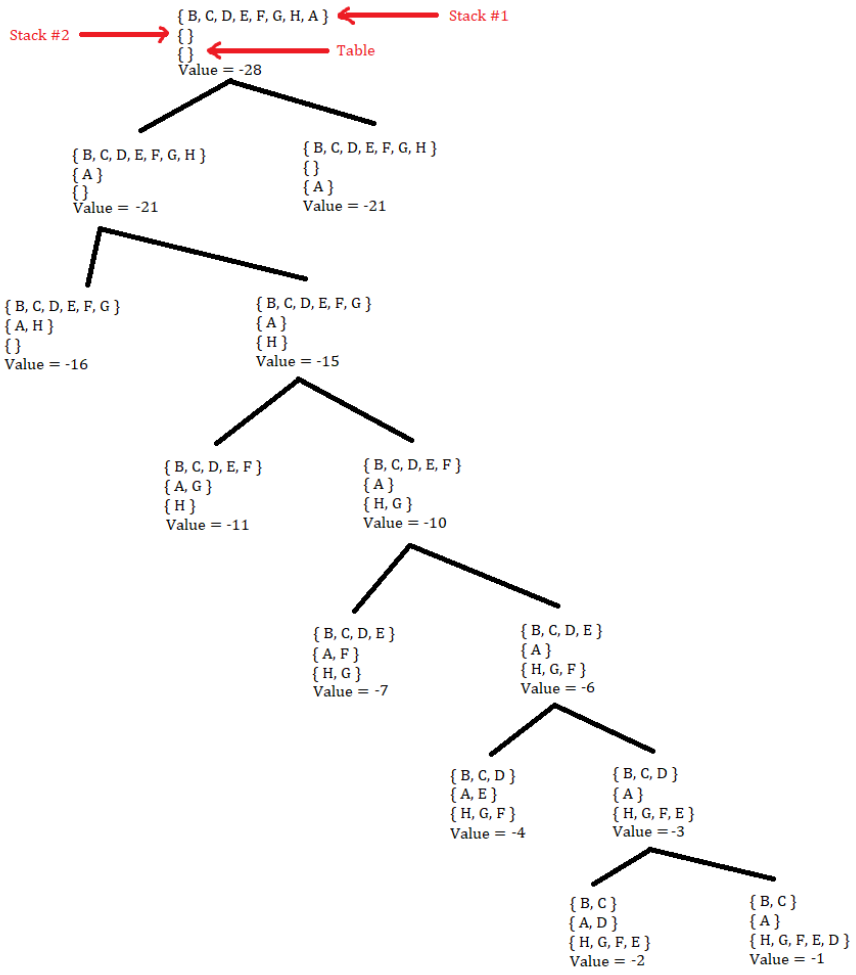
James Hahn
CS1571
M/W 11am-12:15pm
Diane Litman

1. Hill Climbing Search

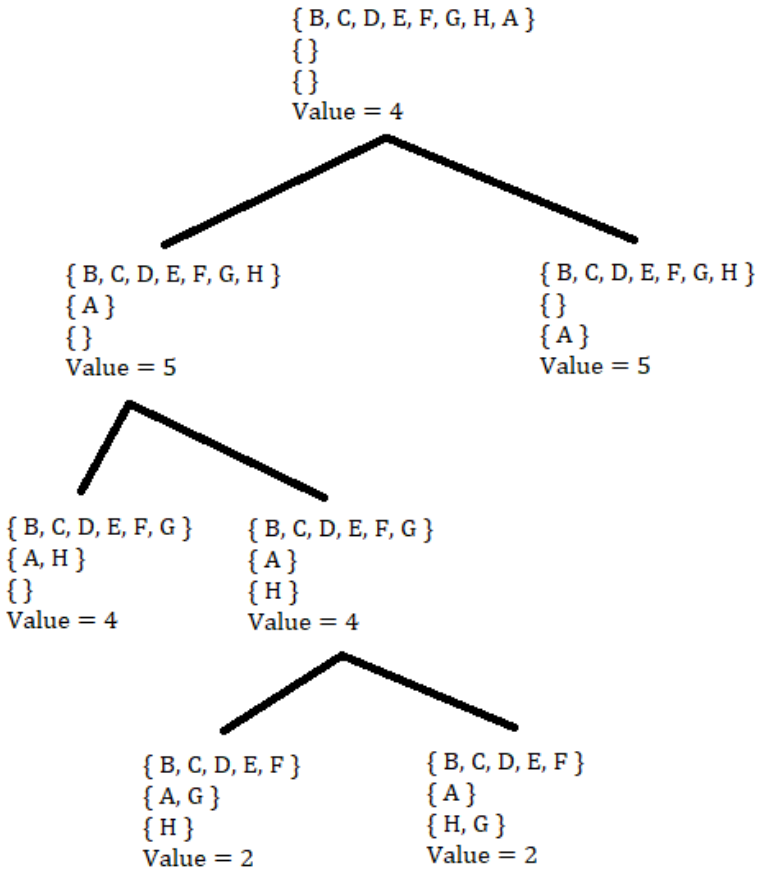
With the initial configuration { B, C, D, E, F, G, H, A } with blocks ordered from bottom to top of the stack, we are given two different heuristic functions to evaluate our navigation through the search space with a hill climbing approach. We can have two stacks at a time and we can place any block we want onto the table. From my understanding, if we place a block onto the table, it counts as being in the incorrect stack even though we are not placing any of the table blocks into a stack, they are just scattered around.

Heuristic 1: The initial state above has a value of -28 and our navigation through the search space looks like below:



Each placement of blocks is represented as a set. Now, keep in mind that the table set is unordered while stack #1 and stack #2 are ordered from bottom to top. As we can see, as we remove the blocks that are placed incorrectly on each stack to another location, our value becomes greater and greater. This is beneficial to us because our initial configuration, while it looks very similar to the goal state, is relatively far off because B, which must be on top of A, is all the way at the bottom. Because of this, we have to remove all the blocks off of B so we can finally put B on top of A. Obviously, this takes quite a few steps, thus indicating we are a significant distance away from the goal state. One great thing about this heuristic function is that as we get closer to developing both stacks, our value increases, indicating that we are travelling closer to the goal state.

Heuristic 2: Now, take a look at our search space for the second heuristic function. The initial state has a value of +4:

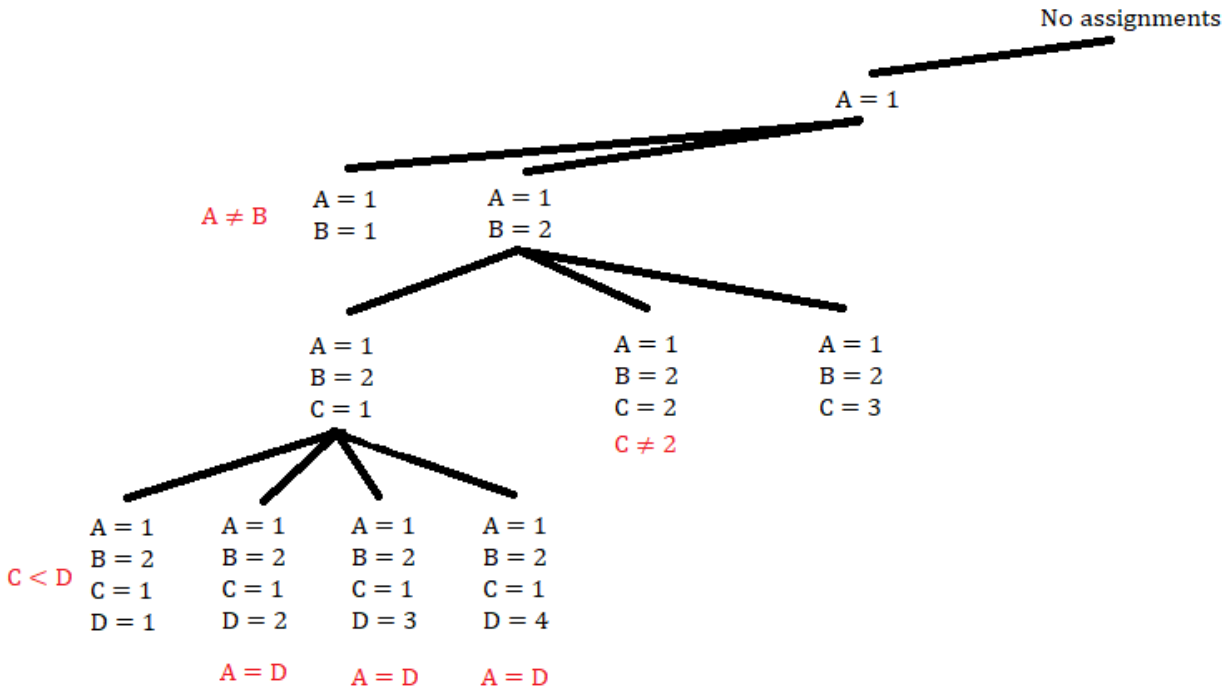


The problem with this heuristic function is that the goal state would have a value of +8. However, we start off with a value of +4. Using common sense, we surely are close to the goal state configuration. However, this approach decreases the value of each state (4 -> 5 -> 4 -> 2) as we travel down the search space. Although it will eventually go back up, this decrease in value indicates we're travelling further away from the goal state. In addition, this heuristic function does not consider how far each block is from its correct placement, but rather if it is locally in the correct location by looking at the block underneath. In addition, every change in value is essentially +1 or -1, providing equal weight to correct and incorrect placement of blocks. However, we should be providing more incentive to blocks placed in their correct positions.

Because of these inherent differences in the two heuristic functions, we can conclude heuristic function #1 is better than function #2 due to its global, rather than local, block placement value assignment. In addition, we move closer to the goal state in terms of value with heuristic #1 (the value increases every iteration instead of decreasing like with heuristic #2). Lastly, our initial configuration, while it seems close to the goal state just by looking at it, is actually far off because we must remove all blocks off of B in order to place B onto A and continue from there, which requires quite a few steps. Heuristic function #2 is too simple of an evaluator and does not accurately represent how far the current state and initial state are from the goal state.

2. Constraint Satisfaction Problems

a. Every red label is the constraint clause that that current state violates. The black labels are the value assignments to the respective variables. Keep in mind this is only a part of the search tree.



b. Below are some examples of each concept would be beneficial if used:

Most constrained variable

With our aforementioned constraints, we can see all packages can be delivered at all four times except for B and C, only able to be delivered at three different times. Out of these two variables, C is more constrained than B. Because of this, we can single out the value of C:

(C ≠ 2) means C can only be 1, 3, or 4

(C < D) means C must have a value greater than it, so 4 is not available; C must be 1 or 3

(E < C) means C must have a value less than it, so 1 is not available; C must be 3

Therefore, we have eliminated three of the four values C can possibly be, thus pruning three-fourths of our tree immediately.

Following that, we have D:

(C < D) means D must be 4 since C is 3 and the only value greater than that is 4

Then, A:

(A = D) means A must be 4 since D is 4

Next, we have B:

(B ≠ 3) means B can only be 1, 2, or 4

(E < B) means B must have a value less than it, so 1 is not available; B must be 2 or 4

(A ≠ B) means B is 2 since A is 4

Finally, E:

(E < A) ^ (E < B) ^ (E < C) ^ (E < D) means E is less than everything; E = 1

As one can see, using the basic constraints provided, we already knew the value of C immediately. C was the most constrained variable, with only 1 valid value to start with, thus pruning a significant chunk of our search space. Because of this, it makes navigating, or setting other values, in our search tree easier due to rules such as (C < D), providing only one value for D, and then (A = D), providing only one value for A. Utilizing the most constrained variable from the beginning reduces the branching factor of our search space.

Most constraining variable

As mentioned briefly in the class slides, most constraining variable is a "tie-breaker among most constrained variables". As such, we can compute how much each variable restricts other variables:

C – must be 3 as shown earlier. Based on C's value alone, we can conclude that D can only hold one value, E can only hold two values, A can hold four values, and B can hold three values

B – must be 2 or 4 if we only consider our initial state having no assignments. From this, A can only hold three values, C can hold three values, D can hold three values, and E can hold three values.

This process continues for the other variables. Obviously, if we choose the variable that constrains the other variables the most, we easily limit the size of our search space. C allows fewer values for D and E than B allows. This is enough merit to show the branching factor of our search space would be smaller and thus easier to navigate.

Least constraining value

This is beneficial because if we assign a value to a variable that allows the most number of value assignments to other variables, we are less likely to fail when traversing down a subtree of the search space. While this may seem computationally inefficient since the search space is larger in theory, we have greater flexibility, or variation, in how we can succeed or fulfill the constraints of the problem. For example, if we used the backtracking approach as mentioned in the project description where we create assignments alphabetically and then chronologically (ABCDE then 1 2 3 4), A can be must be 2, 3, or 4 because one of the constraint clauses is (E < A). We should assign 4 to A because it allows three values to be assigned to E, rather than one if A was assigned to 2. As we can see due to earlier heuristics, A will eventually be 4 in the solution, so this can provide us with the optimal state at each level of the our search space, but the real advantage is more evident in other problems. If a problem had fewer constraints for the solution, allowing more value assignments to E in this specific problem grants us with a greater number of possible solutions in each subtree. That's the true advantage to least constraining value, but may not be the most computationally efficient because we have a greater branching factor at each level of the search tree.

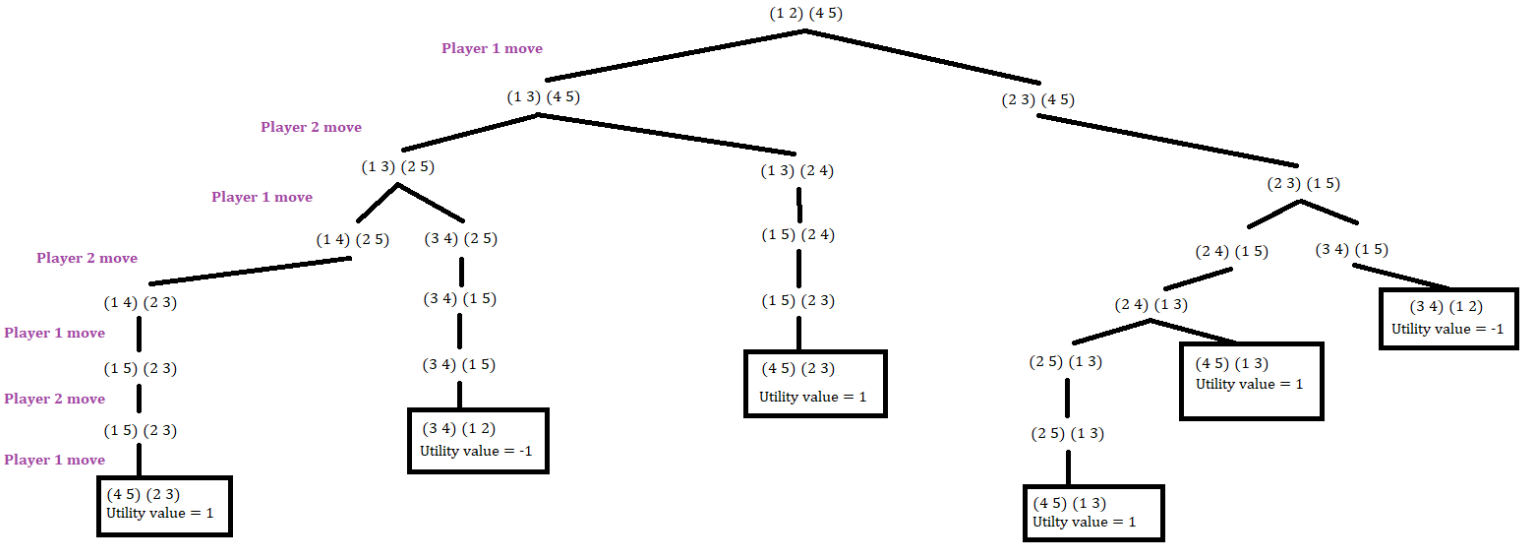
c. Below is the table with forward-checking applied to this problem:

Current assignment being considered	**Domain A**	**Domain B**	**Domain C**	**Domain D**	**Domain E**
A = 1	1 2 3 4	1 2 4	1 3 4	1 2 3 4	1 2 3 4
Failure	*Set to 1*	2 4	1 3 4	1	No values
A = 2	1 2 3 4	1 2 4	1 3 4	1 2 3 4	1 2 3 4
B = 1	*Set to 2*	1 4	1 3 4	2	1
Failure	*Set to 2*	*Set to 1*	3 4	2	No values
B = 4	*Set to 2*	4	1 3 4	2	1
C = 1	*Set to 2*	*Set to 4*	1 3	2	1
Failure	*Set to 2*	*Set to 4*	*Set to 1*	2	No values
C = 3	*Set to 2*	*Set to 4*	3	2	1
Failure	*Set to 2*	*Set to 4*	*Set to 3*	No values	1
A = 3	1 2 3 4	1 2 4	1 3 4	1 2 3 4	1 2 3 4
B = 1	*Set to 3*	1 2 4	1 3 4	3	1 2
Failure	*Set to 3*	*Set to 1*	3 4	3	No values
B = 2	*Set to 3*	1 2 4	1 3 4	3	1 2
C = 1	*Set to 3*	*Set to 2*	1 3 4	3	1
Failure	*Set to 3*	*Set to 2*	*Set to 1*	3	No values
C = 3	*Set to 3*	*Set to 2*	3 4	3	1
Failure	*Set to 3*	*Set to 2*	*Set to 3*	No values	1
C = 4	*Set to 3*	*Set to 2*	4	3	1
Failure	*Set to 3*	*Set to 2*	*Set to 4*	No values	1
A = 4	1 2 3 4	1 2 4	1 3 4	1 2 3 4	1 2 3 4
B = 1	*Set to 4*	1 2	1 3 4	4	1 2 3
Failure	*Set to 4*	*Set to 1*	3 4	4	No values
B = 2	*Set to 4*	2	1 3 4	4	1 2 3
C = 1	*Set to 4*	*Set to 2*	1 3 4	4	1
No values	*Set to 4*	*Set to 2*	*Set to 1*	4	No values
C = 3	*Set to 4*	*Set to 2*	1 3 4	4	1
D = 4	*Set to 4*	*Set to 2*	*Set to 3*	4	1
E = 1	*Set to 4*	*Set to 2*	*Set to 3*	*Set to 4*	1
Success	*Set to 4*	*Set to 2*	*Set to 3*	Set to 4	*Set to 1*

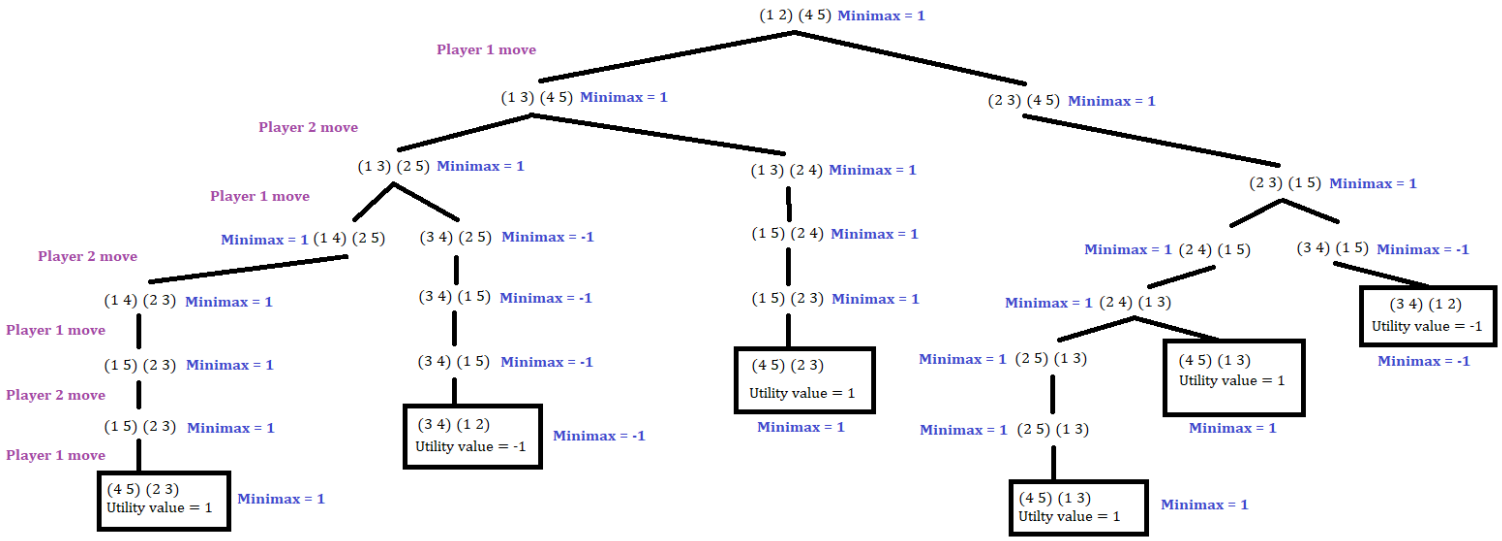
d. Yes, this entire search is severely limited in size if we applied arc consistency. For example, E must be less than A, B, C, and D. There is no other value that E could be other than 1. As we can see above, there are a lot of state assignments where E's domain is just 1. With arc consistency, we would prevent the assignment of A, B, C, or D to 1 at any point in time. This obviously decreases the size of our search tree. The search then becomes quicker and uses less space.

3. Adversarial Search

a. The following is the complete game tree. Instead of labelling every edge with which player made a move, I labelled each "level" of the tree. At every state, the first ordered pair are the white checkers while the second ordered pair is the black checkers. Each number represents which cell each checker is in, from 1 to 5. Each utility value shown in the terminal states is for player one. If the utility value is 1, player one won. Otherwise, if the utility value is -1, player one lost. Please note, we assume player one always makes the first move because the project description does not state otherwise.



b. Below is the same game tree as above but every node has its minimax value for player labelled:



c. Player one would move their checkers to cells 2 and 3; they want to make sure the opponent does not have free reign of their winning cells, cells 1 and 2. Also, if player one makes this move, player two can only make one possible move, so the result is predictable. Next, player two makes their only possible move, taking their black checker at cell 4 and jumping over cells 2 and 3 to reach cell 1; the black checkers are now at positions 1 and 5. As player one predicted, they can just move the white checker at cell 3 to the adjacent cell 4; this is the only possible move he/she can make without losing on the next turn. Then, player two can only move their black checker on cell 5 to cell 3. Finally, player one takes their white checker on cell 2 and jumps over two tiles onto cell 5. The white checkers are now in cells 4 and 5, thus making player one the winner. The following is a sequential list of the actions performed in alternating order of player one and player two respectively:

(1 2) (4 5)
(2 3) (4 5) because player #2 can only make one possible move after this
(2 3) (1 5)
(2 4) (1 5) because player #1 can keep a tile on cell 2 so player #2 can't win on the next move
(2 4) (1 3)
(4 5) (1 3) player #1 wins