Search

Chris Monson

August 12, 2004

Contents

1 Uninformed Search **1**
- 1.1 Breadth-First . . . 2
- 1.2 Depth-First . . . 2
- 1.3 Depth-limited Depth First . . . 3
- 1.4 Iterative Deepening . . . 4
- 1.5 Bidirectional . . . 4

2 Heuristic Search **4**
- 2.1 Greedy Search . . . 4
- 2.2 Uniform Cost Search . . . 5
- 2.3 A* Search . . . 5

3 Continuous and Social Search **5**
- 3.1 Hill Climbing . . . 6
- 3.2 Simulated Annealing . . . 7
- 3.3 Simplex Search (Amoeba Search) . . . 7
- 3.4 Ant Optimization . . . 7
- 3.5 Particle Swarm Optimization . . . 9
- 3.6 Genetic Algorithms . . . 9

4 Machine Learning is Search **10**

1 Uninformed Search

Questions to answer in the presentations:

- Complete?
- Optimal?
- Time complexity?
- Space complexity?
- Give example of running algorithm

Algorithm 1 BFSearch($node$, $goal$, $queue$)

```
while node ≠ goal do
  for all child ∈ node.children do
    queue.push(child)
  end for
  if queue.isempty() then
    break
  end if
  node = queue.pop()
end while
```

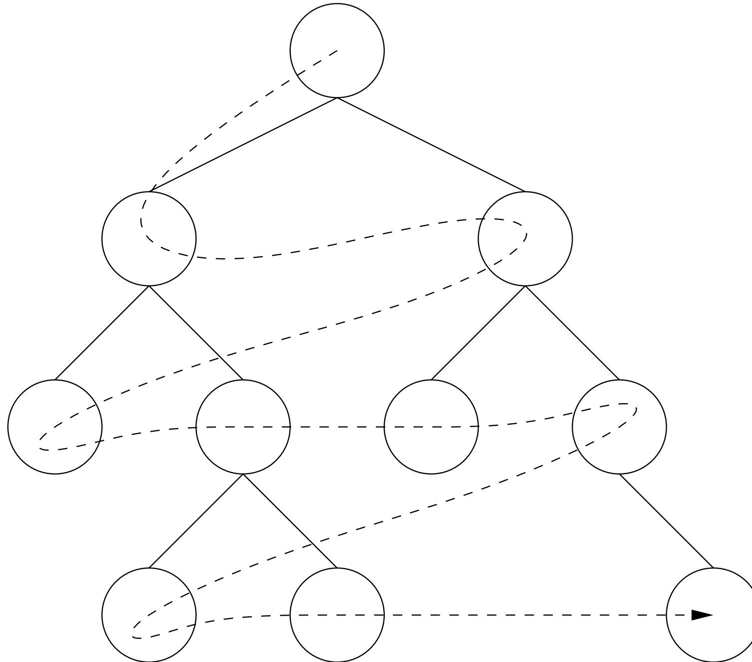


Figure 1: Breadth-first search

1.1 Breadth-First

The basic idea is that each level is expanded at the same time. Thus, we focus on breadth before depth. Starting at the root node, we expand all of its children. Then we expand all of the children of those children, and so on. Algorithm 1 shows pseudo code for this search. See Figure 1 for an example of how this works, more or less.

The space complexity is very high to make this work, since an exponential number of nodes is expanded at each level.

What guarantees do we have with this? If there is a solution, we will definitely find it (provided that we don't run out of resources). We will also be sure to find the shortest possible path to it (in steps, not in cost).

1.2 Depth-First

The basic idea here is that we dig as deep as we possibly can before testing anyone's siblings. Thus, we expand the first child of the root node, then that child's first child, and so on. Algorithm 2 outlines the

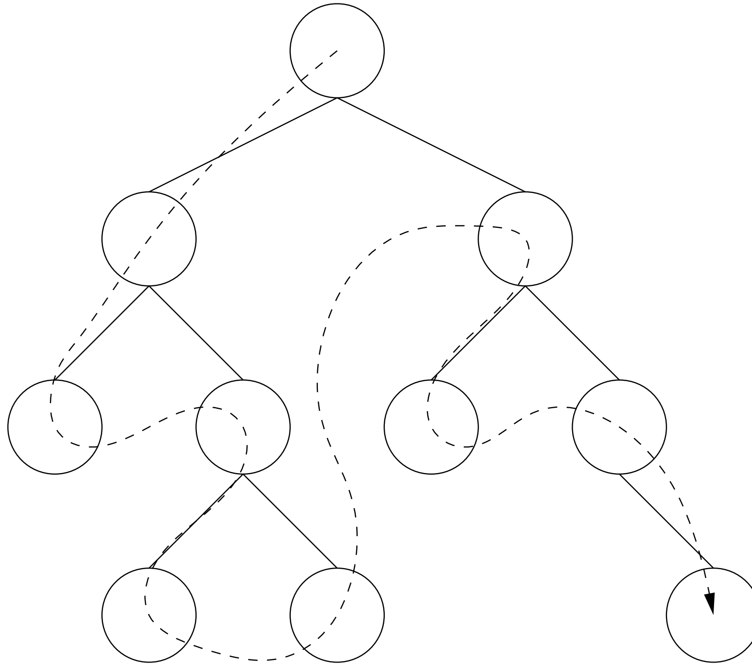


Figure 2: Depth-first search

search without using the call stack. A depiction of the progress of this search is shown in Figure 2.

Algorithm 2 DFSearch($node$, $goal$, $queue$)

---

while $node \neq goal$ **do**
  for all $child \in node.children$ **do**
    $stack.push(child)$
  end for
  if $stack.isempty()$ **then**
    break
  end if
  $node = stack.pop()$
end while

Unlike with breadth-first search, we do not have any guarantees as to whether the solution will ever be found. The space we are searching could very well be infinite, and therefore we will get stuck going deeper and deeper, and never once branch out. This is a major problem with depth-first search.

The space complexity is great, however, and is linear in depth.

1.3 Depth-limited Depth First

This is essentially a depth-first search where a limit is set on the depth. This means that the search will get cut off before running off into infinity. It also means that for finite spaces, we are no longer guaranteed to find the goal, since we may cut things off too early.

1.4 Iterative Deepening

This is a depth-limited search where the depth is gradually incremented. Once a small depth is exhausted, a larger depth is tried. Once that is exhausted, the depth is increased again, and so on forever.

The time complexity of this algorithm is higher than that of depth-first, since we duplicate work. It is surprising to note, however, that it isn't as bad as it seems. The actual complexity will be something that is discussed in class.

1.5 Bidirectional

Here we start at the starting point and at the goal simultaneously, looking for a common midpoint. This is generally done in a breadth-first fashion, though it can be done in other ways. Think hard about it. It's a pain to deal with.

2 Heuristic Search

Uninformed search has several problems. First, there is no concept of distance or cost between the nodes. This means that we can only find that a path exists to get to the node, not that we can necessarily find the shortest path. Additionally, the searches aren't smart about what they are doing at all. They just merrily do their thing without much regard for what is going on in the search space.

Heuristic searches are meant to overcome that particular issue. A heuristic is an estimate. In the case of search, it is an estimate of cost to reach the goal. There are several kinds of heuristics, but we need to focus on *admissible heuristics*. Admissible heuristics never overestimate the distance to the goal. They can get it just right, but they can never overestimate it. The reasons for this will become clear if you think about the way that the following searches work. Briefly though, an inadmissible heuristic will cause the search to find a path that is not minimum cost.

All heuristic searches have roughly the same format as that shown in Algorithm 3. Starting from *node*, the algorithm queues up all of its children in a priority queue, where the priority is given to the child with the smallest associated distance. Then it pops one off and works on that one. The key in all of the algorithms is the nature of the queue and the distance calculator as we'll see below.

Algorithm 3 HSearch(*node*, *goal*, *queue*)

```
while node ≠ goal do
  for all child ∈ node.children do
    distance = calc_distance(child, goal)
    queue.enqueue(distance, child)
  end for
  if queue.isempty() then
    break
  end if
  node = queue.dequeue()
end while
```

2.1 Greedy Search

Greedy search just tries to minimize its heuristic as it goes. Thus the *calc_distance* function simply returns the heuristic and the children are popped off based on their heuristics alone. This is called "greedy" because there is no sense of anything but a myopic estimate. It does not provide us with the lowest cost solution, but rather the shortest one based on heuristics.

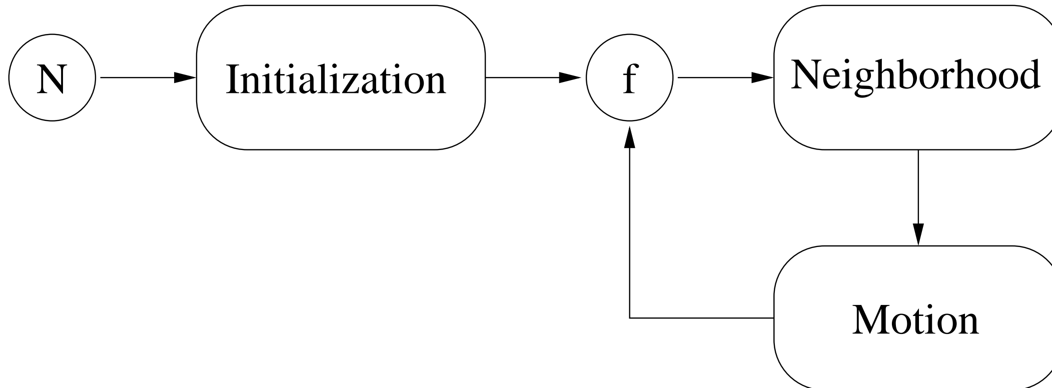


Figure 3: A unified continuous search framework

So, for example, if the heuristic were Euclidean distance to the goal, without regard for cost (admissible and common, but inaccurate), the greedy search would take the shortest possible route to the goal, whether it had to climb vertical cliffs to get there or not.

Because it does not always find the minimum cost path, it is not *optimal*, though it will always find one if it exists, making it *complete*.

2.2 Uniform Cost Search

Uniform cost search does not use the heuristic at all. Instead, it pops nodes from the queue based on the total cost *so far*. It makes use only of the past. In a sense, its heuristic is that it has come as far as it will need to come. It *does* provide us with the minimum cost path and it will always find one if it exists. It is therefore *complete* and *optimal*.

2.3 A^* Search

A^* is one of the coolest searches around. It combines knowledge of the past with its heuristic. The *calc_distance* function adds the total distance so far to the estimate. This gives much better estimates of the overall distance and allows A^* to find the goal much more quickly in general than uniform cost. It is not as fast as greedy to find the goal, but unlike greedy, it is *optimal*. It is also *complete*.

3 Continuous and Social Search

In continuous function domains, we can't exhaustively search anything, so guarantees of optimality and other such stuff fly out the window. We can, however, with a few key assumptions, create some algorithms that do a remarkably good job at searching.

Almost all of these algorithms rely on the concept of a "massless particle" which performs the search. They also all basically follow the same basic format, shown in Figure 3. We initialize the search by tossing some particles into the domain space. If we think of a particle position as an input to the target function and the output of the function as that particle's value, it makes sense that the particle is moving around in the domain space searching for regions of highest (or lowest) value. Thus, we assign values to the particles by plugging their positions into the function and getting the result. Once that evaluation has occurred, we determine neighborhood relationships between the particles and then move them somewhere else in the space. Ideally, we'll end up moving particles to good areas.

For the sake of discussion, it will be assumed that we are always trying to *minimize* a function. Thus, particles are always searching for the lowest possible value. Some of the search techniques have names that

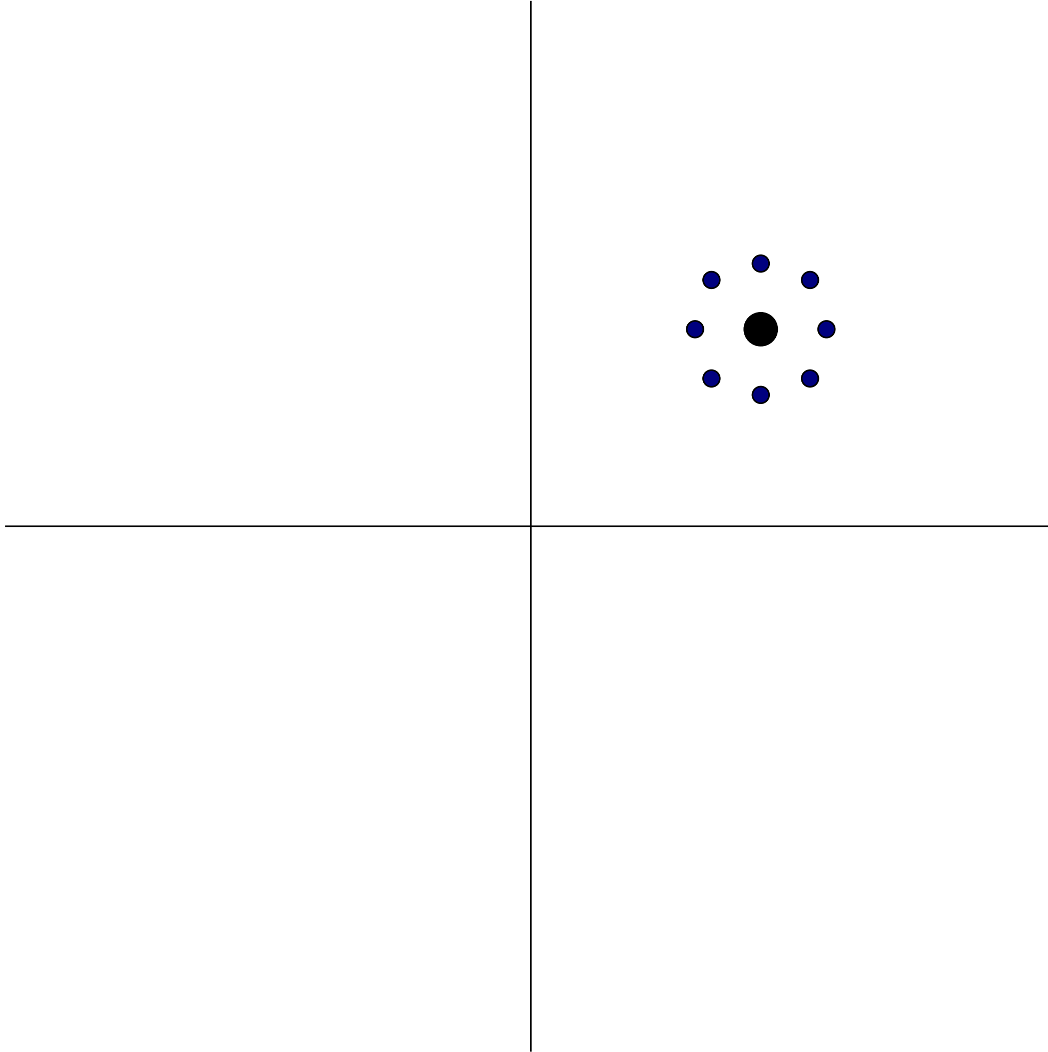


Figure 4: Hill climbing particles

would indicate that maximization is being done (for example, “Hill Climbing”), but the technique applies equally well to minimization. Where confusion is possible, it will be made clear.

3.1 Hill Climbing

This also works for valley-seeking. The concept is exactly the same for minimization as for maximization, it’s just that when minimizing the function, “uphill” is defined as “toward lower values”.

Any number of particles may be used for this search as long as there are at least two of them. The first particle is the “lead particle” and the rest are followers. We place the lead particle into the space and then spread the others around it. The particles can be distributed uniformly as in Figure 4 or randomly.

The neighborhood is set up so that the lead particle uses all of the others. None of the followers care about neighbors at all. The lead particle then moves to the best of its neighbors and the process is repeated.

You have probably noticed that there are some problems with this. First, it tends to get stuck in local optima (little hills or valleys that may not be the highest or lowest). Usually we are searching for the global optimum. If you are creative, you may be able to figure out ways to keep this algorithm from getting stuck

as frequently.

3.2 Simulated Annealing

The idea behind simulated annealing comes from the process of annealing, which heats material up and then cools it slowly so as to encourage molecules to arrange themselves in crystalline structures.

This type of search is traditionally thought of as using exactly one particle, but using the unified framework there are really two of them. It works very similarly to Hill Climbing, but there is always exactly one follower particle.

Additionally, in simulated annealing there is a temperature parameter. This is used to control the optimum-seeking behavior of the lead particle. If the temperature is very low, then the particle will usually respond in two ways:

- The follower will be placed closer to the lead with a lower temperature
- The lead will be less likely to move to worse locations with a lower temperature

Similarly, if the temperature is high, the follower may be placed further away and the lead will be more likely to move to bad locations.

Why would we ever want the lead to move to the location of a "bad" follower? This is to avoid getting stuck in local optima. If there is some probability of going to a position worse than the current position, then there is also a chance that a particle will "jump out" of a local optimum.

The temperature usually starts out high and decreases according to a particular schedule. The schedule is generally carefully chosen so as to ensure that convergence will occur.

If a solution is found, often the annealing process will be restarted (the temperature will be reset). This is called "Simulated Re-annealing" and is a very common extension of the method.

3.3 Simplex Search (Amoeba Search)

This type of search is somewhat unique. The number of lead particles is always $d + 1$ where d is the dimensionality of the search space. This forms a simplex (triangle in two dimensions, tetrahedron in three). The neighborhood is such that every particle is neighbor to every other particle.

The worst particle in the simplex is reflected about the centroid of the other particles as shown in Figure 5. If the new value of that particle is better than the point that used to be the best, then we are onto something, so the particle is moved even further in that direction. This is called an expansion. If it is worse than the worst of the remaining particles, than it was probably a mistake to move that way, so it is moved closer to the center of the simplex. This is called a contraction. Finally, all of the lead points move closer to the center, shrinking the simplex.

The follower particles are simply assigned to the reflection, expansion, and contraction locations at each iteration.

3.4 Ant Optimization

Ant optimization draws its inspiration from the behavior of ant colonies. Ants are very good at not only finding food, but also at finding the shortest path to that food and communicating the path to their peers. Thus, a very short time after a scout finds something interesting, the rest of the workers follow suit. The path gradually gets shorter and shorter until it is of optimum length.

Ants do this by leaving pheromone trails behind as they move. The pheromone dissipates over time so that only well traveled paths continue to be seen as good paths.

This is a little tougher to shoehorn into the framework we have previously discussed, but it is still a particle-based optimization technique and therefore fits into the framework. How to fit it into that setting is left as an exercise for the motivated reader.

Ant Colony Optimization (ACO) is usually used to find minimum cost paths in a space (I bet that you can think of a good application for that right off the top of your head) without doing exponential work. It

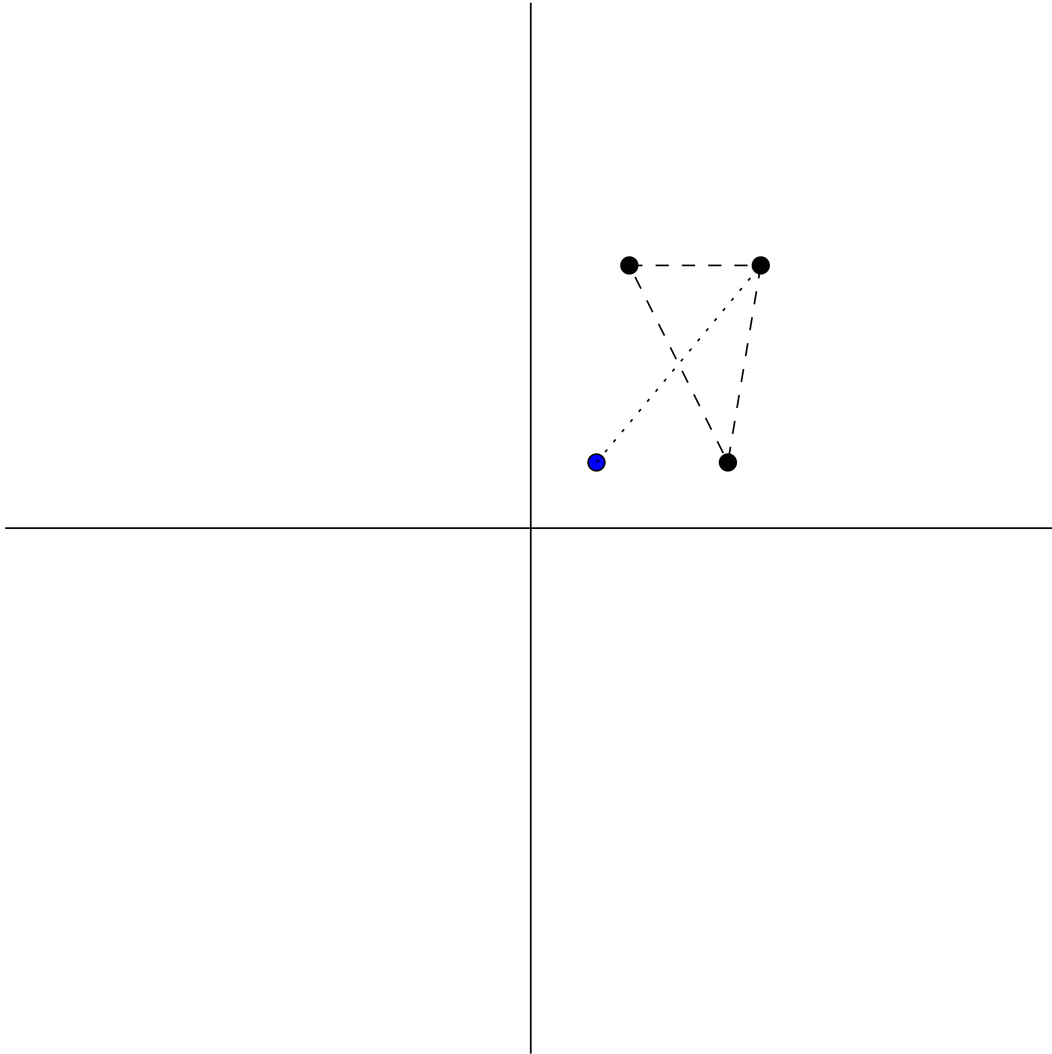


Figure 5: Amoeba Search – Reflection

is a really fun technique to play with, and something that I suggest you look up and learn about. A really excellent website on the topic is found here. If you want to read a good paper about it, go to the "About" section and follow the link for papers on the "ACO metaheuristic". The first paper available for download as a PDF is really good.

This technique is probabilistic. That is a theme that is common to most modern continuous search techniques. Things are not black and white. Some exploration must be done even though a current belief about the goodness of a particular solution may exist. This exploration is often accomplished through means of probabilistically choosing locations even though they are not better. This is evident in simulated annealing, where a high temperature allows a particle to explore parts of the domain space that are not necessarily very promising.

This is a common way of trying to overcome the essentially greedy nature of all of the search algorithms presented here. All of them are interested in finding a better solution than they currently have, yet sometimes to get to that best solution they need to cross through some territory that is not very promising.

Getting back to the original topic, ACO is a very hot topic right now in the optimization and evolutionary algorithms community. It would be a good direction to take if you were looking for interesting research....

3.5 Particle Swarm Optimization

Particle Swarm Optimization (PSO) is the focus of my Ph.D. research and as such is one of my favorite things to fiddle with. That doesn't mean that there will be an unfair focus on that in any tests, but it does mean that I'll probably talk about it from time to time in class. Thanks for your patience.

The idea behind PSO is really very simple, but surprisingly powerful. The idea originally came about while attempting to simulate the motion of birds in a flock. As good motion characteristics were discovered, it was also discovered that the technique could be used effectively to find the global optimum of a function.

With that discovery, the motion algorithm had everything removed from it that did not help in the search for an optimum. The change in motion made the particles look like they formed a swarm of insects more than a flock of birds, and PSO was born.

The motion of particles is really very simple. We first define some values:

$\vec{x}$: Position of the current particle

$\vec{v}$: Velocity of the current particle

ϕ: A random value, drawn from $\mathit{Uniform}(0, 2.05)$ in this case

$\vec{p}$: The best position known by the current particle ("particle best")

$\vec{g}$: The best position seen by all particles ("global best")

χ: A scaling factor used to keep velocities from getting out of control

For each particle, its position is updated using the following simple equations:

$$\vec{v}_{t+1} = \chi\big(\vec{v}_t + \phi_1(\vec{p} - \vec{x}) + \phi_2(\vec{g} - \vec{x})\big) \tag{1}$$

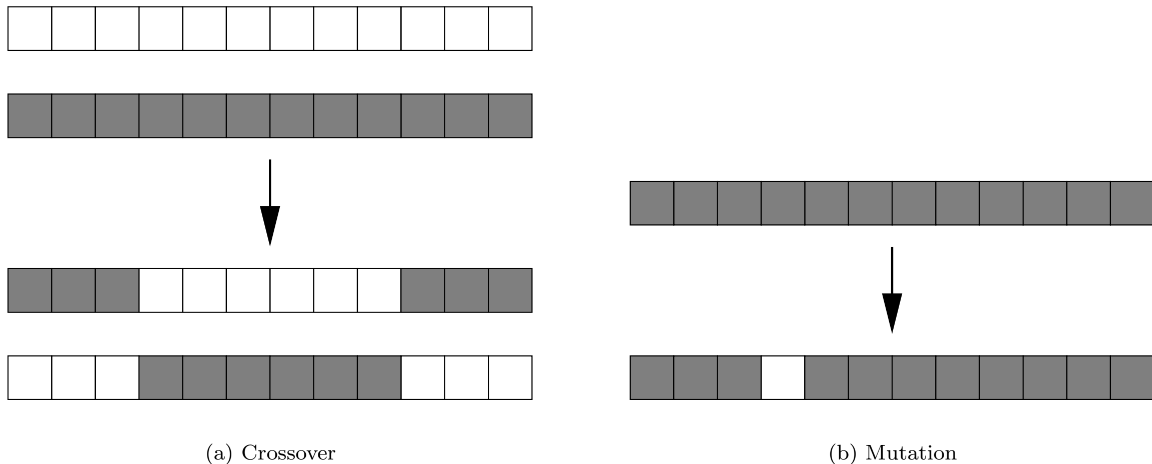
$$\vec{x}_{t+1} = \vec{x}_t + \vec{v}_{t+1} \ . \tag{2}$$

As you can see, this is very simple to implement. In practice, PSO works fairly well, but there are a number of improvements that have been made on it (including my own, of course). It has been found to be a very effective method for things like finding weights in a neural network.

3.6 Genetic Algorithms

These are actually much broader than what we would place into a search setting like this, but they are often used to search for optima, so here they are.

Genetic algorithms (GA) are really quite a neat idea and an extremely hot topic of research right now (in contrast, it appears that PSO research has fizzled out in the last couple of years, but we hope to help



(a) Crossover

(b) Mutation

Figure 6: Genetic operators

resurrect it). The basic idea behind a GA is, of course, inspired from evolutionary processes. Lest you be tempted to think, "But evolution is a wrong idea," just note that evolution is provably happening today within species. It happens all the time, all around us. What I don't believe happens is the evolution of things like monkeys into things like children of God. Regardless, however, of what beliefs we may hold, it turns out that the evolutionary paradigm is very effective at search, besides being a lot of fun to study and play with.

The idea is this: we create a population of particles (they're never called particles in the GA community, though). These particles have three operators applied to them with every "generation". First, several of them are selected to be parents of offspring. This is one of the *selection* steps. Second, pairs of parents mate and generate offspring using some type of crossover operation. This is the *crossover* or *reproduction* step. Third, some of the children are randomly altered, which gives us the *mutation* step. Finally, some of the population is selected to die off, giving us the final *selection* step.

Thus, we apply *selection*, *crossover*, and *mutation* to the population. Over time, we hope that the fittest will survive. In the context of continuous search, the value of the function serves as the fitness (we can decide whether to favor higher fitness values or lower fitness values, depending on whether we are doing maximization or minimization).

The key to making the algorithm converge to good values is to perform selection based on fitness. The least fit particles are selected to die at the end of a generation, and with some probability the best particles are selected for breeding at the start of a generation. Mutation exists to ensure that new areas of the space can be explored. Crossover attempts to combine good solutions in novel ways to create better solutions.

This is a very interesting technique that, as I mentioned earlier, is a hot topic at the moment.

4 Machine Learning is Search

Search is actually one of the more important concepts you will learn if you do anything with machine learning. It has been shown (though it was not necessarily met with much enthusiasm initially) that all of machine learning is search. In fact, I would say that machine learning is usually search through spaces of solutions to extremely hard problems. Optimization is a big part of machine learning, as well, and fits in well with the reinforcement learning community. If you are interested in machine learning in general, you have to understand search. You should also consider taking the class that is completely devoted to it.