COSC 4368: Artificial Intelligence Programming

Brandon Espina

September 29th, 2023

1 Uninformed Search

Consider the vacuum-world problem where the agent seeks to move the vacuum machine to clean all locations. Assume discrete locations, discrete dirt, reliable cleaning, and it never gets any dirtier.

a) Which of the uninformed search algorithms would be appropriate for this problem?

   Breadth-First-Search seems appropriate for an uninformed search for this problem as it is consistent and will find the goal-state every time, despite maybe not being the most efficient search algorithm.

b) Apply your chosen algorithm to compute an optimal sequence of actions for a 3×3 world whose initial state has dirt in the three bottom squares and the agent in the center square.

Algorithm 1 Breadth-First-Search
Input: visited, graph, node
Output: Sequence of ordered nodes visited in order to reach goal-state

function BFS(*visited,graph,node*)

```
graph = {
  '0' : [['1', '3'], 1],
  '1' : [['0', '2', '5'], 1],
  '2' : [['1','5'], 1],
  '3' : [['0', '4','6'], 1],
  '4' : [['1', '3', '5','7'], 1],
  '5' : [['2', '4', '8'], 1],
  '6' : [['3','7'], 1],
  '7' : [['4','6','8'], 1],
  '8' : [['5','7'], 1]
}

visited = []  # List for visited nodes.
queue = []     # Initialize a queue

def bfs(visited, graph, node): # function for BFS
  visited.append(node)
  queue.append(node)

  while queue:     # Creating a loop to visit each node
    graph[node][1] = 0
    m = queue.pop(0)
    print(m, "clean", end = "-")

    for neighbour in graph[m][0]:
      if neighbour not in visited:
        visited.append(neighbour)
        queue.append(neighbour)

# Driver Code
print("Following is the Breadth-First Search")
bfs(visited, graph, '4')    # function calling
```

end function

2 On Probabilistic Search Algorithms: Implementing and Experimenting with Randomized Hill Climbing (RHC)

2.1 Table:

	Seed = 42			Seed = 43		
(x, y)	$(x_{\max}, y_{\max})$	$f_{\max}$	# Sol'n Gen	$(x_{\max}, y_{\max})$	$f_{\max}$	# Sol'n Gen
(2, 2)	(4.185, 4.1921)	95942.5384	1041	(4.1316, 4.1803)	91919.2523	1041
(1, 4)	(4.1386, 4.1947)	94159.3359	1431	(4.1387, 4.171)	91010.0416	1366
(-2, -3)	(-4.1817, -4.1946)	101565.5188	1236	(-4.1288, -4.185)	97677.4437	846
(1, -2)	(4.1248, -4.1941)	101282.1687	1236	(4.1561, -4.1772)	100423.0669	1496

Table 1: p = 65 and z = 0.2

	Seed = 42			Seed = 43		
(x, y)	$(x_{\max}, y_{\max})$	$f_{\max}$	# Sol'n Gen	$(x_{\max}, y_{\max})$	$f_{\max}$	# Sol'n Gen
(2, 2)	(4.1953, 4.1947)	96789.5941	5201	(4.1657, 4.1915)	94977.0315	5201
(1, 4)	(4.1812, 4.1993)	96771.9236	8001	(4.1708, 4.1984)	96154.7425	7201
(-2, -3)	(-4.1926, -4.1997)	102828.7211	5601	(-4.1623, -4.1988)	101221.5681	4801
(1, -2)	(4.1970, -4.1930)	104678.2512	7201	(4.1586, -4.1983)	103540.9148	7201

Table 2: p = 400 and z = 0.2

	Seed = 42			Seed = 43		
(x, y)	$(x_{\max}, y_{\max})$	$f_{\max}$	# Sol'n Gen	$(x_{\max}, y_{\max})$	$f_{\max}$	# Sol'n Gen
(2, 2)	(3.7743,4.1994)	78708.0697	16186	(4.0442, 4.1999)	90551.5893	17746
(1, 4)	(2.9795, 4.1992)	48757.4135	16186	(3.4078, 4.1998)	64051.8407	19436
(-2, -3)	(-3.1559, -4.1999)	58043.3324	9881	(-3.1788, -4.2)	58905.5005	9946
(1, -2)	(3.5982, -4.1994)	77777.8455	21126	(4.1977, -4.1998)	105695.1209	26586

Table 3: p = 65 and z = 0.01

	Seed = 42			Seed = 43		
(x, y)	$(x_{\max}, y_{\max})$	$f_{\max}$	# Sol'n Gen	$(x_{\max}, y_{\max})$	$f_{\max}$	# Sol'n Gen
(2, 2)	(4.1978, 4.1997)	97602.4304	98401	(4.2, 4.2)	97733.0694	100001
(1, 4)	(4.1992, 4.1996)	97649.6984	139201	(4.1996, 4.1999)	97706.7731	139601
(-2, -3)	(-4.1984, -4.1997)	103111.7292	98001	(-4.1997, -4.1997)	103182.0830	97601
(1, -2)	(4.1999, -4.2)	105830.2061	141201	(4.1998, -4.1995)	105762.2103	140401

Table 4: p = 400 and z = 0.01

[H]

	Seed = 42		
(x, y)	$(x_{\max}, y_{\max})$	$f_{\max}$	# Sol'n Gen
(1, -2)	(4.19995, -4.19991)	105823.2454	3260001

Table 5: p = 100 and z = 0.01

2.2 Effects of sp, p, and z

In my 33rd run, I used a large p value, in which I increase the number of neighbors generated. Having a large p value allows us to more carefully take into account the possible neighbors based on our current value. Large p values create more of a pool of randomness so that we can be more confident in our local maximum answer. The larger the pool the more accurate our results. Analyzing the parameters further, our z value controls the randomness of our neighbors generated (x, y) values. It follows if we want a more accurate solution than we want our neighbors to be generated with a small z value. If we have a large z, then we cannot be sure our neighbors will hold a relevant comparison to our current value. Both of these variables affect the algorithm speed and solution quality. If we want to prioritize solution quality we would opt for a large p and a small z. This of course, would take a long time to complete. If we value algorithm speed over solution quality then we would opt for a small p and a large z.

3 Solving Discrete Constraint Satisfaction Problems

3.1 Question

In this problem we were given a series of constraints across 9 variables: $\{A, B, C, D, E, F, G, H, I\}$ with each variable having a domain of $\{1, ..., 125\}$

Here were our constraints

1. $A = B + C + E + F$
2. $A - C = (H - F)^2 + 4$
3. $D = E + E + 21$
4. $G^2 > E * E + 694$
5. $E + I < D$

3.2 Initial Strategies (Pre-improvement)

We are given a few tools to deal with Constraint Satisfaction Problems from the Lecture Slides. I implemented the Brute Force Search as it was the easiest to understand. Loop through every possibility to find a solution.

3.3 Pseudocode

```
def is_valid(B, F, G, H, I):
    # Check the constraints
    if G ** 2 <= ((H - F) ** 2 + 4 - B - F) ** 2 + 694:
        return False

    if I > (H - F) ** 2 + 4 - B - F + 21:
        return False

    return True


def write_solution(B, F, G, H, I):
    # C can be any value between 1-125
    C = 1
    A = C + (H - F) ** 2 + 4
    D = 2 * ((H - F) ** 2) + 8 - 2 * B - 2 * F + 21
    E = (H - F) ** 2 + 4 - B - F
    result = (A, B, C, D, E, F, G, H, I)

    return result


def improved_BF_search():
    nva = 0
    for B in range(1, 125):
        nva += 1
        for F in range(1, 125):
            nva += 1
            for H in range(1, 125):
                nva += 1
                for I in range(1, 125):
                    nva += 1
                    for G in range(27, 125):
                        nva += 1
                        if is_valid(B, F, G, H, I):
                            sol = write_solution(B, F, G, H, I)
                            return sol, nva
    return "no solution"
```



```
solution = improved_BF_search()
print solution
print nva
```

We have a is_valid function that checks to see if the current values of our variables would pass our constraint checks. Then assuming we find a solution we want to set the rest of the variables that are not dependent on the constraints with write_solution(). In order to reach write_solution(), we brute force iterate through each for loop changing values until we have a set of values that pass our constraints.

3.4 Brute Force Improvement: Runtime Reduction and Pre-Analysis

After familiarizing myself with the constraints, I wanted to apply Constraint Propagation in order to reduce the number of constraints that control our search. This would eliminate variables from our search significantly reducing the run time.

Using algebra and substitution I noticed that we have a couple of equal's signs which means we can look for reductions in our constraint set.
For example:
$A = B + C + E + F$ then we can reduce Constraint 2 to:
$B + E + F = (H - F)^2 + 4$
Continuing my reduction, I was able to reduce our original Constraint Set to only two Constraints:

1. $G^2 > ((H - F)^2 + 4 - B - F)^2 + 694$
2. $I < (H - F)^2 + 4 - B - F + 21$

Since we now only have two constraints, only the variables that are involved in these constraints are apart of our search domain: B, F, G, H, I. This means that the rest of the variables depend on these and can be assigned after our search.

1. $C = 1$
2. $A = C + (H - F)^2 + 4$
3. $D = 2 * ((H - F)^2) + 8 - 2 * B - 2 * F + 21$
4. $E = (H - F)^2 + 4 - B - F$

Note: C can be treated like a constant and can be assigned any value from $\{1, ..., 125\}$ since C is not apart of our constraint variables, and will not affect the correctness of our search.

We can go even further. Since our Brute Force Search increments values, we can see that our new constraint, $G^2 > ((H - F)^2 + 4 - B - F)^2 + 694$, must always be greater than the right-hand side. Since we increment in our brute force we can assume in the smallest case of $H = 1$, $B = 1$, and $F = 1$, that $G^2 > 698$ must always be true. It follows that our smallest G value must be 27. So we can use this to improve our brute force search domain.

3.5 Generic

This question seems contradictory. We were asked to solve a specific set of constraints that can be reduced using Constraint Propagation Techniques that may not apply to other constraint sets. We were able to reduce the amount of variables in our specific search domain by using algebraic substitution and domain restrictions. This would not be considered "generic." My code could be used for other constraint problems but the is_valid() and write_solution() and improved_BF_search() functions would need to be appropriately rewritten to consider the new constraint sets.