A Genetic Approach to Planning in Heterogeneous Computing Environments

Han Yu, Dan C. Marinescu, Annie S. Wu
School of Electrical Engineering and Computer Science
University of Central Florida
Orlando, Florida, 32816, USA
hyu, dcm, aswu@cs.ucf.edu

Howard Jay Siegel
Department of Electrical and Computer Engineering
and Department of Computer Science
Colorado State University
Fort Collins, Colorado, 80523-1373, USA
hj@colostate.edu

Abstract

Planning is an artificial intelligence problem with a wide range of real-world applications. Genetic algorithms, neural networks, and simulated annealing are heuristic search methods often used to solve complex optimization problems. In this paper, we propose a genetic approach to planning in the context of workflow management and process coordination on a heterogenous grid. We report results for two planning problems, the Towers of Hanoi and the Sliding-tile puzzle.

1. Introduction and Motivation

Most human activities require some form of planning; thus, it is not surprising that planning is a subject of interest to the artificial intelligence (AI) community. Solving any complex task requires planning, thus, planning is very important in practice as well. Given a set of actions, the objective of planning is to construct a valid sequence of actions, or a plan, to reach a goal state starting from the current state of the system. An intuitive example of planning is the process of solving a puzzle, given the set of pieces with different geometric shapes, scattered on the floor.

Computational grids are heterogenous infrastructures linking many computers together and presenting individual users the image of a large virtual machine capable of performing complex tasks. Planning allows us to create multiple activity graphs, or process descriptions in workflow terminology, and to exploit the resource-rich environment provided by a computational grid. The middleware component of the grid infrastructure includes a set of services to discover available resources and to optimize the use of resources both from the grid and from the user's perspective. Certain tasks performed by the middleware are referred to as *societal services*. Examples of societal services are coordination, planning, brokerage, persistent storage, and authentication.

The objective of planning in the context of the execution of complex tasks on a grid is to construct an activity graph describing a transformation of input data into a different set of data, the desired result. Once this graph is constructed, its description can be provided to a coordination service and then the execution of all the programs involved is supervised by the coordination service. Rather than have a user construct the graph by hand, then select a site A to execute say program $\mathcal{A}$, wait for the results to be produced, then use a file transport protocol to move them to site B where program $\mathcal{B}$ could execute, and so on, the process can be fully automated. Of course, the same result could probably be archived using a script, but the static nature of the script makes handling of dynamic resource allocation problem much more difficult. For example, assume that site B in our previous example is overloaded and there are alternative sites $B_i, i = 1, 2, \ldots n$, capable of executing program $\mathcal{B}$ at lower costs and provide the results earlier. A static script is incapable of taking advantage of the full range of alternatives to carry out a computation, while planning does. Often, users may also require changes in the actual sequence of data transformation. For example, one may wish to increase the accuracy of some computation by filtering or noise re-

duction. The *computation steering* [1] could also be a reason for dynamic altering of the activity graph.

Let us elaborate on the planning process. Data is transformed as a result of the execution of a set of programs, corresponding to the nodes of the activity graph. We assume that we have ontologies describing data, programs, and hardware resources. An *ontology* is a description of the concepts and relationships among them for an agent or a confederation of agents; sometime the scientific community calls this *meta-information*. The description of each program includes a set of *pre-conditions* such as: the type, format, amount, and possibly a history of the input data [2]; the location of the binary and source code; libraries used by the program; and the physical resources required by the program to execute (specified typically as a lower limit or a range of values, e.g., more than 1 GB of main memory, 1 to 3 TB of disk space). In addition to pre-conditions, we have *post-conditions* describing attributes of the results produced by the program, such as: the type, the format, the volume, and the location. A similar description can be built for hardware resources and for data.

An informal description of planning in a computational grid environment is the following steps. First, given a set of initial data and a set of desired results, construct an activity graph to produce the results given the initial data. Then submit the graph description for execution by a coordination service. The graph description can be modified during the execution in response to either information regarding the status of various grid resources or as a result of explicit user desires.

Similarly, in a service grid, multiple versions of services may be provided; planning provides automated means to compose them in a coherent manner. In both cases, planning is an off-line activity; it precedes the execution of the complex task. Therefore, the time required for planning is not a major concern and the genetic approach discussed in this paper is entirely feasible.

Planning may also be done on-line when timing constraints are not violated and resources are not wasted. There are several reasons why a generic activity graph may change dynamically during the enactment of a particular case. The change may be mandated by the change in availability of resources necessary for the execution of individual tasks in the generic activity graph. In this case, the time required by the planning algorithm is of concern and may limit the applicability of our approach. Another reason for change may be the desire to improve the quality of the solution: for example, task α, may be replaced by a sequence of tasks β and γ that are capable of providing a better solution than task α. In this case, we may be able to afford to go through a process of re-planning depending upon the duration of the tasks involved and the benefits of the change.

Thus, we believe that a planning method that offers the possibility of dynamically creating activity graphs [13] is an important component of the workflow management system needed to coordinate the execution of complex tasks in the resource-rich environment provided by a computational or service grid. Genetic algorithms (GAs) have been used in the past for task scheduling in heterogenous systems where an activity graph is provided and the objective is to find an optimum resource allocation scheme (e.g., [4, 11, 19, 20]). Creation of an activity graph for dynamic workflow management as a result of a planning process is a considerably more difficult problem.

There are several formal definitions of a planning problem. We are particularly interested in STRIPS-like domains [6]. In such domains, the change of system state is given by operations which are defined by preconditions and postconditions. We define a planning problem to be a four-tuple

$$\Pi = (\mathcal{P}, \mathcal{O}, \mathcal{I}, \mathcal{G}).$$

$\mathcal{P}$ is a finite set of ground atomic conditions (i.e., elementary conditions instantiated by constants) used to define the system state. $\mathcal{O} = \{\mathcal{O}_i\}$, where $1 \leq i \leq |O|$ is a finite set of operations which can change the system state. Each operation has three attributes: a set of preconditions $\mathcal{O}_i^{pre}$, a set of postconditions $\mathcal{O}_i^{post}$, and a cost $\mathcal{C}(\mathcal{O}_i)$. $\mathcal{I} \subseteq \mathcal{P}$ is the initial state and $\mathcal{G} \subseteq \mathcal{P}$ is the goal state. A plan Δ is a finite sequence of operations. An operation may occur more than once in a plan. An operation is valid if and only if its preconditions are a subset of the current system state. A plan Δ solves an instance of Π if and only if every operation in Δ is valid and the result of applying these operations leads a system from state $\mathcal{I}$ to a state that satisfies all the conditions in the goal state $\mathcal{G}$.

Planning is generally more difficult than a typical search problem. Planning involves larger search spaces, the existence of solution is not guaranteed, and last but not least, the size of the optimal solution cannot be easily anticipated. General search strategies such as breadth first search, though applicable to planning problems, rarely find good solutions efficiently. General planning algorithms such as forward- and backward-chaining are based upon deterministic search methods. To find a good solution, these algorithms require a search over the entire problem space. Therefore, forward- and backward-chaining perform well

[1] The ability to examine partial results and eventually make changes that affect the evolution of a computation, e.g. change a parameter file.

[2] Assume that some 2D image data was collected with a camera with resolution *x*, transformed using a histogram equalization algorithm with parameter *y*, then filtered using a high pass filter with frequency z, then Fourier transformed using an algorithm that required zero-filling up to size *w*. When we decide to use either program A, or B, or C for the next stage of processing, we would like to know the genealogy, or the history of the data. Program A could require a resolution higher than *x*, B could do a filtering in the Fourier domain that would cancel the effect of the histogram equalization, and so on. The history of the data would help in the next stage of planning. The preconditions of either A or B would not be satisfied.

only on small problems with a very limited search space.

GAs are one type of heuristic search method often used to solve difficult optimization problems. GAs are inspired by a fundamental principle of natural selection, the *survival-of-the-fittest*. A genetic algorithm evolves a fixed size population throughout generations. Each individual in the population encodes a candidate solution to a given problem. Initially, these solutions are randomly generated. With each new generation, the GA evaluates the performance of every individual with a fitness function that gives a numeric value as the result of evaluation. Selection of the individuals is based on their fitness. Good solutions have a higher chance of surviving in the population. Selected individuals are subjected to crossover and mutation to explore new search spaces without completely losing the existing solutions that have been evolved. A newly generated population is found. The GA then repeats these evolutionary steps to generate the next population.

In this paper, we propose a non-deterministic approach to planning based upon a GA. Our approach differs from traditional GA applications in several ways. First, we encode the solution as a sequence of floating point numbers to eliminate the existence of invalid operations. Second, we apply three different crossover mechanisms to form new solutions. Third, we divide the search into a number of phases, every phase is an independent GA run and the final solution is the concatenation of the best solutions from individual phases. Experimental results show that our approach is able to solve the 7-disk Towers of Hanoi problem and a 4×4 Sliding-tile puzzle.

This paper is organized as follows. Section 2 provides an overview of the current research in the area of planning algorithms and genetic approaches to planning. In Section 3, we report on our algorithm; we discuss the solution encoding in Section 3.1, the population initialization in Section 3.2, the fitness evaluation in Section 3.3, the genetic operators in Section 3.4, and our multi-pase approach to planning in Section 3.5. In Sections 4.1 and 4.2 we present our experiments related to the Towers of Hanoi problem and to the Sliding-tile puzzle. Finally, we discuss our conclusions.

2. Related Work

A comprehensive analysis of the complexity of domain-independent planning algorithms is found in Erol et al. [5]. The authors study the conditions for the decidability of a planning problem and show that, when planning is decidable, the time complexity of a domain-independent planning algorithm depends on a large number of variables.

The *Graphplan* approach exploits the fact that the operation space is much smaller than the state space of a planning problem [1]. The algorithm first generates a planning graph showing all the possible operations at every time step. Operations that interfere with one another can coexist in the graph. The search for a plan is based on this graph. Experimental result shows that Graphplan outperforms other general planning algorithms in some problem domains.

Jonsson et al. study the efficiency of universal planning algorithms [8]. They conclude that universal planners that run in polynomial time and polynomial space cannot satisfy even the weakest types of completeness. However, if one of the polynomial requirements is removed, constructing a plan that satisfies completeness becomes a trivial problem. They also propose *Stocplan*, a randomized approach to universal planning under a restricted set of conditions. They show that this approach can construct plans that run in polynomial time and use polynomial space and also satisfy both soundness and completeness for these problems. Experiments reveal that the performance of Stocplan is competitive with Graphplan.

Another approach to planning is to partially reuse existing plans. This approach consists of two steps, plan matching and plan modification. Nebel and Koehler [16] analyze the relative computational complexity of plan reuse versus planning from scratch. The study shows that the problem of plan reuse is intractable and the efficiency of this approach is not guaranteed. Generally, reusing an existing plan is harder than planning from scratch. This approach is expected to work better only when the new planning problem is sufficiently close to the old one. Plan matching, as a necessary step in this approach, can be the bottleneck in computation time.

A different direction of planning research is focused on domain-specific planning. Korf and Taylor report on the Sliding-tile puzzle and discuss useful search heuristics for this problem [10]. They present the work on an accurate admissible heuristic function in the $\mathcal{IDA}^*$ 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 $\mathcal{IDA}^*$ search algorithm.

A proposal to use disjoint pattern database heuristics in an evaluation function is discussed in [9]. First, the subgoals are split into disjoint subsets so that an operation 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. This technique was used to search for a solution of the Sliding-tile puzzle and of Rubik's cube. In both cases, it resulted in a decrease of the number of nodes traversed during the search.

Bonet and Geffner [3] show that a heuristic search planner is competitive with the Graphplan or the *SAT* planners. They introduce two planners, *HSP* and *HSP2*; HSP is a hill-climbing planner and HSP2 is a best-first planner. Both planners are forward state planners. This approach assumes that subgoals are independent; therefore, the function is ad-

missible and never overestimates the cost.

All of the above approaches except Stocplan are deterministic approaches that tend to require large coverage of a search space in order to generate a good result. Problem specific heuristics can be used to reduce the size of a search space; however, heuristics for one class of problems may not be applicable to other classes of problems. Evolutionary computation (EC) has emerged as a competitive technique in planning research. Because of an element of randomness in their implementation, the search results of EC methods are not consistent over different runs and these methods are not guaranteed to find an optimal solution. EC methods include genetic programming (GP) that is discussed below and the GA that we use.

Muslea [15] presents a GP approach to planning. *Sinergy* is a general linear planning system built on the GP paradigm. Experiments are performed on a single and 2-Robot Navigation problem and on the Briefcase problem. Results indicate that Sinergy can handle problems that are one or two orders of magnitude larger than the ones handled by *UCPOP* [17], a partial order planner. This approach works only on problems with conjunctive goals.

Another example of a GP based algorithm, called *GenPlan*, is presented in [21]. This approach uses a linear structure to encode the solution. Experiments on two domains show that GenPlan can solve the same problems as Sinergy but with fewer generations. The authors also report on five GP seeding strategies in [22] and show that these strategies improve the search quality on the Blocks World domain problems. Seeding partial solutions and keeping some randomness in the initial population appear to benefit GP performance.

Scheduling and process coordination on a computational grid is a relatively new area of research [2, 12, 13, 14]. However, we could not find any results on applying methods of planning to grid computing.

3. GA planning

In this section, we describe our genetic approach to planning problems. We present a method for solution encoding, then we discuss the fitness evaluation, the genetic operators, and a multi-phase approach to GA planning.

3.1 Solution Encoding

The solution to a planning problem is encoded as a sequence of genes, where each gene represents a single operation in the plan. The simplest way to encode such a plan is to use direct encoding: each operation is represented by an integer and the sequence of integers encodes the sequence of operations in a plan. Because not all operations are valid in every system state, a direct encoding may result in invalid operations.

We use an indirect encoding method. Each gene is represented as a floating point number *x*, $0 \leq x < 1$. Every number in the solution is mapped to a valid operation for the corresponding system state. The result of this mapping depends on the value of the floating point number and the set of valid operations in a given system state. For example, assume that in a given state there are four valid operations, $\mathcal{O}_1$, $\mathcal{O}_2$, $\mathcal{O}_3$, and $\mathcal{O}_4$. Then a floating point number x is mapped as follows:

$$\begin{aligned} 0.00 \leq x < 0.25 &\longmapsto \mathcal{O}_1 \\ 0.25 \leq x < 0.50 &\longmapsto \mathcal{O}_2 \\ 0.50 \leq x < 0.75 &\longmapsto \mathcal{O}_3 \\ 0.75 \leq x < 1.00 &\longmapsto \mathcal{O}_4. \end{aligned}$$

This method ensures that all the genes in a solution will represent valid operations. Thus, the operation represented by a gene depends on the floating point number and the set of operations that are valid for the given system state.

We allow the GA to evolve variable length individuals and set an upper bound, *MaxLen*, on the individual length. The value of *MaxLen* should be chosen to ensure GA search quality while not incurring too much computation time.

3.2 Population Initialization

The members of the initial population are randomly generated. The lengths of the initial population of solutions are set to reasonable values.

3.3 Fitness Evaluation

The goal of the planner is to find a solution that satisfies the following three criteria: (a) no invalid operation is allowed in a solution; (b) the sequence of operations leads the system from the initial to the goal state; and (c) the cost of the solution is minimized. Accordingly, the fitness function has three components: the match fitness f_m, the goal fitness f_g, and the cost fitness f_c.

The *match fitness function*, f_m, $0 \leq f_m \leq 1$, evaluates how well the operations in the solution match their current system state. In a preliminary implementation, when we use the direct encoding method, we go through every operation from the beginning to the end of a solution to calculate the match fitness. We start from the first operation and check if it is a valid operation. If yes, we change the system to the state after this operation is performed. Otherwise, the system stays at the current state. We perform the same check on each succeeding operation until we finish all of the operations in the solution. Match fitness is calculated as:

$$f_m = \frac{number\ of\ valid\ operations}{total\ number\ of\ operations\ in\ a\ solution}. \quad (1)$$

Because we use indirect encoding and all operations are valid, the match fitness is always 1. We do not need to check the matching of operations to system states. However, we still need to change the system state as a result of each operation.

The *goal fitness function*, $f_g, \quad 0 \leq f_g \leq 1$, evaluates the quality of matching between the final state of the solution and the goal state. A better match results in a higher goal fitness. The goal fitness is dependent on the characteristics of the planning problem.

The *cost fitness function*, $f_c, \quad 0 \leq f_c \leq 1$, evaluates the total cost of the solution. The cost of a solution depends upon the cost of individual operations and is problem specific. The cost may be related to the latency of an operation, the number of arithmetic operations, the amount of data to be transferred, and so on. A solution with low cost has a high cost fitness. In the very simple case when all the operations have the same cost, the cost fitness is given by:

$$f_c = \frac{MaxLen - individuallength}{MaxLen} \quad (2)$$

The *overall fitness function* reflects the three figures of merit:

$$f = a \times f_g \times f_m + b \times f_c, \quad (3)$$

where a and b are weights and $a + b = 1$. As the match fitness is always equal to 1 in our approach, it is unnecessary to include it in the fitness calculation. As a result, we use the following function in the experiment.

$$f = a \times f_g + (1 - a) \times f_c. \quad (4)$$

The fitness evaluation time has a significant impact on the overall execution time of a GA. A well designed fitness function is critical to improving the efficiency of a GA.

3.4 Selection and Genetic Operators

3.4.1 Selection

Tournament selection is used to select the individuals for the next generation. In this selection scheme, we randomly pick up two individuals from the current population at each time and compare their fitness values. The individual with the higher fitness value wins and remains in the population.

3.4.2 Crossover

We implement three different crossover mechanisms: random, state-aware, and mixed. In each case, the children created replace their parents.

Random crossover is similar to GA one-point crossover. We randomly pick two individuals from the selected parents and randomly select one crossover point on each parent. The two parents exchange portions of their genetic code relative to the two crossover points. Two children are created; each inherits a portion of the genetic code from both parents.

A potential problem with random crossover is that the selected crossover points can be associated with different system states. Because we use an indirect encoding method, the mapping from floating point numbers to operations is dependent on the system state, see Section 3.1. Therefore, it is likely that the genes to the right of the crossover points will be mapped to a different sequence of operations after crossover although they are still represented by the same floating point numbers.

State-aware crossover is a novel approach that addresses the problem of state mismatching in random crossover. We randomly select a crossover point from the first parent. We restrict the crossover point of the second parent to those that match the first crossover point. Two states match if the same genetic code will be mapped to the same sequence of operations from these two states. If no such crossover can be found, we do not perform the crossover and both parents are included in the population of the next generation. State-aware crossover attempts to preserve partial solutions that have been evolved in the search.

Mixed crossover combines random and state-aware crossover. We randomly select the first crossover point and check if state-aware crossover can be performed. If yes, we perform the state-aware crossover on the two parents. Otherwise, we randomly select the second crossover point and carry out a random crossover.

3.4.3 Mutation

Every gene has equal probability of being mutated. In every mutation, a new randomly generated floating point number replaces the old one.

3.5 The Multi-phase GA

We propose to build a solution to a planning problem incrementally using a multi-phase approach. We divide the GA search into multiple phases. Each phase is an independent GA run and consists of a fixed number of generations. In the first phase, we take the initial state of the system as the state where the search starts. When a phase ends, the best solution found in this phase is stored and the final state

of the solution is taken as the initial state for the search during the next phase. The GA search ends when a valid solution is found at the end of one phase, or after a predefined number of phases. The final solution of a GA run is the concatenation of the best solutions from all the phases.

The procedure of a multi-phase GA consists of following steps.

1. Start GA. Initialize population.
2. While the stopping condition is not met do
   (a) While fewer than the specified number of generations are evolved in the current phase do
      i. Evaluate each individual in the population.
      ii. Select individuals for the next generation.
      iii. Perform crossover and mutate operations on selected individuals.
      iv. Replace old with new population.
   (b) Select the best solution for this phase and keep it.
   (c) If a valid solution is found, go to the next step. Otherwise, randomly initialize population and start the next phase. Set the start state as the final state of the best solution in the previous phase.
3. Construct the final solution by concatenating the best solutions from all the phases.

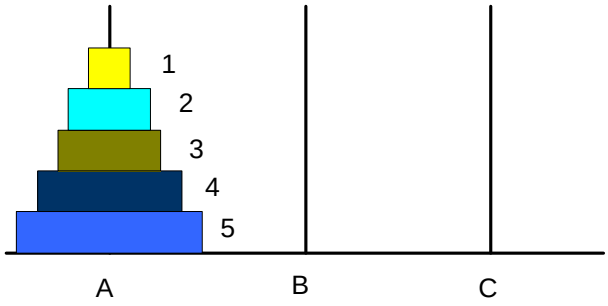

Figure 1. The initial state of the 5-disk Towers of Hanoi problem.

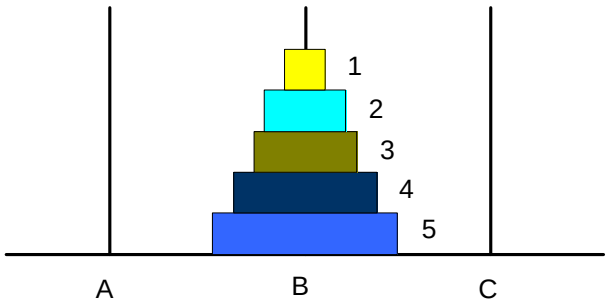

Figure 2. The goal state of the 5-disk Towers of Hanoi problem.

4. Experimental Results

We tested our GA approach to planning on two classical planning problems, the Towers of Hanoi and the Sliding-tile puzzle. Each experiment was run multiple times and the average performance is reported here. Each individual run of the GA was executed using a different random seed.

4.1 Towers of Hanoi

In Towers of Hanoi problem, there are three stakes, A, B, C, and *n* disks, $\mathcal{D}_1, D_2, \ldots, D_n$ of increasing size. $\mathcal{D}_1$ is the smallest disk and $\mathcal{D}_n$ is the largest disk. Initially, all of the disks are on stake A. The goal is to move all of the disks to stake B in a minimum number of steps. In each step, only one disk can be moved from one stake to another stake. Larger disks are not allowed to be moved on top of smaller disks. The minimum number of steps to reach a goal has been proven to be $2^n - 1$. Figures 1 and 2 show the initial and goal states for the 5-disk Towers of Hanoi problem.

In our experiments, we set the size of initial individuals to the length of the optimal solution, $2^n - 1$. To evaluate the goal fitness, we give different weights to disks with different sizes: $\mathcal{D}_i$ has a weight of 2^{i-1}. The total weight of all *n* disks is $2^n - 1$. The goal fitness is calculated using the following equation:

$$f_g = \frac{\textit{total weight of all disks on stake B in the final state}}{2^n - 1} \quad (5)$$

The cost fitness is given by equation 2. For the Towers of Hanoi problem, the value of *MaxLen* is $10 \times (2^n - 1)$.

In this experiment, we use random crossover and test both the single-phase GA and multi-phase GA approaches with the same number of generations. Table 1 shows the parameters for this experiment. For the single-phase GA the maximum number of generations allowed is 500; for the multiple-phase GA every phase contains 100 generations and the maximum number of phases allowed is 5.

We performed ten (10) runs (where each run uses a different initial population) in each case and picked the individual with the highest goal fitness in each run. Then we averaged the fitness and the length of these individuals. We

Parameter	Value
Population size	200
Number of generations	500
Crossover rate	0.9
Mutation rate	0.01
Selection scheme	Tournament (2)
Weight of f_g	0.9
Weight of f_c	0.1
Number of disks	5, 6, and 7
Number of phases in multi-phase GA	5

Table 1. Parameter settings used in the Towers of Hanoi planning experiments.

also calculated the average number of generations required to find a solution. Table 2 summarizes our results.

Our data show that the multi-phase algorithm performs better than the single phase GA. The multi-phase GA can find a valid solution in every run for the 5-disk and 6-disk cases. Although the multi-phase GA cannot find a valid solution in some runs for the 7-disk case, it evolves a solution that has higher goal fitness than the single-phase GA. In the 6-disk case, the multi-phase GA can find a valid solution in two phases (i.e., 200 generations), slightly faster than the single-phase GA.

The multi-phase algorithm evolves longer solutions than the single-phase GA. This is probably because the limit of individual length in multi-phase is larger than the one for single-phase GA. In our experiments, every run can have up to five phases, so the maximum allowed individual length in multi-phase algorithm is five times higher than the one for the single-phase GA.

Still, the multi-phase algorithm is not guaranteed to find a valid solution as the problem size scales up. The problem-specific definition of the goal fitness is partially responsible for this undesirable feature of the algorithm. Even though we give more credit to large disks in evaluating the goal fitness, a partial solution might go into to a state where all disks except the largest one are on stake B. This solution will receive a goal fitness slightly less than 0.5. This state, however, is even farther from the goal state than the initial state. Indeed, to reach the goal, all these disks have to be moved away from stake B before the largest disk can be moved to disk B. This difficulty indicates to us that good heuristic functions still play important roles in improving the performance of our approach.

4.2 Sliding-tile Puzzles

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 [18] discuss the so called 15-puzzle seen in Figure 3. Given some initial state, say the one in Figure 3(a), the goal is to reach the goal state seen in Figure 3(b) by moving the blocks without lifting them from the board. Solutions do not exist for every possible combination of initial and goal states. Johnson and Story showed in a paper published in 1879 that a solution exists only when the starting configuration is an even permutation of the goal configuration [7].

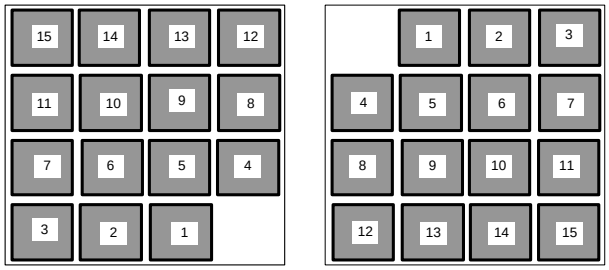

Figure 3. (a) An initial state of the 15-puzzle. (b) The goal state.

In this experiment we set the initial size of a solution as $\frac{n^2 \times (n^2-1)}{2}$, where n is the number of blocks in every row or column. This expression is the number of comparisons needed to sort a set of n^2 values. While the Sliding-tile puzzle is not the same as a sorting problem, e.g. there are restrictions on the tiles that can be exchanged, we believe that this expression gives a reasonable size with which to start the GA. Previous GA studies have found evidence that a variable length GA will evolve to a necessary solution length regardless of the initial individual lengths [23].

The distance between the current state and the goal state is given by the *Manhattan distance* of all tiles [18]. The upper bound on the distance between any two states in a $n \times n$ problem is $(n-1) \times 2 \times (n^2-1)$, where $(n-1) \times 2$ is the longest distance that a single tile may need to move

GA Type	Number of Disks	Average Goal Fitness	Average Size of Solution	Average Number of Generations to Find a Solution
Single-phase	5	1.0	72.3	42.9
	6	0.916	421.3	201.6
	7	0.618	628.0	328.6
Multi-phase	5	1.0	153.4	100
	6	1.0	571.8	200
	7	0.773	799.8	429

Table 2. Experimental results for the Towers of Hanoi problem.

and $n^2 - 1$ is the number of tiles. The goal fitness is:

$$f_g = 1 - \frac{\textit{Manhattan distance between final state and goal state}}{(n-1) * 2 * (n^2 - 1)} \qquad (6)$$

The cost fitness is given by equation 2. For the Sliding-tile puzzle, the value of *MaxLen* is $10 \times \frac{n^2 \times (n^2-1)}{2}$.

In this problem, we test all three crossover mechanisms with up to five individual phases in each run. Table 3 shows the parameter settings for the Sliding-tile puzzle experiments.

We perform 50 GA runs for each experiment and select the individual with the highest goal fitness in every run as the solution. In addition to fitness and individual length, we also recorded the number of runs required to find a valid solution and the average computation time for each run. Table 4 summarizes our results.

The performance of the three crossover types are very close. All of them can find a valid solution in 48 out of 50 runs for the 3×3 case. As the problem size grows, the search performance degrades sharply. The average size of the solutions increases faster than linearly as the number of tiles increases. The computation time depends heavily on the individual length.

We further investigate the contribution of multiple phases to the search of solution. We record the number of runs required to find a valid solution in each phase for the 3×3 case. Table 5 lists the result for all three crossover mechanisms.

In most of the runs, a valid solution is found within the first two phases. State-aware and mixed crossover can find a solution faster than random crossover. Of the three crossover mechanisms, state-aware and mixed crossover have a greater probability of finding a valid solution in the first phase. Random crossover does not search as fast as the other two crossover mechanisms, but using multiple phases helps it to find a valid solution before the end of second phase with a very high probability.

5. Summary

We propose a novel genetic approach to planning. In our algorithm, we use an indirect encoding method to prevent invalid operations in the solution and we test three different crossover mechanisms to combine the partial solutions during the search. In addition, we propose a multi-phase planning algorithm; we divide the search process into a number of serially independent GA runs so that the solution can be built incrementally. The results of experiments conducted on two planning problems, the Towers of Hanoi and the Sliding-tile puzzles, show that our algorithm is capable of finding valid solutions under a number of cases. However, as problem sizes increase, our approach, like all other solutions known in the literature, experiences difficulties in finding a solution. Our results confirm that an accurate goal fitness function is essential to achieving good search performance.

Based on our literature search, GenPlan appears to be one of the most similar approaches to ours. Both are evolutionary in nature and use a linear representation. Nevertheless, there are significant differences between the two approaches. First, GenPlan uses a direct encoding method and it allows solutions to contain invalid operations. Moreover, operations can have arguments. Both operations and their arguments can be evolved during the search. This approach gives GenPlan more freedom to encode and evolve a solution but, at the same time, increases the difficulty of the search due to its increased search space. Second, GenPlan uses larger population size and fewer generations than our approach, which suggests that the performance of GenPlan relies more on the genetic diversity than the evolutionary power of GP.

Our work is one of the earliest studies of genetic algorithms applied to planning problems and certainly one of the first if not the first of planning applied to dynamic workflow management on a grid. There are ample opportunities for research in these areas. We plan to explore multi-phase structures in solution representation, structural goal formation, and more accurate goal fitness functions. We believe that these studies will give us a better understanding of the

Parameter	Value
Population size	200
Number of generations	500
Crossover type	Random / State-aware/ Mixed
Crossover rate	0.9
Mutation rate	0.01
Selection scheme	Tournament (2)
Weight of f_g	0.9
Weight of f_c	0.1
Board size (n)	3 and 4
Number of phases in multi-phase GA	5

Table 3. Parameter settings for the Sliding-tile puzzle experiments.

Type of Crossover	Number of Tiles	Average Goal Fitness	Average Size of Solution	# Runs That Find a Valid Solution	Average Time (seconds)
State-aware	9	0.995	106.96	48	57.70
	16	0.927	865.40	0	415.27
Random	9	0.995	182.52	48	82.35
	16	0.935	831.70	1	408.86
Mixed	9	0.995	131.32	48	60.33
	16	0.928	922.06	1	434.13

Table 4. Experimental results for the Sliding-tile puzzle.

Phase	Random	State-aware	Mixed
1	7	33	36
2	40	13	11
3	1	0	1
4	0	2	0
5	0	0	0

Table 5. The number of runs when a valid solution is found in each phase for the random, state-aware, and mixed crossover strategies.

nature of planning problems and improve the performance of our approach.

6. Acknowledgements

The research reported in this paper was partially supported by National Science Foundation grants MCB9527131, DBI0296107, ACI0296035, and EIA0296179, and by the Colorado State University George T. Abell Endowment.

References

[1] A. L. Blum and M. L. Furst. Fast planning through planning graph analysis. *Journal of Artifi cial Intelligence*, 90:281–300, 1997.

[2] L. Boloni and D. C. Marinescu. Robust scheduling of metaprograms. *Journal of Scheduling*, 5:395–412, 2002.

[3] B. Bonet and H. Geffner. Planning as heuristic search. *Journal of Artifi cial Intelligence*, 129:5–33, 2001.

[4] T. D. Braun, H. J. Siegel, N. Beck, L. L. Boloni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 61(6):810–837, June 2001.

[5] K. Erol, D. S. Nau, and V. S. Subrahmanian. Complexity, decidability amd undecidability results for do main-independent planning. *Journal of Artifi cial Intelligence*, 76:75–88, 1995.

[6] R. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Journal of Artifi cial Intelligence*, 2(3/4):189–208, 1971.

[7] W. W. Johnson and W. E. Story. Notes on the "15" puzzle. *American Journal of Mathematics*, 2(4):397–404, 1879.

[8] P. Jonsson, P. Haslum, and C. Backstrom. Towards effi cient universal planning, a randomized approach. *Journal of Artifi cial Intelligence*, 117:1–29, 2000.

[9] R. E. Korf and A. Felner. Disjoint pattern database heuristics. *Journal of Artifi cial Intelligence*, 134:9–22, 2002.

[10] R. E. Korf and L. A. Taylor. Finding optimal solutions to the twenty-four puzzle. In *Proc. of the International Conference on Artifi cial Intelligence (AAAI 96)*, pages 1202–1207, Portland, OR, 1996.

[11] Y. Kwok, A. A. Maciejewski, H. J. Siegel, A. Ghafoor, and I. Ahmad. Evaluation of a semi-static approach to mapping dynamic iterative tasks onto heterogeneous computing systems. In *Proc. of the 1999 International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN '99)*, pages 204–209, June 1999.

[12] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund. Dynamic mapping of a class of independent tasks onto heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, 59(2):107–131, Nov. 1999.

[13] D. C. Marinescu. *Internet-Based Workflow Management: Towards a Semantic Web*. Wiley, New York, NY, 2002.

[14] D. C. Marinescu and C. L. (Eds). *Process Coordination and Ubiquitous Computing*. CRC Press, New York, NY, 2002.

[15] I. Muslea. SINERGY: A linear planner based on genetic programming. In *Proc. of the 4th European Conference on Planning*. Springer, 1997.

[16] B. Nebel and J. Koehler. Plan reuse versus plan generation: A theoretical and empirical analysis. *Journal of Artifi cial Intelligence*, 76:427–454, 1995.

[17] J. S. Penberthy and D. Weld. UCPOP: A sound, complete, partial-order planner for ADL. In *Proc. of the 3rd International Conference on Knowledge Representation and Reasoning (KR-92)*, pages 103–114, Cambridge, MA, 1992.

[18] S. J. Russel and P. Norvig. *Artifricial Intelligence, a Modern Approach*. Prentice Hall, Upper Saddle River, NJ, 1995.

[19] M. D. Theys, T. D. Braun, Y. Kwok, H. J. Siegel, and A. A. Maciejewski. Mapping of tasks onto distributed heterogeneous computing systems using a genetic algorithm approach. In *Solutions to Parallel and Distributed Computing Problems: Lessons from Biological Sciences*, pages 135–178. John Wiley & Sons, New York, NY, 2001.

[20] L. Wang, H. J. Siegel, V. P. Roychowdhury, and A. A. Maciejewski. Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach. *Journal of Parallel and Distributed Computing*, 47(1):8–22, 1997.

[21] C. H. Westerberg and J. Levine. GenPlan: Combining genetic programming and planning. In *Proc. of the 19th Workshop of the UK Planning and Scheduling Special Interest Group (PLANSIG 2000)*, Open University, Milton Keynes, UK, 2000.

[22] C. H. Westerberg and J. Levine. Investigation of different seeding strategies in a genetic planner. In *Proc. of the 2nd European Workshop on Scheduling and Timetabling (EvoSTIM 2001)*. Springer, 2001.

[23] A. S. Wu, A. C. Schultz, and A. Agah. Evolving control for distributed micro air vehicles. In *Proc. of the IEEE International Symposium on Computational Intelligence in Robotics and Automation*, pages 174–179, 1999.

Biographies

Han Yu

Han Yu is a Ph.D. student in the School of Electrical Engineering and Computer Science at the University of Central Florida (UCF). He received a B.S. degree from Shanghai Jiao Tong University in 1996 and an M.S. degree from the University of Central Florida in 2002. His research areas include genetic algorithms and AI planning.

Dan C. Marinescu

Dan C. Marinescu is Professor of Computer Science at University of Central Florida. He is conducting research in parallel and distributed computing, scientific computing, Petri Nets, and software agents. He has co-authored more than 130 papers published in refereed journals and conference proceedings in these areas.

Annie S.Wu

Annie S. Wu is an Assistant Professor in the School of Electrical Engineering and Computer Science and Director of the Evolutionary Computation Laboratory at the University of Central Florida (UCF). Before joining UCF, she was a National Research Council Postdoctoral Research Associate at the Naval Research Laboratory. She received her Ph.D. in Computer Science & Engineering from the University of Michigan.

H. J. Siegel

H. J. Siegel holds the endowed chair position of Abell Distinguished Professor of Electrical and Computer Engineering at Colorado State University (CSU), where he is also a Professor of Computer Science. He is the Director of the CSU Information Science and Technology Center (ISTeC). ISTeC is a university-wide organization for promoting, facilitating, and enhancing CSU's research, education, and outreach activities pertaining to the design and innovative application of computer, communication, and information systems. Prof. Siegel is a Fellow of the IEEE and a Fellow of the ACM. From 1976 to 2001, he was a professor in the School of Electrical and Computer Engineering at Purdue University. He received a B.S. degree in electrical engineering and a B.S. degree in management from the Massachusetts Institute of Technology (MIT), and the M.A., M.S.E., and Ph.D. degrees from the Department of Electrical Engineering and Computer Science at Princeton University. He has co-authored over 300 technical papers. His research interests include heterogeneous parallel and distributed computing, communication networks, parallel algorithms, parallel machine interconnection networks, and reconfigurable parallel computer systems. He was a Coeditor-in-Chief of the Journal of Parallel and Distributed Computing, and has been on the Editorial Boards of both the IEEE Transactions on Parallel and Distributed Systems and the IEEE Transactions on Computers. He was Program Chair/Co-Chair of three major international conferences, General Chair/Co-Chair of four international conferences, and Chair/Co-Chair of five workshops. He is currently on the Steering Committees of five continuing conferences/workshops. Prof. Siegel was an IEEE Computer Society Distinguished Visitor and an ACM Distinguished Lecturer, giving invited seminars about his research around the country. He is a member of the Eta Kappa Nu electrical engineering honor society, the Sigma Xi science honor society, and the Upsilon Pi Epsilon computing sciences honor society.