Experience-Biased Symbolic Planning

Aram Ebtekar and **Mike Phillips** and **Maxim Likhachev**
Carnegie Mellon University
5000 Forbes Ave,
Pittsburgh, PA 15213

Sven Koenig
University of Southern California
Los Angeles, CA 90089

Abstract

Weighted A* search is the foundation for state-of-the-art domain-independent symbolic planners. An important open challenge for these planners is to utilize experience with similar planning problems to speed up subsequent planning episodes. Experience graphs have recently shown promise as a technique for plan reuse in the context of weighted A* searches in robotics, by biasing them towards a subgraph of the search space, typically chosen to consist of the edges used in previous plans. In this paper, we demonstrate how to augment symbolic planners with experience graphs, using the HSP2 planner as a representative example. The resulting planners are able to trade off seamlessly between plan quality and runtime and thus have advantages over past plan-reuse techniques, such as planning by analogy. We demonstrate experimentally that our planner speeds up HSP2 by a factor of 2 or more on many domains while achieving about the same plan quality.

Introduction

Planning entails finding action sequences that achieve a goal state. In order to apply the language of graph theory, states are often thought of as nodes connected by action transitions. A **plan** is a path from the start state to one of many goal states. Despite the existence of classic polynomial-time search techniques, such as Dijkstra's algorithm, planning remains the crux of many difficult problems in AI.

The reason usually is that the graphs involved are very large; indeed, too large for explicit storage. In motion planning, the state space may be a continuous geometric set whose desired discretization is very fine. In symbolic planning, a planning language such as STRIPS, SAS+, ADL or PDDL [CITE] is chosen to give a compressed symbolic representation.

The graph is exponentially larger than its encoding so, in both cases, states are generated only as needed. The symbolic representation sometimes encodes exploitable structure. In order to create a domain-independent solver for a language such as STRIPS, we need to automatically extract and interpret structural information about the problem.

In forward weighted A* search, a frontier of nodes expands from the start until it hits a goal, at which point a solution is found. Rather than blindly searching the entire exponential-sized space, A* tries to focus toward the goal by expanding nodes which seem likely to lie on a short solution path. The minimum length of a solution passing through a given frontier node is the sum of distances from the start to this node, and from this node to a goal. The better we estimate these, the sooner we'll reach a solution.

Since A* expands in a fairly conservative fashion from the start state onward, the start-to-frontier estimates are very well-approximated by construction of a valid path. Thus, the principal issue is the derivation of frontier-to-goal estimates. Given exact distances, it would be a simple matter to trace a valid path from the start, taking optimal successors at each step, until a goal is reached. In practice, we are forced to use estimates, which sometimes mislead us into local optima, wasting time on dead-end nodes.

These distance approximations are called **heuristics**, and we use them in hopes of finding good plans more efficiently. Algorithms such as A* are designed to find solutions quickly whenever the heuristic is of reasonable quality. If we hope to take subexponential time, we can only afford to examine a vanishing fraction of the states, and it is only for these that we compute the heuristic. If the heuristic is good, we hope not to have to visit (i.e. generate) too many states. Hence, there is a tradeoff between spending more time to compute a more accurate heuristic, vs evaluating a weaker heuristic quickly but at many more states.

In practice, even using state-of-the-art methods to generate heuristics on isolated planning instances, STRIPS problems can be very difficult to solve without expert domain knowledge. Even humans have difficulty when faced with an unfamiliar kind of puzzle, though we get better with experience. Thus, one might hope that a planning agent would similarly learn to generalize solutions from past planning experiences to related new instances.

A recent approach builds **experience graphs** to estimate the high-level connectivity of the free space in motion planning tasks (Phillips et al. 2012). The intuition is to remember previously generated paths so that, when a new start and goals are queried, a new path can be quickly generated by reusing subpaths from the E-graph. A variant of weighted A* is employed which biases its focus toward the E-graph.

While this work demonstrates promising results in robot motion planning, E-graphs have never before been applied to symbolic domain representations such as the STRIPS language.

Our present contribution is to investigate the use of E-graphs in STRIPS domains. In particular, we take the HSP2 planner (Bonet and Geffner 2001) as a representative of the class of modern weighted A* based symbolic planners, and augment its heuristic with a bias toward its E-graph. After presenting the related works, we begin by describing the STRIPS language and the original HSP heuristic. In the following section, we formally introduce E-graphs in technical detail. Then we combine the two approaches, resulting in the new Experience-Biased HSP planner. We present the algorithm and discuss its theoretical properties. This is followed by experiments showing promise for the application of E-graphs to symbolic problem solving. Finally, we conclude with some thoughts on extensions worth investigating.

Related Work

Alot of symbolic planners that have emerged in recent years are based on forward weighted A* search. This paper focuses on HSP2 as a representative example, because it is fairly simple and embodies the core ideas common to many state-of-the-art planners. The use of E-graphs is orthogonal to many of the latest improvements, so our methods can in principle be adapted to any A* planner.

Fast Downward (Helmert 2006) is a planner that decomposes planning tasks using a contruct called the causal graph heuristic. The LAMA planner (Richter and Westphal 2010), which won the International Planning Competition 2008, replaces the causal graph with landmark heuristics.

While many of these planners perform preprocessing in an attempt to automatically extract knowledge about the domain, none are capable of using their own fully generated plans to inform later searches. A few works were made in similar directions, for example (Fikes, Hart, and Nilsson 1972) are able to generalize their plans, (Veloso 1992) is able to relate by analogy to known solutions, and (Borrajo and Veloso 2012) probabilistically chooses whether to step along past plans or explore new paths.

STRIPS Language

A STRIPS problem is a tuple $P = \langle A, O, I, G\rangle$ consisting of an atom set A, operator set O, initial state $I \subseteq A$ and goal condition $G \subseteq A$. Each operator $op \in O$ is defined by its cost, preconditions, add effects and delete effects: $Cost(op) \in \mathbb{R}^+$ and $Prec(op), Add(op), Del(op) \subseteq A$.

The problem P defines a directed state graph (V, c) where V is the power set of A (i.e. states $S \in V$ correspond to collections of atoms), and the edge costs are

$$c(S, S') = \min\{Cost(op) \mid S \supseteq Prec(op) \text{ and } S' = (S \setminus Del(op)) \cup Add(op)\}$$

By default, $c(S, S') = \infty$ when there is no operator directly transitioning from S to S'. Given a STRIPS problem P, we seek a low-cost path from the initial state $I \in V$ to any of the goal states $S \in V$ such that $S \supseteq G$. In this paper, we are particularly concerned with sequences of problems in which A and O are fixed, but various pairs $\langle I, G\rangle$ are queried.

HSP Heuristic

Many of today's state-of-the-art domain-independent STRIPS planners are based on the HSP2 planner (Bonet and Geffner 2001), which we now describe. It's common to compute heuristics by solving an easier, relaxed version of the original problem. In STRIPS, one might ignore the delete lists of operations. Since having more atoms makes preconditions and the goal condition more likely to hold, this can only make the problem easier, so the resulting heuristic never overestimates the true cost.

However, even the relaxed STRIPS planning problem includes set-cover as a special case, making it NP-hard. Intuitively, we see that the relaxed search space remains exponential. To reduce it, HSP2 decouples the atoms, instead estimating the cost of achieving each individual atom.

Let's say we want a heuristic estimate of the distance from S to a goal that contains G. HSP2 estimates the cost to achieve an atom $a \in A$ from S by $g_S(a) =$

$$\begin{cases} 0 & \text{if } a \in S \\ \min_{op|a \in Add(op)} \left(g_S(Prec(op)) + Cost(op)\right) & \text{if } a \notin S \end{cases}$$

The HSP heuristic (also used in HSP2) is then defined by

$$h^{HSP}(S) = g_S(G),$$

the estimated cost of achieving all atoms in G. Note that $g_S(\cdot)$ is evaluated on certain atom sets, namely $Prec(op)$ and G. To keep the computations feasible, define $g_S(P) = \max_{p \in P} g_S(p)$ for atom sets P. The resulting HSP-max heuristic is an underestimate; indeed, it satisfies a stronger property called **consistency**.

Definition 1. *For $\epsilon \geq 1$, a heuristic h is called ϵ-**consistent** if $h(S) = 0$ when S is a goal and $h(S) \leq \epsilon c(S, S') + h(S')$ for all $S, S' \in V$. h is **consistent** if it is 1-consistent.*

Theorem 1. *Given an ϵ-consistent heuristic and expanding no node more than once, A* is guaranteed to find a path costing no more than ϵ times the optimal path cost. [CITE ARA*]*

If the atoms were truly independent, a more accurate estimate would be $g_S(P) = \sum_{p \in P} g_S(p)$. Despite its inconsistency in general, the latter HSP-plus heuristic is often useful in practice, as it uses more information and biases more greedily toward the goal.

From a computational perspective, the HSP heuristic must compute $g_S(a)$ for all $a \in A$ whenever a new state S is generated. This can be done by dynamic programming, iterating the recursive formula to a fixpoint in Bellman-Ford-like fashion. Pseudocode is listed in Algorithm 1.

In a sense, a lot of work is wasted by estimating the cost to reach every atom, when we only require $g_S(G)$. Later, we'll see how to make better use of this computation.

Experience Graphs

E-graphs can improve total search time over a series of related queries on the same graph. Formally, an E-graph is a

Algorithm 1 ComputeG(S)

```
for all a ∈ A do
  if a ∈ S then
    g_S(a) ← 0
  else
    g_S(a) ← ∞
  end if
end for
repeat
  for all op ∈ O do
    cost_estimate ← g_S(Prec(op)) + Cost(op)
    for all a ∈ Add(op) do
      g_S(a) ← min(g_S(a), cost_estimate)
    end for
  end for
until no further change in g_S-values
```

subgraph of the state space. We update it between queries, typically by adding the solution path from the previous search query.

Suppose we have a consistent heuristic $h(S, S')$ for the distance between arbitrary pairs of states. That is, h obeys the triangle inequality, is never dominated by an edge cost, and $h(S, S) = 0$ for all $S \in V$. The E-graph heuristic h^E biases the search to follow E-Graph edges instead of h by penalizing the latter by an inflation factor $\epsilon^E > 1$. To be precise, define

$$h^E(S_0) = \min_{N, S_N, \pi} \sum_{i=1}^{N} \min\{\epsilon^E h(S_{i-1}, S_i), c^E(S_{i-1}, S_i)\}$$

with the minimization taking place over all goal states S_N and paths $\pi = \langle S_0, S_1, ..., S_N \rangle$. c^E are E-graph edge costs, or ∞ if the E-graph edge does not exist.

In the limit as $\epsilon^E \to 1$, E-graph edges offer no advantage as $h(S_{i-1}, S_i) \leq c^E(S_{i-1}, S_i)$. Hence, using the triangle inequality to coalesce the sum,

$$h^E(S_0) = \min_{N, S_N, \pi} \sum_{i=1}^{N} h(S_{i-1}, S_i) = \min_{S_N} h(S_0, S_N)$$

Conversely, as $\epsilon^E \to \infty$, the inflated terms dominate. Hence, the minimizing path becomes the one which incurs the least estimated cost outside the E-graph. In particular, h^E has the following property:

For sufficiently large ϵ^E, once the A frontier touches an E-graph component which includes a goal state, no node outside the E-graph will ever again need to be expanded.*

This means, for instance, that if the E-graph already contains a path from the start to a goal, then A*, guided by h^E with large bias parameter ϵ^E, will never leave the E-graph. If the E-graph is small, a solution would promptly be discovered. In this manner, little work is needed to solve previously encountered subproblems.

We cannot afford to compute h^E according to its literal definition, as there are far too many paths to consider. Fortunately, there exists a practical means of computing it. By the triangle inequality, any consecutive pair of $h(S_{i-1}, S_i)$ terms can be merged into one. Thus, we lose no generality in restricting S_i to lie on the E-graph for $0 < i < N$. The heuristic computation can focus on the E-graph, using $h(S_{i-1}, S_i)$ to estimate the cost of leaving the E-graph at S_{i-1} before re-entering it at S_i.

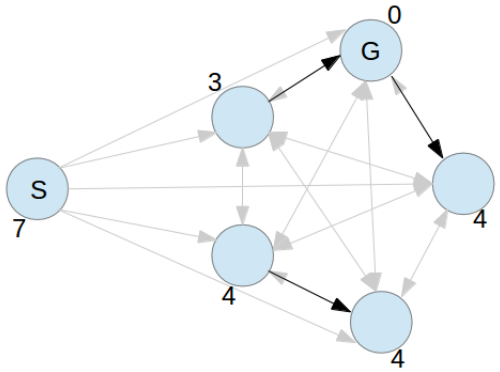
Let V^E be the union of the goal states and the E-graph's vertices. In addition to following E-graph edges, we imagine it's possible to "jump" from S to S' at cost $\epsilon^E h(S, S')$. That is, let $c'(S, S') = \min\left(c^E(S, S'),\ \epsilon^E h(S, S')\right)$

In a preprocessing stage before the main search, we apply Dijkstra's algorithm once in reverse with the costs c' to compute the estimated distance-to-goal $h^E(S)$ from every $S \in V^E$. Ignoring the computation of h (presently considered as a blackbox), this preprocessing takes $O(|V^E|^2)$ time, which is insignificant for small E-graphs and goal sets.

Later, when generating $S \notin V^E$, we see that we have already precomputed costs of paths consisting of all but the first edge of π according to the definition of $h^E(S)$. Thus, we derive the mathematically equivalent but much cheaper computation

$$h^E(S) = \min_{S' \in V^E} \left(\epsilon^E h(S, S') + h^E(S')\right)$$

Theorem 2. *h^E is ϵ^E-consistent. (Phillips et al. 2012)*



The above figure illustrates the idea. Suppose the dark edges have cost 3 and make up the E-graph. Off-E-graph edges are not shown, as they are only accessed indirectly via h; instead, light edges represent the jump estimates $h(S_i, S_j) = 2$ (with one exception: $h(S, G) = 5$), which we inflate by a factor $\epsilon^E = 2$ to make 4 (or 10, respectively). Dijkstra's algorithm follows dark and light edges backward to compute the h^E values shown alongside each node.

G is on the E-graph, though this need not be the case in general. V^E is the pentagon on the right, and its h^E values are precomputed before the search. For $S \notin V^E$, $h^E(S)$ is computed only when A* generates S.

Experience-Biased HSP

Now we combine the ideas from HSP2 and E-graphs to construct a domain-independent STRIPS planner which learns from experience. In STRIPS, a consistent heuristic $h(S, S')$ is given by the HSP-max estimate $g_S(S')$. We can compute $g_S(S')$ for all $S, S' \in V^E$ in precisely $|V^E|$ runs of Algorithm 1: each run provides us with $|V^E|$ of these results, yielding $(|V^E|)^2$ values in total.

As before, Dijkstra's algorithm combines an inflation of these estimates with the E-graph edges to derive $h^E(S)$ for all $S \in V^E$. Note that in STRIPS, V^E need not include every goal state: the minimal goal G suffices as it's always the easiest to reach in the heuristic relaxation.

Upon encountering a new state $S \notin V^E$, we run Algorithm 1 again to compute $g_S(a)$ for all atoms $a \in A$. Then it's a simple matter to compute the E-graph heuristic by

$$h^E(S) = \min_{S' \in V^E} \left(\epsilon^E g_S(S') + h^E(S') \right)$$

We remark that, when E-graphs are involved, the values $g_S(S')$ on the right-hand side range over a variety of different states S'. Compare this to the formula for h^{HSP}, which uses only $g_S(G)$. HSP2 would still compute all the g_S-values despite using them only once, so here we get to consult the E-graph essentially for free. The only additional work is the minimization over $S' \in V^E$, which takes O($|V^E||A|$) time. A pseudocode implementation is listed in Algorithm 2. The parameter ϵ^E controls bias toward the E-graph, while the weight ϵ provides a uniform goal-directed bias.

Algorithm 2 Search()

```
for all S ∈ V^E do
    ComputeG(S)
    for all S' ∈ V^E do
        c'(S, S') ← min ( c^E(S, S'), ε^E g_S(S') )
    end for
end for
Compute h^E on V^E by reverse Dijkstra from G with c'.
Run A* on the full graph from I with heuristic ε h^E:
for S ∉ V^E when generated by A* do
    ComputeG(S)
    h^E(S) ← min_{S' ∈ V^E} ( ε^E g_S(S') + h^E(S') )
end for
if A* successfully found a path to some goal state S ⊇ G
then
    Add all edges of the solution path to the E-graph.
    return solution path
else
    return failure
end if
```

ComputeG(S) implements the dynamic programming computation of g_S-values. Each iteration of the outermost loop permanently fixes at least one $g_S(a)$ value, so it does at most $|A|+1$ iterations. Multiplying nested loop iterations together and dropping cardinality signs for brevity, the runtime bound on Algorithm 1 is $O(\mathbf{A^2O})$. Thus, each state generated costs $O(\mathbf{A^2O} + \mathbf{V^E A})$ time.

The preprocessing cost is $O(\mathbf{V^E A^2 O} + (\mathbf{V^E})^\mathbf{2}\mathbf{A})$, asymptotically equivalent to preemptively generating every state in V^E. In fact, most of the preprocessing requires no knowledge of G, so it can be done in advance of the new planning query. Once the query $\langle I, G \rangle$ is given, the remaining work consists of completing the $c'(S, G)$ computations from $g_S(a), a \in G$, and then running Dijkstra's algorithm from G. This costs only $O(\mathbf{V^E}(\mathbf{A} + \mathbf{V^E}))$ time.

For comparison, standard HSP2 without E-graphs takes $O(\mathbf{A^2O})$ time (with essentially the same constant factors) to generate each state, and does not incur the cost of generating a state unless the A* frontier encounters it. When HSP2 is augmented with small E-graphs, the same asymptotic bound continues to hold, provided that the number of E-graph vertices is limited to $O(\mathbf{AO})$. Furthermore, we retain completeness and the ability to configure suboptimality bounds by adjusting the bias parameters:

Theorem 3. *Provided there exists a path from initial state I to a goal state $S \supseteq G$, the experience-biased HSP algorithm finds a path which is at most $\epsilon\epsilon^E$-suboptimal.*

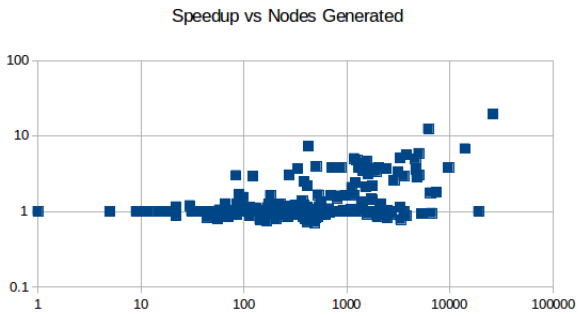
This follows directly from Theorems 1 and 2, since the weighted heuristic ϵh^E is $\epsilon\epsilon^E$-consistent. However, E-graphs make no guarantee of improving the search time. Indeed, biasing toward bad regions could trap the planner inside large local optima. Potential gains depend entirely on choosing a good small set of edges to form our E-graph. In our experiments, we simply keep the previous solution path; different choices remain a topic for future investigations.

Finally, we remark that the Experience-Biased HSP algorithm is highly parallelizable, even if A* is retricted to expanding nodes sequentially. Recall that the sum, min and max functions are computable in logarithmic time using linearly many processors. Therefore, with minor adjustments, ComputeG(S) has parallel depth $O(\mathbf{A} \log \mathbf{A} \log \mathbf{O})$. With $\mathbf{A} \max(\mathbf{O}, \mathbf{V^E})$ processors, the total cost of generating a state is $O((\mathbf{A} \log \mathbf{O} + \log \mathbf{V^E}) \log \mathbf{A})$. Using even more processors, the cost of *expanding* a state can be reduced to the same by generating its successors in parallel. If still more processors are available, some may dedicate to the task of preemptively computing h^E for states that are likely to be generated in the future, e.g. near the A* frontier.

Experimental Setup

We augmented the HSP2 planner code (Bonet and Geffner 2001) which won [AI COMPETITION IN 2000] with E-graphs. Our aim is to observe the effect of E-graphs on planning time given that we've already solved a closely related problem. To measure this, we run three searches for each problem instance. First, we run the original problem using standard HSP2. Then, we generate a modified problem by moving the start and goal nodes randomly for X steps each. The second search solves the modified problem in the standard way. Finally, the third search solves the modified problem using as E-graph a random Y% of the path obtained from the first search. The reason for saving only a portion of the previous experience is to simulate the effects of disconnected E-graphs, as might arise after multiple searches. Our main comparisons will be between the number of nodes generated in the second and third searches. In practice, the times followed similar ratios to the number of nodes generated; as one would expect from the analyis in [PREVIOUS SECTION] since the E-graph resulting from Y% of a single path is very small.

Experimental Results



The plot shows speedup (i.e. ratio of nodes generated in the second and third searches) against a measure of problem size (i.e. nodes generated in the second search). Planning times followed similar trends as node generation counts, as one would expect from the analyis since the E-graph resulting from a half-path is very small. We see considerable speedups in many domains, while the plan costs stayed roughly the same (in fact, they became slightly shorter on average). Many domains saw no discernable effect, but none were significantly hurt by E-graphs.

Domain	0 steps	10 steps	20 steps
BlocksWord	0	0	0
Grid	0	0	0
Satellite	0	0	0

Analysis of HSPr (section to be deleted)

Whether or not E-graphs are used, the call to ComputeG(S) at each node generation is quite substantial. For this reason, a variant of HSP, called HSPr, searches the regression space of STRIPS, where each node represents a "subgoal". The search proceeds backward from the goal. In this case, the heuristic must estimate the cost to achieve a given state S (considered as a subgoal) from the initial state I. A single preprocessing call to ComputeG(I) allows each $h^{HSPr}(S) = g_I(S)$ to be computed in $O(\mathbf{A})$ time.

With E-graphs, in backward as well as forward search, we call ComputeG(S) for all $S \in V^E$; only now, V^E includes I instead of G. From I, a forward Dijkstra yields $h^E(S)$ for all $S \in V^E$. Then for $S \notin V^E$, the E-graph heuristic estimate of the cost from I to S is given by

$$h^E(S) = \min_{S' \in V^E} \left(h^E(S') + \epsilon^E g_{S'}(S) \right)$$

Since $g_{S'}$ was precomputed, the cost of generating a state is reduced to just $O(\mathbf{V^E A})$. Nonetheless, the overhead of using E-graphs is much more substantial in backward than in forward search, contributing a $|V^E|$ factor on the state generation cost.

Since a major benefit of backward HSP is being able to generate states in $O(\mathbf{A})$ time, it appears E-graphs in their present form are less likely to be of much help. However, in domains where forward search is more practical, we saw that E-graphs can effectively provide free hints.

Conclusions

We saw that by reusing pieces from past plans, planning time can often be reduced. Many interesting directions remain for future investigation. The E-graph cannot be allowed to grow without bound over countless planning episodes, so a better scheme is needed for expansion and pruning as the agent learns.

Distinct states in STRIPS can actually be very similar, differing only by a few irrelevant atoms; hence, experiences should be transferable to inexact matches.

What if graph changes? Anytime incremental...

Another use of E-graphs: if something goes wrong along the path, we can replan using the E-graph to get a faster and less disruptive result.

References

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1):5–33.

Borrajo, D., and Veloso, M. 2012. Probabilistically reusing plans in deterministic planning. In *Proc. of ICAPS-12 Workshop on on Heuristics and Search for Domain-Independent Planning*, 17–25.

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

Helmert, M. 2006. The fast downward planning system. *J. Artif. Intell. Res.(JAIR)* 26:191–246.

Phillips, M.; Cohen, B. J.; Chitta, S.; and Likhachev, M. 2012. E-graphs: Bootstrapping planning with experience gfvraphs. In *Robotics: Science and Systems*.

Richter, S., and Westphal, M. 2010. The lama planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research* 39(1):127–177.

Veloso, M. M. 1992. Learning by analogical reasoning in general problem solving. Technical report, DTIC Document.