TECHNICAL REPORT NO. 285

Extending Distributed Genetic Algorithms to Problem Solving: The Case of the Sliding Block Puzzle

by

Jung-Yul Suh and Chan-Do Lee

July 1989

COMPUTER SCIENCE DEPARTMENT
INDIANA UNIVERSITY
Bloomington, Indiana 47405-4101

TECHNICAL REPORT NO. 285

Extending Distributed Genetic Algorithms to Problem Solving: The Case of the Sliding Block Puzzle

by

Jung-Yul Suh and Chan-Do Lee

July 1989

COMPUTER SCIENCE DEPARTMENT
INDIANA UNIVERSITY
Bloomington, Indiana 47405–4101

Extending Distributed Genetic Algorithms To Problem Solving: The Case of the Sliding Block Puzzle

Author: Jung-Yul Suh, Chan-Do Lee

Computer Science Dept., Indiana University

1 The Need of Parallel Scheme for Problem Solving

As parallel machines are increasingly available for use in recent years, much research is devoted to exploiting the full power of parallelism in these machines to solve problems which have been tackled by sequential programs in sequential machines. Much research attempts to *parallelize* the known sequential algorithm. In the case of AI, there have been attempts to parallelize typical search schemes for problem solving, such as A^* [4]. Since most of these schemes were originally designed for sequential machines, they are not quite amenable to parallelization. The result is that the computing speed increases sublinearly with the number of parallel processors. It is known to be quite difficult to achieve a linear speed-up[1] in parallel machines. This is usually due to the fact that a significant portion of these schemes cannot be parallelized by design. This portion called *sequential overhead* offsets whatever gains are made in the parallelizable portion of the schemes. The situation is no different in more knowledge-based scheme for problem solving. The only difference is that more knowledge is exploited in searching for a desired solution.

Thus, what is needed is a new scheme for problem solving that is fundamentally different from any scheme which was designed for a sequential machine in mind. What is required is an inherently parallel structure. We would like to propose the **Operator-oriented Distributed Genetic Algorithm (ODGA)** as a possible new alternative. In [16], it is shown that the Genetic Algorithm has an inherent parallel structure which can exhibit massive parallelism. Its implementation is done on the *Butterfly Machine* of *Bolt Beranek and Newman, Inc.*. The empirical test shows that it can exhibit almost linear speed-up, after the *selection operation* was modified properly so that the whole process can be distributed, parallel and asynchronous. The parallel version is called *Distributed Genetic Algorithm (DGA)*, designed to be executed by the *MIMD* machine. A high degree of parallelism comes from the fact that the population can be distributed over all processors evenly. The communication requirement of each processor is quite small, yet this relatively small amount of communication can improve the performance significantly. The parallelism can speed up the computation up to the size of population.

---

[1] with respect to, obviously, the number of processors

To briefly describe the general behavior of *DGA*, it consists of n processors, each of which has its own candidate solution of a problem in the local memory. Each has its own heuristics or knowledge to guide its search and they evolve their local candidate solutions until some of them find solutions or the whole population of candidate solutions converges. Thus n processors are randomly interacting with one another asynchronously, producing its offspring using a repertoire of genetic operators, namely *cross-over, mutation,* and *selection operations*. These operations can embody much of knowledge or heuristics, if necessary.

Encouraged by the success in *Traveling Salesman Problem (TSP)*, we explore the possibility of extending the application of *DGA* to the more complicated task of problem solving, such as the *Sliding Block Puzzle* or *Parsing a Sentence*. In this paper, only the case of *Sliding Block Puzzle* is investigated. At the moment, the kind of knowledge or heuristics exploited is not sophisticated. However, in the future, each processor can be equipped with domain-specific knowledge or heuristic and some additional operators , which may even contain some traditional sequential search algorithms such as A^*. The bulk of search is still done by genetic operators. However, the sequential deterministic search can be effective when the this search scheme find a candidate which appears close enough to the real solution. In a small search space, the deterministic sequential search can be quite useful.

Therefore, this paper is a preliminary investigation into the possibility of parallel distributed search scheme which is inherently parallel. This inherent parallelism actually is not surprising because the evolution of species in the nature, from which Genetic Algorithm originated, is a parallel process. Many animals produce offspring and the offspring are selected for by their performance in their surrounding environment. All these events happen in parallel fashion. The purpose of this paper is, in a way, to apply this effective scheme of natural evolution to solving complex problems exploiting its transparent parallelizability.

To solve a problem like the *Sliding Block Puzzle*, the representation scheme of candidate solutions should be needed. Also the proper design of genetic operators is necessary.

2 The Issue of Representation

2.1 Introduction

When GA first came out, it was used for finding near-optimal solutions of nonlinear real-valued functions. The representation for parameters is done by using binary representation of real numbers. After this application yielded some promising results, many tried to extend the application of GA to more difficult optimization tasks with some success. These attempts include applying GA to the *Traveling Salesman Problem(TSP)*[15, 7], Job Shop Scheduling Problem [13], Bin Packing Problem[2], and other combinatorial optimization problems[1, 3, 5, 6, 11, 12]. In particular, GA was quite successfully used in *TSP*. For details, refer to [15]. The level of difficulty rests on the degrees of constraint in structures to be optimized. There may be other ways to characterize the difficulty of optimization tasks. In this paper, we will only focus on the amounts of constraint in a structure.

In the task of optimizing real-valued functions, the representation of real numbers (which are parameters of the function) has little constraint. The only constraint is that each position of arrays representing real numbers must have either 0 or 1 as their value. There is no context sensitivity involved. That is, the change of bit value in one position does not affect values in the other positions. The change in one position creates a new bit vector which is another legitimate structure in the search space. In short

- S is the set of bit vectors of size n.

- if $v = (v_1, \ldots, v_n)$ is in S, then $w = (w_1, \ldots, w_n)$ is in S where $w_i = v_i$ or $1 - v_i$ for all $i = 1, \ldots, n$.

In many combinatorial optimization problems, structures need to satisfy higher degree of constraint than in the case of binary bit vectors. Let us illustrate this point by using *TSP* as an example. In *TSP*, the structure can be represented as a vector of integers between 1 and n where n is the number of cities to be visited. A structure can be encoded as follows (see also Figure 1):

> a structure is a tour encoded as $T = (t_1, t_2, \ldots, t_n)$ where $1 \rightarrow t_1 \rightarrow t_{t_1} \rightarrow t_{t_{t_1}} \rightarrow \ldots \rightarrow 1$ is a tour[7, 15].
>
> It is clear that T is always a permutation of size n. That is, number i $(1 \leq i \leq n)$ occurs only once in T as a component.

Since a tour should be always a permutation (i.e., no city can be visited more than once), a change in one position of T leads to at least one or more changes in other positions. Thus, this representation is more constrained than a binary bit vector. Every time a cross-over or (directed) mutation (refer to section 3 for definitions) is applied, whole structures may be affected to accommodate a local change introduced by these operators. This problem is dealt with by the use of a specialized cross-over operator called heuristic cross-over[7 , 15].

Already extensive modifications are needed to apply GA to many combinatorial optimization problems. Thus, it is quite clear that an attempt to apply GA to even more constrained structures is faced with a serious challenge.

$T[i] = j$ means that the tour T has an edge going from i to j.

$$T[1] = 4,\ T[2] = 3,\ T[3] = 1,\ T[4] = 5,\ T[5] = 2$$

$$1 \rightarrow T[1] \rightarrow T[T[1]] \rightarrow T[T[T[1]]] \rightarrow T[T[T[T[1]]]]$$

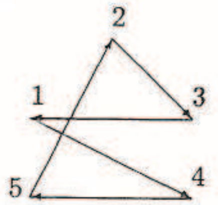


Figure 1: **The Representation of Tour**

Another problem which lies outside the reach of GA is common problem solving such as the *Sliding Block Puzzle (SBP)* and *Parsing a String(Sentence) with a Grammar (PSG)*. Search Schemes for these problems should be able to find a sequence of rules (or operators or moves) which can derive a goal state from the initial state[2]. These problems have three features which separate them from known GA optimization problems.

First, the length of the solution sequence of rules is not known. Usually it is not known in advance, even if the solvability of the problem is.

Second, rules have preconditions which must be satisfied in order to apply the rules. That is, a rule application can be context-sensitive. A sequence of rules will derive a state which may or may not satisfy preconditions for another sequence to be applied, so that two sequences may not necessary be concatenated. Cross-over operations should be more complicated than the case of *TSP*, let alone the case of function optimization. By similar reasoning, directed mutation will be not trivial.

---

[2] In *SBP*, a sequence of moves which will lead the initial board arrangement to the goal board arrangement. In *PSG*, a sequence of rules which will transform the initial symbol (usually denoted as S) into the sentence.

Third and most importantly, solving these problems will require the shift of focus in optimization tasks. Instead of structures, sequences of operators should be the objects to be evolved by GA if GA is to be applied. In these problems, we are less interested in whether there can be a solution than in how the goal can be reached from the initial state,that is, knowing that there is a solution.

It is the objective of this paper to present possible ways to resolve these difficulties. Before presenting these ideas, let us define and analyze problems of this class in order to motivate the subsequent discussion. We will call this class the **Metric-Operator Class**.

2.2 Defining the Class of Problems

The following is the definition of Metric-Operator Class to be investigated in this paper.

given

I: an initial structure.

G: a goal structure.

R: a set of operators (rules) $\{r_1, \ldots, r_k\}$ each operator has preconditions which should be satisfied in order for the operator to be applied to the structure.

$S = \{I \cdot \gamma \mid \gamma = r_{i_1} \cdot r_{i_2} \cdots r_{i_m}$, for some integer m $(m \geq 0)$ where r_{i_l} is applicable to $I \cdot r_{i_1} \cdots r_{i_{l-1}}$ for all l $(1 \leq l \leq m)\}$

$\Gamma = \{\gamma \mid I \cdot \gamma \in S\}$.

d_G: a non-negative real-valued function of S such that $d_G(s) = 0$ if and only if $s = G$. (we assume that for any $s \in S$, $d_G(s)$ is easy to compute, (that is, not consume a large computing resource).

δ_G : a non-negative real-valued function of Γ such that $\delta_G(\gamma) = d_G(I \cdot \gamma)$, for all $\gamma \in \Gamma$.

Type I : Find $\gamma \in \Gamma$ such that $I \cdot \gamma = G$ using a distance function δ_G to direct the search process. That is, search for $s \in S$ such that $\delta_G(s) = 0$. (It is always assumed that such a sequence exists for a given I and G.)

Type II: Find $s \in S$ which minimizes $d_G(s)$.(Note that d_G may not reach 0.)

2.3 Examples of This Class of Problems

1. **Sliding Block Puzzle: Type I**: given an initial board arrangement, find a sequence of moves of the empty tile which will lead to the goal board arrangement. (it is

I :

1	2	3
4	5	6
7	8	

G:

	1	3
4	2	6
7	5	8

R : `U` : if the empty tile is not at the top edge, move it up by one square.

`D` : if it is not at the bottom edge, move it down by one.

`L` : if it is not at the left edge, move it to the left by one.

`R` : if it is not at the right edge, move it to the right by one.

Γ : all legal operator sequences from the initial board I.

δ_G : $\delta_G(\gamma) = \Sigma_{i=0}^{8}\mu(i,\gamma)$ where $\mu(i,\gamma) =$ manhattan distance between the i-th tile in I and the i-th tile in $I \cdot \gamma$. Here, 0-th tile means the empty tile.

Figure 2: **Sliding Block Puzzle**

assumed that such a sequence is known to exist.) Figure 2 contains one example of this puzzle, the 8 puzzle on 3 by 3 board.

2. **Best-Fit Sliding Block Puzzle: Type II**: given an initial board arrangement, find a valid board arrangement (the one derived from the initial board by a sequence of legal moves) which will make the board arrangement as close as possible to the goal board arrangement. The degree of closeness can be measured in many different ways. One measure is to count number of tiles which are out-of-position compared with the goal board. Obviously, the task is to minimize d_G. This is a weaker version of the puzzle above. (In this case, there may be no sequence of moves which leads to the goal board.) Here, I, G, R are the same as above. S, d_G can be defined in a straightforward way using the definitions of Γ, δ_G in the above.

Grammar : s → np vp
np → a n | n
n → I | lunch | You | dinner
vp → v np
v → have | skip
a → a | the

G : parse tree starting with s.

I : a sentence: `I have a book`

R : the parsing rules: In the following, X and Y stand for arbitrary list.

ν_1 : (np X) (vp Y) → (s ((np X) (vp Y)))
ν_2 : (a X) (n Y) → (np ((a X) (n Y)))
ν_3 : (n X) → (np ((n X)))
ν_4 : w → (n (w)) where w = `I, lunch, You, dinner`
ν_5 : (v X) (np Y) → (vp ((v X) (np Y)))
ν_6 : w → (v (w)) where w = `have, skip`
ν_7 : w → (a (w)) where w = `a, the`

Note: ν_i^j means that apply ν_i starting at j-th position in the current structure.

Γ : any legal sequence of parse rules on I.

δ_G : Let us define the δ_G^* first. Then we will define δ_G in terms of δ_G^*.
$\delta_G^*(\gamma)$ = the number of parsed components after the rule sequence γ is applied to I. That is,

1. $\delta_G^*(\epsilon)$ = the number of parsed components of (`I have a book`)$\cdot\epsilon = 4$, where ϵ is the zero-application of rules.
2. $\delta_G^*(\gamma_0)$ = the number of parsed components of (n (I)) (v (have)) (a (a)) (n (book)) $= 4$, where $\gamma_0 = \nu_4^1\nu_6^2\nu_7^3\nu_4^4$).
3. $\delta_G^*(\gamma_0\nu_2^3\nu_5^2)$ = the number of parsed components of (n (I)) (vp ((v (have)) (np ((a (a)) (n (book)))))) $= 2$.

$$\delta_G(\gamma) = \begin{cases} \delta_G^*(\gamma) & \text{if } I \cdot \gamma \text{ does not contain a component with } s \text{ at its head} \\ \delta_G^*(\gamma) - 1 & \text{otherwise} \end{cases}$$

Figure 3: **Parsing of a Sentence According to a Grammar**

3. **Parsing of a String According to a Grammar:** Let G be a grammar with a set of derivation rules R and the initial symbol s and a set of terminal symbols A. Given a string $\alpha \in L(G)$ find the sequence of derivation rules $r_1, \ldots, r_k$ (such that $s \rightarrow_{r_1} \rightarrow_{r_2} \cdots \rightarrow_{r_k} \alpha$). See Figure 3 for an example.

4. **Best-Fit Parsing:** A weaker version of the parsing task above. Given a string $\alpha \in A^*$, find $\beta \in L(G)$ that is the closest fit to α. (Again, it is possible that $\alpha \notin L(G)$) I, G, R, S, d_G are defined in a manner analogous to that of SBP.

2.4 Inadequacy of Relaxation Method

The first and simplest way to deal with high degree of constraints is the relaxation of constraints. Instead of strictly searching for valid structures, the search space is extended to a larger space which contains the original subset and obeys far fewer constraints. Here, the mutation or cross-over operator will be more likely to generate structures which can satisfy the relaxed constraints, or at least those which are quite close to legal structures under relaxed constraints. The evaluation function is modified to work on the enlarged search space. For any structure, if it is a structure which does not satisfy original constraints, it is given a penalty value which depends on the degree of violation. This technique is used with the hope that the the search starts with the enlarged space but search quickly concentrates on the original search space and eventually finds the optimum in the original space. However, if the enlarged space is too big compared with the original space, this

$$T[1] = 2,\ T[2] = 3,\ T[3] = 4,\ T[4] = 2,\ T[5] = 1$$

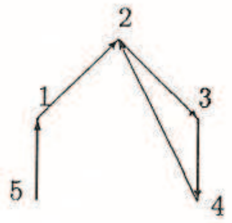


Figure 4: **Relaxed Tour**

technique ceases to be effective. The following example will show how that could be the case.

2.4.1 Example: The Relaxation Method Applied to *TSP*

The most common relaxation of *TSP* is to allow each structure to be a one-dimensional array of size n where every position can take any value between 1 and n. An example of relaxed tour is shown in Figure 4. The corresponding evaluation function can be defined as follows: whenever a number i $(1 \leq i \leq n)$ appears more than once as a component of the array, a positive penalty value can be added to the value of original evaluation. The global minimum of this new evaluation measure will be the optimal tour. It appears that *TSP* can be transformed into a much easier problem. Close examination, however, shows that this is not the case. Suppose that we want to solve the 50-city *TSP*. The relaxed search space will contain 50^{50} structures. The space of all legal tours will contain 50! structures (tours).

The simple calculation shows that

$$10^{-21} < \frac{50!}{50^{50}} < 10^{-20}$$

Thus, finding a legal tour itself is turned into a difficult search problem since the set of legal tours is such a tiny fraction of the relaxed search space. What usually happens is that GA can find a legal tour or something close to a legal tour, but will never get close to the optimal legal tour.

2.5 Patching-Up Method

The other common method in optimizing constrained structures is to patch up a structure produced as a result of cross-over or mutation into a correct structure (one which satisfies required constraint(s)). The cross-over or mutation usually generates structures which do not satisfy all required constrains. These structures are patched up so that they can be a correct structure. This method was successfully used in GA implementation of *TSP*. In *TSP*, the cross-over is modified so that it can patch up the offspring tour properly to make sure that the offspring is a valid tour. Many combinatorial optimization problem such as Job Shop Scheduling and Bin Packing can be solved by GA using this technique.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Initial Board

A	**B**	*C*	**D**
E	*F*	**G**	*H*
I	**J**	*K*	**L**
M	*N*	**O**	*P*

Lettered Position

Rules for determining if a specific arrangement is possible or impossible to obtain

1. Let N be a number in position A of the array to be achieved. Count how many numbers smaller than N are in positions higher lettered than A. Count the blank as 16.
2. Do this for all 16 positions (A through P), and add up the count.
3. If the blank square is the one marked by a bold-face letter(**B, D, E, G, J, L, M** or **O**), add 1 to the sum. Do not change the sum if the blank square was the one marked by a plain roman letter (A, C, F, H, I, K, N, or P).
4. The new array is **POSSIBLE** if the sum is **EVEN**.
5. The new array is **IMPOSSIBLE** if the sum is **ODD**.

Figure 5: **Algorithm for a Derivable Board from the Initial Board**

2.6 The Difficulty of Metric-Operator Class

In case of example 2, it is not difficult to solve the problem because there is an efficient algorithm to determine whether the given goal board arrangement can be derived from the initial board arrangement. (For details, see Figures 5, 6[3]). This algorithm can be used to construct a new board arrangement from two previous board arrangements by cross-over or from a single board arrangement by mutation. Given any board configuration, it can be

---

[3] This algorithm is from [14]

7	8	9	10
6	1	2	4
5	4	3	12
	15	14	13

A. Possible

15	14	13	12
11	10	9	8
7	6	5	4
3	2	1	

B. Impossible

determining the sums for the arrangements shown in figure A and B above.

lettered	numbers smaller than N	
position	A	B
A	6	14
B	6	13
C	6	12
D	6	11
E	5	10
F	0	9
G	0	8
H	3	7
I	2	6
J	1	5
K	0	4
L	0	3
M	3	2
N	2	1
O	1	0
P	0	0
sum	41	105
blank position	+1	+0
total	42	105

Figure 6: How to Use the Algorithm

transformed into an achievable arrangement, with little modification because all you have to satisfy is to make the sum of numbers[4] even. This problem is not likely to be solved successfully using the Relaxation Method. But, it can be solved by the Patch-up Method as suggested above.

For Example 4, there is no efficient algorithm known to us which can determine the grammatical correctness of a sentence. This makes Example 4 a hard problem even if we are just interested in the correctness of a sentence rather than the whole derivation sequence through which the sentence can be produced from the starting symbol S. The complexity of patching up a structure generated via cross-over or mutation in order to make it fit to grammatical constraints is more difficult than that of *TSP* due to a higher degree of constraints on this task. It is even more unlikely that this problem can be solved by the Relaxation Method.

For Example 1 and 3, it is seemingly impossible to solve them with just a structure, that is, a board configuration or generated string of symbols (both terminal and non-terminal). The task is to find a sequence of operators which will lead the initial state to the goal. Thus, we need more than a structure to represent the problems properly. It is quite clear that neither the Relaxation Method nor the Patch-up Method can be useful for these problems because they are to be used for structures, not sequences of operators

---

[4] Refer to the Figure 5

2.7 Shift of the Focus: from Structure to Operator

As has been argued in the previous subsection, the Relaxation Method or Patch-up Method will not work for optimizing most structures with high constraints nor for applying GA to finding a solution path for problem-solving tasks. What , then can be the alternative approach to the representation of these structures ?

We propose the **Operator-Oriented Representation** as a possible solution. This is a new way of representing structures, in that structures are represented indirectly. It is possible in many cases that the search space of GA can be represented as the set of sequences of transformation acting on an initial structure rather than the set of structures. Thus, we shift our focus to the transformation process rather than its final result (which is a structure). We regard a structure as a transformed version of a fixed initial structure. Thus, instead of searching for an optimal structure, GA performs the search for the optimal transformation. The essence of this approach is stated in Figure 7. In the case of *SBP*, the search space S is the set of all board arrangements that can be derived by a sequence of legal moves from the initial board. In case of *PSG*, the search space is all (terminal or non-terminal) symbol strings derivable from initial symbol S. Constraints can be represented as a set of simple rules which consist of its precondition and its action which is to be applied to a structure. All valid structure can be generated by repeatedly applying these rules correctly in sequence. Thus, **Operator-Oriented Representation** provides a universal framework by which a highly constrained structure can be represented in a simple and

Let S be a search Space.

Let $s_0 \in S$ be an initial structure.

Define an *operator* σ of S as a mapping such that $Dom(\sigma_s)$, $Ran(\sigma_s) \subseteq S$ and $s_0 \in Dom(\sigma)$.

Assume that for all $s \in S$, there exists at least one operator σ_s of S such that $s_0 {\cdot} \sigma_s = s$.

$\Sigma = \{\sigma \mid \sigma \text{ is a transformation of } S\}$.

thus, it is clear that optimization in S can be translated into the equivalent task in Σ[5]as follows:
$optimize_{s \in S} F(s)$ where F is a evaluation measure($F : S \rightarrow R$)
is equivalent to
$optimize_{\sigma \in \Sigma} G(\sigma)$ where $G : \Sigma \rightarrow R$ is defined as $G(\sigma) = F(s_0 \cdot \sigma)$.

Figure 7: **The Essence of Operator-Oriented Representation**

uniform way. The representation is a sequence of operators. One may view this approach as a scheme which shifts difficulties of a problem from representation to the design of genetic operators. As has been pointed out in subsection 2.1, the design of genetic operators will be more difficult than the case of *TSP* or function optimization. Not only must cross-over and mutation produce valid sequences of operators, but they also must generate better sequences in general. Satisfying these two requirements will be the main task in the design of this new variant of GA.

Thus, the success of Operator-Oriented Representation depends on how well these genetic operators perform. This is to be investigated with empirical research as discussed in Section 4. We need to make attempts to solve a variety of *Metric-Operator Class* problems. Then we can find out if this approach has real advantages over the conventional structure-oriented approach. If its intended benefits can be achieved, we will have a chance to reformulate the Operator-Oriented Representation scheme in light of new findings from the empirical research. In Section 3, we will discuss the *15 Puzzle Problem* which is a form of *SBP* presented in Section 2.3. Since it is a simpler problem than other kind of Metric-Operator Class problem (e.g.: *PSG*), we decided to use this problem as our first empirical research. At the same time, we will continue discussion on *ODGA*. Note that we will only present a special case of *Operator-Oriented DGA*(**ODGA**), *ODGA* for *SBP*. This is because we have not applied *ODGA* to other kind of problems in the Metric-Operator Class. Thus, we do not have a clear idea of the general form of *ODGA*.

---

[5] it is important that the size of Σ should not be prohibitively large compared with the size of S. If so, this massive enlargement of size will effectively cancel out any gains obtained by this representation scheme. This situation is quite analogous to that of Relaxation Method discussed previously.

3 Sliding Block Puzzle

3.1 The Review of Distributed Genetic Algorithm (DGA)

The Genetic Algorithm operates on a population $P = (p_1, \ldots, p_m)$ where m is the size of the population. The individual $p_i (i = 1, \ldots, m)$ of P is an element of the search space S on which the search is conducted, that is, each p_i is a candidate solution. The representation of each p_i can be in many different forms. It can be an array, a tree, a linked list or a graph. Here, it will be defined as a 1-dimensional array of size l (l is any positive integer) for the simplicity of argument with l the same for all p_i. Each position of these arrays will be filled with values of a certain range. Typically, integers are used. They can be real numbers as well. Individual p_i's of the population P interact with one another using two genetic operators, **cross-over and selection**, or else they modify themselves in isolation using the third operator, **directed mutation (local improvement)**. Each operator is designed to improve the population by applying itself to individual(s). The population should have a tendency to contain better individuals after the application(s) of operators. This trend of improvement toward a better population is statistical, i.e., does not strictly hold after each application. The whole algorithm is described in Figure 8. Figure 9 describes the three genetic operators. For details of DGA and its genetic operators, refer to [16].

Unlike many other search algorithms, DGA keeps a population of candidate solutions, each representing many different regions of search space. Through the genetic operators,

Setting:

m processors, each of which contains an individual;
processor P_i has an individual p_i for $i = 1, \ldots, m$;
$p_1, \ldots, p_m$ form a population $\mathcal{P}$.

Processor P_i executes the following using its locally stored individual p_i. They all run in parallel. In case of selection and cross-over, P_i accesses, via built-in communication network, p_j in another processor P_j. If more than one processor tries to access p_j at the same time, one of them has to wait.

```
initialize();  (comment: initialize a local individual)
while (termination condition is not satisfied) do
{
  selection();
  cross-over();
  directed-mutation();
}
```

Figure 8: **the Description of the DGA**

For the simplicity of presentation, the representation of each individual p_i in the population $\mathcal{P}$ is a bit vector and any combination of bits can constitute a valid representation (minimal constraint).

```
cross-over : Given an individual $p_i$, choose its mate $p_j$ where $p_i = (p_{i1}, \ldots, p_{il})$, $p_j = (p_{j1}, \ldots, p_{jl})$.
    construct their offspring $q$ as follows where $q = (q_1, \ldots, q_l)$:
    copy $p_i$ to $q$;
    (comment: $C_{freq}$ is a parameter provided from outside. It specifies how large a portion of $p_j$ will be incorporated into $p_i$ to yield $q_k$)
    for $i = 1$ to $C_{freq}$ do {
        randomly generates a number $k$ between 1 and $l$;
        $q_k = p_{jk}$;
    }
    The output of cross-over is $q$, which replaces $p_i$.
```

```
directed-mutation : Given an individual $p_i$, modify it as follows
    (comment: $M_{freq}$ is a parameter provided from outside. It specifies how large a portion of $p_i$ will undergo the directed-mutation.)
    for $i = 1$ to $M_{freq}$ do {
        mutate $p_i$ to $\hat{p_i}$ ;
        if $\hat{p_i}$ is better than $p_i$
            then replace $p_i$ with $\hat{p_i}$;
        else keep $p_i$;
    }
```

```
selection : Given an individual $p_i$, choose its opponent $p_j$ randomly
    if $p_i$ is better than $p_j$
        then copy $p_i$ into $p_j$;
    (comment: otherwise do nothing)
    (note: it is assumed that a routine or procedure exists to determine, given two individuals, which of the two is a better individual [a].)
```

---

[a] Relative comparison between individuals is enough. There is no need for evaluation measures which return a number as a performance score of the individual.

Figure 9: Three Genetic Operators

the representatives from unpromising regions (i.e., those with poor performance) will be eventually eliminated (**selection**), and those from unexplored regions will be generated (**cross-over, directed mutation**). DGA does not narrow its attention to any small region. Instead, it always keeps a broad overview of the whole search space by maintaining many candidate solutions in a population. As the search progresses, the scope of the search is eventually reduced to smaller regions in which many good candidate solutions have been found. This way, the possibility of being trapped in a local optimum will be greatly reduced. This is why DGA is capable of reaching a global (near)-optimum.

3.2 The Sequential Simulation of DGA

The sequential version of DGA can be defined. Even though the sequential version may not perfectly simulate the real parallel DGA[6], it is much easier to program and maintain than the parallel version. Thus, it is a useful temporary step before a program is implemented in *Butterfly* or other *MIMD* parallel machine. Its purpose is to find out if a given problem can be efficiently solved by using DGA. If so, we can convert this sequential version into the parallel version in a straightforward manner[7][16].

Next, the description of Sequentiual Simulation of DGA (*SSDGA*) will be presented.

---

[6] The parallel version is an asynchronous algorithm while the sequential version is synchronous. However, this difference affects the performance only marginally.

[7] Ease of parallelization makes this *ODGA* implementation of *SBP* so attractive

```
initialize(pop);  (comment: initialize the population)

while (termination condition is not satisfied) do
{
  selection-loop(pop, newpop);
  cross-over-loop(pop, newpop);
  directed-mutation-loop(pop, newpop);
}
```

Figure 10: **Sequential Simulation of DGA**

3.2.1 Program Organization

The program is built around two main arrays for population, pop and `newpop` whose size is m. Only one of them might be enough, but we use two for the sake of the efficiency of the program. Genetic operations are done by three loop modules, `select-loop`, `cross-over-loop`, `directed-mutation-loop`, each executing one of three genetic operators.

The layout of *SSDGA* is shown in Figure 10, and the description of three loop modules appears in Figure 11.

3.3 The Description of the Sliding Block Puzzle (SBP)

Consider the initial board of the puzzle shown in Figure 12 and let the board shown in Figure 13 be a goal board (the empty tile is represented by the number 0).

The objective of the Sliding Block Puzzle[8] is to reach the goal board starting from the comment:

---

[8] This classic board game, sometimes called boss puzzle or 14–15 puzzle, was created by Samuel Loyd

1. **marked**: the one-dimensional array **marked** of size **n** is used to record which individual will overwrite which. at the start of **select-loop**, it is initialized to -1's. Using this array, we can ensure that no individual would have more than 2 copies of itself in the whole population.
2. `rand()`: a function which returns a random real number between 0 and 1.
3. `random(n)`: a function which returns a random integer between 0 and n-1.

```
select-loop(pop, newpop)
{
  for idx = 0 to m-1 do {
    inx = random(m);
    if pop[idx] is better than pop[inx] {
      newpop[inx] = pop[idx];
      marked[idx] = inx;
    }
  }
  for idx = 0 to m-1 do
    if (marked[idx] >= 0)
      copy newpop[marked[idx]] into
      pop[marked[idx]];
}

cross-over-loop(pop, newpop)
{
  for idx = 0 to m-1 do
    if (rand() < probability-of-cross-over {
      inx = random(m);
      cross-over pop[idx] with pop[inx] and
      produce newpop[idx];
      copy newpop[idx] to pop[idx];
   }
}

directed-mutation-loop(pop, newpop)
{
  for idx = 0 to m-1 do
    apply directed mutation to pop[idx]
    and produce newpop[idx];
  swap pop and newpop (copy newpop into pop);
}
```

Figure 11: Three Loop Modules of SSDGA

1	2	3	4
5	6	7	8
9		10	11
12	13	14	15

Figure 12: **An initial board**

1	2	3	4
6	9		8
5	10	7	11
12	13	14	15

Figure 13: **A goal board**

initial board using a sequence of valid moves. Given an empty tile position, there are at most 4 moves possible:

`L: move the empty tile to the left.`

`U: move the empty tile upwards.`

`R: move the empty tile to the right.`

`D: move the empty tile downwards.`

The only precondition required for applying a move is that it should not move the empty tile out of the board. For example, a sequence of moves which transforms the board shown in Figure 12 into the board shown in Figure 13 is (`L, U, R, D, R, U`).

In this paper, we will assume that the length of a solution sequence for a given SBP is

---

about 1878.

known in advance. Thus, the exact formulation of the problem is:

> *Given an initial and goal board arrangement and the length of the solution move sequence, find the move sequence.*

In other search algorithms such as A*, the length of a solution sequence is not known in advance. We think that this problem can be taken care of once we are successfully implement **ODGA** on *SBP*. Therefore, in our implementation, the solution length will be provided to ODGA. The reason we need the solution length to solve *SBP* is that we do not have to implement the mechanism to guess the solution length. We can impose a certain range around the given length, then whenever cross-over or directed mutation creates a new sequence, its length is within the specified range. This will simplify the implementation, because now, we are interested in finding out whether *SBP* can be effectively solved. The success of this restricted version will enhance the chance of full-fledged ODGA for *SBP* with unknown solution length.

In the following subsections we will show more in detail how this problem is actually implemented in **GA**.

3.4 The Description of ODGA for SBP

3.4.1 Representation

Sequences of operators[9] will make up the population which will be evolved by *DGA*. All sequences in a population don't have to be the same length. Also a sequence can shrink and be extended as will be explained in Section 3.7.3. This presents some complications for DGA because DGA has to prevent the length of sequences from wildly changing. Again, details on this subject will be discussed later. There are two different ways of representing an operator sequence, **LOCATION INDEPENDENT SCHEME** and **LOCATION DEPENDENT SCHEME**.

Location Independent Scheme

Each sequence in a population is represented as a string of 0, 1, 2 and 3. These numbers denote moves L, U, R and D, respectively. The precondition for each move is that the move should not push the empty tile off the edge of the board. The leftmost number in a sequence is the first move to be made from the initial board and the rightmost, the last. The above problem is solved by a sequence with alleles 0, 1, 2, 3, 2 and 1, in this order. The position of a certain allele in a certain location of the sequence is most important in this representation scheme.

---

[9] Note that these operators are the moves that can be made on the board: `L, U, R, D`.

Location Dependent Scheme

Each move which makes up a sequence now takes the form of $((x, y), o)$ where (x, y) denotes the location from which the move originates, and o is one of `L`, `U`, `R`, `D`. (x, y) means that the location is x rows away from the top row and y columns away from the leftmost column. Thus, the solution for the problem is :

((2, 1), 0) ((2, 0), 1) ((1, 0), 2) ((1, 1), 3) ((2, 1), 2) ((2, 2), 1)

Here, the precondition is different from that of the first scheme above. Not only should the move make the empty tile stay in the board, but the location (x, y) should also be the location of the empty tile before the move is applied. This additional precondition will play a role in defining GA operators. As a result, this definition is different from that of the location independent scheme. As will be explained later in Section 3.4.1, this will reduce the context-sensitivity of the representation scheme. Empirical results show that this scheme is better than the first one (See Section 4). This is used in the results discussed in the later part of this report.

3.5 Main Module

The layout of *ODGA* for *SBP* is no different from that of standard Genetic Algorithm or DGA. The difference is in the representation scheme (Operator-oriented scheme) and two genetic operators which manipulate these represented individuals in the population.

Currently our *ODGA* program is implemented and tested in a sequential machine (Vax

8800). If this sequential version turns out to be a success, we can easily convert it into a parallel version which can run in **Butterfly** Machine which is a *MIMD* machine that can have up to 256 processors running in parallel. If it succeeds, it implies that we can have a highly effective parallel algorithm which can be used for complicated problem-solving tasks. This point will be elaborated in Section 3.7.4.

3.5.1 Parameters

There are some adjustable parameters for fine-tuning the program to the specific problem presented.

LENGTH Length of an initial structure. This is usually set to the number close to the *Manhattan distance.*

MAXLEN The maximum length a structure can grow. If a structure reaches this length when crossover or mutation is being performed, the rest of the alleles in the parent are lost.

MINLEN The minimum length a structure can shrink. While doing crossover, if a structure reaches this length, the operation is aborted and a new crossover is started all over again.

GENERATION The maximum number of generations when evolution stops. Nature doesn't stop, but we have to have some maximum number for our system in order to avoid an infinite loop.

POPNUM The number of structures in a population.

CPOR Crossover rate. The percentage of the population which undergoes the crossover.

MUT-RANGE Mutation range. The number of alleles to be mutated.

3.5.2 Termination Condition

The program terminates when it finds a solution to the problem or it reaches the prefixed total number of evaluations(GENERATION), in which case an approximate solution is found.

3.5.3 Initialization

In the initialization phase, several problem specific and move generating variables, e. g. boards, data structures etc. , are initialized and the initial population is chosen randomly.

An initial structure is generated as follows. Starting from the initial position, legal moves are generated. After a certain number of move generating steps (LENGTH), this process is stopped, and the structure is cut at the allele point where the best result is produced (getbest). Next, the gap between the blank space position of the current board and that of the goal board is bridged by adding more moves (append)[10], yielding one complete structure for the evolution.

---

[10] The two operations, getbest and append, are used in crossover and mutation, too.

The above processes are repeated until a desired number of structures (POPNUM) are produced in a population.

3.6 Evaluation

In order to apply genetic algorithms to SBP, we need to formulate the problem as a function optimization problem.

In order to define *f*, the function to be optimized, we need to introduce some extra notation. We will denote the initial board by `IB` and the goal board by `GB`. Let $(x_1, \ldots, x_n)$ be a sequence of valid moves, we denote by $\mathtt{IB}(x_1, \ldots, x_n)$ the board which is obtained by applying the sequence of moves $(x_1, \ldots, x_n)$ to `IB`.

Consider the boards $\mathtt{IB}(x_1, \ldots, x_n)$ and `GB`. For each tile (except the empty tile) in $\mathtt{IB}(x_1, \ldots, x_n)$, compute the *Manhattan distance* between the tile's position in $\mathtt{IB}(x_1, \ldots, x_n)$ and its position in `GB`. We define the *performance*$(x_1, \ldots, x_n)$ as the sum of all these Manhattan distances.

The evaluation measure function[11] is

$$f(x_1, \ldots, x_n) = min\{performance(x_1, \ldots, x_i) | 1 \le i \le n\}$$

i.e. , the value of a structure $(x_1, \ldots, x_n)$ is defined as the performance of the subsequence$(x_1, \ldots, x_i)$ whose corresponding intermediate board $\mathrm{IB}(x_1, \ldots, x_i)$ comes closest to GB. It should be clear that whenever $f(x_1, \ldots, x_n) = 0$, the sequence $(x_1, \ldots, x_n)$ contains a sub-sequence $(x_1, \ldots, x_i)$ which is a solution to the SBP.

---

[11] The appearance of the definition of f suggests that it takes $O(n^2)$ (square) time to compute it. However, this can be computed in $O(n)$ (linear) time.

3.7 Genetic Operators

3.7.1 Selection

The selection operator is the same both in **Location Independent Scheme** and in **Location Dependent Scheme**.

3.7.2 Genetic Operators under Location Independent Scheme

Cross-Over: The crossover process here is similar to that in the TSP. Suppose that two sequences of operators are given.

1. We pick the first operator from each sequence.
2. Apply each operator to the initial board to see which operator yields a new board closer to the goal board.
3. Choose with high probability the operator which yields the closer board, i.e., the one with the better performance. Notice that here we employ heuristic information about the Sliding Puzzle Problem. Indeed, it seems likely that we should try to obtain intermediate configurations that get closer to the goal board. We do not

always choose the better operator, however, because this may eventually lead to a bad sequence whose performance we can not improve as long as it starts with that particular operator, meaning, we could get stuck in the local optimum. However, it is our assumption that in general, it is more likely the case that selecting the better operator will contribute to constructing a good sequence. In case the two operators have the same performance, pick either of them randomly. Once the operator is chosen, it becomes the first operator of the new sequence and the board is updated accordingly.

4. Now we pick the second operators of each sequence. Again, we will take the one with the better performance. It may however be the case that one or both of them is no longer legal, i.e., it pushes the empty tile off the edge of the board. This is possible because the operator chosen for the new sequence is not necessarily the one which preceded the current two operators. In case only one of the operators is illegal, choose the one which is legal. Otherwise, randomly generate a legal one. It becomes the second operator of the new sequence. Again update the board.

5. This process is repeated until we reach the end of the two sequences.

Directed Mutation: The directed mutation process is performed on a single structure.

1. Randomly pick m positions $(0 \leq m \leq n)$ in the sequence. For the left-most position, make the corresponding board arrangement by applying the operators in the sequence

preceding this position.

2. Randomly generate a legal operator in that position.

3. Check if this new operator is acceptable by comparing it with the old operator using the boltzman distribution test (i.e., perform simulated annealing). This test goes as follows: Accept the new operator if it yields a board closer to the goal board than the old one does. Otherwise, accept it with the probability according to a boltzman distribution. If the temperature in the boltzman distribution is high, the new operator will be accepted with high probability, even if its performance is bad. If the temperature is low, the operator is accepted with less probability. (Temperature decreases exponentially. We chose the temperature $T = T_0 \rho^{gen}$, where T_0 is a initial temperature, $0 < \rho < 1$ and gen is the number of generations).

4. If the new operator is accepted, it replaces the old one in the sequence. If not, the old one is kept.

5. Now proceed to scan the given sequence to the right until the second initially chosen position at which directed mutation will be performed is reached, checking if, along the way, any of the operators should be updated due to the replacement of the previous operator. If an operator has to be updated, replace it with a legal one.

6. Repeat this process for all the initially chosen positions at which directed mutation will be performed.

7. Upon completion of this process, the whole new sequence is compared with the initial sequence and accepted according to the boltzman distribution test.

Problems with Location Independent Scheme:

It appears that, while this scheme has fewer preconditions for each move, the mechanism of cross-over and directed mutation violates the essence of Genetic Algorithm. That is, the thrust of GA is that it combines good building blocks from two individuals (cross-over) and improves an individual by replacing the bad parts of an individual with good ones (directed mutation).

It is important that good buiding blocks remain generally good ones even if they are used to construct new operators. In the case of *SBP*, building blocks or parts are segments of operator sequences. If the goodness(badness) of these building blocks depends too much on what moves are made in other parts of the entire operator sequence, cross-over and directed mutation can hardly effective. A good part in one sequence may turns out to be dismally bad as a part of another sequence.

From this point of view, the Location Independent Scheme can cause too much context-sensitivity. An operator (or move) taken from one sequence can be used to create a new sequence. However, the role of an operator (move) is dramatically changed if a move is applied in a different location. The `U` move from the location $(2, 1)$ can have an entirely different effect on the performance of a sequence than the `U` move from $(1, 2)$. See Figure 16 for an example.

To prevent the effect of each move or a segment of move sequence from being heavily context-sensitive, we need to consider the location to which the move is applied as well as the move itself. This motivated the Location Dependent Scheme. It will not remove the context-sensitivity entirely, but, it appears to be less context dependent than the Location Independent Scheme, and the actual implementation shows just that. Refer to Section 4 for details.

3.7.3 Genetic Operators under Location Dependent Scheme

Crossover:

1. A random number between 0 and 1 is generated and tested to see if it is less than the predetermined threshold ($0 \leq$ threshold < 1), CPOR. If so, crossover is performed, otherwise skip to the next individual.

2. In the actual crossover routine, a individual is randomly chosen for crossover with the target individual. Then a random position is chosen in each parent for the crossover. It still isn't possible to exchange the segments to the right of the crossover point as is the case with most simple GA implementations, since the move to be applied next may not be applicable to the current board. The Location Dependent Scheme requires that the location of the empty tile, after a subsequence of operators to the left of cross-over point of the one sequence is applied, should be the location where the first operator of the subsequence to the right of cross-over point of the other sequence

starts from (see Figure 17).

3. To make it work, the location of the blank tile at the designated positions in each parent is determined and the random moves are generated to bridge the gap between those locations.

   The child is made up of the first portion of target sequence (individual), the bridging moves, and as much as of the second portion of the chosen sequence so that the length of the child does not exceed MAXLEN.

4. All of the above steps are wrapped up in a higher level loop to ensure the child's length is at least MINLEN. If not, it continues trying new crossovers until a legal child is produced.

5. After a new child is born, to make sure that the empty tile reaches its location in the goal board, it goes through two more steps, getbest and append as explained in Section 3.5.3, yielding a final version of a new structure.

Directed Mutation: Mutation is applied to each structure in the new population after selection and crossover.

1. A starting position for the mutation is chosen randomly, the ending position is calculated using mutation range (MUT-RANGE) and the new path between these positions is randomly generated.

2. The child is made up of the moves up to the starting position, the new moves, and as much of the remainder of the target sequence(individual) starting at (`the ending position + 1`), without the length exceeding MAXLEN (see Figure 18).

3. After it is cut at the best position and the gap between its empty tile position and goal board's empty tile position is bridged as the case of crossover, the child is accepted according to the Boltzmann distribution (for more on simulated annealing, see [8]) in Figure 19 to avoid premature convergence.

Variable Length of Sequences (individuals): In both crossover and directed mutation, the length of a new sequence can be different from the old one(s). This is due to the following reasons:

First, when cross-over points are chosen, they are done randomly. Since a new child sequence is made by connecting two subsequences from its parents via randomly generated bridge of moves, quite conceivably the child could have different length. Second, in the directed mutation, a segment of the sequence is replaced with the randomly generated moves. Since there is no such requirement that these two segment should have the same length, directed mutation can change the length of the sequence. It should be also noted that the mutation can occur at the beginning or the end of a sequence. The length of each sequence(individual) in the population is maintained between *MINLEN* and *MAXLEN*. *MINLEN* and *MAXLEN* are determined by the value of the length of the solution sequence known at the beginning of the program

3.7.4 Possible Advantage of ODGA as a Problem Solver

One may wonder what possible gains can be made by the application of GA to solve this class of problems other than just to prove that GA can really be used for more than combinatorial optimization tasks. There is at least one other potential benefit. In [16], it has been shown that GA can run in parallel with **almost no sequential overhead**. The resulting computing time of the parallel GA, the DGA, can be far shorter than any other competing sequential search algorithms. This tendency may not be true for problems of small size. However, as the size of problem grows larger, DGA is shown to outperform other alternate sequential algorithm. Thus, GA application can be a quite efficient search scheme for solving the problems of the above class which are of significantly large size. For example, it may solve the 4 by 4 Sliding Block Puzzle requiring more than 100 operator applications far faster than any sequential search. This provides us with a search scheme suited to complex large scale problems, which has been a difficult task using known search techniques such as A*.

4 Experiments and Results

We now describe how our GA has been applied to solve the actual Sliding Block Puzzle. The algorithms were coded into *Chez Scheme* and were run on VAX-8800. In the experiments reported here, we used 3 × 3 and 4 × 4 puzzles. The results of experiments show much improvement over our initial reports [15].

4.1 3 × 3 puzzles

We presented 12 different problems to the system. The moves needed to solve them range from 5 to 40. 4 of the 12 problems are either shown in the literature [10, 14] or hand-generated. The rest are machine-generated from a given goal board as shown in Figure 20 by applying the moves backwards until we obtain a board after the predetermined number of moves (5 to 40 increased by 5, which makes 8 different boards), which we used as initial boards. One of the initial boards generated this way is also shown in Figure 20. Using very little computation time (some solutions were found from the initial population, and at most 36 generations were needed for the most difficult puzzle), we always found a solution using the following parameter settings: initial population = 20 for the easy puzzles, and 50 for the difficult one (see Figure 21), initial length of a structure = 12 for the easy, and 30 for the difficult, crossover rate = 70 %, and mutation range = 3. In Figure 21 we show one "difficult" problem introduced by Martin Gardner [14]. Up until 1965 the best solution on record required 36 moves. Later, computers were used and they found 634 methods that required less than 36 moves[14].

```
 10 methods requiring 30 moves
112 methods requiring 32 moves
512 methods requiring 34 moves
```

We ran our system 5 times for this problem. It always produced solutions: 2 times with 30 moves, one with 32, 36, and 38 moves, respectively.

4.2 4 × 4 puzzles

In the case of 4 × 4 Sliding Block Puzzles, we often found solutions, and other times we found near-optimum solutions. We presented 13 different problems to the system and the moves needed to solve them ranged from 20 to 40. As in the case of 3 × 3 puzzles, 8 were chosen from the literature[14] and the rest were generated by the program from the goal board shown in Figure 22 (required number of moves range from 20 to 40). The reason why we generated some problem sets this way, rather than making them randomly, is that it is shown only one-half of the 20 trillion (16!) possible permutations the pieces can assume are possible from any beginning position 9. Out of 55 runs (5 runs for each genuine problems, 3 for machine-generated problems) we could find 18 correct answers using the following parameters: population size = 50, initial length of a structure = 20, number of generations = 75, crossover rate = 60 %, and mutation range = 3. The boards we reached (for the cases unsolved) are within 4 to 5 in Manhattan distance from the goal configuration, hence the majority of tiles were in place.

In Figure 23 , we show a case of an unsolved puzzle and a board reached by a GA. For this problem, the Manhattan distance is 28 and the final board has a Manhattan distance of 5 from the goal board. For the puzzle described in the paper presented to IJCGA-87[15], we could find 2 solutions out of 5 trials. Both involved 41 moves (see Figure 24.) Overall, we made progress solving 4 × 4 puzzle, yet rooms for improvement remains.

5 Conclusion

We think that the current implementation of *ODGA* for *SBP* has trouble in solving *15 Puzzle* problems which require quite a long solution sequence because the routines which generate moves to bridge the gap (cross-over) or replace the old segment (directed mutation) cannot cover all possible segments of moves. The generation routine is too simple at the moment because It only generates step-like moves (see Figure 25). Attempts should be made to eliminate this problem and that would be our next immediate objective

In ODGA, the representation of individuals, a sequence of operators, is theoretically unbounded in its size. For example, in the *Sliding Block Puzzle*, it is usually not known in advance how many moves it will take to reach a goal board arrangement from an initial board arrangement. Eventually, *ODGA* for *SBP* needs to have a mechanism to make a guess on the size of the solution path and to make adjustments, if its initial guess fails. This will usually take the form of extending the maximum length for all paths (operator sequence) in a population. The extending mechanism has not been implemented yet in any ODGA program.

In most *SBP* problems, the task is to find a solution path which will lead the initial board to the goal board. The application of GA in this situation can exploit a important piece of information which is not usually available in many GA optimization task: the global optimum (or minimum) is already known. It is obviously 0. Whenever a search by ODGA produces a local optimum, it is easy to find if the local optimum is also a global one. If it is 0, it is a global optimum. Otherwise, it is not. As long as it is computationally inexpensive to find whether a given structure is a local optimum (minimum), it is easy to escape from local optimum traps because the local optimum is easy to detect. If this can be implemented, it will serve as a rough equivalent of the backtracking mechanism in known sequential searches.

As we said in Section 3.7.4, *ODGA* has a potential to be a quite effective problem solver for large scale search problems. This is recognized from the inherent parallelizability of evolutionary processes of the nature from which *ODGA* has originated. Many sequential algorithms for problem-solving task are not quite amenable to parallelization. They either create too much sequential overhead or the factor of parallelization is far less than linear with respect to the number of parallel processors [4].

Ultimately, the success of this project will provide the new search scheme, which is inherently parallel and thus can be quite efficient. We may add other operations to the repertoire of genetic operators, thus making *ODGA* more sophisticated and intelligent. One possibility is the addition of a known sequential search algorithm such as sl A*. Since *ODGA* is a random stochastic algorithm, it may waste time when many members of its population are close to the solution. When a sequence is a few moves away from the solution, a deterministic sequential search is more effective. Thus, when *ODGA* generated a solution sequence s whose metric score $d_G(s)$ is quite close to 0, the program should switch its search operation from crossover and directed mutation to a deterministic sequential search (such as A^*). This way, each processor is equipped with genetic operators and a sequential search operation, and exploits two different modes. This can constitute a powerful search scheme which make use of two concepts: **parallel stochastic cooperative search**, and **sequential deterministic search**.

References

[1] Coombs, Susan and Davis, Lawrence (July 1987), "Genetic Algorithms and Communication Link Speed Design: Constraints and Operators", Proc. of the 2nd Int'l Conf. on Genetic Algorithms and Their Applications, pp. 257-260.

[2] Davis, Lawrence (July 1985), "Job Shop Scheduling with Genetic Algorithms" Proc. of an Int'l Conf. on Genetic Algorithms and Their Applications, pp. 136-140.

[3] Davis, Lawrence and Coombs, Susan (July 1987), " Genetic Algorithms and Commu-

nication Link Speed Design: Theoretical Considerations", Proc. of an Int'l Conf. on Genetic Algorithms and Their Applications, pp. 252-256.

[4] Finkel Raphael A., Fishburn, John P. "Parallelism in Alpha-Beta Search" Artificial Intelligence, 19(1). September 1982, pp 89-106

[5] Fourman, Michael (July 1985), " Compaction of Symbolic Layout Using Genetic Algorithms", Proc. of an Int'l Conf. on Genetic Algorithms and Their Applications, pp. 141-153.

[6] Greene, David Perry and Smith, Stephen F. (July 1987), " A Genetic System for Learning Models of Consumer Choice", Proc. of the 2nd Int'l Conf. on Genetic Algorithms and Their Applications, pp. 217-223.

[7] Grefenstette, J. J., Gopal R., Rosmaita, B. J. and Van Gucht, D. (July 1985), "Genetic Algorithms for the Traveling Salesman Problem", Proc. of an Int'l Conf. on Genetic Algorithms and Their Applications, pp. 160-168.

[8] Kirkpatrick, S. , C. D. Gelatt and M. P. Vecchi (1983). "Optimization by Simulated Annealing", Science. Vol. 220 (4598), pp. 671-680.

[9] Loyd, Samuel (1976). *Sam Loyd's Cyclopedia of 5000 Puzzles, Tricks and Conundrums with Answers.* New York: Pinnacle Books.

[10] Rich, Elaine (1983). Artificial Intelligence. New York: McGraw-Hill.

[11] Raghavan, Vijay V. and Agarawal, Brijesh (July 1987), "Optimal Determinations of User-Oriented Clusters: an Application for the Reproductive Plan", Proc. of the 2nd Int'l Conf. on Genetic Algorithms and Their Applications, pp. 241-246.

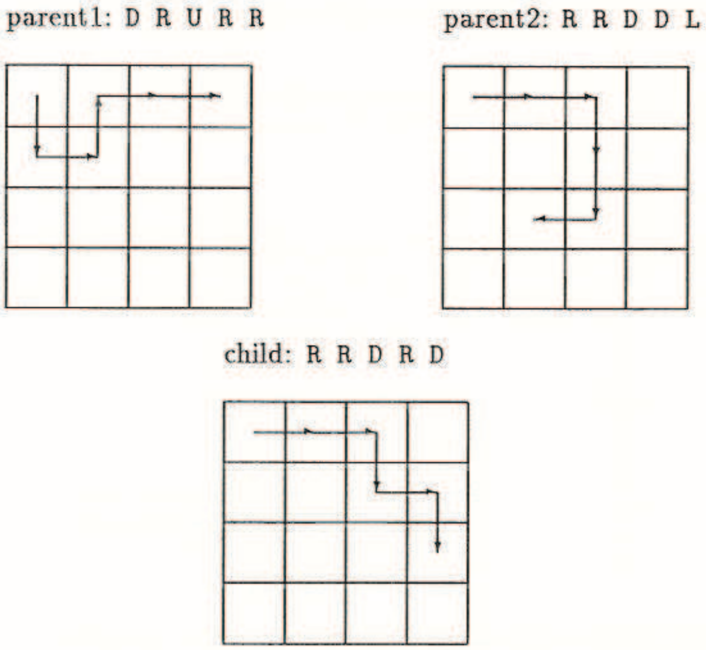
[12] Sirag, David. J. and Weisser, Paul T. (July 1987), "Toward a Unified Thermodynamic Genetic Operator", Proc. of Int'l Conf. on Genetic Algorithms and Their Applications, pp. 116-122.

[13] Smith, Derek. (July 1985), "Bin Packing With Adaptive search", Proc. of an Int'l Conference on Genetic Algorithms and Their Applications, pp 202-206.

[14] Spencer, Donald D. (1975). Game playing with computers. Rochelle Park, New Jersey: Hayden Book Company.

[15] Suh, J. Y. and Van Gucht, Dirk (July 1987), "Incorporating Heuristic Information into Genetic Search", Proc. of a Second Int'l Conference on Genetic Algorithms and Their Applications, pp. 100-107.

[16] Suh, J. Y. and Van Gucht, Dirk (July 1987), "Distributed Genetic Algorithms", Technical Report No. 225, Indiana University.



parent1: D R U R R

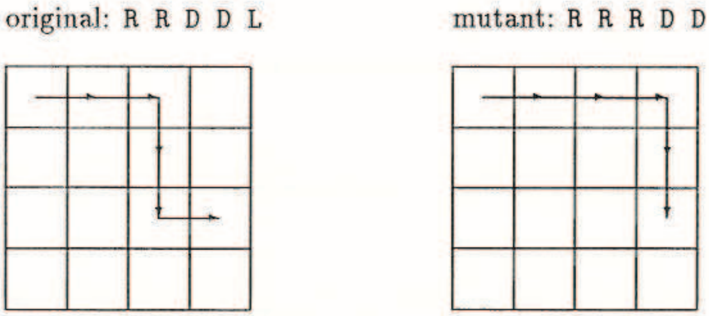
parent2: R R D D L

child: R R D R D

How a child is constructed

position	parent1	parent2	child	current board	reason
1	D	R	R	I	$\delta_G(\texttt{D}) < \delta_G(\texttt{R})$
2	R	R	R	$I \cdot \texttt{R}$	R is only choice and is also applicable to the current board.
3	U	D	D	$I \cdot (\texttt{RR})$	U is illegal on the current board, but D is legal.
4	R	D	R	$I \cdot (\texttt{RRD})$	$\delta_G(\texttt{RRD} \cdot \texttt{R}) < \delta_G(\texttt{RRD} \cdot \texttt{D})$
5	R	L	D	$I \cdot (\texttt{RRDR})$	R is illegal on the current board. L offsets the preceding R. This move is discarded for the sake of efficiency.

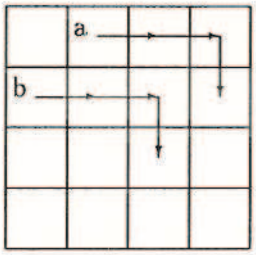
Figure 14: **Cross-Over: location independent version**


original: `R R D D L`

mutant: `R R R D D`

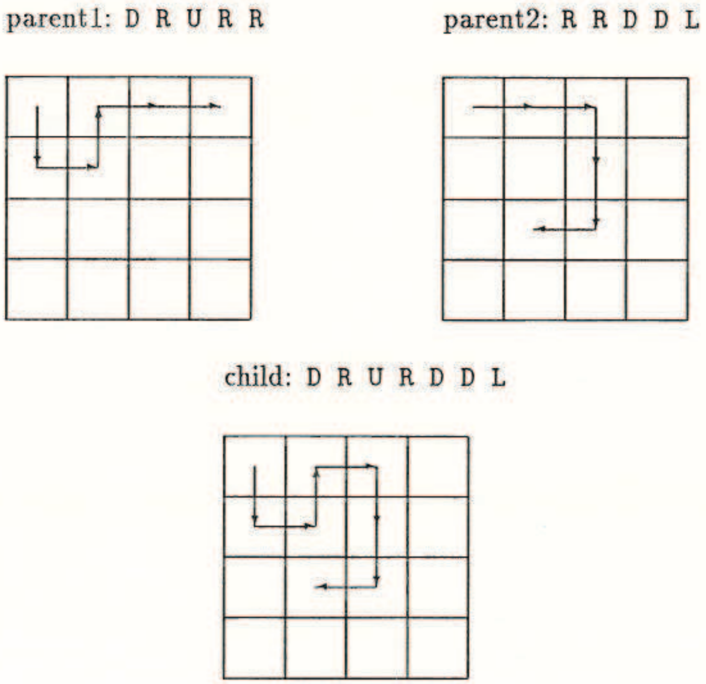
1. Pick the 3rd operator `D` of the original for mutation.
2. Mutate it to `R`.
3. If $\delta_G(\texttt{CB}) \cdot \texttt{R} < \delta_G(\texttt{CB} \cdot \texttt{D})$ where `CB` is the current board (that is, $\texttt{CB} = I \cdot (RR)$), the 3rd operator will be `R`.
4. If the 3rd move is `R`, the 4th move of the original `D` is still legal. But, the 5th one is no longer legal on its current board. Thus, pick a new one randomly (in this case, `D`).

Figure 15: **Directed Mutation: location independent version**



The sequences a, b in the above transforms the board configuration differently, even though the operator sequence as moves is identical. This is because they are applied at different locations on the board.

Figure 16: **The Context Sensitivity**



parent1: D R U R R

parent2: R R D D L

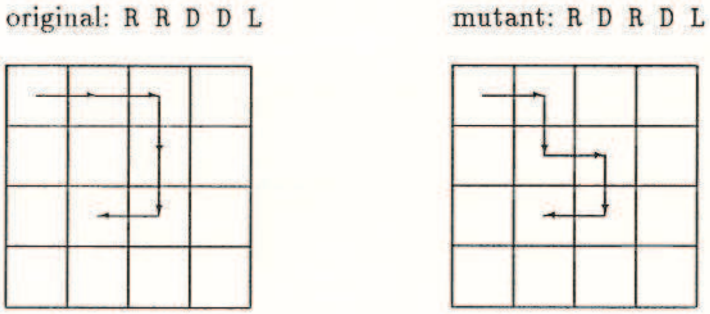
child: D R U R D D L

(a) Get first 3 moves (D R U) from parent1.

(b) Get last 3 moves (D D L) from parent2.

(c) Join two subsequence by a move R. that is, (D R U)(R)(D D L) = (D R U R D D L). **Note** that the locations of operators from either parent remain unchanged.

Figure 17: **Cross-over: location dependent version**



1. pick the 2nd and the 3rd moves for the mutation.
2. replace them with moves D R, yielding the mutant.
3. note that the location of the other moves remain unchanged.

Figure 18: **Directed Mutation: location dependent scheme**

```
boltzmann(old, new, temperature)
{
  return((old>new) or
         (rand()<exp((old-new)/temp)));
}
```

In the experiments reported below, the following annealing schedule is used.

$$\texttt{Temperature} = (\texttt{initial average deviation}) \times (0.975^{gen})$$
$$\texttt{where gen denotes the generation}$$

Figure 19: **Boltzman Distribution**

1	2	3
8		4
7	6	5

2	8	3
1	6	4
7		5

Figure 20: One of the goal boards and an initial board generated from it

8	7	6
5	4	3
2	1	

1	2	3
4	5	6
7	8	

Figure 21: The initial and the goal board introduced by Gardner

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

1	11	10	6
5	8	7	2
9		4	3
13	14	15	12

Figure 22: One of the goal boards for 4 × 4 puzzle and an initial board generated from it

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

1	2	3	4
12	13	14	5
11		15	6
10	9	8	7

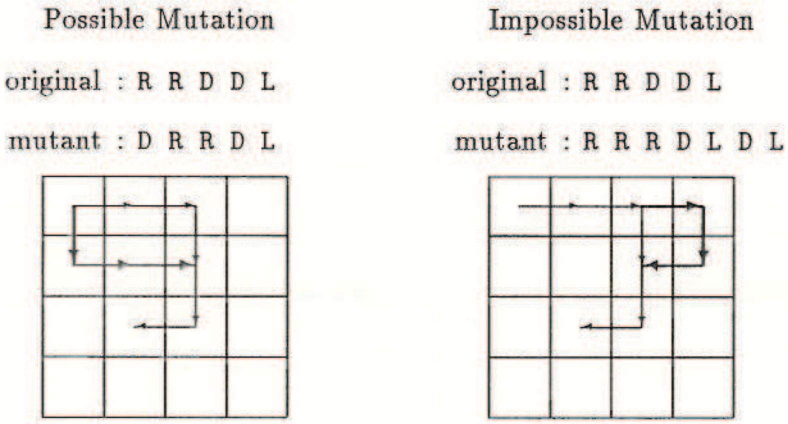
1	2	3	4
12	13	14	5
	9	11	6
10	15	8	7

Figure 23: The initial and the goal board of a 4 × 4 SBP and a board obtained by a GA after 36 moves

1	2	3	4
5	6	7	8
9		10	11
12	13	14	15

10	1	6	2
3	8	15	4
9	7	13	11
5		12	14

Figure 24: The initial and goal boards presented from Suh's paper



Possible Mutation

original : R R D D L

mutant : D R R D L

Impossible Mutation

original : R R D D L

mutant : R R R D L D L

Figure 25: **Possible and Impossible Generation**