CSC Problem Set 06 +X Report

Name: John Cheek
Email: jwcheek2@ncsu.edu
Section: 001
Collaborators: N/A

Summary:

For my +X report I decided to attempt to solve the problem using exactly what we were told *not* to do when solving these problems. I tried to solve the problem with an approach that doesn't use any sort of proper pathfinding or planning. I tried a variety of things, simply using Manhattan distance, randomization, and ultimately settled on similar ‘improved randomization’ methods to those I used in problem set one. After spending a large amount of time on this approach ultimately (and unsurprisingly), an approach without pathfinding could not solve the problem. However, interestingly for every single map the agent found all the Chips but struggled pathfinding to the goal after achieving all the chips.

"+X" Concept:

In class, we have talked about the benefits of A* pathfinding and the applications of certain pathfinding algorithms, for my +X for this problem set I wanted to experiment with those and try to find out *why* poor pathfinding methods do not work for problems such as these. I also made many modifications. I tried a variety of things that were inspired by past problem sets. First I tried a direct distance approach. It would pick a goal (keys, locked doors, chips, the goal) then it would move straight towards the selected objective. Obviously this ran straight into impassable tiles but it was a decent starting place. I then built upon this direct approach to move randomly if it got stuck. It would tell it got ‘stuck’ if it tried to take the same move more than a set threshold amount of times (I used 5). This would not solve any of the problems so I built in the ‘improved randomization’ logic I used in my first problem set where it would act randomly but prevent taking moves that undid the previous one (I.E. if the agent randomly chooses up, it won’t choose down directly after). The final thing I added was if the agent got stuck, it would keep track of visited tiles and backtrack to unvisited ones. The technical implementation of my final approach is as follows:

Technical Implementation of +X:

In a similar manner to my proper solution, this approach first selects a goal in the order of nearby keys, then nearby doors, (which needed to be added for the +X percent), then chips, then the goal. To find these goals, it used a simple distance calculation:

```
private Position findClosestTarget ( final Position start, final ArrayList<Position> targets ) {
    return targets.stream()
            .min( Comparator.comparingInt(
                    t -> Math.abs( start.getRow() - t.getRow() ) + Math.abs( start.getCol() - t.getCol() ) ) )
            .orElse( null );
}
```

To reach these different goals it does not use any sort of planning, but a combination of rather an ‘improved random’ movement system that backtracks to unvisited tiles when stuck. Firstly, it will move directly towards the selected goal with the following code:

```
private Action determineMove ( final Position current, final Position target ) {
    final int dRow = target.getRow() - current.getRow();
    final int dCol = target.getCol() - current.getCol();

    if ( Math.abs( dRow ) > Math.abs( dCol ) ) {
        return dRow > 0 ? Action.MOVE_DOWN : ( dRow < 0 ? Action.MOVE_UP : Action.DO_NOTHING );
    }
    else {
        return dCol > 0 ? Action.MOVE_RIGHT : ( dCol < 0 ? Action.MOVE_LEFT : Action.DO_NOTHING );
    }
}
```

Then, if it gets stuck on a wall or other impassable tile (if it takes the same move an amount of times >= the ‘Stuck threshold’ [which was five in this case]) it will move randomly with the improved random logic and the following code with help from the backtrack stack:

```
      if ( target != null ) {
          backtrackStack.push( currentPos );
          chosenAction = determineMove( currentPos, target );
      }
      else if ( !backtrackStack.isEmpty() ) {
          final Position backtrackTarget = backtrackStack.pop();
          chosenAction = determineMove( currentPos, backtrackTarget );
      }

      if ( chosenAction == lastAction && chosenAction != Action.DO_NOTHING ) {
          repeatCount++;
      }
      else {
          repeatCount = 0;
      }
      lastAction = chosenAction;

      if ( repeatCount >= STUCK_THRESHOLD ) {
          chosenAction = improvedRandomMove();
          repeatCount = 0;
      }

      return chosenAction;
  }

  private Action improvedRandomMove () {
      final List<Action> possible = new ArrayList<>(
              Arrays.asList( Action.MOVE_UP, Action.MOVE_DOWN, Action.MOVE_LEFT, Action.MOVE_RIGHT ) );
      final Action inverseLast = inverseOf( lastAction );
      possible.remove( inverseLast );
      possible.removeIf( action -> isVisited( getNewPosition( env.getRobotPosition( this ), action ) ) );

      return possible.isEmpty() ? Action.DO_NOTHING : possible.get( random.nextInt( possible.size() ) );
  }
```

Evaluation and Results:

For the sake of testing, I removed the maximum interaction count and increased the timeout to 45 seconds. My finished combination of a simple pathfinding algorithm without using any sort of planning or pathfinding passed Test Cases 1, 2, 3, 5, and 6. This is far from all 10 cases, but more than I thought I would be able to achieve without planning or pathfinding.

Interestingly, with this logic the agent collected every chip in each trial but failed to reach the final goal. Upon analyzing this, I realized the agent would collect all the chips, then get stuck in a corner. The reasoning for this was likely the logic I was using when dealt with being ‘stuck’. My logic to handle that involved backtracking until nearby an unvisited tile then moving to that tile, however; since my approach prioritizes unvisited tiles when finding goals, it is likely that every tile had already been visited by the time all chips had been collected.

Ultimately, while I knew I would not be able to create a perfect solution without using planning, I was surprised about what I was able to achieve.