Using Available Memory to Transform Graphplan's Search

Terry Zimmerman & Subbarao Kambhampati
Department of Computer Science and Engineering
Arizona State University, Tempe AZ 85287-5406
{zim,rao}@asu.edu

Abstract

We present a major variant of the Graphplan algorithm that employs available memory to transform the depth-first nature of Graphplan's search into an iterative state space view in which heuristics can be used to traverse the search space. When the planner, PEGG, is set to conduct exhaustive search, it produces guaranteed optimal parallel plans 2 to 90 times faster than a version of Graphplan enhanced with CSP speedup methods. By heuristically pruning this search space PEGG produces plans comparable to Graphplan's in makespan, at speeds approaching state-of-the-art heuristic serial planners.

1 Motivation and Approach

Despite the recent dominance of heuristic state-search planners over Graphplan-style planners, the Graphplan approach [Blum and Furst 1997] is still one of the most effective ways to generate so-called "optimal parallel plans". While state-space planners are drowned by the exponential branching factors of the search space of parallel plans, Graphplan excels due to the way it combines an IDA* style iterative search [Bonet and Geffner, 1999] with a highly efficient CSP-based, incremental generation of valid action subsets. We present here a system called PEGG, that addresses weaknesses in Graphplan's approach by employing available memory to: 1) reduce the redundant search Graphplan conducts in consecutive iterations, and 2) more importantly, to transform Graphplan's IDA* search into iterative expansion of a select set of states that can be traversed any order.

A shortfall of the IDA*'s approach to search is the fact that it regenerates many of the same nodes in each of its iterations. This can be traced to using too little memory in many cases; the only information carried over from one iteration to the next is the upper bound on the f-value. Given that consecutive iterations of search overlap significantly, we investigated using additional memory to store a trace of the explored search tree to avoid repeated re-generation of search nodes. With a representation of the explored search space, we can transform the way this space is extended during the next iteration. In particular, we can (a) expand search trace nodes in the order of their heuristic merit and (b) we may also consider iteratively expanding a select set of states. This strategy is too costly for normal IDA* search, but Graphplan's type of IDA* search is particularly well-suited to these changes as the kth level planning graph provides a compact way of representing the search space traversed by the corresponding IDA* search in its kth iteration. The state space view provided by the search trace allows us to transform Graphplan's search from its depth-first default to a more informed traversal of the space.

2 Design and Experiments

As would be expected for IDA* search there is great similarity (redundancy) in the search space for successive search episodes as the plan graph is extended. In fact, the search conducted at any level k+1 of the graph is essentially a replay of the search conducted at the previous level k with certain well-defined extensions. Specifically, *every* set of subgoals reached in the backward search of episode n, starting at level k, will be generated again by Graphplan in episode n+1 starting at level k+1.

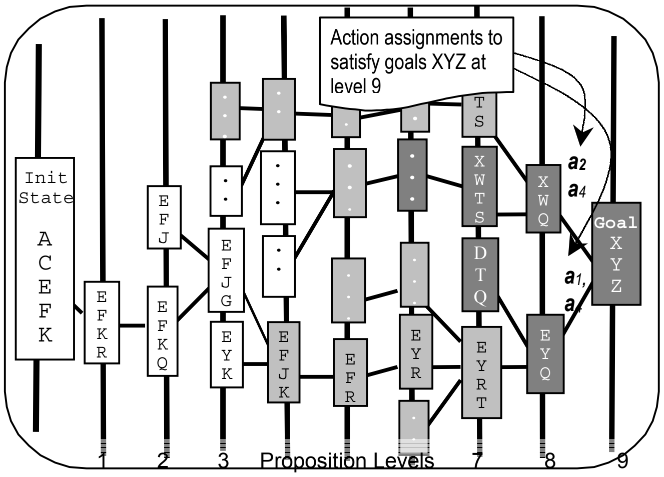

***Figure 1**. State space view of Graphplan's search space: 3 consecutive search episodes leading to a solution*

Figure 1 depicts the state space tree structure corresponding to Graphplan's search over three consecutive search iterations on a hypothetical problem. The dark shaded states are first produced during Graphplan's attempt to satisfy the XYZ goal at level 7. They are generated again in the next episode, each at one level higher, along with some new states (shown in lighter shade). Finally, in the third episode Graphplan regenerates the dark and lighter shaded states, attempting to satisfy XYZ at level 9, and finds a solution.

EGBG [Zimmerman, Kambhampati, 1999] used memory to aggressively record the experience in each search episode, essentially avoiding all redundant effort. That approach confronted memory constraints on larger problems, but it suggests a more powerful use for a pared-down search trace: exploiting the snapshot view of the entire search space of a Graphplan iteration to focus on the most promising areas. This transformation frees us from the depth-first nature of Graphplan's search, permitting movement about the search space to focus on its most promising sections first -or even exclusively.

A summary of PEGG (for details; [Zimmerman and Kambhampati, 2003]) relies on these definitions: **Search segment**: a state generated during Graphplan's regression search from the goal state, indexed to a specific plan graph level. It holds the state's goal list, a pointer to the parent search segment, and the actions assigned in satisfying the parent's goals. **Search trace (ST):** the linked set of search segments (states) representing the search space visited in a Graphplan backward search episode. The sets of states in each of the three shadings of Figure 1 can be seen as the ST after each of the three episodes. **Transposition:** The extant trace of search segments (states) after search episode *n* is *transposed* up one planning graph level for episode *n+1* as follows: For each ST search segment associated with graph level *j* associate it with level *j+1* for episode *n+1*. **Visiting** a search segment: the goals of segment S_p are memo checked at their associated level and if valid, PEGG initiates Graphplan's CSP-style search to satisfy them. The process is enhanced by a cadre of efficiency techniques such as a bi-level plan graph, domain preprocessing, explanation based learning (EBL), dependency directed backtracking (DDB), and goal & action ordering. Whenever the goals of S_p are validly assigned, a child segment is created containing S_p's goals regressed over the assigned actions, and linked to S_p (thus extending the ST).

The PEGG algorithm: the graph is built until the problem goals appear non-mutex, and the first regression search episode is conducted ala Graphplan fashion. During this search, the initial trace is constructed, concisely capturing all 'states' generated during the search process. If no solution is found, the ST directs the search process for future iterations. This search is 2-phased: select a promising ST state, then Graphplan's depth-first, CSP-type search on the state's subgoals is conducted. Another ST search segment is heuristically selected if search fails. Since the ST provides a state space view, PEGG can use 'distance based' heuristics (c.f. HSP-R [Bonet and Geffner, 1999] and AltAlt [Nguyen and Kambhampati, 2000]). For results reported here, the 'adjusted sum' heuristic from the latter is used.

Table 1 compares PEGG against standard Graphplan and a highly enhanced version (GP-e), which has been augmented with the efficiency methods mentioned above. Two PEGG modes of operation are reported; 1) *so-PEGG*: make-span optimal, ST search segments ordered according to a state space heuristic, *all* segments visited, 2) *PEGG*: Ordering the ST search segments as for 1, beam search on only the heuristically 'best' fraction. The first approach maintains Graphplan's guarantee of step optimality while the latter sacrifices the guarantee of optimality in favor of pruning search in *all* search episodes *and* bounds the size of the search trace that is maintained in memory. Empirically we find that optimal make-span plans are generally found by PEGG *regardless*, (one exception shown in bold) and speedups as high as two orders of magnitude over enhanced Graphplan are achieved.

Problem	**Graphplan** cpu sec (steps/acts) **Stnd.**	**GP-e**	**so-PEGG** cpu sec (steps/acts)	**PEGG** cpu sec (steps/acts)	**Speedup** (PEGG vs. GP-e)
bw-large-B	194.8	13.4 (18/ 18)	12.2	3.1 (18/ 18)	4.3x
bw-large-D	~	~ *(36/ 36)*	~	340 **(38 / 38)**	> 5x
att-log-b	~	~	~	120 (11/ 79)	>15x
gripper-15	~	~ *(36/ 45)*	47.5	16.7 (36/ 45)	>107x
gripper-20	~	~ *(40/ 59)*	~	44.8 (40/ 59)	> 40x
tower-9	~	~ *(511/ 511)*	118	23.6 (511/ 511)	> 76x
TSP-12	~	~ *(12/ 12)*	7.2	6.5 (12/ 12)	>277x
AIPS '98, '00, '02 Competition Problems					
gripper-x-4	~	190 (19/ 29)	73.9	30.9 (19/ 29)	6.1x
gripper-x-5	~	~	512	110 (23/ 35)	> 16x
log-y-4	~	470 (11/ 60)	366	330 (11/ 58)	1.4x
blocks-10-1	~	95.4 (34/ 34)	18.7	11.0 (34/ 34)	8.7x
blocks-16-2	~	~ *(54/ 54)*	~	58.7 (54/ 54)	> 31 x
logistics-10-0	~	30.0 (15/ 56)	21	8.9 (15/ 55)	3.4x
logistics-12-1	~	~	1101 (15/75)	119 (15/ 75)	>15.1x
freecell-2-1		98.0 (6/ 10)	80.1	62.9	1.5x
depot-6512	239	5.1	5.0	2.1	2.4x
depot-1212	~	~ *(22/ 55)*	~	127 (22/ 56)	> 14x
driverlog-2-3-6	~	27.5 (7/ 20)	1.9	1.9 (7/ 20)	14.5x
driverlog-4-4-8	~	~	~	589 (10/ 24)	> 3x
ztravel-3-7a	~	~	1434 (10/23)	222 (10/ 21)	> 8x
ztravel-3-8a	~	972 (7/ 25)	11.2	2.3 (7/ 25)	423x

Table 1. PEGG vs. Graphplan and enhanced Graphplan (GP-e)
GP-e: enhanced Graphplan (see text) *so-PEGG*: step-optimal, search via the ST. *PEGG*: beam search on best 20% of search segments in ST
~ indicates failure in 30 min limit, cpu time
Parentheses next to cpu time give # of steps/ # of actions in solution.
Allegro Lisp, runtimes (excl. gc time) on Pentium 900 mhz, 384 MB RAM

References

Blum A. and Furst M.1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90(1-2).1997.

Bonet, B. and Geffner, H. 1999. Planning as heuristic search: New results. In *Proceedings of ECP-99,* 1999.

Nguyen, X. and Kambhampati, S. 2000. Extracting effective and admissible state space heuristics from the planning graph. In *Proceedings of. AAAI-2000.*

Zimmerman, T. and Kambhampati, S. 1999. Exploiting Symmetry in the Planning-graph via Explanation-Guided Search. In *Proceedings of AAAI-99,* 1999.

Zimmerman, T. and Kambhampati, S. 2003. Using memory to transform search on the planning graph. ASU Technical Report (available at http://rakaposhi.eas.asu.edu/pegg-tr.pdf)