Comp 424: Artificial Intelligence

Jacob Reznikov

February 14, 2022

Abstract

My course notes for Artificial Intelligence

1 Search and AI

1.1 Our focus

The focus of this course will be on AI with 3 major components

- **Perception**: The part of the AI that allows it to get information about the task at hand.
- **Actions**: The available actions that the AI agent can make to achieve its goals.
- **Reasoning**: The mechanism by which the AI takes its current representation of the world and turns it into an action.

The most Naive way to approach most tasks is the brute force search method, to use it we must first define a search problem formally

Definition 1.1.1. *A search problem consists of the following*

- ***State Space**: A set S of all possible configurations of the underlying domain.*
- ***Initial State**: A distinguished state $s_0 \in S$ in which we start.*
- ***Goal States**: A subset of states $G \subseteq S$ which we consider preferable.*
- ***Operators**: Actions A that move us from one state to another.*

Associated with these are the following terms

- ***Path**: A sequence of states and operators.*
- ***Path cost**: A number c associated with a path.*
- ***Solution**: A path from the starting state s_0 to any state $s_g \in G$ which is a goal state.*
- ***Optimal Solution**: A solution path of minimal cost.*

We have the following few basic assumptions to simplify our task

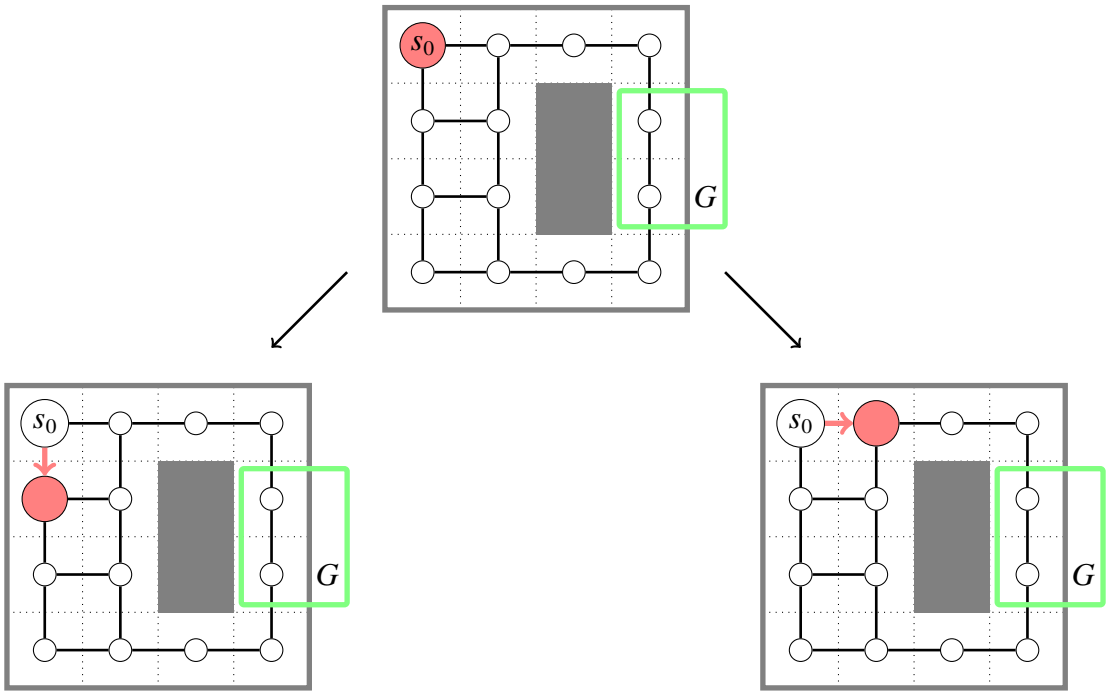
- **Static**: The operators do not change.
- **Observable**: We have perfect knowledge of which state we are in.
- **Discrete**: We have a finite amount of states.

- **Deterministic**: Our actions have no randomness involved and always result in the same outcome.

A very helpful way to represent our task is to define a State Space Graph.

Definition 1.1.2. *A State Space Graph(SSG) of a search problem* (S, s_0, G, A) *is the directed graph with vertex set* S *where two states* (s_1, s_2) *are joined by an edge if there exists an operator* $A_0 \in A$ *that move you from state* s_1 *to* s_2*.*

Example 1.1.3. As an example consider the following state graph



Here from the initial state of s_0 we can make two moves, giving us the new states. This would be the first level of a search tree of the task in the diagram.

Furthermore we can add costs to the operators/edges of the SSG to represent how preferable that path is. This is equivalent to choosing a weighting of the graph. We assume that the weights are all non-negative.

Remark 1.1.4. The SSG is NOT the same as the graph of a search tree.

We now have all the tools necessary to define the main data structure for search algorithms.

Definition 1.1.5. *A search tree is a tree where each node contains:*

- *A state id.*

- *The parent state and the operator used to generate it.*
- *The cost of the path so far.*
- *The depth of the current node.*

We expand a search tree from a leaf with state s by checking all operators we can apply to state s and adding all such states as children of said leaf.

This then leads to the most general search algorithm:

Algorithm 1.1.6. Generic Search Algorithm (GSA)

Data: Search task
Result: Path to goal state
initialize search tree from initial state;
while *there are candidate nodes to expand* **do**
  node ← select node according to some strategy; **if** *node contains a goal state* **then**
    return *the path to this node*;
  else
    foreach *applicable operator* **do**
      succ ← generated successor state;
      add succ as child of node in the tree;
    end
  end
end
return *failure*

The above is a more abstract pseudo code, but it can be implemented using a queue.

Algorithm 1.1.7. Generic Search Algorithm Implementation (GSAI)

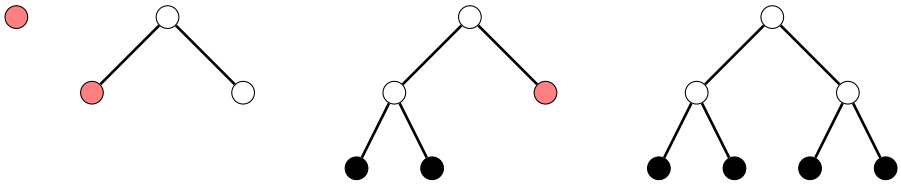
Data: Search task
Result: Path to goal state
queue ← node with initial state;
while *queue isn't empty* **do**
  node ← dequeue front of queue; **if** *node contains a goal state* **then**
    return the path to this node;
  else
    foreach *applicable operator* **do**
      succ ← generated successor state;
      insert succ into queue;
    end
  end
end
return *failure*

Algorithm 1: Generic Search Algorithm Implementation

But note that we can choose how to insert successor nodes into the queue, this is how we can change a search algorithm to fit our requirements.

This leads to the simplest such algorithm, BFS.

Algorithm 1.1.8. Breadth-First Search (BFS)
BFS enqueues nodes at the end of the queue. This means that all the children of any node are visited before any of its grandchildren are.



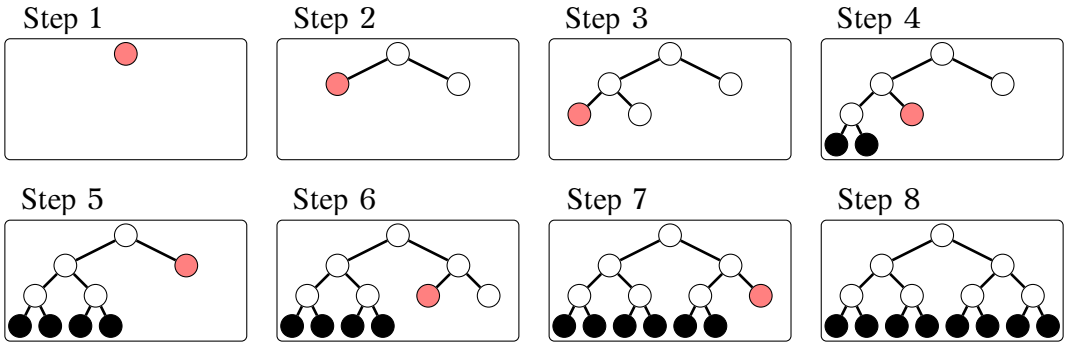
An example traversal is shown above.

In some cases there may be cycles in the SSG and so we want to avoid infinite traversals, because of this we will keep a set (list with no repetitions) of nodes that were visited so far.

Remark 1.1.9. Sometimes it is preferable to allow cycles, either because there are too many states and we can't keep track of which ones we visited or because cycles could allow us to reduce path costs.

A similar algorithm is DFS.

Algorithm 1.1.10. Depth-First Search (DFS)
DFS enqueues nodes at the start of the queue. This means that children of a node are always visited first before its siblings.



An example traversal is shown above.

In some cases there may be cycles in the SSG and so we want to avoid infinite traversals, because of this we will keep a set (list with no repetitions) of nodes that were visited so far.

As we get familiar with more and more algorithms we will need to be able to compare them to one another. We will have many metrics by which we want to weigh them and so we introduce the property stamp.
We will add to this stamp as we continue learning more about AI, additionally a table at the end of this section will summarize all algorithms in the section. We already have some properties we can discuss.

Example Stamp

Name:	Example
Completeness:	True/False
Optimality:	True/False
Space Complexity:	$O(f(\cdot))$
Time Complexity:	$O(f(\cdot))$

- **Completeness**: Are we guaranteed to find a solution if one exists?
- **Optimality**: Is our solution the best solution that exists (in terms of cost)?
- **Space Complexity**: How much storage is needed?
- **Time Complexity**: How many operations does our algorithm perform?

Before we analyse our known algorithms we must first figure out what parameters of the problem they depend on. For this purpose we define two new concepts

Definition 1.1.11. *A **branching factor** b of a state space is the maximal number of operators that can be applied to a single state.*

*The **solution depth** d is the number of operators in the shortest path between the initial state and a goal state.*

*The **depth** m is the maximal length of a path in the state space.*

With this we can now pin down the stamps for our previous two algorithms

BFS Stamp

Name:	BFS
Completeness:	True
Optimality:	True[a]
Space Complexity:	$O(b^d)$
Time Complexity:	$O(b^d)$

[a] If the edges are unweighted

DFS Stamp

Name:	DFS
Completeness:	False
Optimality:	False
Space Complexity:	$O(bm)$
Time Complexity:	$O(b^m)$

Well already we have found a complete optimal algorithm in BFS! This is very good news if this was our goal, however, the complexity of BFS is extremely bad, while the time complexity can be circumvented by just weighting (or parallelization) the space complexity is a much harder problem to fix.

On the other hand DFS solves the memory problem but its time complexity is based on the depth of the entire space not the solution depth, this makes DFS extremely inefficient if the search space is very large. DFS is also not complete since it can get stuck in loops and the solution it finds is almost never optimal.

Additionally neither of these algorithms consider the cost of edges and so that is our next goal. As always we first give names to the sub-cases we can encounter.

Definition 1.1.12. *A **unit cost** problem is a problem where each action has the same cost. A **general cost** problem is a problem where actions can have different costs.*

We also say an algorithm is General Cost Optimal (GCO) if it is optimal for a general cost problem.

So now to solve our issue we consider a slight modification to BFS.

Algorithm 1.1.13. Uniform Cost Search (UCS)

UCS is a modification of BFS where instead of using a normal queue we use a priority queue, we then insert into this queue based on the cost of path to the successor node we are considering, with lower costs being first.

This basically runs BFS but orders the nodes by cost instead of by path length. Because of this algorithms is GCO.

UCS Stamp

Name:	UCS
Completeness:	True
Optimality:	True
GCO:	True
Space Complexity:	$O(bm)$
Time Complexity:	$O(b^m)$

But now let us return to the problem of memory complexity. We saw before that BFS and DFS are two extremes of search algorithms in terms of their properties, it seems then natural to ask if there are any algorithms that combine the two techniques to get some sort of middle ground.

Algorithm 1.1.14. Depth-Limited Search (DLS)

The DLS algorithms modifies the DFS algorithm by terminating any path that is longer than some constant maximum depth d_{max}. This forces the search to always terminate solving the being stuck in loops problem we had before, however, it may not find a solution if the solution is further than its maximum depth allows it to explore.

DLS Stamp

Name:	DLS
Completeness:	False
Optimality:	False
GCO:	False
Space Complexity:	$O(bd_{max})$
Time Complexity:	$O(b^{d_{max}})$

To solve the issue of never finding a solution a clever trick is to iteratively run DLS but increase its maximum depth every iteration.

Algorithm 1.1.15. Iterative Deepening Search (IDS)

IDS does exactly that, it will run DLS repeatedly each time increasing the maximum depth by 1. This actually does not increase the time complexity as the exponential dependence on depth means the lower depth iterations barely contribute.

IDS Stamp

Name:	IDS
Completeness:	True
Optimality:	True
GCO:	False
Space Complexity:	$O(bd)$
Time Complexity:	$O(b^d)$

Remark 1.1.16. Even though the asymptotic behavior of IDS is technically the same as BFS in practice BFS is usually much faster if there is a unique path to the solution.

This algorithm is exactly what we were looking for uninformed search, it has low space complexity and is still complete and unit cost optimal.

1.2 Informed Search

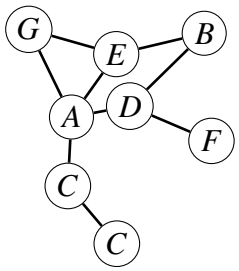
So far we have just been making random moves until we find the goal, but in most tasks we have some idea of what the goal looks like, the idea of how far we are from the goal state is formalized by a distance function $d(n, n_{goal})$.

Definition 1.2.1. *For a task (S, s_0, G, A) we define for any state s the distance to the goal $d(s)$ as the smallest number of operators needed to be applied to reach a goal state $s_g \in G$.*

In the case that (S, s_0, G, A) has associated costs $c : A \to \mathbb{R}^+$ we instead have $d(s)$ being the sum of costs along the optimal path to a goal state $s_g \in G$.

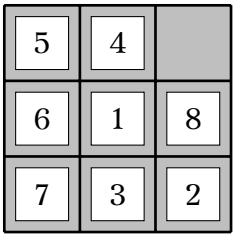
Usually we do not have an *exact* distance function but we instead have some estimate of it called a heuristic.

Example 1.2.2. Consider the task of finding the shortest path along roads between two cities G and F with defined positions as seen on the right. We can have an estimate of the shortest distance to a city by calculating the straight line distance between every city and F.

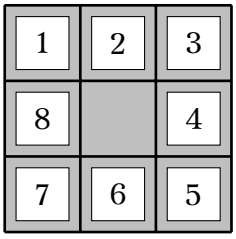


These are not exact shortest distances since the path along the roads is not exactly the straight one. But it is a good estimate.

Example 1.2.3. Let us consider another example called the eight-puzzle. In this puzzle our available operators are to switch the empty tile with any tile adjacent to it.



(a) Start state



(b) Start state

An example of a good heuristic is the number of misplaced tiles, lets call this heuristic $h_1(s)$. Another good heuristic is the sum of distances between the tiles and their goal position, we will call this heuristic $h_2(s)$.

These heuristics here seem sort of natural so its good to ask where do they come from. It turns out that they come from exact distances of *relaxed* versions of the same tasks.

Note that if we relaxed te task to allow a tile to move anywhere from anywhere then h_1 is the exact distance function. On the other hand if we allow a tile to move to any adjacent tile, not just an empty one, then h_2 is the exact distance function.

Heuristics lead to a whole new class of search algorithms that depend on them.

Algorithm 1.2.4. Best-First Search (BFS*)
This algorithm is still based on the generic search algorithm but now we insert into the queue so that the most **promising node** is first according to the heuristic.

When we reach a tie we break it according to shallowest-depth. This makes BFS* act similarly to BFS. We can also consider BFS* as the complement of UCS,

- UCS considers cost-so-far.
- BFS* considers cost-to-go.

Also note that BFS* is a **greedy** algorithm. It tries to maximize short-term advantage without worrying about long term consequences. Because of this fact the performance of BFS* is highly dependent on the quality of its heuristic function. Its stamps are as follows:

BFS* (Worst Heuristic) Stamp

Name:	BFS* (Worst Heuristic)
Completeness:	If finite
Optimality:	False
GCO:	False
Space Complexity:	$O(b^d)$
Time Complexity:	$O(b^d)$
Needs Heuristic:	True

BFS* (Best Heuristic) Stamp

Name:	BFS* (Best Heuristic)
Completeness:	If finite
Optimality:	False
GCO:	False
Space Complexity:	$O(bd)$
Time Complexity:	$O(bd)$
Needs Heuristic:	True

Unfortunately this last algorithm is not optimal because it ignores the cost of the path so far. We can fix this by including the cost of the path so far g in the algorithm.

Algorithm 1.2.5. Heuristic Search (HS)

This fixed search is exactly the same as BFS* but instead of greedily choosing according to the lowest heuristic value h we choose according to the sum of h and g. This is a better estimate of the cost of the current path.

Heuristic search is, however, not an improvement of BFS* in terms of any of our properties.

HS Stamp

Name:	HS
Completeness:	If finite
Optimality:	False
GCO:	False
Space Complexity:	$O(bd)$
Time Complexity:	$O(bd)$
Needs Heuristic:	True

The algorithm is not optimal since if h is sufficiently large it might be more valued than g resulting in unwanted behavior. We fix this with a specific subtype of heuristic function.

Definition 1.2.6. *We say that a heuristic h of task (S, s_0, G, A) is **admissable** if*

$$h(s) \leq d(s), \forall s \in S$$

where d is the distance function for the task (S, s_0, G, A).

Algorithm 1.2.7. A* Search (A*)
This is just HS but with an admissable heuristic function. It fixes the optimality problem of HS. We can actually have an even more restrictive heuristic to improve our properties.

Definition 1.2.8. *An admissable heuristic is called* ***consistent*** *if for every state s and successor state s' we have*

$$h(s) \leq c(s, s') + h(s')$$

An A* algorithm that uses a consistent heuristic is labeled A*+. A*+ is complete since it visits paths from the start state in order decided by f and lower nodes cannot be hidden by other nodes since for any successor s' of s we have

$$f(s) = g(s) + h(s) \leq g(s) + c(s, s') + h(s') = f(s').$$

Thus our optimal solution will eventually be visited and so A*+ is complete.

A* Stamp

Name:	A*
Completeness:	If finite
Optimality:	True
GCO:	True
Space Complexity:	$O(bd)$
Time Complexity:	$O(bd)$
Needs Heuristic:	True

A*+ Stamp

Name:	A*+
Completeness:	True
Optimality:	True
GCO:	True
Space Complexity:	$O(bd)$
Time Complexity:	$O(bd)$
Needs Heuristic:	True

It is useful for actual implementation to have some way to compare two heuristics.

Definition 1.2.9. *Let $h_1(s)$ and $h_2(s)$ be two heuristics of (S, s_0, G, A). We say that h_1* ***dominates*** *h_2 if*

$$h_1(s) \geq h_2(s),\ s \in S$$

Theorem 1.2.10. *Let $h_1(s)$ and $h_2(s)$ be two admissable heuristics of (S, s_0, G, A) where h_1 dominates h_2. A* will always perform better with h_1 than with h_2.*

Intuitively this is true because a dominating admissable heuristic is closer to the optimal heuristic which is $d(s)$.

We can also use a similar trick on A* as on DFS.

Algorithm 1.2.11. Iterative Deepening A* (IDA*)

The algorithm is similar to A* where we set an f-limit starting at 1. We then do a loop where each iteration doubles the limit and runs A* with that limit (A* runs along a path until its f value exceeds f-limit) and see if it finds a solution. This solves our memory problem even if the the heuristic function is really bad.

IDA* Stamp

Name:	IDA*
Completeness:	True
Optimality:	True
GCO:	True
Space Complexity:	$O(bd)$
Time Complexity:	$O(bd)$
Needs Heuristic:	True

We can also learn heuristic functions based on examples. Given some features of the state which we can enumerate $x_1(s), \ldots, x_n(s)$. We can learn a heuristic function of the form

$$h(s) = \sum_{k=1}^{n} c_k x_k(s)$$

where we specifically learn the vector of coefficients $\{c_k\}_{k\in[n]}$.

Property Table Constructive Algorithms

Name:	BFS	DFS	UCS	DLS	IDS	BFS*	HS	A*	A*+	IDA*
Completeness:	True	False	True	False	True	If Finite	If Finite	If Finite	True	True
Optimality:	True	False	True	False	True	False	False	True	True	True
General Cost Optimality:	False	False	True	False	False	False	False	True	True	True
Space Complexity:	$O(b^d)$	$O(bm)$	$O(b^d)$	$O(bd_{max})$	$O(bd)$	$O(bd)$	$O(bd)$	$O(bd)$	$O(bd)$	$O(bd)$
Time Complexity:	$O(b^d)$	$O(b^m)$	$O(b^d)$	$O(b^{d_{max}})$	$O(b^d)$	$O(bd)$	$O(bd)$	$O(bd)$	$O(bd)$	$O(bd)$
Needs Heuristic:	False	False	False	False	False	True	True	True	True	True

1.3 Optimization

So far we dealt with tasks where the 'cost' of a state can be easily expressed as the sum of costs along edges. This assumption is not always valid for the tasks we deal with and so instead we introduce a new type of task.

Definition 1.3.1. *An optimization task* $(S, Eval)$ *consists of*

- ***Solution Space****: S set of solutions to the task (not states of the world).*
- ***Cost Function****: $Eval : S \to \mathbb{R}^+$, evaluates how good a solution is.*

So far we dealt with *constructive* methods that start from scratch and build up a solution. There are also *iterative* methods that start with a solution and improve it.
The most basic such algorithm is what we call a local search algorithm.

Algorithm 1.3.2. Generic Local Search (GLS)

Data: $(S, Eval)$, X_0
Current $\leftarrow X_0$ **while** *Not satisfied* **do**
    Neighbours $\leftarrow$ Generate set of neighbours of Current and evaulate them;
    Next $\leftarrow$ select one of the Neighbors;
    Current $\leftarrow$ Next;
end

Specific algorithms are all variants of GLS and will vary in their satisfaction criteria and the way they select their Neighbors.

Algorithm 1.3.3. Hill Climbing (HC)

In the Hill Climbing Algorithm we simply select the neighbor with the maximum evaluation which is better than our current solution and we are satisfied if no such neighbor exists.

Hill Climbing already has a tuning parameter which is the size of the generated neighborhood. If the neighborhood is large it takes longer to evaluate but wil likely give more accurate results, if the neighborhood is small it is faster to evaluate but will likely be less accurate.

Definition 1.3.4. *Given an optimization task* $(S, Eval)$ *we say that a solution* $s \in S$ *is a* ***Global Optimum*** *if it evaluates to the highest number under Eval over all* $s' \in S$*.*

Given a fixed neighborhood generating function of $(S, Eval)$ *we say that a solution* $s \in S$ *is a* ***Local optimum*** *if it evaluates to the highest number under Eval between it and all its neighbors.*

These concepts are important because an HC algorithm can get stuck at local optimums and miss the global optimums of the solution space. We can try to improve this algorithm by randomly restarting when we plateau to try and find a different region of the state space where we can optimize further. Another thing we can do is to randomly pick any improvement neighbor instead of the best neighbor, this is called randomized hill climbing (RHC).

However, these fixes rarely have large improvements on the outcome, since sometimes we simply need to make some bad moves in order to reach a global optimum. This leads us to the next algorithm.

Algorithm 1.3.5. Simulated Annealing (SA)
SA work similarly to RHC but when we randomly select from all neighbors and then we check if the neighbor evaluates higher. If it does then we just select it, otherwise we have some probability p to still select that bad neighbor and keep going with it.

The value p is something we can control in SA, here are some options:

- We can keep it fixed.
- We can let it decay over time.
- We can make it depend on how bad the move is.

A popular choice for p is $e^{-(E'-E)/T}$ where T is some hyperparameter of the system and E is our current evaluation and E' is the evaluation of the candidate neighbor. This makes the probability quickly go to 0 when the neighbor is much worse than our current node. T can also be changed per iteration.

Algorithm 1.3.6. Parallel Search (PS)
Another strategy to circumvent local optimums is to run many separate searches in parallel starting from random locations. This strategy is especially good recently as advances in GPU computing allow us to run many complex computations at the same time.

Algorithm 1.3.7. Local Beam Search (LBS)
Another idea similar to PS is to connect the parallel searches and fix some constant integer k where each iteration we keep the best k states over all neighbors of all nodes in the previous iteration.

Algorithm 1.3.8. Genetic Algorithms (GA)
We can also borrow ideas from biology and implement genetic algorithms that combine solutions together based on their fitness score.

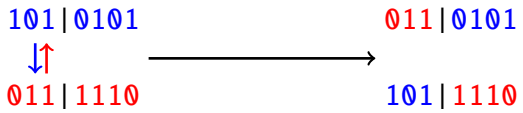
Typically this is done by letting each solution be an 'individual' where the score is that individual's 'fitness'. A set of such individuals is called a 'population' which is typically split into 'generations' where each generation is generated based on the previous generation.

The way this is done is that we randomly pick individuals in a generation based on their fitness (higher fitness individuals are more likely to survive) and then we mix them with mutations and crossovers to get new individuals that hopefully perform better.

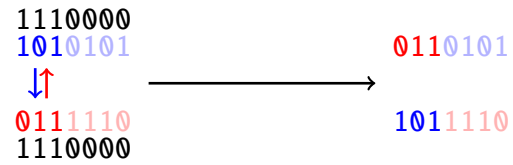
Practically we often store the individual as a binary string representing its DNA where we have a one to one mapping

$$\text{Characteristics} \iff \text{DNA}$$

We can then 'crossover' two strings of DNA to create two new strings, as an example if we have two strings 1010101 and 0111110 we can cross them over as so:



this would be called a crossover with *mask* 1110000 since all the bits in the mask with a 1 get swapped, it is called a mask since we can think of it as covering the strings with a mask and then only swapping the ones with a 1.



We can then choose an arbitrary mask to implement different types of crossovers.

In pseudo-code the genetic algorithm would look like this:

Data: Fitness function F, threshold t, p, r, m
$P \leftarrow$ generate p random individuals;
Compute $F(h)$ for all $h \in P$;
while $\max_{h \in P} F(h) < t$ **do**
  $R \leftarrow$ Select $(1-r)p$ members of P to remain;
  $N \leftarrow$ Crossover $rp/2$ pairs of individuals using some cross over masks;
  $P' \leftarrow R \cup N$;
  Randomly mutate $m \cdot p$ members of P';
  $P \leftarrow P'$;
  Compute $F(h)$ for all $h \in P$;
end
return $\operatorname{argmax}_{h \in P} F(h)$

This, however, is still pretty vague regarding how we select which individuals to remain, here are some common techniques:

- Fitness proportionate selection: We choose the probability that individual h gets selected as
$$P(h) = \frac{F(h)}{\sum_{h \in P} F(h)}$$
- Tournament selection: We uniformly pick two individuals and select the better one with probability p, repeat until reached desired amount. A simple calculation shows that this is equivalent to a linear probability distribution over your individuals.
- Rank selection: We give each individual h probability proportional to its rank when sorted by fitness.
- Softmax Selection: Our probability is
$$P(h) = \frac{\exp[F(h)/T]}{\sum_{h \in P} \exp[F(h)/T]}$$
for some parameter T that we can choose.

It is important to know the ups and downs of Genetic Algorithms

1.3.1 The Good

- Very intuitive.
- Can be extremely effective if properly implemented and tuned.

1.3.2 The Bad

- Performance is HIGHLY dependent on how you encode your problem which in practice is very difficult to get right.
- The more complicated the problem the more parameters there will be tweak in the encoding and in the crossover operations, and bad parameters can result in extremely poor performance.
- If the mutation rate is too small one can often encounter generations full of identical individuals due to not enough evolution occurring.

1.4 Specification

So far we have assumed that we have perfect knowledge of the entire search space (or at least local knowledge) where we know all possible solutions. This is often not the case as real life problems are much easier to describe with a list of *constraints* rather than a list of solutions.

This leads to our next class of search problems, *Searches With Constraints*. The simplest way to incorporate a constraint is to eliminate all solutions that do not satisfy the constraint from some larger all encompassing search space. But we can often do much better than this by cleverly using the constraints to augment our search algorithms.

Definition 1.4.1. *A Constraint Search Problem (CSP) is defined by:*

- *Variables V_i which take their values from their domain D_i.*
- *Constraints $C_j \subseteq \prod_i D_i$ which represent allowed variable combinations.*

Constraints will often be given in the form of a boolean function

$$F_j : \prod_i D_i \to \{True, False\}$$

which are to be interpreted as $C_j = F_j^{-1}(True)$ i.e. all combinations satisfying this function.

Definition 1.4.2. *A **Solution** to a CSP is defined as an assignment*

$$(v_1, \ldots, v_n) \in \prod_i D_i$$

such that

$$(v_1, \ldots, v_n) \in \bigcap_j C_j$$

i.e. $(v_1, \ldots, v_n)$ satisfies all constraints.

Example 1.4.3. An example of a Search With Constraint problem is the coloring problem of a map, where we are given a graph G and a set of colors C and we are asked to color the vertices $V(G)$ such that no adjacent vertices share the same color. A solution would then be an assignment satisfying the above.

Example 1.4.4. Another example is a sudoku grid where we have an initial state of the sudoku grid and we want to place numbers (our variables) such that our grid has each number in every row and every column and the 9 3 by 3 squares. A solution would then be exactly a solution to the sudoku puzzle.

A CSP can often be described by the types of variables that are at play

- If the variables are boolean these are often called **satisfiability** problems.

- If the variables are some discrete set these are often called **coloring** problems.
- If the variables are some infinite discrete set these are often called **integer** problems.
- If the variables are continuous these are called **continuous** problems.

Example 1.4.5. A Linear Program (LP) is a good example of a continuous problem.

The complexity of a CSP can range anywhere from polynomial time like with LP's to undecidability like with some satisfiability problems.

Definition 1.4.6. *A constraint C_j is said to involve a variable V_i if there exist values $v_i, v_i' \in D_i$ and an assignment $(v_1, \ldots, v_i, \ldots, v_n)$ such that*

$$(v_1, \ldots, v_i, \ldots, v_n) \in C_j, \text{ but } (v_1, \ldots, v_i', \ldots, v_n) \notin C_j$$

Intuitively a constraint involves a variable if the value of the variable can change whether the constraint is satisfied.

Constraints can be categorized according to the number of variables they involve, unary constraints involve 1 variable, binary constraints involve 2 variables and so on.

In real life CSP's are often brought about when there are limited resources that need to be allocated.

- When we are scheduling something or deciding on a timetable the resource to be allocated is time.
- When we are choosing a configuration for a computer we are allocating cost and our goal is performance.

Example 1.4.7. An example could be a 4x4 Chess Board where we are tasked to place 4 queens such that none of the queens attack each other.

There are two types of approaches to solve CSPs

1.4.8 Constructivist Approach

This approach is akin to the search space we started the course with, we start with some base initial state and then gradually assign more and more variables while satisfying the constraints.

This approach is very general and works for almost all CPSs. More formally we can get a search space from a CSP as follows.

- Our states S are partial assignments of values to our variables, i.e. $S = \prod_i \{D_i + \varepsilon\}$ where ε represents no assignment.
- Our initial state is no assignments, i.e. $(\varepsilon, \ldots, \varepsilon)$.

- Our operations are to change an unassigned ε an assigned value in D_i.
- A Goal state is a state in $\bigcap_j C_j$ i.e. satisfying all constraints (note that this implicitly required that all variables be assigned).

A strength of this formulation is that the depth of the search tree for this search space is bounded by the number of variables n and so with a moderate amount of variables DFS is a very good algorithm.

However, a naive implementation of a search algorithm directly on this search space can very quickly run into two problems.

- Firstly, DFS or any other search algorithms for that matter cannot recognize the fact that the order in which we assign variables does not matter and so they will over-count possible states by a factor of $n!$ (very big).
- Secondly, a search algorithm cannot recognize that when a constraint is violated any further assignments cannot 'un-violate' it and so it will keep going down undesirable paths.

To fix both of these we have the next algorithm

Algorithm 1.4.8. Backtracking Search (BS)
In BS we fix these problems by selecting only the next variable at each level of the tree, this means that the order is fixed which gets rid of the over counting.

Additionally if a variable assignment violates a constraint then we immediately go back to the previous variable avoiding that entire branch.

In practice this is the basic uninformed algorithm for CSPs and it can solve n-queens for around $n = 25$.

We can also apply several heuristics to the order in which we choose variables to try and make a more efficient tree to traverse.

- A simple heuristic would be to choose the variable with the fewest legal values as choosing it means we are eliminating the largest amount of options.
- Another heuristic would be to choose the variable involved in the largest amount of constraints as they often lead to the fewest legal values. In practice this heuristic is often used to break ties coming from the previous heuristic.
- A more complex heuristic is to instead choose the variable with most legal values as that would more quickly lead to a solution as it is easier to select values for.

1.4.9 Local Search Approach

This approach is more similar to the optimization problems we have been doing, we start with an assignment to all variables that could be broken, but then we gradually fix all broken constraints by re-assigning variables. This approach often uses the aforementioned optimization algorithms to decrease the number of broken constraints.

1.5 Informed Constraint Search Problems

Often times we have a bit more information than the purely abstract problems we encountered before.

Definition 1.5.1. *A Constraint Graph is a graph for CSP, where vertices are variables V_i and two vertices V_i, V_k are connected by an edge if and only if there is a constraint that involves both of them.*

We can use a Constraint Graph data structure to reduce the search space.

Example 1.5.2. If we have two variables V_1, V_2 with

$$D_1 = D_2 = \{1, 2, 3\}$$

then the constraint $V_1 < V_2$ instantly reduces the search space since we know that V_2 cannot be 1 and V_1 cannot be 2.

Definition 1.5.3. *A variable V_i is called **arc consistent** if every assignment $V_i = v_i \in D_i$ **can** satisfy all constraints that involve it.*

A Constraint Graph is arc consistent if all its variables are arc-consistent.

We can optimize our search space by pruning variables that are not arc consistent.

Example 1.5.4. Let V_1, V_2, V_3 be a CSP with

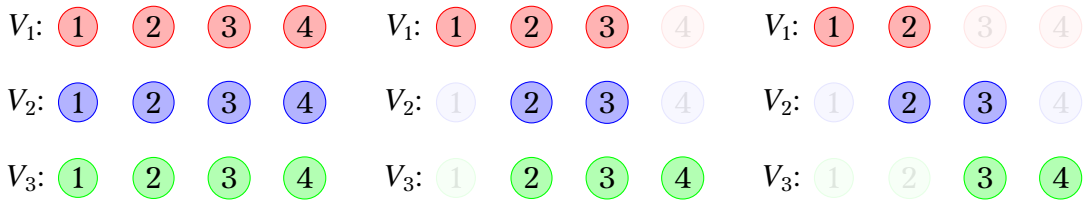
$$D_1 = D_2 = D_3 = \{1, 2, 3, 4\}$$

with constraints $V_1 < V_2$ and $V_2 < V_3$.

On the first round of pruning we find that V_1 is not arc consistent because $V_1 = 4$ violates $V_1 < V_2$ and so we reduce D_1 to $\{1, 2, 3\}$, similarly $V_2 = 1$ violates $V_1 < V_2$ and $V_2 = 4$ violates $V_2 < V_3$ and so we also reduce D_2 to $\{2, 3\}$, we also have $V_3 = 1$ violates $V_2 < V_3$ and so we reduce D_3 to $\{2, 3, 4\}$.

Now this was one round of pruning but we can now preform further pruning. Note that with the changed domains we now also have $V_1 = 3$ violating $V_1 < V_2$ and $V_3 = 2$ violating $V_2 < V_3$ and so we prune the two domains to $D_1 = \{1, 2\}$ and $D_3 = \{3, 4\}$ which leaves the network consistent.

The three steps are shown in the diagram below.



Now after any given inference (pruning) that we compute there are several possible outcomes

- At least one domain is left empty, this instantly means that no solution is possible.
- At least one domain contains exactly 1 element, that variable's value is uniquely determined.
- In any other case we continue as normal and either do more pruning or continue onto the search.

This is then formalized in the following algorithm.

Algorithm 1.5.5. AC-3 Algorithm (AC-3)
The AC-3 Algorithm takes as input some constraint graph and outputs a consistent version of it after pruning.

Data: Constraint Graph G
W $\leftarrow$ all arcs of G;
while *W not empty* **do**
  $(X, Y) \leftarrow$ W.pop();
  prune $dom(X)$ by considering the constraints involving Y //$O(d^2)$;
  if $dom(X)$ *empty* **then**
    return *No solution*;
  end
  else if $dom(X)$ *changed* **then**
    for *N neighbor of X which isn't Y* **do** //$O(e)$
      W.push((N,X));
      This loop adds back all the arcs that might have been undone by the changing of $dom(X)$;
    end
  end
end

Remark 1.5.6. The work-list W *must* be initialized with both directions of each edge as otherwise the algorithm can fail.

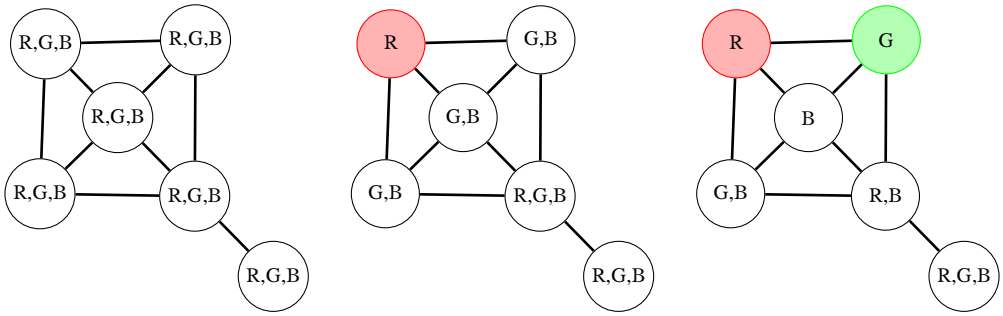
The runtime of the AC-3 algorithm is $O(ed^3)$ where e is the number of edges in the constraint graph G and d is the maximal size of the domains D_i. Note that each run of the loop takes $O(e)$ time and each pruning can generate one such loop. Since the maximal number of prunings is $O(d)$ that gives us $O(d \cdot d^2 \cdot e) = O(ed^3)$.

Another algorithm that uses this extra information is a modification to our uninformed BS algorithm.

Algorithm 1.5.7. Forward checking (FC)
In FC we modify future possible assignments of variables given our constraint graph and current assignments. In essence we are looking into the future to remove options that don't work before we have to try them (hence the name).

This is similar to how, in sudoku, one often writes down the possible numbers to put in a square and removes options as they fill in the rest of the board.

Example 1.5.8. Consider the graph G below and how assignments of colors change the possible domains of unassigned vertices.



In particular the middle vertex has only one available value by the time we reach it, that is a consequence of the FC method.

This algorithm has worst case complexity $O(d^n)$ where n is the number of variables and n is as before. At first this might seem bad but this worst case is really only applicable to very strange graph structures, in practice for most problems we can take advantage of the structure of the graph to reduce its complexity.

As an example if we find that the graph has two disjoint components then we can solve those separately which reduces our complexity from $O(d^n)$ to $O(d^{n*})$ where $n*$ is the size of the largest component. This is a massive change as this complexity decreases exponentially the more components there are.

Further more if the graph happens to be a tree the tree structure allows us to drastically reduce the complexity to $O(nd^2)$ (this is done by starting from a single node of the tree and propagating changes in layers).

In fact even when the graph is 'close' to be being a tree we can get a good reduction. Specifically if we need to remove c vertices to make the graph an induced tree we get that the complexity of the algorithm is $O(d^c(n-c)d^2)$.

2 Advanced Environments

So far we have assumed that our environments are fully observable and deterministic. But in most real world applications we cannot make these assumptions. So we must adapt our techniques to be able to deal with these more advanced environments.

2.1 Sources of uncertainty

We can categorize sources of uncertainty by what information the agent has access to.

Environments may not allow the agent to observe what state it is in. We categorize environments as,

- Observable.
- Partially observable.
- Non-observable.

An agent might not have knowledge about the outcomes of its actions. We categorize environments as

- Deterministic.
- Non-deterministic.

It is important to notice that the state space in these cases **is still there** it is just that the agent has no knowledge about its initial state or the way its operators act.

2.1.1 Non-observable case

Since the state space cannot model the level of 'knowledge' our agent has we must find a different mathematical model to describe it. To do that we introduce *beliefs*.

Definition 2.1.2. *Let $P = (S, s_0, G, A)$ be a search problem we define a **belief** of P to be some nonempty subset $I \subseteq S$ of states.*

Beliefs transform under operators as

$$O(I) = \{O(s) : s \in I\}.$$

Beliefs represent the agent's current options for where it can be. Beliefs induce a much larger search space containing all possible beliefs called the belief space with the operators being the same as above.

It is intractable to consider the entire belief space as it contains $2^{|S|} - 1$ beliefs but note that that a lot of beliefs are not actually reachable from an initial state. This allows us to reduce our belief space considerably.

Definition 2.1.3. *The reachable belief space for a task* (S, s_0, G, A) *is the space defined by,*

$$(B, S, G, A)$$

where

$$B = \{\ \textit{Belief}\ \subseteq S\ :\ \textit{Belief is reachable from } S\ \textit{using operators in } A\}$$

Example 2.1.4. Let (S, s_0, G, A) be the search problem with

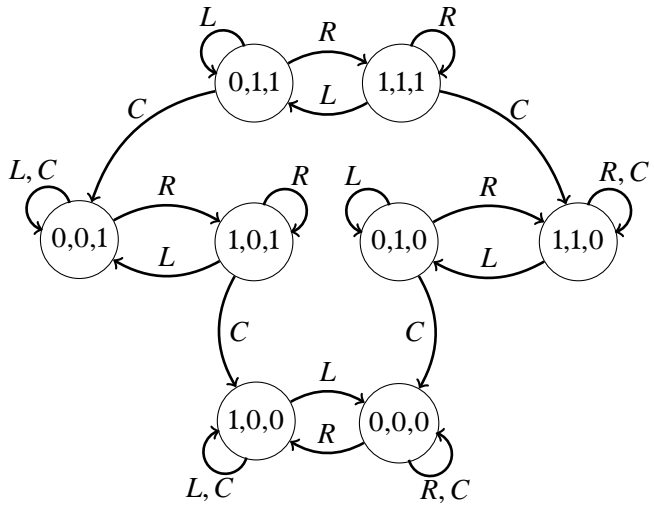
$$S = \{0,1\}^3,\ s_0 = (0,1,1)$$

encoding the position of a rumba in the first bit and the cleanliness of a side of the room with the other two bits.

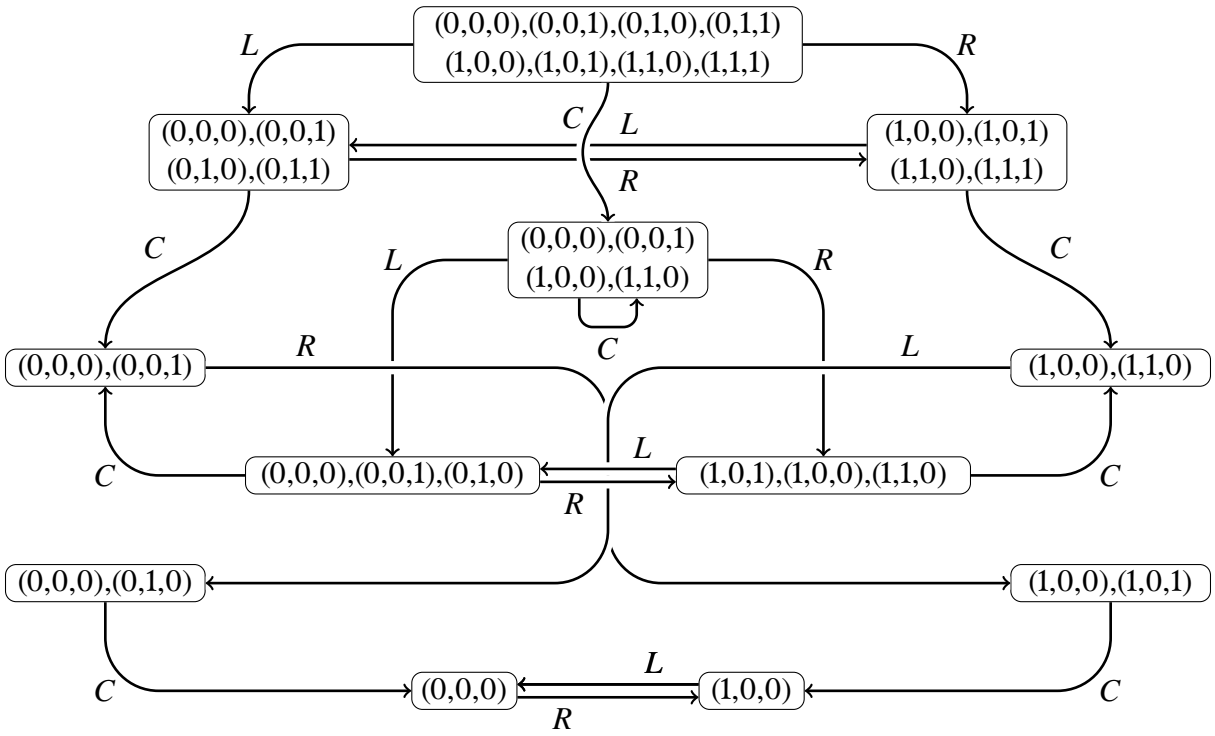
Then we define the goals to be $\{(0,0,0),(1,0,0)\}$ i.e. both sides are clean and then we have 3 operators.

$R(x_0, x_1, x_2) = (1, x_1, x_2)$	move right
$L(x_0, x_1, x_2) = (0, x_1, x_2)$	move left
$C(x_0, x_1, x_2) = \begin{cases} (x_0, 0, x_2) & : x_0 = 0 \\ (x_0, x_1, 0) & : x_0 = 1 \end{cases}$	clean whichever side of the room you are on

We can represent this search problem with its state space graph



In contrast the reachable belief space for this task has initial state S i.e. all possible states are possible since we have no starting information. Then any operator actually reduces the amount of states we have.


(0,0,0),(0,0,1),(0,1,0),(0,1,1)
(1,0,0),(1,0,1),(1,1,0),(1,1,1)
L
R
(0,0,0),(0,0,1)
(0,1,0),(0,1,1)
C
L
R
(1,0,0),(1,0,1)
(1,1,0),(1,1,1)
C
L
R
C
(0,0,0),(0,0,1)
(1,0,0),(1,1,0)
C
R
L
(0,0,0),(0,0,1)
(1,0,0),(1,1,0)
C
(0,0,0),(0,0,1),(0,1,0)
L
R
(1,0,1),(1,0,0),(1,1,0)
C
C
(0,0,0),(0,1,0)
(1,0,0),(1,0,1)
C
L
R
(0,0,0)
(1,0,0)
C