Karen Shay West
9 Shannon Marie Way, North Easton, MA 02356

508-844-9776 http://www.linkedin.com/in/karenshaywest KarenWest15@gmail.com

SUMMARY OF RECENT ONLINE TECHNICAL – Python Used For Artificial Intelligence Algorithms And a Given Python PacMan Game – see my Linked In Project Section for details and github links to files:

- **Applied Python to Artificial Intelligence Algorithms:**
  - We used Python and a provided Pac Man game written in Python with the goal of the Pac Man not bumping into walls, or ghosts that would eat them, and to instead find food that gave them points, as they got through various mazes to reach their goal. We learned about:
    - **Search and Planning: Depth First Search, Breadth First Search, Uniform Cost, A-Star Search, Greedy Search, Heuristics and Optimality):**
      - **Solution Search applied to Pacman game:** We implemented breadth-first-search using a FIFO data structure to explore graph paths, depth-first-search using a stack data structure for exploring graph paths, uniform cost search and a-star-search, using priority queues to explore graph paths and had to compare the results. We ran the algorithms on different sized mazes for Pacman to explore and made cost comparisons. With a varying cost function, Pacman may choose different paths. We explored charging more for for more dangerous steps in a ghost-ridden environment and charging less in food-rich areas. A rational Pacman agent would adjust his behavior in response to the cost comparison of the path taken. Each search algorithm is similar except in how they manage the fringe of path exploration (data structures mentioned above). BFS expands outward in every direction, but is not sensitive to exploration cost. Uniform Cost is similar to BFS except that it gets farther in the lower cost area than BFS in getting to the goal node. It is not as good as A-star search, because it explores every direction and has no information about the goal. It starts at the top and works systematically down, finding the cheapest path in terms of cost. DFS goes to the highest depth before going further in the search. We did not do greedy search, but goes straight to the solution node and is not optimal, since it goes through high cost areas to get to the goal. A-star hedges its bets, explores a bit in the wrong direction, just to be sure because the heuristic it uses is imperfect, but it does a very small amount of work, not much worse than greedy, and it is optimal. A-star uses backward costs and estimates of forward costs, that it is optimal with admissible and consistent heuristics, and that heuristic design is the key to it, and sometimes it makes sense to use a relaxed problem, what to use in practice.
    - **Constraint Satisfaction Problems:** Backtracking Search, Constraint Propagation (forward checking, arc consistency, exploiting graph structure)
    - **Game Trees and Decision Theory:** game trees, structured computation, mini-max algorithm, expecti-max agorithm, combinations, evaluation functions, approximations, alpha beta pruning, Baye's Nets, decision theory, preferences, rationality, utilities, maximum expected utility
      - **Multi Agents Added To Pacman, Including Ghosts, and Using Minimax Search Design:** We designed agents for the classic version of the Pacman game which was provided to us, including ghosts. We implemented both minimax and expectimax

searches and tried out evaluation function design. We designed a Reflex Pacman Agent that had to consider both food locations and ghost locations in order to perform well. We found that this agent would likely die if 2 ghosts were on the default board, unless the evaluation function was quite good. We altered the distance to food by using reciprocal values to try a new feature. The evaluation function evaluates state-of-game with an action-for-that-game-state as a pair, so that in later parts of the project, you could evaluate states. Default ghosts were random, but you could also use directional ghosts, and to determine whether our agent was improving, we ran with fixed random seeds (same random choices every game). We then wrote an adversarial agent for Pacman in the Minimax Agent class and it had to work with any number of ghosts, with multiple min layers, one for each ghost, for every max layer. We were given an evaluation function which scored the leaves of the minimax tree. A single search ply was considered to be one Pacman move and all the ghosts' responses, so depth 2 search will involve Pacman and each ghost moving 2 times. The solution was to explore the correct number of game states, to know whether or not you did your implementation correctly. The evaluation function was given to us and checked that we were evaluating game states rather than actions, as we were for the reflex agent, who evaluated both food and ghost locations to perform well. Look ahead agents evaluate future states where as reflex agents evaluate actions from the current state.

  - **Multi Agents Added To Pacman, Including Ghosts, and Using Expectimax Search Design and Writing Better Evaluation Function:** Large boards: Pacman was good at not dying but quite bad at winning. He thrashed around and when next to a dot, would not eat it because he did not know where he'd go after eating the dot. If Pacman thought his death was unavoidable, he tried to end the game ASAP, because of the constant penalty for living, and with random ghosts not always the best choice, but Minimax agents always assume the worst. Alpha-Beta pruning agent: to more efficiently explore the minimax tree. Alpha-Beta Agent's minimax values: identical to the Minimax Agent minimax values, although the actions it selects can vary because of different tie-breaking behavior. Code worked if: correct number of game states were explored - important that we performed alpha-beta pruning without reordering a node's children. Successor states had to be processed in the correct order. Alpha-Beta pruning was not done on all of a node's children, so an equal amount of alpha-beta pruning was not done for each game state explored. Expectimax Agent: useful for modeling probabilistic behavior of agents who may make suboptimal choices. Random ghosts are not optimal minimax agents. Expectimax Agents: no longer take the min over all ghost actions, but the expectation according to your agent's model of how the ghosts act. To simplify: assumed we would be running against an adversary which chose amongst their legal action list uniformly at random. Noticed: a more cavalier approach in close quarters with ghosts. If Pacman perceives that he could be trapped but might escape to grab a few more pieces of food, he'll at least try. A better evaluation function: it evaluated future states, rather than actions from the current state, as the reflex agent did. Evaluation function features: based on how important you think it is, such as distance to food, locations of ghosts, etc.

- **Markov Decision Processes:** policies, rewards and values, value iteration and **Reinforcement Learning:** TD/Q Learning, Exploration and Approximation
  - **Reinforcement Learning Project:** We implemented Value Iteration and Q-learning,

and tested our agents first on GridWorld and later to a simulated robot controller (Crawler) Pacman. Reinforcement learning: an agent that takes actions on an environment, and the output state and reward from the action are passed back to the agent that took the action. It is a Markov decision process, non-determininsitic. You have a set of states with a set of associated actions, and a transition function, a probability of one of the successor outcomes, a model of the dynamics, along with the reward of the associated transition. Markov means that given the present state, the future and past are independent. For decision processes, Markov means action outcomes depend only on the current state - just like search, where the successor function could only depend on the current state and not its history. Deterministic single agent search problems: wanted an optimal plan, or sequence of actions, start to goal. For Markov Decision processes, we want an optimal policy for each state. The idea was to create a Pacman that could learn from his environment. Once Pacman was done training, he should win very reliably in test games, since now he is exploiting his learned policy. This worked for small but not medium grids. To correct this, we had to implement a q-learning agent that learns weights for features of states, where many states might share the same features. Approximate q-learning assumes the existence of a feature function over state and action pairs. Approximate-q-agent uses the an extractor to assign a single feature to every (game state,action) pair. Even much larger layouts should be no problem for approximate-q-learning agent. We then had a learning Pacman agent for small, medium or large grids.