (denoted *Opt*) against four fixed search routines making use of one of the search techniques available to the parametrized planner. For greedy best-first search, we use the Fast Downward's implementation. The other baselines are implemented by setting the parameters of our parametrized planner accordingly. Overall, the set of baseline planners includes:

- plain greedy BFS (*GBFS*);
- ϵ-greedy BFS with $\epsilon = 0.5$ ($R = 0$, $c = 0$) (*ϵ-greedy*);
- greedy BFS following node expansion with $R = 5$ random walks of length $L = 10$ on condition that the lowest known heuristic value h_{min} has not changed for last $S = 10$ expansions ($c = 0$) (*RW*);
- local BFS with a span of $C = 200$ expansions ($R = 0$, $c = 1$) (*Local*);
- a combination of all of the above, instantiating the parametrized planner template with $\epsilon = 0.5$, $S = 10$, $R = 5$, $L = 10$, $C = 200$ and $c = 0.5$ (*Mixed*).

The motivation behind the *mixed* configuration is to assess performance of an algorithm making a balanced use of all the available techniques. In this configuration, the planner interleaves between 100 expansions using the global open list and 100 expansions of local search. It follows the expansions with random walks if the search has not progressed for the last 10 expansions. A random node is selected for expansion instead of the one with lowest heuristic value with the probability of 0.5.

We remark that a direct comparison against our previous work (Gomoluch, Alrajeh, and Russo 2019) would be difficult due to its requirement to train on substantially smaller problems, with time limits of the order of 5 seconds.

The IPC scores obtained by each of the considered planners are reported in Table 1 (generated test sets) and Table 2 (IPC problem sets). The learning approaches score higher than any of the baselines in all cases except *No-mystery* set from IPC, where they narrowly trail plain greedy BFS. In general, in *No-mystery*, the performance of greedy and ϵ-greedy search and the learned policy is very similar. In *Transport* and *Parking* the baseline reaching performance closest to the learners is local search. Importantly, the *mixed* configuration obtains the lowest overall scores, clearly showing the advantage of a learned policy over a naive combination of all the search techniques.

Interestingly, the stateless variant of our approach sometimes performs better than the full state-dependent search policies. In particular, it achieves higher scores in the *Parking* and *Transport* domains. This seems surprising given the fact that the state-dependent policies are strictly more expressive. A possible explanation is that, given the same computational resources, the training process is able to better explore the significantly smaller parameter space (5 parameters in the stateless version as opposed to 104 in the neural search policy). On the other hand, the state-dependent policy performs much better in the *Elevators* domain, where it allows the planner to adopt a different approach depending on the problem size. For comparison, the stateless approach in the *Elevators* case resorts to GBFS and fails to match the performance of the GBFS baseline. On aggregate, the state-dependent policy outperforms the stateless variant on problems following the training distribution, but not on the problems sourced from IPC. We conjecture that this effect may be partially due to better generalization of the simpler model. In the following subsection, we investigate the effective behaviour of the learned policies in more detail.

	E	F	N	P	T	Sum
GBFS	14.67	2.24	8.18	9.24	2.6	36.93
ϵ-greedy	13.07	2.64	8.93	7.44	2.7	34.78
RW	14.63	0.47	6.78	7.95	3.6	33.42
Local	15.97	1.91	7.15	11.85	4.48	41.36
Mixed	11.6	1.25	6.69	6.14	2.9	28.58
Opt	14.64	**3.5**	8.86	**13.81**	**5.39**	46.18
NSP	**16.37**	3.28	**9.04**	12.93	5.12	**46.74**

Table 1: IPC scores for randomly generated test problems (average over 10 sets). *Elevators* (E), *Floortile* (F), *No-mystery* (N), *Parking* (P) and *Transport* (T).

	E	F	N	P	T	Sum
GBFS	16.11	4.13	**8.84**	8.08	0	37.16
ϵ-greedy	13.64	6	8.43	5.91	0	33.98
RW	14.7	2.34	6.58	6.57	0.48	30.66
Local	15.32	4.5	5.79	10.78	0.73	37.12
Mixed	10.93	2.86	7.49	5.4	0.43	27.11
Opt	15.62	6.2	8.78	**12.69**	**2.72**	**46.01**
NSP	**16.64**	**6.44**	8.56	11.37	1.01	44.02

Table 2: IPC scores for IPC problem sets (average of 10 runs). *Elevators* (E), *Floortile* (F), *No-mystery* (N), *Parking* (P) and *Transport* (T).

Qualitative evaluation

In the following, we examine the search policies learned for each of the domains. For the stateless variant, we report the values of the search parameters in Table 3. For the state-dependent policies, we observe the search parameters designated by the policies while solving selected validation problems, interpret the resulting search behaviour and relate it to the stateless case.

Elevators In this domain, the easiest problems are solved with GBFS ($c = 0$, $\epsilon \approx 0.01$). This is in line with the search parameters learned in the stateless case. A large number (> 15) of relatively short random walks (3 steps) is used when the search fails to progress for around 30 node expansions. In larger instances, the value of ϵ is even smaller

	ϵ	S	R	L	C	c	Remarks
Elevators	0.050	0	0	0	3	0.123	ϵ-GBFS with $\epsilon \sim 0.05$
Floortile	0.169	1	0	10	178	0.090	161 global and 17 local expansions, $\epsilon \sim 0.17$
No-mystery	0.349	1	2	1	24	0.067	22 global and 2 local expansions, $\epsilon \sim 0.35$
Parking	0.030	6	1	0	6	0.654	2 global and 4 local expansions, $\epsilon \sim 0.03$
Transport	0.086	5	2	1	33	0.965	$\sim$ local search with a span of 32 expansions

Table 3: The search parameters optimized for each of the problem distributions.

($\epsilon \approx 0.001$), the random walks get longer (9 steps) and there is more of them (> 30). This makes sense in the logistics-style domain of *Elevators*: random walks may increase the chance of finding a solution but also increase its cost. Long random walks are better avoided when the problem is likely to be solved in time without them. On the other hand, they are useful when the time limit becomes an issue. For large problems, a small number of local expansions is performed towards the end of the search (2 in a cycle of 50), but it is not clear whether this has substantial effect on the results.

Floortile In *Floortile*, the policy results mostly in ϵ-greedy BFS. It starts with a very high value of $\epsilon \approx 0.8$, which means that the heuristic value is ignored most of the time. This is not entirely surprising given the performance of baseline planners guided by the same heuristic. For the largest problems solved within the time limit, the value of ϵ gradually falls towards the end of the search. The policy performs better than the baselines, but still fails to solve most of the problems. In the stateless case, the learned value of ϵ is smaller (0.17), but results in quantitatively similar performance.

No-mystery The values of ϵ in *No-mystery* are even more extreme than in *Floortile*, ranging between 0.75 and 0.98 and increasing as h_{min} decreases. Neither random walks nor local search are used. On the one hand, the policy shows that a *delete relaxation* heuristic is of limited use. On the other hand, it also brings up a limitation of the parametrized search algorithm in its current form. While the value of the heuristic is virtually unused, it still needs to be computed for every encountered state, for the purpose of ordering the open list. We aim to address this issue in future work. As in *Floortile*, the ϵ found directly, without the state-dependent policy, is lower, at 0.35, but results in similar overall performance.

Parking In small *Parking* instances, the policy's behaviour is close to GBFS. Interestingly, this is achieved by setting the cycle length C to a very small value (a cycle with 0 or 1 local expansion is equivalent to global search). For larger instances, this approach is mixed with evenly balanced cycles of several global and local expansions. The ϵ values are typically around 0.1. Optimizing the parameters directly leads to more extensive use of local search (4 expansions in a cycle of 6), which allows the planner to solve, on average, 1 more problem of each batch of 20.

Transport On smaller *Transport* problems, the learned policy's behaviour is close to global ϵ-greedy BFS, with the value ϵ ranging between 0.08 and 0.2. This is achieved by using a very short cycle with no local expansions. In larger instances, the policy performs iterated local search of varying span. The number of expansions in a single cycle typically varies through the search, between the values of 10 and 150. Extensive use of local search makes intuitive sense in the logistics-style *Transport* domain, where focusing on a subset of the search space can often allow for faster progress. Occasionally, the search approaches global BFS by decreasing C to very low values. The value of ϵ increases as the search progresses, up to about 0.25. The stateless variant uses local search with a span of 32. This leads to similar performance on test problems following the training distribution (Table 1), but allows for solving 2 more problems of the IPC set (Table 2), which appears harder than the randomly generated problems of the same size.

8 Conclusion

We have introduced a parametrized search algorithm flexibly combining multiple search techniques in a way determined by the values of its parameters. We further constructed a simple neural policy model for designating the values of the algorithm's parameters given a high-level representation of the current state of the search. Using CEM, we trained the model on problems from five different planning domains. The empirical evaluation shows that the learners are able to discover effective distribution-specific search policies. We also considered a simpler setting, in which CEM is used directly to find fixed values for the search parameters. The simpler approach, learning the search parameters directly is also effective. However, it can not match the performance of state-dependent policies on problem sets where changing the search strategy is beneficial.

Besides the strengths of the approach, the empirical evaluation has also shown some of its limitations. In some cases, the state-dependent policies perform worse than the stateless approach, despite using a strictly more expressive model. This suggests that there is scope for improvement in how the parameters of the model are optimized. One possible approach is to apply quality-diversity optimization (Cully and Demiris 2017), so that each iteration of the training process considers a diverse set of possible solutions, not necessarily bound by a common Gaussian distribution.

Furthermore, while the current approach has the capacity to ignore the value of a heuristic it finds misleading, it can not replace it. One possible solution would be to make to choice of heuristic subject to the learning process, for example by allowing the algorithm to work with multiple open lists. It is also possible to extend the algorithm with other state-of-the-art techniques from classical planning, such as *preferred operators* or novelty-based search (Lipovetzky and Geffner 2017).

References

Asai, M., and Fukunaga, A. 2017. Exploration Among and Within Plateaus in Greedy Best-First Search. In *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling, ICAPS 2017*, 11–19.

Cenamor, I.; De La Rosa, T.; and Fernández, F. 2016. The IBaCoP planning system: Instance-based configured portfolios. *Journal of Artificial Intelligence Research* 56:657–691.

Chrabaszcz, P.; Loshchilov, I.; and Hutter, F. 2018. Back to basics: Benchmarking canonical evolution strategies for playing atari. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018*, 1419–1426.

Coles, A., and Smith, A. 2007. Marvin: A Heuristic Search Planner with Online Macro-Action Learning. *Journal of Artificial Intelligence Research* 28:119–156.

Conti, E.; Madhavan, V.; Such, F. P.; Lehman, J.; Stanley, K. O.; and Clune, J. 2018. Improving Exploration in Evolution Strategies for Deep Reinforcement Learning via a Population of Novelty-Seeking Agents. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018*, 5032–5043.

Cully, A., and Demiris, Y. 2017. Quality and diversity optimization: A unifying modular framework. *CoRR* abs/1708.09251.

de Boer, P.-T.; Kroese, D. P.; Manor, S.; and Rubinstein, R. Y. 2005. A Tutorial on the Cross-Entropy Method. *Annals of Operations Research* 134(1):19–67.

Domshlak, C.; Hoffmann, J.; and Katz, M. 2015. Red-black planning: A new systematic approach to partial delete relaxation. *Artificial Intelligence* 221:73–114.

Fikes, R. E.; Hart, P. E.; and Nilsson, N. J. 1972. Learning and executing generalized robot plans. *Artificial Intelligence* 3(1972):251–288.

Garrett, C. R.; Kaelbling, L. P.; and Lozano-Perez, T. 2016. Learning to Rank for Synthesizing Planning Heuristics. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016*, 3089–3095.

Gerevini, A.; Saetti, A.; and Vallati, M. 2009. An Automatically Configurable Portfolio-based Planner with Macro-actions: PbP. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS 2009*.

Gomoluch, P.; Alrajeh, D.; and Russo, A. 2019. Learning Classical Planning Strategies with Policy Gradient. In *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling, ICAPS 2019*, 637–645.

Hansen, N., and Ostermeier, A. 2001. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation* 9(2):159–195.

Helmert, M.; Röger, G.; and Karpas, E. 2011. Fast Downward Stone Soup: A Baseline for Building Planner Portfolios. *ICAPS 2011 Workshop on Planning and Learning* 28–35.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research* 26:191–246.

Hoffmann, J., and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research* 14:263–312.

Leckie, C., and Zukerman, I. 1998. Inductive learning of search control rules for planning. *Artificial Intelligence* 101:63–98.

Lipovetzky, N., and Geffner, H. 2017. Best-First Width Search: Exploration and Exploitation in Classical Planning. *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence* 3590–3596.

Mannor, S.; Rubinstein, R.; and Gat, Y. 2003. The Cross Entropy Method for Fast Policy Search. In *ICML'03 Proceedings of the Twentieth International Conference on International Conference on Machine Learning*, 512–519.

Richter, S., and Westphal, M. 2010. The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *Journal of Artificial Intelligence Research* 39:127–177.

Rubinstein, R. 1999. The Cross-Entropy Method for Combinatorial and Continuous Optimization. *Methodology and Computing in Applied Probability* 1:127–190.

Salimans, T.; Ho, J.; Chen, X.; and Sutskever, I. 2017. Evolution strategies as a scalable alternative to reinforcement learning. *CoRR* abs/1703.03864.

Valenzano, R.; Sturtevant, N. R.; Schaeffer, J.; and Xie, F. 2014. A Comparison of Knowledge-Based GBFS Enhancements and Knowledge-Free Exploration. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling, ICAPS 2014*.

Virseda, J.; Borrajo, D.; and Alcazar, V. 2013. Learning heuristic functions for cost-based planning. *Preprints of the ICAPS'13 PAL Workshop on Planning and Learning* 6–13.

Wierstra, D.; Schaul, T.; Glasmachers, T.; Sun, Y.; Peters, J.; and Schmidhuber, J. 2014. Natural evolution strategies. *J. Mach. Learn. Res.* 15(1):949–980.

Xie, F.; Müller, M.; and Holte, R. 2014. Adding Local Exploration to Greedy Best-First Search in Satisficing Planning. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, 2388–2394.

Yoon, S.; Fern, A.; and Givan, R. 2008. Learning Control Knowledge for Forward Search Planning. *The Journal of Machine Learning Research* 9:683–718.