Experience-Biased Planning in STRIPS

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

Abstract

Experience graphs provide a means to augment existing heuristics with a bias toward using a select set of edges, typically taken from solutions to past queries. Thus, a planner can learn to bias its search toward well-understood regions and reuse parts of old plans. We show how to extend the HSP STRIPS planner with E-graphs at minimal cost, and in doing so speed up subsequent planning episodes across multiple domains.

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 start to goal. 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, too large to store explicitly. By either recording newly observed states or using a compressed representation of the graph's structure, states may be generated dynamically as the search progresses. In motion planning, the space may be a continuous geometric set which is dynamically discretized.

In high-level logical planning, a domain language such as STRIPS, SAS+, ADL or PDDL encode a graph whose size may be exponential in its description length. These short descriptions sometimes encode exploitable structure. In order to create a domain-independent solver for a language such as STRIPS, we need to automatically extract and interpret information about the problem.

We may view planning as a process that generates distance-to-goal labels for each node of a graph. Given these distance labels, it's a simple matter to generate a plan by tracing steps from the goal, successively taking the optimal predecessor. Since there is no worst-case polynomial-time algorithm that can solve the shortest-paths problem in STRIPS (recall the graph is exponential-sized), we may try to roughly approximate the distance labels. The approximate labels we call **heuristics**, and we use them in hopes of finding plans more efficiently. If the heuristic precisely reflects the true distances, we saw how to immediately find a path. Algorithms such as A* are designed to find solutions quickly whenever the heuristic is of reasonable quality. Note that in practice, any subexponential algorithm can only afford to look at a small fraction of all states, so it is only for these that we compute the heuristic. If the heuristic is good, we hope not to have to look at (i.e. generate) too many states. Hence, there is a tradeoff between spending more time to compute a tighter heuristic, vs evaluating a weaker heuristic more quickly but at many more states.

In practice, even with a domain-independent means of generating heuristics, STRIPS problems can be very difficult. 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. The intuition is to remember previously generated paths so that, when a new start and goal is queried, a new path can be more quickly generated by reusing subpaths from the E-graph. However, E-graphs have never before been applied to high-level representations such as STRIPS. This is our present contribution.

In section [???], we present the HSP heuristic for STRIPS. Then, we discuss E-graphs in more technical detail. This paper's novel contributions follow thereafter, where we extend E-graphs to STRIPS planning by using it to augment the HSP heuristic. The resulting A* search is complete with a configurable suboptimality bound. Finally, we present experiment results in which this use of E-graphs is demonstrated to achieve considerable speedups over plain HSP in several planning domains.

Related Work

HSP

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 and its precondition, add and delete lists: $Cost(op) \in \mathbb{R}^+$ 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

$$c(S, S') = \inf\{Cost(op) \mid S \subseteq Prec(op) \text{ and } S' = S - Del(op) \cup Add(op)\}$$

Naturally, $c(S, S') = \infty$ 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 $S \supset G$.

Many of today's state-of-the-art domain-independent STRIPS planners are based on the HSP planner, 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 is admissible.

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, we decouple the atoms, instead estimating the cost of achieving each individual atom.

Let's say we want to compute a distance-to-G heuristic from a state S. Estimate the cost to achieve an atom $a \in A$ from S by $g_S(a) =$

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

Define the HSP-max heuristic by

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

the estimated cost of achieving all atoms in G. In order to cheaply compute $g_S(G)$ and $g_S(Prec(op))$, we define $g_S(P) = \max_{p \in P} g_S(p)$ for sets of atoms P. This is admissible and consistent. If the atoms were truly independent, a much tighter estimate would be $g_S(P) = \sum_{p \in P} g_S(p)$. Despite its inadmissibility 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. In a sense, it seems we are wasting a lot of work by estimating the cost to reach every atom when we only require $g_S(G)$. Later, we'll see how to make fuller use of this computation.

Experience Graphs

E-graphs can improve total search time over a series of related queries on the same graph. Between queries, we update the set of edges which constitute the E-graph, typically by adding the solution path from the previous search. In cases where the heuristic compution is done by searching on a state space that was reduced by projection, the E-graph is considered as a subgraph of the reduced space.

Suppose we have a consistent heuristic $h(S, S')$ for the distance between arbitrary pairs of states. That is, h obeys the triangle inequality with respect to edge costs, and $h(G) = 0$ whenever G is a goal. In order to derive a related heuristic which biases in favor of using the E-Graph, we inflate the heuristic by a factor $\epsilon^E > 1$, but allow it to use uninflated "shortcuts" on the E-graph. To be precise, the E-graph heuristic is defined by

$$h^E(S) = \min_\pi \sum_i \min\{\epsilon^E h(S_i, S_{i+1}), c^E(S_i, S_{i+1})\}$$

over all paths $\pi = \langle S_0, S_1, ..., S_N \rangle$ with $S_0 = S$ and S_N being any goal. c^E are E-graph costs, or ∞ if the E-graph edge doesn't exist.

In the limit as $\epsilon^E \to 1$, taking "shortcuts" along the E-graph loses its advantage, so

$$h^E(S) = \min_\pi \sum_i h(S_i, S_{i+1}) = \min_\pi h(S_0, S_N) = h(S, G)$$

by the triangle inequality, where G is the minimizing goal.

Conversely as $\epsilon^E \to \infty$, the minimizing path becomes the one which uses the fewest edges outside the E-graph, and the cost remains finite iff S is in an E-graph component which also contains a goal state.

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 pair of consecutive $h(S_i, S_{i+1})$ terms can be combined into one. Thus for $0 < i < N$, we only need be concerned with states S_i lying on the E-graph.

Let V^E consist of all goal states plus all endpoints of the E-graph's edges. In a preprocessing stage before the main search, we can apply Dijkstra's algorithm once in reverse to compute the estimated distance-to-goal $h^E(S)$ from every $S \in V^E$, optimizing over combinations of E-graph edges and "jumps" of cost $\epsilon^E h(S_i, S_{i+1})$. If the blackbox heuristic h is cheap to compute, this preprocessing runs in $O(|V^E|^2)$ time, which is negligible for small E-graphs.

Later, when generating $S \notin V^E$, we see that we have already precomputed all but the first "leg" of the path in the definition of $h^E(S)$. Thus, we derive the mathematically equivalent but much cheaper computation

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

h^E is ϵ^E-consistent [cite].

Algorithm Discussion and Analysis

Now we combine the ideas from HSP and E-graphs to construct a domain-independent STRIPS planner which learns from experience. In STRIPS, the consistent heuristic $h(S, S')$ is given by the HSP-max estimate $h_S^{HSP}(S')$. We can compute $h_S^{HSP}(S')$ for all $S, S' \in V^E$ in precisely $|V^E|$ runs of the dynamic programming we saw before.

Thus, our preprocessing work is roughly equivalent to that of generating $|V^E|$ states, which will be small. 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 dynamic programming procedure 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

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

Here we remark that, when E-graphs are involved, the values $g_S(S')$ appear on the right-hand side for multiple states S', not just $g_S(G)$. Since HSP would be computing all the g-values anyway, we get to consult the E-graph essentially for free. The only additional work here is the minimization over $S' \in V^E$, which is negligible for small E-graphs. A pseudocode implementation is listed in Algorithms 1 and 2.

Algorithm 1 ComputeG(S)

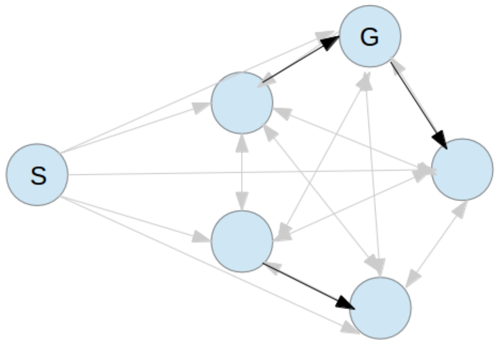
```
for all $a \in A$ do
  if $a \in S$ then
    $g_S(a) \leftarrow 0$
  else
    $g_S(a) \leftarrow \infty$
  end if
end for
repeat
  for all $op \in O$ do
    $pCost \leftarrow g_S(Prec(op))$
    for all $a \in Add(op)$ do
      $g_S(a) \leftarrow \min(g_S(a),\ pCost + Cost(op))$
    end for
  end for
until no further change in $g$-values
```

Algorithm 2 Search()

```
Let $c(S, S')$ for $S, S' \in V^E$ represent E-graph edge costs.
for all $S \in V^E$ do
  ComputeG(S)
  for all $S' \in V^E$ do
    $c'(S, S') \leftarrow \min(c(S, S'),\ \epsilon^E g_S(S'))$
  end for
end for
Compute $h^E$ on $V^E$ by reverse Dijkstra from $G$ with $c'$.
Do $\epsilon$-weighted A* from $I$ on full graph with heuristic $h^E$:
for $S \notin V^E$ when generated do
  ComputeG(S)
  $h^E(S) \leftarrow \min_{S' \in V^E} (\epsilon^E g_S(S') + h^E(S'))$
end for
if A* successfully found a path to some goal state $S \supseteq G$
then
  Add all edges of the solution path to the E-graph.
  return solution path
else
  return failure
end if
```

ComputeG implements the dynamic programming computation of g-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 everything together and ignoring cardinality signs for brevity, the runtime bound is $O(\mathbf{A^2O})$. Thus, each state generated costs $O(\mathbf{A^2O} + \mathbf{V^E})$ time, and the preprocessing cost is asymptotically equivalent to preemptively generating every state in V^E.

By comparison, standard HSP without E-graphs takes $O(\mathbf{A^2O})$ time (with essentially the same constant factors) to generate a state, and does not incur the cost of generating a state unless the A* encounters it. The E-graph search maintains completeness and $\epsilon\epsilon^E$-suboptimality bounds on the path found. However, it makes no guarantee of improving the search time. Potential gains depend entirely on choosing a good small set of edges to call our E-graph. In our experiments we simply keep the previous solution path, but different choices remain a topic of investigation for future research.



Experimental Setup

We augmented the HSP planner code [CITE] 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 HSP. 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 returned 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

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

Conclusions and Extensions

Generalization from experiences

Alternative E-graph edge-selection schemes (complete graph on V^E?)

What if graph changes? Anytime incremental...