International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:2, No:6, 2008

Genetic-Based Planning with Recursive Subgoals

Han Yu, Dan C. Marinescu, Annie S. Wu, and Howard Jay Siegel

***Abstract*—**In this paper, we introduce an effective strategy for subgoal division and ordering based upon recursive subgoals and combine this strategy with a genetic-based planning approach. This strategy can be applied to domains with conjunctive goals. The main idea is to recursively decompose a goal into a set of serializable subgoals and to specify a strict ordering among the subgoals. Empirical results show that the recursive subgoal strategy reduces the size of the search space and improves the quality of solutions to planning problems.

***Keywords*—**Planning, recursive subgoals, Sliding-tile puzzle, subgoal interaction, genetic algorithms.

I. Introduction

PLANNING is an artificial intelligence (AI) problem with a wide range of real-world applications. Given an initial state, a goal specification, and a set of operators, the objective of planning is to construct a valid sequence of operators, or a plan, to reach a state that satisfies the goal specifications starting from the initial state of a system.

Much effort has been devoted to building computational models for a variety of planning systems. Our work is based on STRIPS-like domains [7] in which the change of system state is given by the operators and their preconditions and postconditions. In addition, we are interested in the linear planning problem where solutions are represented by a total order of operators that must be executed sequentially to reach the goal.

Definition 1: A Planning problem is a four-tuple

$$\prod = \{P, O, I, G\}.$$

P is a finite set of ground atomic conditions (i.e., elementary conditions instantiated by constants) used to define the system state. $O = \{o_i\}$, where $1 \leq i \leq |O|$, is a finite set of operators that can change the system state. Each operator has three attributes: a set of preconditions o_i^{pre}, a set of postconditions o_i^{post}, and a cost $C(o_i)$. o_i^{post} consists of two disjunctive subsets: o_i^{post+} and o_i^{post-}. o_i^{post+}, called the add list, is a set of conditions that must be true for a system state after the execution of the operator; o_i^{post-}, called the delete list, consists of a set of all conditions that do not hold after the execution of the operator. $I \subseteq P$ is the initial state and $G \subseteq P$ is the set of goal conditions. A plan Δ contains a finite sequence of operators. An operator may occur more than once in a plan. *An operator is valid* if and only if its preconditions are a subset of the current system state. A plan Δ solves an instance of $\prod$ if and only if every operator in Δ is valid and the result of applying these operators leads a system from state I to a state that satisfies all the conditions in G.

Planning is generally more difficult than a typical search problem not only because it involves an extremely large search space but also because the existence of solutions is not guaranteed. In addition, the size of an optimal solution cannot be easily estimated. As a result, it is difficult to quantify the time and space complexity of planning algorithms.

This paper presents a planning strategy called recursive subgoals for problems with conjunctive goals (i.e., the goal of problems can be expressed as the conjunction of multiple goals). The main idea of this strategy is to decompose the goals recursively into a sequence of subgoals so that reaching one subgoal reduces a planning problem to the same problem but at a smaller scale. We give a formal definition of recursive subgoals and incorporate this strategy with a genetic-based planning algorithm. Experiments on the Sliding-tile puzzle show that this strategy is able to significantly improve the performance of planning algorithms to problems in which recursive subgoals maintain the subgoal serializability.

II. Subgoal Ordering and Interaction

Korf presents a detailed study on the interaction of subgoals for a planning problem with conjunctive goals [11]. He classifies three different types of interactions between subgoals: independent subgoals, serializable subgoals, and non-serializable subgoals. If a set of subgoals is independent, reaching any arbitrary subgoal does not affect the difficulty of reaching the rest of the subgoals. Problems with independent subgoals are easy to solve because we can reach the problem goal by approaching every subgoal individually. As a result, the cost of the search is the total amount of cost devoted to every individual subgoal. This type of interaction, however, rarely occurs in planning problems. In some planning problems, it is possible to specify an ordering of the subgoals that have the following property: every subgoal can be reached without violating any subgoal conditions that have been met previously during the search. Such subgoals are called *serializable subgoals*. The search becomes easier if we are able to recognize this type of subgoal correlation and specify a serializable ordering. On the other hand, if such an ordering does not exist among the subgoals, the subgoals are

Han Yu, Dan C. Marinescu, and Annie S. Wu are with the School of Computer Science in the University of Central Florida, P. O. Box 162362, Orlando, FL 32816-2362 USA (corresponding author: Han Yu. phone: 407-823-5602; fax: 407-823-5419; e-mail: {hyu, dcm, aswu}@ cs.ucf.edu).

Howard Jay Siegel is with Department of Electrical and Computer Engineering and Department of Computer Science in Colorado State University, Fort Collins, CO 80523-1373 USA (e-mail: hj@colostate.edu).

called *non-serializable subgoals*. There is no universal method of dividing and ordering subgoals into serializable subgoals. In addition, proving the serializability of a sequence of subgoals is as difficult as proving the existence of solutions for a planning problem [11]. Therefore, Korf's classification of subgoal interactions is not appropriate for predicting the difficulty of a planning problem.

Barrett and Weld [2, 3] extend the classification of serializable subgoals based on the probability of generating a sequence of serializable subgoals from a randomly ordered set of subgoals. They define *trivially serializable subgoals* for those subgoals that are always serializable given any possible sequences. If a set of subgoals is not trivially serializable, violation of previously met goal conditions might occur during the search for the complete solution. As the cost of backtracking the previous subgoals is exponentially high, a planning problem is tractable only if the probability of a random sequence of subgoals being non-serializable is sufficiently low so that the cost for backtracking does not dominate the average cost of the algorithm. Otherwise, a planning problem is intractable. These subgoals are called *laboriously serializable subgoals*.

A correct ordering among subgoals is critical for the performance of planning algorithms. Thus, the study of subgoal correlations has drawn the attention of the planning community. One school of thought attempts to pre-process the control knowledge gained from the specifications of operators and goals to construct a total order on a group of subgoals, before the search begins [4, 6, 10, 15]. A second approach includes online ordering methods that focus on detecting and resolving goal condition conflicts from an existing partially ordered plan [5, 8].

III. Planning with Recursive Subgoals

In this paper, we introduce a strategy of dividing planning goals into a sequence of serializable subgoals. Informally, our strategy is to decompose a planning problem recursively into a set of subgoals and then to define a strict ordering of these subgoals.

A. State Space Graph

We begin our formal description of recursive subgoals with the introduction of the state space graph of a planning problem.

Definition 2: Let $S = \{s_1, s_2, \ldots\}$ be a set of all possible states of a planning system. Let $O = \{o_1, o_2, \ldots\}$ be a set of operators defined for a planning problem. The goal of a planning problem can be represented by G as a set of atomic conditions (see also **Definition 1** in Section 1).

Definition 3: The state space of a planning problem can be represented by a directed graph $GR = \{V, E, f_e, s_{init}, S_{goal}, f_s, f_o\}$, where

1. $V = \{v_1, v_2, \ldots\}$, a set of vertices.
2. $E = \{e_1, e_2, \ldots\}$, a set of directed edges.
3. Every edge e_i connects a pair of vertices $\{v_j, v_k\}$, where v_j and v_k are source and destination vertices of an edge, respectively. $f_e: E \rightarrow V$ is a function that maps an edge to its source and destination vertices.
4. s_{init} is the initial state of a planning problem. $s_{init} \in S$.
5. S_{goal} is the set of all system states that meet every condition in G. $S_{goal} \subseteq S$.
6. f_s: $V \rightarrow S$ is a function that maps every vertex v_i in V to a distinct system state s_i that can be reached from the initial state s_{init}. $f_s(v_i) = s_i$. $f_s(V) \subseteq S$. A planning problem is solvable if $S_{goal} \cap f_s(V) \neq \phi$. For the rest of the notation in Section III, we assume that a planning problem is solvable.
7. Edges represent the transitions between two system states in $f_s(V)$. $f_o: E \rightarrow O$ is a function that maps every edge e_i in E to an operator o_i. This function does not enforce a one-to-one mapping, i.e., $\exists\, i$ and j, where $i \neq j$ and $f_o(e_i) = f_o(e_j)$.

B. Subgoals

Definition 4: Let $GOAL = \{g_1, g_2, \ldots, g_n\}$ be a set of subgoals defined for a planning problem. Any subgoal g_i of a planning problem can be represented by P_i as a set of atomic conditions with the following four properties:

1. $P_i \subseteq G$. Subgoals are easier to reach than the goal of a problem because the conditions for subgoals are subsets of the conditions for the problem goal.
2. $G = \bigcup P_i$, $1 \leq i \leq n$. The problem goal can be reached when we reach a state that meets the conditions for all the subgoals.
3. Let $f_{gs}: GOAL \rightarrow S$ be a function mapping a subgoal g_i to a set of all states that can be reached from s_{init} and meet the conditions for g_i. Clearly, $S_{goal} \subseteq f_{gs}(g_i) \subseteq f_s(V)$. If $P_i = \phi$, $f_{gs}(g_i) = f_s(V)$; if $P_i = G$, $f_{gs}(g_i) = S_{goal}$.
4. Let GR_i be the state space graph that consists of all states in $f_{gs}(g_i)$ and transitions between the states. GR_i is a subgraph of GR.

C. Serializable Subgoals

According to Korf [11], a set of subgoals is serializable if a specific ordering among them exists. Although an optimal solution is not guaranteed to be found, this ordering ensures that a problem is always solvable by following the sequence of the subgoals without ever violating any previously reached subgoals. We use this definition and give a formal definition of serializable subgoals based on the state space graph of a planning problem.

Definition 5: A set of subgoals in $GOAL$ is serializable if it has the following properties:

1. $GOAL$ contains an ordered list of subgoals. g_1 is the first subgoal and g_n is the last subgoal. The search for a solution follows the order of the subgoals.
2. $P_n = G$ and $f_{gs}(g_n) = S_{goal}$. That is, the set of conditions for the last subgoal is the same as the goal of the problem. If the last subgoal is reached, the problem is solved.
3. $P_1 \subseteq P_2 \subseteq \ldots \subseteq P_{n-1} \subseteq P_n$. That is, the set of conditions for a subgoal is a subset of the conditions for all subsequent subgoals.
4. $f_{gs}(g_n) \subseteq f_{gs}(g_{n-1}) \subseteq \ldots \subseteq f_{gs}(g_2) \subseteq f_{gs}(g_1)$. That is, the set of all states that satisfy the conditions for a subgoal is a

subset of all states that satisfy the conditions for every preceding subgoal. This property indicates that the state space of a search algorithm can be reduced after reaching intermediate subgoals.

5. Let $GR_i = \{V_i, E_i, f_i, s_{init}, S_{goal}, f_s, f_o\}$ be the state space graph of subgoal i, $V_n \subseteq V_{n-1} \subseteq V_{n-2} \ldots \subseteq V_1 \subseteq V$. As a result, GR_i is a subgraph of GR_j, for every i and j, where $1 \leq j \leq i \leq n$.

6. Define $Adjacent(v_i, v_j, GR) = true$ if there exists an edge in G that connects v_j from v_i. Define $Connect(v_i, v_j, GR) = true$ if $Adjacent(v_i, v_j, GR) = true$ or, $\exists\, v_k$, $Connect(v_i, v_k, GR) = true$ and $Adjacent(v_k, v_j, GR) = true$. In other words, $Connect(v_i, v_j, GR) = true$ if and only if there is a sequence of edges that connects vertex v_j from v_i.

If a sequence of subgoals is serializable, a graph GR_i that corresponds to any subgoal g_i has the following property: for any $v_j \in V_i$, $\exists\, v_k \in V_{i+1}$, $Connect(v_j, v_k, GR_i) = true$. That is, every state that meets the conditions of subgoal g_i can reach at least one state within the state space of subgoal g_{i+1} without violating the conditions set for subgoal g_i. Therefore, serializable subgoals ensure that a solution can be found if it exists.

D. Recursive Subgoals

The recursive subgoal strategy offers a simple and effective solution to the formation and ordering of subgoals from a single goal. This strategy divides the goal of a planning problem recursively into a sequence of subgoals. These subgoals, which will be shown by an example in Section V, have the following property: reaching one subgoal results in a reduction of a problem to the same problem at a smaller scale. A formal definition of recursive subgoals is given below.

Definition 6: A sequence of subgoals is recursive if it meets the following condition:
Let PR be a set of the same problems of different scales. $PR = \{PR_1, PR_2, \ldots, PR_m\}$. PR_i is smaller than $PR_{i'}$, if $i < i'$. Then reaching subgoal g_j in PR_i and reaching subgoal g_{j+1} in PR_{i+1} are essentially the same problem for $1 \leq j \leq i < m$. Let $GR_{i,j}$ be the state space graph corresponding to subgoal g_j of PR_i. Then $GR_{i,j} \cong GR_{i+1,j+1}$; i.e., $GR_{i,j}$ and $GR_{i+1,j+1}$ are isomorphic.

The division of recursive subgoals does not guarantee serializability among subgoals. We consider three different scenarios as to the applicability of this approach.

1. If a solution exists in any configuration of problems (i.e., any given initial and goal states for a problem) at any scale, the division of recursive subgoals always preserves the subgoal serializability. An example of a domain belonging to this category is the Tower of Hanoi [1], in which any two configurations are reachable from each other.

2. If a solution does not always exist in any configuration of a problem at any scale, but reaching one recursive subgoal never leads a problem at a smaller scale to an unsolvable configuration, we can still preserve the subgoal serializability on this problem. We show in Section V that the Sliding-tile puzzle falls into this category.

3. Recursive subgoals are non-serializable if we cannot avoid the situation of backtracking any previous recursive goals during the search for a complete solution.

IV. The Recursive GA-Based Planning Algorithm

The recursive planning heuristic is incorporated into the genetic-based planning algorithm. This algorithm differs from the traditional GA approaches in two aspects. First, operators are encoded as floating-point numbers to eliminate invalid operators in a plan. Second, the search process is divided into multiple phases, with each phase an independent GA run. Thus, we can build the solutions incrementally by combining the solutions found in each individual phase. In addition, the fitness of a solution is evaluated with two independent aspects: the goal fitness evaluates the quality of a plan (how well the plan reaches goal specifications); the cost fitness evaluates the efficiency of a plan. A detail description of this planning algorithm can be found in [17].

If the goal of a planning problem is divided into recursive subgoals, we can apply a multi-phase GA to search for solutions to reach every subgoal. The number of necessary phases to reach a subgoal depends on the difficulty of subgoals. Only when a subgoal is reached in a phase can GA proceed to search for the next subgoal in subsequent phases. The final solution is the concatenation of the solutions to all subgoals that have been attempted in a single GA run. The following pseudo code illustrates the search procedure of this algorithm.

(1) Start GA. Initialize population.
(2) Set the first subgoal of the problem as the current search goal.
(3) While the specified number of phases are not finished and the final goal is not reached, do
  (a) While the specified number of generations for a phase are not finished, do
    (i) Evaluate each individual in the population.
    (ii) Select individuals for the next generation.
    (iii) Perform crossover and mutation.
    (iv) Replace old population with new population.
  (b) Select the best solution for this phase and keep it.
  (c) If the current subgoal is reached, set the next subgoal as the current search goal.
  (d) Randomly initialize population and start the next phase. The search starts from the final state of the best solution in the previous phase.
(4) Construct the final solution by concatenating the best solutions from all phases.

V. Case Study: the Sliding-tile Puzzle

Sliding-tile puzzles consist of a number of moving blocks and a board on which the blocks can slide. Such problems are sometimes used in AI textbooks to illustrate heuristic search methods. For example, Russell and Norvig [16] discuss the 4×4 Sliding-tile puzzle shown in Fig. 1.

Given an initial configuration, say the one in Fig. 1(a), the aim is to reach the goal configuration in Fig. 1(b) by sliding the blocks without lifting them from the board. Solutions do not exist for every possible combination of initial and goal configurations. Johnson and Story show that a solution exists only when the initial configuration is an even permutation of the goal configuration [9].

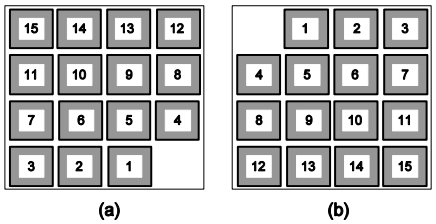

Fig. 1 The initial and goal configurations of a 4×4 Sliding-tile puzzle (a) The initial configuration. (b) The goal configuration

There have been some important studies on applying domain-specific knowledge to the Sliding-tile puzzles. Korf and Taylor introduce several search heuristics [14] that are useful for defining an accurate admissible heuristic function in the *IDA** search algorithm. The heuristics used include the linear conflict heuristic, last moves heuristic, and corner-tile heuristic. These heuristics are shown to improve the search performance of the *IDA** search algorithm. The computational costs of their algorithm, however, seem to be heavily dependent on the initial state of the problem. The number of nodes that are generated during a search can differ as many as 100 times for the same size but different configurations of the problem. The execution time on the 5×5 Sliding-tile puzzles can be as long as three months. Korf and Felner [13] use a disjoint pattern database heuristic in some planning domains including the Sliding-tile puzzles. With this heuristic, the subgoals are first split into disjoint subsets such that an operator affects only the subgoals in one subset. The values obtained for each subset are then combined to form the result of the heuristic evaluation function. Their approach is guaranteed to find optimal solutions and has been applied successfully to different instances of 5×5 Sliding-tile puzzles. The results indicate that this heuristic improves the search efficiency by decreasing the number of nodes traversed during the search. Nevertheless, the computational cost of this approach still increases very quickly with the increase in problem size. Problems larger than the 5×5 puzzle were not tested due to the high computational cost [12].

Fig. 2 shows one approach to create recursive subgoals for solving a 4×4 Sliding-tile puzzle. The first subgoal is to have the tiles located in the fourth row and fourth column in their desired positions, see Fig. 2(a). After the first subgoal is reached, the problem is reduced to a 3×3 Sliding-tile puzzle. Then we work on the second subgoal: moving the remaining tiles in the third row and third column to the correct positions, shown in Fig. 2(b). After the second subgoal is reached, the problem is reduced to a 2×2 Sliding-tile puzzle, which is very easy to solve. The puzzle is solved after the third subgoal is reached, as shown in Fig. 2(c).

Johnson and Story also show that if we move any tiles in the Sliding-tile puzzle, we can always maintain the parity of the permutation between the current configuration and the goal configuration [9]. If in the original problem the initial configuration is an even permutation of the goal configuration (i.e., the original problem is solvable), after reaching one recursive subgoal we can always find an even permutation between the current configuration and the goal configuration in the reduced problem. Hence, the reduced problem is solvable as long as the original one is solvable. The goal serializability is preserved in the Sliding-tile puzzle because we are able to reach a subgoal without moving the tiles that have been set in place in previous subgoals.

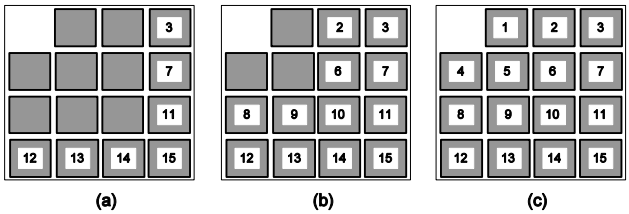

Fig. 2 The steps for solving a 4×4 Sliding-tile puzzle using the recursive subgoal strategy. (a) The first subgoal. (b) The second subgoal. (c) The third subgoal

The recursive strategy can be applied to any possible configuration of a Sliding-tile puzzle. In a goal configuration, the empty tile can be located at any position. If the empty tile is already in one of the corners, we choose those tiles in the row and column that are farthest to that corner to be in the first subgoal. If the empty tile is not in a corner, we first move it to the nearest corner. The number of moves depends on how far a tile is from the nearest corner. In a $n \times n$ Sliding-tile puzzle, if n is odd, at most n-1 moves are needed; if n is even, at most n-2 moves are needed. After the relocation of the empty tile, the new configuration replaces the original one as the goal configuration of the problem. As every operator in the Sliding-tile puzzle is reversible, a reversed sequence of the operators that move the empty tile to the corner will lead the system from the new goal configuration to the original one. The final solution is the solution to the new goal configuration appended by this reversed sequence of operators. Fig. 3(a) and Fig. 3(b) show an example of changing the goal configuration in a 4×4 Sliding-tile puzzle. In our experiments, the empty tile is always in top-left corner in the goal configuration.

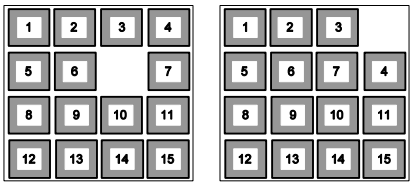

Fig. 3 An example showing the reconfiguration of problem goals for the recursive subgoal strategy (a) The original goal configuration. (b) The new goal configuration in which the empty tile is moved to the nearest corner

VI. Experimental Results

In this section, we test our strategy on the $n \times n$ Sliding-tile puzzle discussed in Section V. We evaluate the effectiveness of the recursive subgoal strategy by comparing the performance of the genetic-based planning approach with and without the subgoal strategy incorporated (also called single goal approach). Table I shows the parameters for this experiment.

TABLE I
PARAMETER SETTINGS USED IN THE EXPERIMENT

Parameter	Value
Population Size	200
Crossover Rate	0.9
Mutation Rate	0.01
Selection Scheme	Tournament
Tournament Size	2
Number of Generations in Each Phase	100

In the single-goal approach, the goal fitness is evaluated with the Manhattan distance of all n^2 -1 tiles between the final state of the plan and the goal configuration. The smaller the distance, the higher the goal fitness. In the recursive subgoal approach, we decompose the $n \times n$ Sliding-tile puzzle into n-1 subgoals, $\{g_1, g_2, \ldots, g_{n-1}\}$. After the first subgoal is reached, the problem is reduced to a $(n - 1) \times (n - 1)$ Sliding-tile puzzle. For every subgoal g_i, we focus on the $2 \times (n - i) + 1$ tiles that need to be moved to the correct positions. The goal fitness is evaluated as the Manhattan distance between the final state and the goal configuration for the $2 \times (n - i) + 1$ tiles.

We test both the recursive subgoal strategy and the single-goal approach on 4×4, 5×5, 6×6, 7×7, and 8×8 Sliding-tile puzzles. For each problem size, we run both approaches 50 times. In a 4×4 problem, each run has up to 15 phases. We double the number of phases each time the problem size increases by one scale, but use the same population size of 200 for all problem sizes.

The experimental results show that the single-goal approach finds solutions in 10 out of 50 runs on the 4×4 sliding-tile problem and none for any larger problems. Table II shows the number of runs that allow us to reach every subgoal for experiments where the recursive subgoal strategy is incorporated. The recursive subgoal strategy significantly improves the search performance. It finds solutions to the 4×4 Sliding-tile puzzle in 34 out of 50 runs and the performance even improves as the problem size increases because more phases are allowed for all subgoals. Table III reports the average number of phases needed to reach each subgoal from those runs that find a valid solution. The result indicates that reaching a subgoal does not make the subsequent subgoals more difficult. We observe that the number of phases needed to reach subgoal g_i is very close to the number of phases needed to reach subgoal g_{i+1} in the next larger problem.

TABLE II
THE NUMBER OF RUNS OUT OF 50 RUNS THAT THE RECURSIVE SUBGOAL STRATEGY CAN REACH EACH SUBGOAL G_1 - G_7

Problem Size	4×4	5×5	6×6	7×7	8×8
g_1	44	50	50	50	50
g_2	37	50	50	50	50
g_3	35	50	49	50	50
g_4	N.A.	50	49	50	50
g_5	N.A.	N.A.	49	50	50
g_6	N.A.	N.A.	N.A.	50	50
g_7	N.A.	N.A.	N.A.	N.A.	50

TABLE III
THE AVERAGE NUMBER OF PHASES THAT THE RECURSIVE SUBGOAL STRATEGY NEEDS TO REACH A SUBGOAL FROM ITS PREVIOUS SUBGOAL

Problem Size	4×4	5×5	6×6	7×7	8×8
g_1	6.86	9.34	18.50	28.56	40.74
From g_1 to g_2	1.36	5.02	8.32	16.14	23.00
From g_2 to g_3	1.07	2.34	5.65	8.74	12.96
From g_3 to g_4	N.A.	1.00	2.12	5.34	10.68
From g_4 to g_5	N.A.	N.A.	1.00	2.70	5.64
From g_5 to g_6	N.A.	N.A.	N.A.	1.00	2.32
From g_6 to g_7	N.A.	N.A.	N.A.	N.A.	1.00

Next, we study the effect of the parameters on the performance of the approach. We use the parameter settings in Table I as the baseline settings and vary the population size, the crossover rate, and the mutation rate separately. We keep the other parameters the same as the baseline settings while varying each of the above parameters. We test problem sizes from 4×4 to 8×8 Sliding-tile puzzles, run each test case 50 times, and calculate the number of successful runs (i.e., the runs that find valid solutions) and the average number of phases needed in successful runs. We also evaluate the efficiency of the approach by calculating the average computational time of 50 runs in each case.

Fig. 4 and Fig. 5 show the performance comparison in cases with different population sizes. The results indicate that noticeable performance gains can be achieved with larger populations, which give the GA better sampling of the search space. A population size of 100 is not sufficient to produce competitive results as compared to larger populations. The runs with a population size of 400 need fewer phases to find solutions than runs with the baseline population of 200. A large population, however, incurs higher computational cost. Fig. 6 shows the average execution time of 50 runs in each test case. Execution time increases as the population size increases. The only exception is in runs on the 8×8 Sliding-tile puzzles, where the execution time of GA runs using a population size of 200 is shorter than those using a population size of 100.

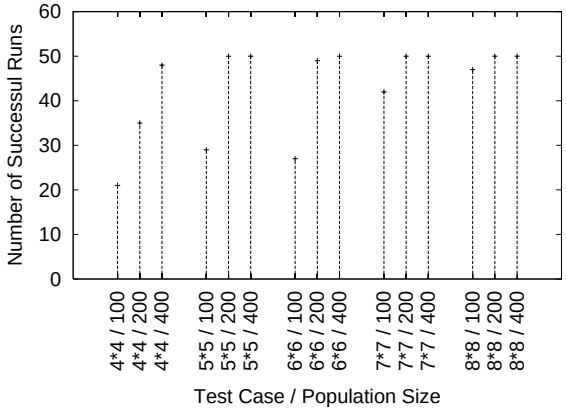

Fig. 4 The number of successful runs (out of 50) for population size from 100 to 400

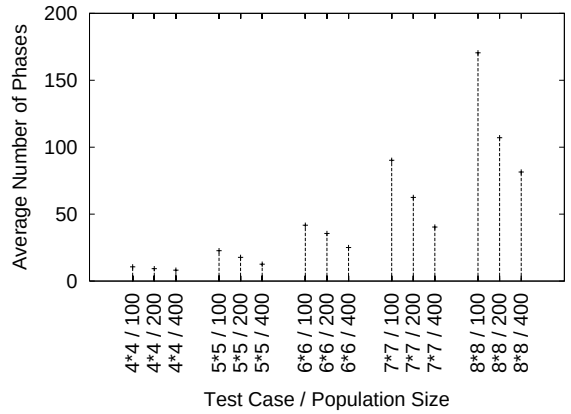

Fig. 5 The average number of phases needed to find a solution for successful runs with population size varying from 100 to 400

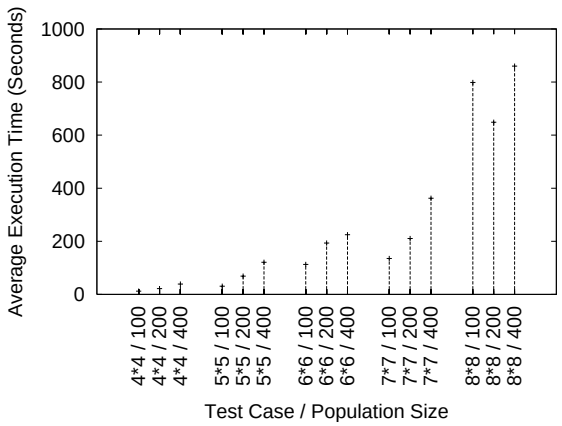

Fig. 6 The average execution time (of 50 runs) for population size varying from 100 to 400

Fig. 7 and Fig. 8 show the performance comparison in cases with different crossover rates. We test crossover rate of 0.5, 0.8, and 1.0 as well as the baseline settings of 0.9. The results indicate that varying the crossover has little effect on the search performance.

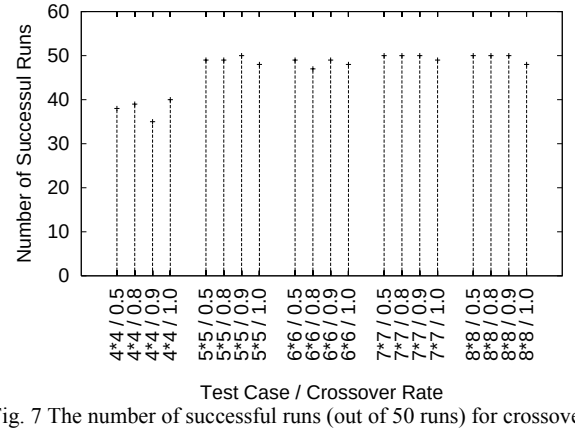

Fig. 7 The number of successful runs (out of 50 runs) for crossover rate varying from 0.5 to 1.0

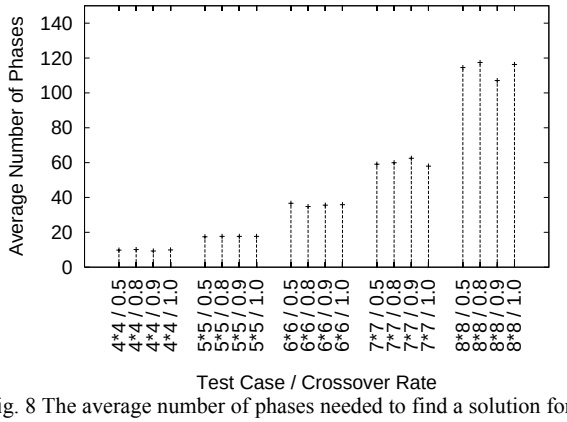

Fig. 8 The average number of phases needed to find a solution for successful runs with crossover rate varying from 0.5 to 1.0

Fig. 9 and Fig. 10 show the performance comparison in cases with varying mutation rates. A lower mutation rate (0.005) and a higher mutation rate (0.05) in addition to the baseline settings are tested. All test cases exhibit consistent search results, which indicate that the mutation rate has little effect on the search performance. We suspect the reason is that the crossover method applied in this approach is very disruptive and it already produces ample opportunities for exploring the search space. As a result, the usefulness of a mutation operator is significantly reduced.

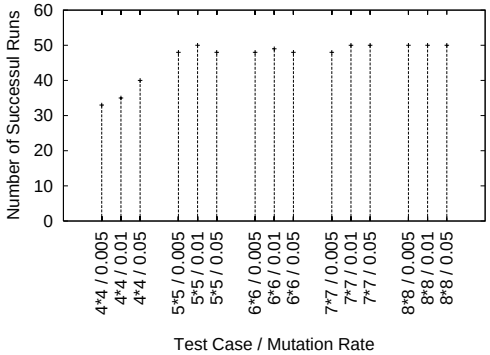

Fig. 9 The number of successful runs (out of 50 runs) for mutation rate varying from 0.005 to 0.05

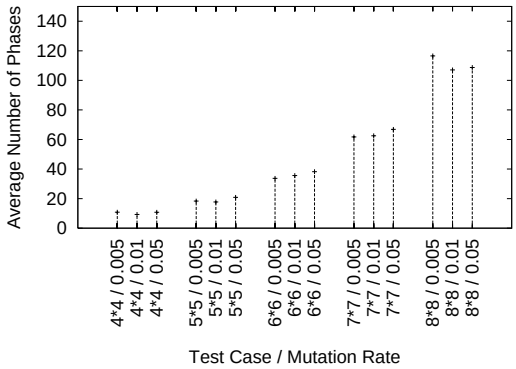

Fig. 10 The average number of phases needed to find a solution for successful runs with mutation rate varying from 0.005 to 0.05

VII. Conclusion and Future Work

In this paper, we introduce a search strategy for planning problems with conjunctive goals and combine this search strategy with a novel GA-based planning algorithm. Our strategy transforms the goal of a planning problem into a sequence of recursive subgoals. As a result, the search for a complete solution consists of a number of independent stages. After reaching a subgoal, the problem is reduced to a similar problem but at a smaller scale. This strategy is applicable to a larger class of problems characterized by the fact that the construction of recursive subgoals guarantees the serializability of the subgoals. The experimental results on the Sliding-tile puzzle indicate that, although the recursive subgoal strategy may not find optimal solutions, it is able to achieve better search performance than the traditional single-goal planning approach and solve larger instances of problems than existing domain-specific planning approaches. Additional experiments on the GA parameters reveal that the population size has much stronger influence on the performance of the search than crossover and mutation rates have. A large population improves the quality of search but it also results in higher execution time.

Although we identify three classes of planning domains relative to the applicability of this strategy, a crisp criterion to decide if our strategy is applicable for a given problem proves to be a formidable task. It is also very difficult to define the concept of "similar" planning problems. Informally, we say that a 5×5 sliding block puzzle is reduced to a 4×4 one and it is intuitively clear why these problems are similar, but formalizing this concept is hard. Our future work will address these open problems.

VIII. Acknowledgment

This research was supported in part by National Science Foundation grants MCB9527131, DBI0296035, ACI0296035, and EIA0296179, and the Colorado State University George T. Abell Endowment.

References

[1] Tower of Hanoi, http://www.cut-the-knot.com/recurrence/hanoi.shtml.
[2] A. Barrett and D. S. Weld. "Characterizing subgoal interactions for planning." In *Proc. of the 13th International Joint Conference on Artificial Intelligence* (IJCAI-93), pages 1388-1393, Chambery, France, 1993.
[3] A. Barrett and D. S. Weld. "Partial-order planning: evaluating possible efficiency gains." *Journal of Artificial Intelligence*, 67:71-112, 1994.
[4] J. Cheng and K. B. Irani. "Ordering problem subgoals." In *Proc. of the 11th International Joint Conference on Artificial Intelligence* (IJCAI-89), pages 931-936, Detroit, USA, 1989.
[5] M. Drummond and K. Currie. "Goal ordering in partially ordered plans." In *Proc. of the 11th International Joint Conference on Artificial Intelligence* (IJCAI-89), pages 960-965, Detroit, USA, 1989.
[6] O. Etzioni. "Acquiring search-control knowledge via static analysis." *Journal of Artificial Intelligence*, 62:255-301,1993.
[7] R. Fikes and N. Nilsson. "STRIPS: A new approach to the application of theorem proving to problem solving." *Journal of Artificial Intelligence*, 2(3/4):189-208, 1971.
[8] J. Hertzberg and A. Horz. "Towards a theory of conflict detection and resolution in nonlinear plans." In *Proc. of the 11th International Joint Conference on Artificial Intelligence* (IJCAI-89), pages 937-942, Detroit, USA, 1989.
[9] W. W. Johnson and W. E. Story. "Notes on the '15' puzzle." *American Journal of Mathematics*, 2(4):397-404, 1879.
[10] J. Koehler and J. Hoffmann. "Planning with goal agendas." *Technical Report 110, Institute for Computer Science*, Albert Ludwigs University, Freiburg, Germany, 1998.
[11] R. E. Korf. "Planning as search: A quantitative approach." *Journal of Artificial Intelligence*, 33:65-88, 1987.
[12] R. E. Korf, personal communication, 2003.
[13] R. E. Korf and A. Felner. "Disjoint pattern database heuristics." *Journal of Artificial Intelligence*, 134:9–22, 2002.
[14] R. E. Korf and L. A. Taylor. "Finding optimal solutions to the twenty-four puzzle." In *Proc. of the Thirteenth National Conference on Artificial Intelligence* (AAAI 96), pages 1202–1207, Portland, OR, 1996.
[15] F. Lin. "An ordering on subgoals for planning." *Annals of Mathematics and Artificial Intelligence*, 21(2-4):321-342, 1997.
[16] S. J. Russell and P. Norvig. "Artificial Intelligence: A Modern Approach." Prentice Hall, Upper Saddle River, NJ, 1995.
[17] H. Yu, D. C. Marinescu, A. S.Wu, and H. J. Siegel. "A genetic approach to planning in heterogeneous computing environments." In *the 12th Heterogeneous Computing Workshop (HCW 2003), CD-ROM Proc. of the 17th International Parallel and Distributed Processing Symposium* (IPDPS 2003). IEEE Computer Society Press, Los Alamitos, CA, ISBN 0-7695-1926-1, 2003.