Exploration and Exploitation Without Mutation: Solving the *Jump* Function in $\Theta(n)$ Time

Darrell Whitley[1(✉)], Swetha Varadarajan[1], Rachel Hirsch[1], and Anirban Mukhopadhyay[2]

[1] Colorado State University, Fort Collins, CO 80523, USA
{whitley,swetha.varadarajan,rachel.hirsch}@colostate.edu
[2] University of Kalyani, Kalyani, Nadia 741235, West Bengal, India
anirban@klyuniv.ac.in

Abstract. A number of modern hybrid genetic algorithms do not use mutation. Instead, these algorithms use local search to improve intermediate solutions. This same strategy of combining local search and crossover is also used by stochastic local algorithms, such the LKH heuristic for the Traveling Salesman Problem. We prove that a simple *hybrid* genetic algorithm that uses only local search and a form of deterministic "voting crossover" can solve the well known *Jump* Function in $\Theta(n)$ time where the jump distance is $log(n)$.

1 Introduction

The *Jump* function is a function of unitation that uses the $OneMax(x)$ function as an intermediate form in the construction of the evaluation function. Functions of unitation [15,16] are pseudo-Boolean functions where all bit string inputs that have the same number of 1 bits have exactly the same evaluation. The *Jump* evaluation function first computes $OneMax(x)$, the number of 1 bits in string x. A "gap" or "moat" is then created that the search must jump across to reach the global optimum, where the global optimum is the string of all 1 bits [1].

This paper proposes a novel approach to solving the *Jump* function. A hybrid genetic algorithm is used that improves the population using local search. The hybrid genetic algorithm also uses a form of multi-parent recombination. This hybrid genetic algorithm easily solves instances of the *Jump* function in $\Theta(n)$ time when the jump distance is $log(n)$. The use of a hybrid genetic algorithm is, in fact, common practice in that part of the evolutionary computation community concerned with real world applications. Furthermore, if local search can provide diversity, there is no need for mutation. Many highly effective evolutionary algorithms do not use mutation. Thus, a meta-level goal of this work is to make theory more relevant to the community as a whole by focusing on hybrid genetic algorithms, the aggressive use of recombination, and the role of diversity in genetic algorithms.

A. Auger et al. (Eds.): PPSN 2018, LNCS 11102, pp. 55–66, 2018.
https://doi.org/10.1007/978-3-319-99259-4_5

2 Background and Basics

Let x denote a bit string, let n denote string length, and let m be width of the gap that must be jumped to reach the global optimum. The $Jump$ function is then defined as follows.

$$Jump_{m,n}(x) = \begin{cases} m + OneMax(x) & \text{if } OneMax(x) \leq (n-m) \\ & \text{or } OneMax(x) = n \\ n - OneMax(x) & \text{otherwise} \end{cases}$$

where $OneMax(x)$ denotes the number of 1 bits in string x. The $Jump_{m,n}$ function is illustrated in the left side of Fig. 1. Another way to think about the $Jump$ function is also shown in the right side of Fig. 1. The $Jump$ function represents a worst case situation in as much as the global optimum is located on a single point surrounded by a moat (the "gap" that must be jumped). Otherwise, the entire landscape is a symmetric hill. We can think of the edge of the "moat" as being a ridge encircling the global optimum, since all points at the edge of the moat have the same evaluation. There are $Choose(n, m)$ points in the search space at the edge of the moat. The local optima at the edge of the moat are not a connected plateau under 1 bit flip, since flipping 1 bit increases or decreases the number of 1 bits in the string. Thus all of the $Choose(n, m)$ points at the edge of the "moat" are distinct local optima. But the same level set of points are a single connected plateau under a 2 bit flip neighborhood operator.

Under "Black Box Complexity" the number of calls to the evaluation function is typically used in place of the true runtime cost. In this paper we consider the true runtime cost. Under normal black box optimization, each execution of the $Jump$ evaluation function takes $\Theta(n)$ time, since one must count all of the 1 bits in string x in order to compute $OneMax(x)$. However, we can also create an incremental, partial evaluation function that will execute in $\Theta(1)$ time.

We create an auxillary function $Jump_{m,n}(x) = Jump1_{m,n}(x'_i, eval(x'))$ where string x is created by flipping the bit x'_i in string x' to generate string x, and $eval(x') = Jump_{m,n}(x')$ stores the evaluation of string x'.

This will allow us to create alternative implementations of $OneMax$ where we invert the $eval(x') = Jump_{m,n}(x')$ function to calculate OneMax(x').

$$OneMax(x') = \begin{cases} Jump_{m,n}(x') - m & \text{if } Jump_{m,n}(x') > m \\ n - Jump_{m,n}(x') & \text{otherwise} \end{cases}$$

We can then create an incremental update to calculate OneMax(x).

$$OneMax(x) = \begin{cases} OneMax(x') + 1 & \text{if } x'_i = 0 \\ OneMax(x') - 0 & \text{otherwise} \end{cases}$$

Lemma 1. *An incremental implementation of the Jump evaluation function can be executed in $\Theta(1)$ time when evaluating $Jump_{m,n}(x) = Jump1_{m,n}(x'_i, eval(x'))$ assuming $eval(x_i)$ is given, and x and x'_i are Hamming distance 1 apart.*

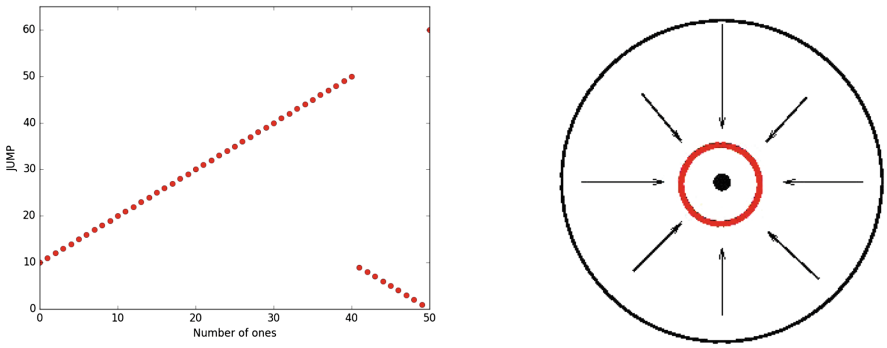

Fig. 1. An instance of the $Jump_{m,n}(x)$ function, with $m = 10$ and $n = 50$ is shown on the left. Another way of visualizing the $Jump$ function is shown on the right. The global optimum is at the center of a hill, surrounded by a "moat". At the edge of the moat (shown in red) are $Choose(n, m)$ local optima under a single bit-flip neighborhood with exactly the same evaluation. For crossover to be effective, there must be a diverse set of solutions distributed along the edge of the moat. (Color figure online)

Proof. Assuming $eval(x'_i)$ is given, then the $Jump_{m,n}(x')$ function can be inverted in order to compute $OneMax(x')$ in $\Theta(1)$ time. $OneMax(x)$ can be computed in $\Theta(1)$ time given x'_i and the $OneMax(x')$ evaluation. The original $Jump$ function now executes in $\Theta(1)$ time given the $OneMax(x)$ evaluation. □

2.1 Jansen's and Wegener's Classic Result

The $Jump$ function suggests two very simple questions: given a multi-modal, nonlinear function, can an evolutionary algorithm jump across short barriers in the search space? Second, can crossover be useful in solving this class of functions, or any other class of functions?

Jansen and Wegener [1] prove that a $(1 + 1)ES$ using a mutation rate of $1/n$ and a gap of m bits has an expected running time of $\Theta(n^m + n\ log\ n)$. They then consider a relatively standard steady state genetic algorithm, with two restrictions. First, they disallow duplicate strings, meaning that the same string cannot occur in the population more than once. Second, the crossover probabilities were unusually small.

When a genetic algorithm moves along a trajectory from a randomly generated population to a location on the edge of the moat, it tends to converge to a small localized region on the edge of the moat because the population has lost diversity. But for crossover to jump across the moat, the population must be diverse. In effect, the population needs to surround the moat.

After the population reaches the edge of the moat, mutation must be high enough to scatter and spread the population around the edge of the moat. At the same time, crossover and selection must be low enough to allow this increase in diversity. Thus, as stated by Jansen and Wegener, only in the final phase of

search is crossover critical. They provide the following expected running time for their genetic algorithm:

$$\Theta(\mu n(m(n)^2 + log(\mu n)) + 2^{2m(n)}/pc)$$

where μ is the population size, and pc is the probability of crossover.

3 Hybrid Genetic Algorithms

The work of Jansen and Wegener was groundbreaking. However, there is still a tendency in the theory community to overly focus on the (1+1)ES, or Holland's Simple Genetic Algorithm.

We consider instead a "hybrid genetic algorithm" or"memetic algorithm" [11] where the population is improved by applying local search to all of the individuals in the population every generation. We will use the "next ascent bit climber" introduced by Dave Davis [4] to the genetic algorithm community. The next ascent bit climber generates a random permutation, then flips the bits in the order indicated by the permutation. The "next ascent bit climber" accepts each improvement as it is found. After every bit has been flipped once, the process is repeated with a new permutation until a local optimum is reached.

In the case of the *Jump* function, the initial population that is improved by local search will either (1) include the global optimum by chance, or (2) reaches the global optimum by improving a randomly sampled string adjacent to the global optimum, or (3) all of the strings are at the edge of the "moat". Sampling the global optimum by chance occurs with probability $1/2^n$ per sample. Sampling a point adjacent to the global optimum occurs with probability $n/2^n$; however, the probability that "next ascent" will select exactly the right bit to improve first is only $1/n$. Thus, we will conservatively assume the global optimum is not found without recombination.

Lemma 2. *Assume a random initial population has been generated that is then improved by the "next ascent bit climber" local search. Assuming the global optimum is not generated randomly, or discovered by local search, then any constant size population improved by using the "next ascent bit climber" will be uniformly distributed around the edge of the "moat" in* $\Theta(n)$ *time.*

Proof. It requires $\Theta(n)$ time to evaluate each member of the initial population using a standard (not incremental) form of the $Jump_{m,n}(x)$ function. Assuming the population size is bounded by a constant, the initial population is evaluated in $\Theta(n)$ time. Next, a $\Theta(1)$ implementation of the *Jump* function can be used to implement local search. Because "next ascent bit climber" tries every bit in the string once before flipping any bit a second time, the bit climber must reach the edge of the moat in at most n evaluations for every string in the population. We can confirm the point is a local optimum by attempting another n bit flips. Since each initial string was randomly selected, and the order of improving moves is randomized as well, local search is equally likely to yield any point on the edge of the moat. This work requires $\Theta(n)$ time. □

3.1 Deterministic Crossover: 3-Parent Voting Crossover

A number of simple test problems can be solved by exploiting low level hyperplane information. This is also true for the *Jump* function; it is trivial to compute the averages of the first order hyperplane subspaces for functions of unitation, and it is trivial to prove that any order-1 hyperplane with a single 1 bit in any position is better on average than any order-1 hyperplane with a single 0 bit in any position for the *Jump* function. Any crossover operator that can effectively exploit first order hyperplane averages has a clear advantage.

Recently, a number of deterministic crossover operators have been proven to be highly effective on classic NP-Hard problems. The *partition crossover* operator has produced excellent results on large, one million variable NK-Landscapes [2,17] without using mutation. The LKH search algorithm for the TSP also uses Iterative Partial Transcription (IPT) [9,10,12]. In the area of scheduling, Deb and Myburgh [5] used a deterministic form of *block crossover* and deterministic repair operators (which they call "mutation" operators) to generate near optimal solutions to one billion variable cast scheduling problems. All of these algorithms use deterministic crossover operators to solve classic NP-Hard problems but none of these very modern evolutionary algorithms uses random mutation operators.

Perhaps the best operator for exploiting low level hyperplane information is "Voting Crossover." Voting crossover uses an odd number of parents (e.g., 3 parents), and then each parent "votes" for a 1 bit or a 0 bit in every bit position, where the majority wins. Thus, the result is deterministic. This operator was first introduced at the PPSN conference in a highly cited paper by Eiben et al. in 1994 [6]. It was given the name "Occurrence Based Scanning" crossover, and in its most general form it could use any number of parents. (We argue that the name "Voting Crossover" is more intuitive, more descriptive and easier to remember.) A randomized version of this crossover was also introduced called "Uniform Scanning" crossover [6]. Under "Uniform Scanning" crossover, the "vote" is interpreted probabilistically. For example, given 3 parents, if 2 parents have a 1 bit and 1 parent has a 0 bit, then the 1 bit is inherited with 2/3 probability.

Eiben et al. [6] reported that multi-parent crossover operators yields superior results on the classical DeJong test suite. On other benchmarks they considered, the results were more mixed, but overall, multi-parent crossover operators were competitive with classical operators such as 2 parent uniform crossover. A follow up study also suggested that multi-parent crossover operators are more effective on NK-Landscapes with low-epistasis [7]. This should not be surprising. One of the problems with classical uniform crossover is that all bits that are shared are inherited from the parents, but when the 2 parents differ, the bit assignment is completely random. This means that uniform crossover just randomly picks a string drawn from the largest hyperplane subspace that contains both parents.

We use 3 Parent Voting Crossover for two reasons. First, is very easy to mathematically characterize the outcome of using just 3 parents because crossover is deterministic. Second, using just 3 parents allows for some diversity to remain in the search process and the population; using 5 parents or 7 parents would create

more selection toward the bit pattern that (already) most commonly occurs in the population in that particular position. This does not matter for the *Jump* function, but could be important for other objective functions.

3.2 The Probability of Success (POS) for 3-Parent Voting Crossover

Lemma 3. *Given 3 random parent strings with m 0 bits and $(n-m)$ 1 bits (strings on the edge of the gap), Voting Crossover yields the global optimum with probability $POS(n,m)$ for the Jump function, where:*

$$POS(n,m) = \frac{(n-m)!/(n-3m)!}{(n!/(n-m)!)^2}$$

Proof. Assume you have n buckets, and have m red marbles, m green marbles and m blue marbles. We can use the red marbles to construct a bit string with m bits of 0, and $n-m$ 1 bits. Place each red marble in a random bucket that does not already contain a red marble. This yields a bit string: a bucket without a red marble is assigned a 1 bit, and a bucket with a red marble is assigned a 0 bit. A second string is constructed using the green marbles, and a third string is constructed using the blue marbles.

After 3 strings are generated, if every bucket contains at most 1 marble then 3-parent voting crossover will jump to the global optimum: in every bit position there is at most 1 vote (1 marble) for 0, and therefore 2 or more votes for 1.

Place the 1st red marble and 1st green marble and 1st blue marble randomly into a bucket. The probability of zero conflicts for these 3 events is $n/n \cdot (n-1)/n \cdot (n-2)/n$ because the marbles can go anywhere.

Place the 2nd red marble and 2nd green marble and 2nd blue marble randomly into a bucket. The probability of zero conflicts for these 3 events is:

$$((n-3)/(n-1)) \cdot ((n-4)/(n-1)) \cdot ((n-5)/(n-1))$$

Generalizing, as each marble is placed, the enumerator is decreases by 1. But the denominator is decreasing by 1 only after 1 marble of each color has been placed, every 3 steps. This yields the following general result:

$$\frac{n!/(n-3m)!}{n!/(n-m)! \cdot n!/(n-m)! \cdot n!/(n-m)!} = \frac{(n-m)!/(n-3m)!}{n!/(n-m)! \cdot n!/(n-m)!}$$

□

3.3 A Lower Bound on the Probabilities

An alternative way to calculate POS (equivalent by simple algebra) is as follows:

$$POS(n,m) = \frac{(n-m)!/(n-2m)!}{n!/(n-m)!} \cdot \frac{(n-2m)!/(n-3m)!}{n!/(n-m)!}$$

We will use this form to compute a simple but sufficient bound on the probability $POS(n, m)$. Since there are m integers in the following sequence, we automatically obtain the following result by taking the smallest number in the enumerator and the largest number in the denominator.

$$\frac{(n-(2m-1))^m}{n^m} < \frac{(n-m)!/(n-2m)!}{n!/(n-m)!}$$

By identical logic:

$$\frac{(n-(3m-1))^m}{n^m} < \frac{(n-2m)!/(n-3m)!}{n!/(n-m)!}$$

This yield a bound

$$\frac{(n-(2m-1))^m}{n^m} \cdot \frac{(n-(3m-1))^m}{n^m} < \frac{(n-m)!/(n-2m)!}{n!/(n-m)!} \cdot \frac{(n-2m)!/(n-3m)!}{n!/(n-m)!}$$

See Fig. 2. From this bound, and we can obtain the following theorem:

Lemma 4. *Assume $m = Floor(log_2(n))$. Then for all positive integers:*

$$Bound(POS(n, log_2(n)) = \frac{(n-(2m-1))^m}{n^m} \cdot \frac{(n-(3m-1))^m}{n^m}$$

is a non-decreasing function that converges in the limit to probability 1 for large n. This function is a lower bound on $POS(n, log_2(n))$.

Proof. When n is a power of 2, the following inequalities hold by simple algebra:

$$\frac{(n-(2m-1))^m}{n^m} < \frac{(2n-(2(m+1)-1))^{(m+1)}}{(2n)^{(m+1)}}$$

and

$$\frac{(n-(3m-1))^m}{n^m} < \frac{(2n-(2(m+1)-1))^{(m+1)}}{(2n)^{(m+1)}}$$

This is sufficient to prove the *Bound* function is non-decreasing.

Again assume that n is a power of 2. Now consider any integer $n + x$ such $n < n + x < 2n$ where m is given by $m = Floor(log_2(n))$. Then, by simple algebra:

$$\frac{(n-(2m-1))^m}{n^m} < \frac{((n+x)-(2m-1))^m}{(n+1)^m}$$

and

$$\frac{(n-(3m-1))^m}{n^m} < \frac{((n+x)-(3m-1))^m}{(n+1)^m}$$

□

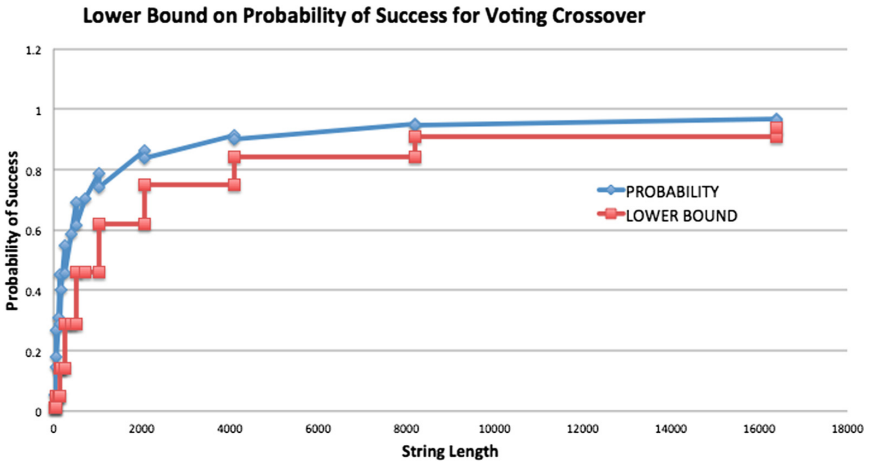

Fig. 2. The Probability of Success (POS) for Voting Crossover and the Lower Bound on the Probability of Success. The true probability is not monotonic, but the Lower Bound on the POS is a non-decreasing function that asymptotically converges to 1.0.

4 Probabilities and Populations

For Voting Crossover, even a single crossover yields a high probability of reaching the global optimum (e.g., >50%) for $n > 512$. We can use the population to boost that probability. But we will apply crossover more frequently than is normally the case in a simple genetic algorithm. Because we are using 3-parent crossover, we will allow each parent to be involved in up to 3 crossover events. This is related to the concept of "Brood Selection", where 2 parents can generate multiple offspring. "Brood Selection" is critical to the highly successful EAX algorithm for the Traveling Salesman Problem, where the number of offspring generated each generation is typically 30 times the population size [13,14]. This allows crossover to be utilized as an exploration operator.

We will assume each crossover event must be independent. For example, if we recombine the three parents $P1, P2, P3$, we will then not allow the crossover of parents $P1, P2, P4$, since $P1$ and $P2$ have already been paired in the previous crossover. For example, with a population size of 6, an independent set of recombination events might include the following subsets of parents:

$$\{\{P1, P2, P3\}, \{P1, P4, P5\}, \{P2, P4, P6\}, \{P3, P5, P6\}\}$$

where each parent is involved in 2 recombination events, but no pair of parents occurs in more than 1 recombination event. Let μ denote the population size and let λ denote the number of offspring generated in one generation. Table 1 calculates the probabilities of discovering the global optimum in the first generation of our hybrid genetic algorithm.

We next show both the theoretical results and empirical results based on 1000 runs of our hybrid genetic algorithm with Voting Crossover. Because our

Table 1. Theoretical probability analysis that the algorithm will converge to the global maximum using 3 Parent Voting Crossover. A probability of "1" in this case means that the probability is greater than 0.999999999. "μ" denoted population size, and λ denoted the number of independent offspring generated.

N	Bound	Probability ($\lambda = 1$)	$\mu = 7$ ($\lambda = 7$)	$\mu = 11$ ($\lambda = 13$)	$\mu = 13$ ($\lambda = 22$)	$\mu = 15$ ($\lambda = 35$)	$\mu = 25$ ($\lambda = 75$)
32	0.011	0.05242	0.31402	0.57747	0.69412	0.84810	0.98237
64	0.051	0.14659	0.67031	0.87263	0.96941	0.99610	0.99999
128	0.143	0.29076	0.90972	0.98851	0.99947	0.99999	1
256	0.290	0.45779	0.98622	0.99964	0.99999	1	1
512	0.461	0.61534	0.99875	0.99999	1	1	1
1024	0.622	0.74326	0.99992	0.99999	1	1	1
2^{11}	0.750	0.83653	0.99999	1	1	1	1
2^{12}	0.843	0.89954	0.99999	1	1	1	1
2^{13}	0.910	0.94830	1	1	1	1	1
2^{14}	0.943	0.96470	1	1	1	1	1
2^{15}	0.967	0.97959	1	1	1	1	1
2^{16}	0.981	0.98834	1	1	1	1	1
2^{17}	0.989	0.99340	1	1	1	1	1
2^{18}	0.993	0.99629	1	1	1	1	1
2^{19}	0.996	0.99793	1	1	1	1	1
2^{20}	0.998	0.99984	1	1	1	1	1

calculations are precise and because Voting Crossover operator is deterministic, the theoretical results and empirical results match more or less perfectly. This can be seen in Fig. 3. We can now state the overall result.

Theorem 1. *Let the gap used by the Jump function be $m = Floor(log_2(n))$. For sufficiently large n, a hybrid genetic algorithm which uses (1) a $\Theta(1)$ time incremental evaluation function, (2) a population size bounded by a constant, (3) "next ascent bit climber" local search and (4) Voting Crossover using 3 parents converges to the global optimum of the $Jump_{m,n}$ function in $\Theta(n)$ time with probability approaching 1 in one generation using no mutation, assuming each parent is allowed to be involved in up to 3 recombinations.*

Proof. Lemma 1 establishes that local search can evaluate potential solutions in $\Theta(1)$ time after the initial population has been evaluated. Lemma 2 demonstrates that the population will be uniformly distributed along the edge of the "moat" in $\Theta(n)$ time after the first generation is improved by next ascent bit climbing. By Lemma 3, the probability of generating the global optimum by a single 3 parent Voting Crossover is at least $POS(n, log_2(n)) = \frac{(n-m)!/(n-3m)!}{(n!/(n-m)!)^2}$ and by Lemma 4 this probability is bounded by a nondecreasing function that

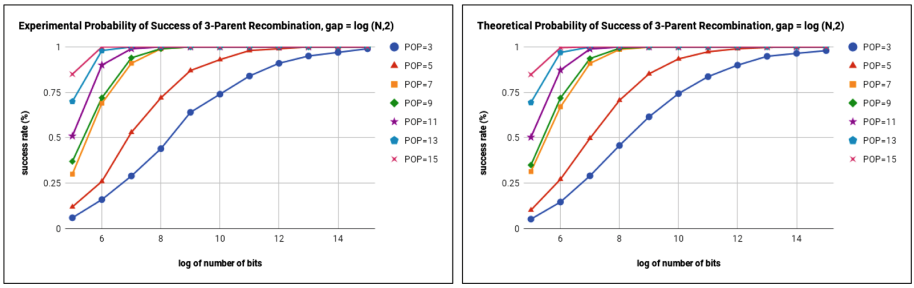

Fig. 3. The probability of finding the global optimum (jumping across the gap) in one generation for a hybrid genetic algorithm using Voting Crossover as a function of population size. The experimental data and the theoretical data match almost perfectly.

in the limit converges to 1 for large n. For smaller values of n we can use the population and multiple recombinations to boost the probability of finding the global optimum. Let $p = POS(n, log_2(n))$ and let $\lambda \leq 3\mu$ denote the number of offspring generated by *independent* recombinations, where the parents are located on the edge of the "moat" on the *Jump* function. Then the probability of finding the global optimum in the first generation is given by: $1 - (1 - p)^{\lambda}$. For all $n \geq 64$ and all $\mu = 25$, the probability of finding the global optimum is greater than 0.99999. This probability also converges to 1 in the limit for large n. The time to find a set of independent crossovers is bounded by a constant when the population is of constant size. Each crossover takes $\Theta(n)$ time. But the total number of applications of crossover is a constant when the population size is bounded by a constant. □

5 Other Crossover Operators

While multi-parent crossover and voting crossover are in literature and have been shown to be effective on some benchmarks and on low epistasis NK Landscapes, one might ask what happens when a more conventional crossover operator is used. The first answer is that the probability of successful crossover depends on m but is independent of n when m is also independent of n. Thus, if we fix m at some value such as $m = 8$, the ability of crossover to find the global optimum in $O(n)$ time can be preserved. But the "constant" involved (or more precisely, the cost depending on m) can still vary dramatically.

"Uniform Crossover" is perhaps the worst possible choice of crossover operators. Assuming maximally different parents (that are still local optima), there are 2^{2m} possible offspring, only one of which is the global optimum.

A better choice would be the HUX (Half Uniform Crossover) operator of CHC [8]. HUX determines which bits are different in two parents, then selects exactly one half of the non-shared bits from one parent and the remaining non-shared bits from the other parent. If the two parents are maximally distant from each other on the edge of the moat, then they have $2m$ bits that differ, and HUX

will select m bits from each parent. Thus, there are $Choose(2m, m)$ possible offspring (compared to 2^{2m} possible offspring under Uniform crossover). The CHC algorithm also boosts the probability that parents are maximally different by "incest prevention." Thus, when recombining parents, it prefers to pair parents that are maximally different. When the populations used by CHC are improved using next ascent bit climbing, CHC also finds the global optimum for JUMP functions as long as the rate of crossover and the population size is sufficient to allow on the order of $Choose(2m, m)$ recombinations of maximally different parents. For $m = 8$, $Choose(16, 8) = 12,870$ recombinations with an associated probability of success of 0.000077, while $2^{16} = 65,536$ recombinations is needed for Uniform crossover, with an associated probability of success of 0.000015.

6 Conclusions

We have shown that a hybrid genetic algorithm which improves the initial population using next ascent bit climbing, and which uses 3 parent Voting Crossover can solve the $Jump_{m,n}$ function when $m = log_2(n)$ with probability asymptotic to 1 for sufficiently large n in $\Theta(n)$ time. We would argue that this is not at all surprising. Theoretically, it is easy to prove that the $Jump_{m,n}$ is also solved in $\Theta(n)$ time by calculating the averages of all of the order-1 hyperplanes. Any combination of crossover and local search that actively exploits this property of the $Jump$ function should also arrive at the global optimum.

It is also the case that the $Jump$ function becomes easier to solve for recombination operators as n increases, because $Choose(n, m)$ becomes larger and the probability that two parents are different enough to have a successful recombination increases. On the other hand, for mutation to make a jump of length m becomes harder as n increases. Assuming that m mutations happen at once, there are still $Choose(n, m)$ ways to select m bits. To make the jump from a local optima, exactly the right m bits must be selected.

The work presented here highlights the advantages of using crossover to enhance exploration. There is no reason that two parents should have only 1 or 2 offspring (as is normally the case for genetic algorithms); there are many biological species where two parents might have dozens of offspring at a time. This idea, while common in the mutation driven $(\mu + \lambda)$ES, has probably not received the attention it deserves in a crossover driven genetic algorithm.

This work also emphasizes the critical role that diversity places in genetic algorithms. We also want to acknowledge the work by Dang et al. [3] looking at how diversity mechanisms, such as fitness sharing, and the use of an Island model, can also make crossover more effective when solving the $Jump$ function when using a $(\mu + 1)$ genetic algorithm. These are all very simple ideas, and common strategies in genetic algorithm applications.

Acknowledgements. This work was supported by a grant from the US National Science Foundation CISE/ACE, SSI-SI2. Dr. Mukhopadhyay was supported by a Fulbright-Nehru Academic and Professional Excellence Fellowship.

References

1. Jansen, T., Wegener, I.: The analysis of evolutionary algorithms-a proof that crossover really can help. Algorithmica **34**, 47–66 (2002)
2. Chicano, F., Whitley, D., Ochoa, G., Tinos, R.: Optimizing one million variable NK landscapes by hybridizing deterministic recombination and local search. In: Genetic and Evolutionary Computation Conference (GECCO), pp. 753–760. ACM (2017)
3. Dang, D., et al.: Escaping local optima with diversity mechanisms and crossover. In: Genetic and Evolutionary Computation Conference (GECCO), pp. 645–652. ACM (2016)
4. Davis, L.: Bit-climbing, representational bias, and test suit design. In: Booker, L., Belew, R. (eds.) International Conference on Genetic Algorithms, pp. 18–23. Springer, Heidelberg (1991)
5. Deb, K., Myburgh, C.: Breaking the billion variable barrier in real world optimization. In: Genetic and Evolutionary Computation Conference (GECCO), pp. 653–660. ACM (2016)
6. Eiben, A.E., Raué, P.-E., Ruttkay, Z.: Genetic algorithms with multi-parent recombination. In: Davidor, Y., Schwefel, H.-P., Männer, R. (eds.) PPSN III 1994. LNCS, vol. 866, pp. 78–87. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58484-6_252
7. Eiben, A.E., Schippers, C.A.: Multi-parent's niche: N-ary crossovers on NK-landscapes. In: Voigt, H.-M., Ebeling, W., Rechenberg, I., Schwefel, H.-P. (eds.) PPSN IV 1996. LNCS, vol. 1141, pp. 319–328. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61723-X_996
8. Eshelman, L.: The CHC adaptive search algorithm: how to have safe search when engaging in nontraditional genetic recombination. In: Foundations of Genetic Algorithms (FOGA), vol. 1, pp. 265–283. Morgan Kauffman (1991)
9. Helsgaun, K.: General k-opt submoves for the Lin-Kernighan TSP heuristic. Math. Program. Comput. **1**(2–3), 119–163 (2009)
10. Helsgaun, K.: DIMACS TSP challenge results: current best tours found by LKH (2013). http://www.akira.ruc.dk/keld/research/LKH/DIMACSresults.html
11. Moscato, P.: Memetic algorithms: a short introduction. In: Corne, D., Dorigo, M., Glover, F. (eds.) New Ideas in Optimization, pp. 219–234 (1999)
12. Möbius, A., Freisleben, B., Merz, P., Schreiber, M.: Combinatorial optimization by iterative partial transcription. Phys. Rev. E **59**(4), 4667–4674 (1999)
13. Nagata, Y., Kobayashi, S.: Edge assembly crossover: a high-power genetic algorithm for the traveling salesman problem. In: International Conference on Genetic Algorithms (ICGA), pp. 450–457. Morgan Kaufmann (1997)
14. Nagata, Y., Kobayashi, S.: A powerful genetic algorithms using edge assemble crossover the traveling salesman problem. INFORMS J. Comput. **25**(2), 346–363 (2013)
15. Rowe, J.: Population fixed-points for functions of unitation. In: Foundations of Genetic Algorithms (FOGA), vol. 5, pp. 69–84. Morgan Kauffman (1998)
16. Srinivas, M., Patnaik, L.M.: On modeling genetic algorithms for functions of unitation. IEEE Trans. Syst. Man Cybern. Part B (Cybern.) **26**(6), 809–821 (1996)
17. Tinós, R., Whitley, D., Chicano, F.: Partition crossover for pseudo-boolean optimization. In: Foundations of Genetic Algorithms, pp. 137–149 (2015)