Ministry of Silly Walks (Evolutionary Walker)

CS672

Matt Harren and Allen Wang

Walker is a 2D ball-and-stick construct that uses genetic programming with mutation, crossover, and elitism to evolve a program to move itself. The programming language governing its actions is capable of doing standard arithmetic operations as well as if-then operations and numerical comparisons. We have made a few simplifications and assumptions in the relaxed physics model used to simulate Walker. Through our experiments we have discovered that the genetic programming technique is robust, though slow to converge to a solution at times.

1 Introduction

Genetic programming is a form of evolutionary computation that has computer programs as members of its population.

We chose to use Genetic programming in attempting to make the construct in figure 1.1 walk. The goal we set for the walker to achieve: move as far to the right of the starting position as possible, under the constraints set forth by the physics system.

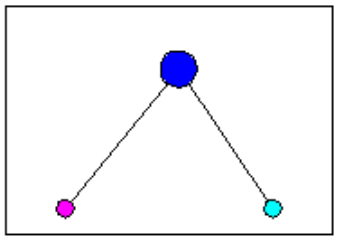

Figure 1.1: The walker model.

1.1 Rationale

Why use Genetic programming in this situation?

-Genetic programming is perhaps best suited to be used in problem areas where humans may find the input data difficult to correlate with desired output results. The nature of computer-generated programs means that the variation of things tried will be enormous when compared to the relatively narrow thought process most humans follow, and there will be chances to discover even the most obfuscated of relations between inputs and outputs.

-Genetic programs search from a set of points, as opposed to many search methods which base their search on one point at a time -- local search. This makes it easier to break out of local minima (since it is highly improbable that the entire population is trapped within one local minimum) and the variety of points being searched at once increases the chance of finding a global optimum.

-Genetic programs make no assumptions about the search space and only need an objective function value. There is no need for the function to be continuous or differentiable, and the only required information is a notion of fitness for each individual in a given generation.

1.2 Problem definition

The walker is situated in a world that is simply a ground, with no walls. It is free to attempt to move in any direction through the following two possible output operations:

hop: We can imagine that a spring has been embedded in each of the feet of our walker. This operation, when applied to a given foot, causes an upward force, aligned with the leg connecting the foot with the body, to be applied to the walker.

specify-angle: The body of the walker can specify a change in the angle between its legs. If both feet are in the air, this affects both legs identically. Otherwise, if one foot is on the ground, this command is equivalent to rotating the other leg about the body. If both feet are on the ground, the angle between the legs is fixed in place and may not be changed.

To determine what operations to use, and what amount to set the leg angle to, the walker may access the following inputs for each vertex (of which there are three -- the two feet and the body):

X: This is the x-coordinate of the vertex.

Y: This is the y-coordinate of the vertex.

VX: This is the velocity in the x direction of the vertex.

VY: This is the velocity in the y direction of the vertex.

Time is quantized into frames for the purposes of this project. At each frame, the walker provides information which allows us to calculate what will happen in the next frame.

1.3 Technical details

We implemented Walker and all the other components of our project using Java 1.3.0, and the visualization part (see section 2.5) was done using Java Swing.

2 Program Structure and Information

2.1 Walker

Each Walker contains a Vector of 3 `Vertex` objects and a Vector of 3 `RealExpr` objects.

The `Vertex` objects each represent one of the vertices in our walker, and contain information about the location and velocities of that vertex.

Each of the `RealExpr` objects are programs that evaluate to real values. There is one program assigned to every vertex, and these programs represent how the walker will decide to move. For the "body" vertex, output of the program controls the angle between the legs; for the "feet" vertices, a positive result is interpreted as a *hop* command. The programming language they use is described in section 2.3.

Each walker also contains a `step` function, which runs the program for each vertex, interprets the outputs, and applies the physics model to calculate the positions and velocities of all objects in the next time step.

2.2 Physics Model

The relaxed physics model we are using incorporates realistic models of gravity and momentum. However, some simplifications have been made to keep the system more manageable.

Here are the main assumptions:

-Only the body has non-negligible mass. Swinging the feet around will not cause the body to move.

-Perfect friction. The feet do not slide.

-The feet have built in springs which direct force along the legs towards the body. (This is necessary in order for the walkers to lift their legs and escape the ground's fiction.) These springs are activated when the walker decides to use the *hop* output operation.

-The feet may pass through each other.

2.3 Programming Language

We created a simple functional-style programming language for use with this system. Although this language is not Turing-complete, it has a variety of available operators and inputs and is therefore capable of performing a reactive task like walking.

There are three types in this language: real values, boolean values, and indices that are used to label the three vertices. Here is a definition of the language:

real ::= *real* + *real*
| *real* – *real*
| *real* * *real*
| *real* / *real*
| getX(*index*)
| getY(*index*)
| getX_velocity(*index*)
| getY_velocity(*index*)
| if (*bool*) then *real* else *real*
| (constants between –1 and 1);

bool ::= *real* < *real*
| *real* > *real*
| closeTo(*real*, *real*)
| true
| false;

index ::= 0
| 1
| 2;

The operators getX and getY access the physical model of the walker to retrieve the X position and Y position, respectively, of the vertex specified by their argument. Similarly, getX_velocity and getY_velocity retrieve the velocities of that vertex.

Programs and subprograms are generated using a random, recursive algorithm. Mutations can occur on any node, and consist of regenerating that node. To ensure type-safety, we only do crosses between real-valued nodes.

2.4 Genetic Algorithm

Here is pseudocode for the genetic algorithm that we used to evolve the walkers:

```
Genetic Algorithm:

Initialize population

For maxGen generations {
Calculate fitness of every individual

If all individuals are all at base fitness, re-initialize the population and skip the rest of the loop

Replace the weakest individual with the fittest individual from the last population (if generation > 1)

Select the pairs of parents that will make children for the following generation

With probability crossProb perform crossover on the pairs of parents

With probability mutationProb perform mutation on the resulting offspring from crossover

}
```

Listing 2.4.1 pseudocode listing for genetic algorithm

The following sections describe the various features of the algorithm.

2.4.1 Mapping to a population

The population at each generation is a series of walker objects. The genetic operations all work on the "chromosomes" of the walkers, which, in our case, have been designated to be the 3 programs associated with each walker as its genetic identifying sequence.

2.4.2 Fitness

As we are trying to get our walkers to walk, we have arbitrarily chosen a direction in which we would like them to move. The fitness metric we are using is the x-coordinate reached by the body of the walker after *t* time steps (always a positive number -- if the fitness is less than 0 we set it to 0). If at any time during those *t* steps the walker's body touches the ground, it is considered "dead" and assigned a fitness of 0.

2.4.3 Selection

Selection is used to choose the parents for the population in the next generation. Two parents are selected for each child. The only restriction is that the same parent cannot be

selected twice to create one child. We tried two methods:

-Roulette Wheel selection. Each walker is selected probabilistically based on its fitness (normalized with respect to the sum fitness of the entire population). We select two parents for each child in the next generation.

-"Weak" Tournament selection. For each child that we wish to create for the next generation, the population is arbitrarily divided into two segments, and we choose the fittest individual from each segment to become two parents of the child.

2.4.4 Crossover

Crossover is applied to each pair of parents to generate a child.

Because it does not make sense to combine programs from different vertices (there is no reason why a "good" right foot program would enhance a body program), crossover is strictly limited to alterations between programs assigned to the same type of vertex.

In each crossover process, a subtree of the second parent is randomly selected to replace a subtree of the first parent. This newly modified first parent becomes the child. This is essentially the standard form of single-point crossover for genetic programming.

In our implementation of the genetic algorithm, there is a chance that crossover does not occur. In this case we simply assign the first parent to be the new child.

2.4.5 Mutation

After the new children for the next generation have been created through the crossover method, there is a chance that each of them may be mutated. Mutation in this case is a random regeneration of a program subtree. Each vertex's program is equally likely to be chosen for the mutation, though only 1 of the 3 may be mutated in one mutation operation.

2.4.6 Elitism/Ultra-Elitism/Euthanasia

Elitism, in standard genetic algorithm terminology, refers to always preserving the fittest individual of a generation. We have implemented a variation of this which is perhaps a slight improvement on the method. We save the fittest individual of the current generation before performing selection, crossover, and mutation. Then, in the next generation, after all fitness calculations of the new children, we replace the least-fit child with the previous fittest individual. If a more fit individual has been located, that is then preserved for insertion to the next generation, and we carry on with selection, crossover, and mutation as usual.

Ultra-Elitism is a term we have created to mean the pre-emptive seeding of a newly initialized population with the best individual found from *all previous runs* of the genetic algorithm. This is basically a jump-start on evolution for all new runs. It may result in an evolutionary defect in that we are channeling the evolution in an overly restrictive way, (and not reaping the full benefits of the multiple-point search inherent in genetic algorithms) but there are clear advantages as well, such as immediate high fitnesses for many members of the population in generation 2 and onwards.

In the event that *all* members of the population are lacking in fitness (this is the case when no individuals have been able to move from the starting point) we euthanize the entire population by re-initializing it for the next generation, as the members of the current generation clearly have nothing to offer the walkers in an evolutionary sense.

2.5 Visualization

The Java Swing application that we used for visualization of the walker shows the animated action of the walker object given to it. The current time step and the walker body's x location are displayed at the top of the screen.

Whereas our fitness function used a fixed number of time steps, the visualization tool will run the walker as long as the program is not stopped.

3 Results/Analysis

In comparing heuristic algorithms, whether to themselves (running on different parameters) or to other algorithms, we need a good metric that still represents a notion of how "good" an algorithm with a certain set of parameters is, regardless of context of the problem. We decided to use the number of fitness evaluations as a metric. Fitness evaluations generally take up the most time in the main loop of any heuristic algorithm, and is a reasonable approximation of what is probably the best real-world metric, clock time. Thus, within these comparisons we maintained the same number of fitness evaluations.

The tests below were based on averages of 5 trials for each permutation of parameters tested.

3.1 Parameter variation comparison

As shown in figures 3.1.1 and 3.1.2, the effect of alterations in the mutation and crossover rates is not particularly pronounced. Even in the case of what might be called “outliers” (such as the 0.3 mutation runs or the 0.3 crossover runs, which had the unlikely occurrence of randomly induced introduction of a powerful archetype into the population) the average fitness still does not vary more than ± 30.

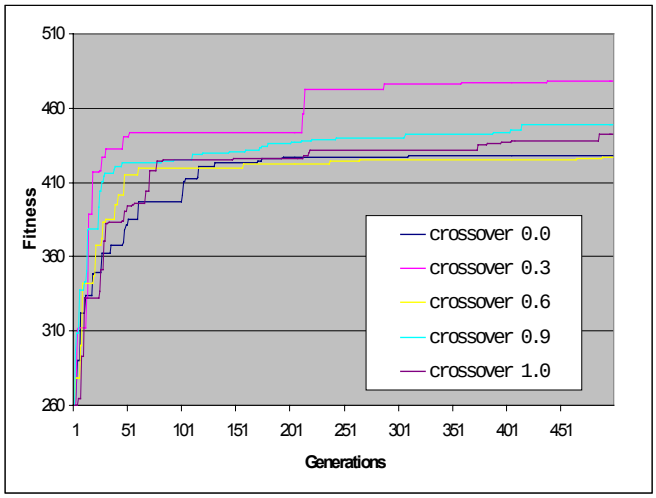

Figure 3.1.1: The effect of crossover rates.

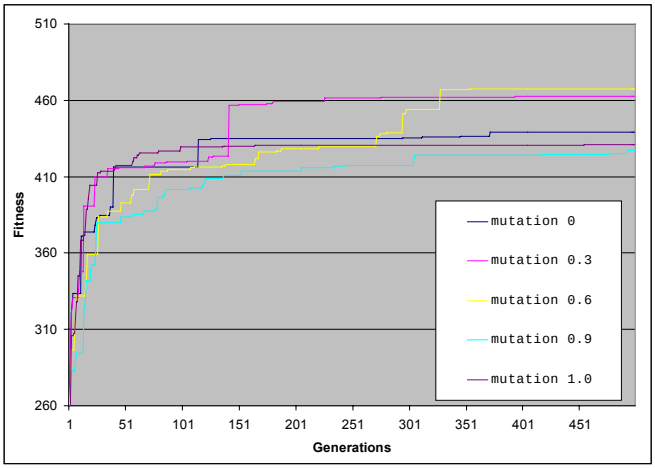

Figure 3.1.2: The effect of mutation rates.

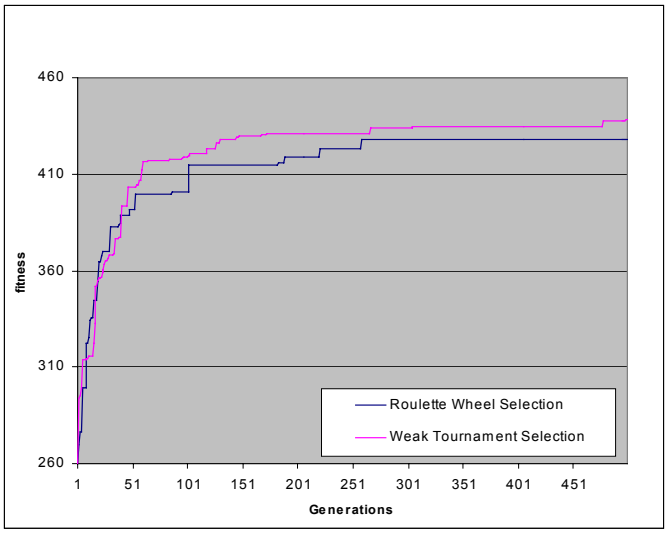

Figure 3.2.1: The effect of the selection method.

This lack of deviation can be interpreted as a side-effect of certain characteristics in the search space for our problem. We postulate that the space is saturated with local minima that are very difficult to escape. The global minima are likely few and far between.

The local minimum that is the most frequent occurrence within these trials is simply that of the archetype we have detailed in 3.4.2, where the walker falls over to the right.

3.2 Selection variations

Again, in figure 3.2.1, there is little noticeable difference between the two selection types. Perhaps in a search space with fewer (or shallower) local minima, the selection process would play a greater role in the advancement of a population's fitness.

3.3 Population/Generation tradeoff

Figure 3.3.1 exhibits a manifestation of the outlier phenomenon mentioned above. We can see that the runs corresponding to [Population 100, Generations 250] suddenly gain a very significant lead over the other runs at about 14000 fitness evaluations.

This was in fact due to the sudden discovery/evolution of a powerful archetype, documented below as the "hopper". This one individual alone (fitness of around 1500, and increasing steadily) was enough to skew the results drastically for the rest of the runs in its permutation category.

Other than that, alteration of population size and number of generations did not seem to greatly affect the average best fitness found by our genetic algorithm.

3.4 Some Walker "Types"

During our trial runs, these were the most prominent archetypes that emerged. Note that the ones that actually were able to display some form of sustained locomotion were the windmill, the hopper, and the sprinter.

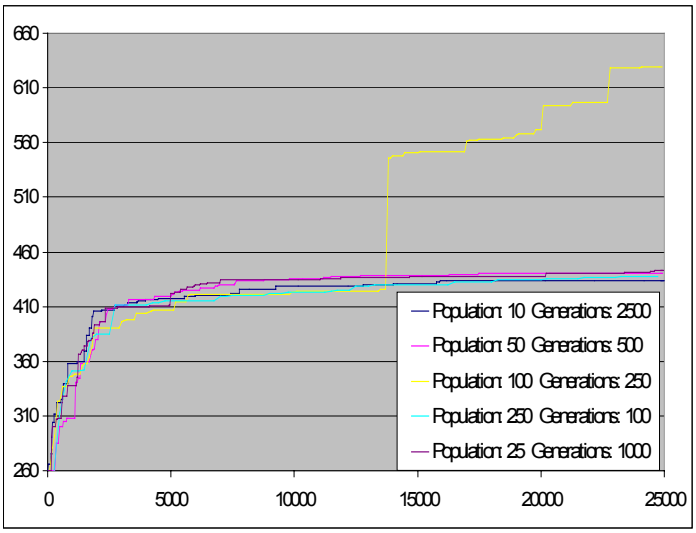

Figure 3.3.1: The tradeoff between population size and generations for constant fitness evaluations.

3.4.0 Statue

In this case the walker simply stands in place without even attempting to move a leg.

3.4.1 Bird

The "bird" walker jumps in place, repeatedly, flapping its legs while airborne in a manner reminiscent of wings.

3.4.2 Leaning Tower of Pisa

This breed of walker starts out by taking a step, and then leans over to a seemingly unstable angle and then remains still for the duration of the time allotted to the experiment.

3.4.3 Windmill

This walker spins its legs alternately, carrying itself forward with each step. Occasionally it pauses and performs full rotations with one leg.

3.4.4 Hopper

The "hopper" leaps to new positions with legs outstretched, and inches its rear leg towards the leading one in preparation for the next leap.

Hopper is the most stable type we have found, which continue almost indefinitely.

3.4.5 Sprinter

This type of walker leans over near the start of its run, allowing its hops to provide it with more forward momentum than usual. The initial large leap is followed by many bursts of skipping followed by rapid steps.

The sprinter is the fastest walker that has evolved (goes the farthest in the limited time frame we have chosen to use for our fitness evaluations), but is unstable at times due to being on the *bleeding edge* of walker evolution and enjoys driving itself to the limit, and into the dirt shortly thereafter.

3.5 Outputs of the Genetic Programming

As a result of many generations of mutation and crossover, the successful walkers often had very large, very obfuscated programs. (One Hopper-like solution had about 20,000 nodes in its program.) Even those programs which were short were generally very difficult to read and understand.

4 Conclusion

Genetic algorithms have proven to be a robust form of search. The program was able to locate solutions even for problems such as this one, which has hard-to-escape local minima.

Our program succeeded in finding the simple walking algorithm we had expected it to discover (the Windmill) as well as a few more powerful strategies that we had not thought of (Hopper, Sprinter).

5 Future Study

It may be interesting to look at the following issues as further research:

-Different terrains for the physical model, including terrains with hills and slippery surfaces, or moving obstacles.

-Walkers with more than three nodes, and the physics model to handle this. (The physics model will need to be significantly more intricate in order to handle these more complicated models.)

-A Turing-complete programming language, including subroutines and local variables.

-The ability for the walkers to “hop” with varying strengths.