Ann Tupek
CS405
Midterm

1: See code

2:

1. Megaman is a robot created by Dr. Light. Dr. Light creates only good robots. Therefore, Megaman is a good robot.
2. Dr. Wily creates only evil robots. Dr. Wily created Cutman, Bombman, Fireman, Iceman, Electicman, and Gutsman. Therefore, Cutman, Bombman, Fireman, Iceman, Electicman, and Gutsman are evil robots.
3. The robot who lives in the quarry is killed by the good robot's weapon. Therefore, the robot who lives in the quarry is killed first.
4. The weapon of the evil robot who lives in the forest can kill the robot who lives in the city. Therefore, the robot who lives in the forest must be killed before the robot who lives in the city.
5. Fireman's domain is underwater.
6. After killing Electricman, Megaman would have to kill two more robots before he can kill Bombman. Therefore, Electricman's weapon cannot kill Bombman, and Electricman cannot be killed later than the third kill, and Bombman cannot be killed earlier than the third kill.
7. The third robot killed lives in the tundra.
8. Electricman needs to be killed after Fireman. Therefore, Fireman's weapon can kill Electricman.
9. Iceman can be killed by the robot in the tundra. Therefore, Iceman must be killed after the robot in the tundra, and Iceman does not live in the tundra.
10. Iceman defeats the robot who lives in the forest. Therefore, Iceman must be killed before the robot in the forest, and Iceman does not live in the forest.
11. The robot who lives in the city has to be killed last.
12. Cutman lives in the quarry. Therefore, because of P3, Cutman must be killed first.

Order	Domain		P9, P10	P6, P8		Robot Order	Domain Order
1	Quarry (P3)			Fireman(i)		Cutman (P3, P12)	Quarry
2				Electricman(i+1)		Fireman	Underwater (P5)
3	Tundra (P7)		Tundra	XXX(i+2)		Electricman	Tundra
4			Iceman	XXX(i+3)		Iceman	Volcano (by elimination)
5	Forest (P4)		Forest	Bombman(i+4)		Gutsman(by elimination)	Forest
6	City (P11)					Bombman	City

The order and domains are as follows:

1. Cutman, quarry (P3, P12)
2. Fireman, underwater (P5)
3. Electricman, tundra (P6, P7, P8)
4. Iceman, volcano (P9, P10)
5. Gutsman, forest (P4)
6. Bombman, city (P11)

3:

Informed searches incorporate a heuristic, or previous knowledge of the search, in its search of the given domain, while uninformed searches do not. Uninformed searches use a brute-force method to exhaustively search for the solution. Uninformed searches are inefficient. They are blind searches and, because they do not keep track of previous searches, they can get caught in loops and potentially never find a solution even though one might exist in the search's domain. On the other hand, because informed searches incorporate a heuristic in their search, they are able to be guided into selecting the next node to search, as opposed to blindly searching through the tree.

Depth-First Search and Breadth-First Search are both uninformed searches, and they both have modifications that can be made to them for new search algorithms. In Depth-First Search, the nodes are followed all the way down to the leaves before backtracking to the top of the tree. A Depth-Limited Search is a modification of Depth-First Search in which the maximum search depth is predetermined and the search does not continue down past that level.

In Breadth-First Search, the first level of nodes is first searched before going down to the next level, and the leaves are searched last for each branch. A Bidirectional Search performs two Breadth-First Searches simultaneously. One search begins from the root, and the other from the goal. The Bidirectional Search requires that the goal is known, and can be used to find a path from the root node to the goal node.

Depth-First Search and Breadth-First Search can be combined into an Iterative Deepening Search. This search begins by implementing a Depth-First Search down to a given level, at which time it implements a Breadth-First Search.

If a weighted tree is being searched for the least-cost path to a node, then the Uniform Cost Search should be implemented. It maintains a list of nodes in order of descending cost, and the weights of the next nodes are added to them. Once all of the nodes have been traversed, the list of nodes is sorted by ascending cost, which will yield the path of least cost.

Breadth-First Search, Bidirectional Search, and Uniform Cost Search are complete in that they can find the solution if it is present in the tree. These three search algorithms are also optimal in that they can find the lowest-cost solution available. In addition, Iterative Deepening Search is also optimal, though it is not complete. Depth-First Search and Depth-Limited Search are neither complete nor optimal.

The efficiency of search algorithms is most often less than desired, and uninformed searches are inefficient as many of them search through the entire tree. The time efficiencies of uninformed searches are as follows:

- Depth-First Search: O(b^m) where b is the branching factor and m is the tree's depth
- Depth-Limited Search: O(b^l) where b is the branching factor and l is the depth limit
- Breadth-First Search: O(b^d) where b is the branching factor and d is the depth of the solution
- Bidirectional Search: O($b^{d/2}$) where b is the branching factor and d is the depth of the solution
- Iterative Deepening Search: O(b^d) where b is the branching factor and d is the depth of the solution
- Uniform Cost Search: O(b^d) where b is the branching factor and d is the depth of the solution

Uninformed search algorithms should be implemented when the search space is small and when time is not a factor. Of the uninformed searches, I would be likely to use Breadth-First, Bidirectional Search, and Uniform Cost Search since they are both optimal and complete.

Informed searches, because they incorporate heuristics, can be more efficient than uninformed searches. Best-First Search, A* Search, and the Hill Climbing Search are all examples of informed searches.

Best-First Search incorporates a function to evaluate the nodes in the graph. The evaluative function f(n)=g(n)+h(n) where g(n) is the estimated cost and h(n) is the heuristic function The nodes are kept in an open list until they are evaluated, at which time they are moved to a closed list. The nodes in the open list are sorted in order of the heuristic function. This is a greedy algorithm because it always chooses the best local option. Two uninformed searches are comparable to Best-First Search: Breadth-First Search is where f(n)=h(n) and Uniform Cost Search is where f(n)=g(n). Best-First Search is not optimal, but it is complete.

A* Search also uses an evaluative function, however it continually re-evaluates the cost function for each node as it re-encounters them. For this reason, A* can find the minimum path from the root node to the goal node. A* is both optimal and complete.

The Hill-Climbing Algorithm is similar to Best-First Search except that it does not allow for backtracking. At each step in the search, only the best node is chosen to be followed. This can lead to finding local maximums, instead of a global maximum. The Hill-Climbing Algorithm is complete in that it can find a solution, but it is not optimal.

The informed searches are more efficient than uninformed search algorithms. The time complexity of Best-First Search is O(b^m) where b is the tree's branching factor and m is the depth of the tree. A* Search is O(b^d) where b is the tree's branching factor and d is the depth of the solution.

Though their performance for worst-case scenarios are comparable to those of uninformed searches, informed searches should be implemented when some characteristic of the anticipated result is known, particularly when looking for the shortest path to the result.

Since we are using heuristics in our checkers game for searching for the best move (ie: more red pieces than black pieces, the ability to gain a king, ownership of the middle of the board, etc), as well as having a time limit for the search, I consider our checker's program to be an informed search. Since we are looking for optimal board positions, we will be able to exclude certain nodes and branches from our search. Our Alpha-Beta Search will be incorporating our heuristics, and is therefore an informed search.

4:

In classical, zero-sum games where valid moves are transparent to both players and one player's gain is the other player's loss, artificial intelligence can be utilized to quickly search a tree of possible moves and, using heuristics that reflect the game's goals, find a move that maximizes the computer's move and minimizes its opponent's move. In classical games such as Checkers and Chess, an artificial intelligence is able to analyze the board quicker than a human opponent. It can also look further into the game and see future results of its current move. It can also keep track of these moves far better than a mediocre human player. It is in these classical games that artificial intelligence can be a real challenge to a human opponent.

However, artificial intelligence in video games is lacking. In many cases, it is little more than a finite state machine that makes 'decisions' based on a set of changing states. Different actions are available to an non-player character, and its decisions are based on various inputs from the environment and its own internal state. Most of the time, its actions are based on looking up information in a table, and it is a stretch to call this table-lookup artificial intelligence.

In video games, artificial intelligence can also be used for path finding, to detect the quickest path to get from its origin to its destination. In path finding, the map is broken up into connected nodes, but because tree-searching algorithms are so expensive, the non-player character's movements are often reduced to another lookup table. This is not always effective. I haven't had the time recently to play many video games, but I recall path finding for NPC's to be particularly poor in a game (I think it was Oblivion). I would be out in the wilderness and some creature would spot me and try to approach, but there would be some large boulder in the way and the creature would keep on bumping into it (because at least the collision-detection was good), and could not get around it. In this particular case, the path finding algorithm did little good, for had the creature only gone around the boulder, it would have been able to attack.

Traditional artificial intelligence techniques such as neural nets are well adapted to classical games. However, because in video games, much of the CPU is taken over by graphics, there is little processing power for artificial intelligence. If console designers would devote an entire chip to AI processing, then I think that video games would be drastically improved and could present more challenging AI to pit against players.

5: See code