VNUHCM – UNIVERSITY OF SCIENCE
Faculty of Information Technology

Artificial Intelligence

PROJECT 01 - FINDING OPTIMAL PATH REPORT

Class: 19CLC5
Members:

- 19127038 – Phạm Trọng Vinh Khuê
- 19127191 – Ngô Văn Anh Kiệt

Hồ Chí Minh, ngày 18 tháng 7 năm 2021

Contents

1. Adversarial Search theory summary

1.1. Adversarial Search

In multiagent environments, the unpredictability of other agents introduces contingencies into the searching process. Adversarial search is a game-playing technique where each agents have conflicting goals which created a competitive environment.
Four main types of Games in AI including perfect and imperfect information, deterministic and non-deterministic games (i.e., Chess, Poker). The typical assumptions in game theory usually are deterministic, two players (MIN and MAX), turn-taking, rational players, zero-sum games of perfect information. Those can generalize to stochastic games, multiplayers, non-zero sum, etc. Games are modeled as a search problem and heuristic evaluation function which help to model and solve games in AI. Solution for games is strategy, with time limits force an approximate decision to be taken and efficiency is based on evaluating the current game position.
To play a game, a game tree will be created to know all the possible choices and to find the optimal strategy. Games formularization as search will have:

- Initial state: Game set up
- Player(s): MAX/MIN
- Action(s) – Successor function: List of legal moves
- Result(s, a) – Transition model: result of move *a* on state *s*
- Terminal-Test(s): End state
- Utility(s, p) – Utility function: A numerical value of a terminal state *s* for a player *p*

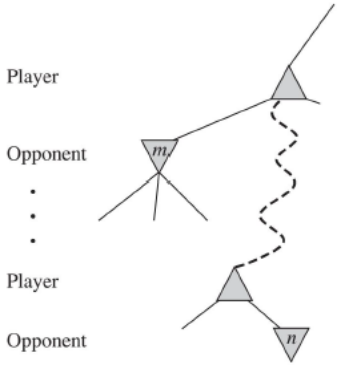
1.2. Types of algorithms in Adversarial search

- Minmax Algorithm

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$

  - ✓ The Minimax algorithm tries to predict the opponent's behaviour. It predicts the opponent will take the worst action from MAX viewpoint.
  - ✓ Perfect play for deterministic, perfect-information games.
  - ✓ A complete depth-first exploration of the game tree.
  - ✓ Time complexity : $O(b^m)$
  - ✓ Space complexity: $O(b * m)$ where b is branching factor and m is the maximum depth of the tree.
  - ✓ Problem: Number of game states is exponential in the number of moves

- Alpha-beta Pruning
  - ✓ Alpha-beta pruning: Prune away branches that cannot possibly influence the final decision.
  - ✓ If a node n is determined to be worse than its previous node, that n node can be pruned.
  - ✓ α is the best value to MAX up to now for everything that comes above in the game tree. Similar for β and MIN.
  - ✓ Algorithm: Depth-first search – only consider nodes along a single path at any time.
  - ✓ Good move ordering improves effectiveness of pruning.
  - ✓ With "perfect ordering", time complexity = $O(b^{m/2})$
  - ✓ Transposition table avoids re-evaluation a state.



In real-time decisions, those searches above have impractical depth as they reached terminal state. Hence, standard approaches for practical implementation are introduced:

- Cutoff test
  - ✓ Limit depth
  - ✓ Iterative deepening
  - ✓ Quiescence search: Expand nonquiescent positions until quiescent positions are reached.
  - ✓ More sophisticated tests include Horizon effect, Singular extension.
- Evaluation function
  - ✓ When search is cut off, current state is evaluated by estimating its utility. This becomes an estimation for the desirability of position.

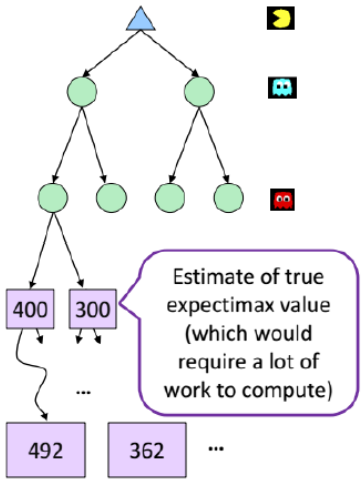
$$\text{H-MINIMAX}(s,d) = \begin{cases} \text{EVAL}(s) & \text{if CUTOFF-TEST}(s,d) \\ \max_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if PLAYER}(s) = \text{MIN}. \end{cases}$$

Other improvements include beam search and search vs. look up.

1.3. Stochastic games

Games have chance elements like dices that resulted in uncertain outcomes.

- Expectimax search is introduced to maximize average score by interleave chance nodes with min nodes.
- The underlying uncertain-result problems can be formulated as Markov Decision Processes.
- Expectimax requires the full search tree to be explored. No pruning can be done as the value of a single unexplored utility can change the expectimax value.
- It is sensitive to monotonic transformations in utility values which mean magnitudes of the evaluation function values matter.
- Expectimax can be implemented using recursive algorithm. The function is called recursively until reach a terminal node, then return the utility for that state. If the current state is a max node, return the maximum of the state values of the nodes successors. If state is a chance node, expand all probability then return the expected utility given by $\sum_i P(i) \times Max(i)$.



2. Source Code Running

The program is implemented in Python, so that it is albe to run the program from the command line, but you must ensure that your Python has installed following modules:

- Pillow (PIL): using for image execution
- Numpy: using for array manipulation
- Heapq: using for priority queue
- Itertools: using for iteration
- Math: using for calculation
- Typing: using for typing support
- Re: using for regular expressions

These modules have significant impacts on our program.

After having installed modules, you can activate our program by directly running the *main.py* file from the command line. After running, many instructions will let you know how to use our program.

3. A* searching algorithm

3.1. A* searching function

```
def A_Star_Search(graph, heuristic):
    initialize frontier – a priority queue
    initialize graph.start
    frontier.push(graph.start)
    while (!get result or fail)
```

```
cell = frontier.pop()
if (frontier is empty)
    return False
if (cell is the goal)
    graph.tracePath()
    return True
cell.visited = True
moveList = possible move from cell
for nextCell in moveList
    calculate f(n) = g(n) + h(n)
    if (nextCell.visited = False and !frontier.hasItem(nextCell))
        nextCell.parent = cell
        update the path that will be traveled by the next cell
        frontier.push(nextCell)
    else
        if (frontier.hasItem(nextCell) and the new heuristic is lower)
            nextCell.parent = cell
            update the path that will be traveled by the next cell
            change heuristic value of the cell in frontier
```

A searching algorithm pseudocode*

A* search algorithm basically a pathfinding and graph traversal technique. It efficiently plots a walkable path among multiple nodes on the graph. By introducing heuristic, A* can estimate the minimum cost between start node and target node, hence can find an optimal path much faster. First, a priority queue (frontier) is created to store the unexamined cells, starting from the start cell. The cell being examined is marked as visited. The algorithm then considered all accessible adjacent cells (nextCell) of the most recently added cell in the frontier queue using a for loop. Here, function $f(n) = g(n) + h(n)$ is calculated where:

- $f(n)$ = total estimated cost of path through cell n
- $g(n)$ = cost so far to reach node n
- $h(n)$ = estimated cost from node n to the target node. This is the heuristic part in A* search algorithm.

Those adjacent cells then being check whether they had been visited or put in the queue. If not, push those next cells in the frontier queue. Otherwise, if the queue was already stored the nextCell and the new heuristic is lower, change heuristic value of the cell in frontier. Both cases required marking the current cell as their nextCell's parent and storing the travelling path.
This process is recursively repeated (while loop) until there are no unvisited cell left (frontier empty) or it had found the shortest path (target cell is reached).

3.2. Graph implementation

Graph is a special class implemented to ultilize program's performance. The Graph class consists of two 2D array: one 2D array simulate the topographic map with *m* x *n* cell (*m* and *n* are dynamic, depend on the input file) containing the "height" of this point; one 2D *m* x *n* array containing "information" of corresponding cells including index of the parent cell, indices of children cells, information of interaction and visited, value g(n), ... The graph have 2 tuples represent the start point and the goal point, along with the value *m* for calculating requirements. Some functions that support manipulating data also implemented in the class.

3.3. possibleChild function

```
def possibleChild(cell):
        A = list of 8 neighbor cells
        For neigbor in A
            if (neighbor is a valid cell)
                if (neighbor is a valid child)
                    cell.child.append(neighbor)
                if (neighbor.interacted == False)
                    neighbor.interacted = True
                    totalInteracted += 1
```

possibleChild pseudo code

possibleChild is a funcion implemented in Graph class that search for possible children of a cell. It scan 8 neighbor cells for valid children and check if the child is interacted or not (for calculating purposes).

3.4. tracePath function

```
def tracePath():
        outImg = image in RGB mode

        current = getGoal()
        while (current != start)
            redraw current as red color
            current = current.parent
        save outImg to output path
        outfile = open output file to write
        outfile.write(goal.g(n), totalInteracted)
        outfile.close()
```

tracePath pseudo code

tracePath is a function implemented in Graph class that highlight the path found by A* algorithm when reaching the goal. First, it convert the input grayscale image to RGB image and redraw on that image. The highlight color is setted at 255 (red). By repeatly redraw and trace back to the start point, the path is highlighted. The function also save the redrawn image to a file and print required parameter to another output file.

4. Heuristic functions

4.1. Euclidean heuristic

```
def h_Euclidean(nextPoint, goal)
    dx = goal.x – nextPoint.x
    dy = goal.y – nextPoint.y
    da = goal.a – nextPoint.a
    return sqrt(dx * dx + dy * dy + da * da)
```

Euclidean heuristic pseudo code

Euclidean heuristic applies the Euclidean distance between two points as the heuristic value by using (x,y,a) as coordinates in 3D coordinate system. In mathematics, Euclidean distance is the length of a line segment connecting two points. The Euclidean distance can be defined as the equation:

$$d = \sqrt{(\Delta x^2 + \Delta y^2 + \Delta z^2)},$$

where Δx, Δy, Δz are differences in coordinates.
In the case of this problem, the Euclidean distance represents the shortest path from nextPoint to goal with the assumption that there is no obstacle between them.

4.2. Manhattan heuristic

```
def h_Manhattan3D(nextPoint, goal)
    dx = abs(goal.x – nextPoint.x)
    dy = abs(goal.y – nextPoint.y)
    da = abs(goal.a – nextPoint.a)
    return float(dx + dy + da)
```

Manhattan heuristic pseudo code

Manhattan heuristic applies the Manhattan distance between two points as the heuristic value by using (x,y,a) as coordinates in 3D coordinate system. In geometry, Manhattan distance, also known as taxicab metric, is the sum of the absolute differences of their Cartesian coordinates.

$$d = |\Delta x| + |\Delta y| + |\Delta z|,$$

where Δx, Δy, Δz are differences in coordinates.

In the case of this problem, Manhattan distance defines the shortest path from nextPoint to goal that only consists of horizontal, vertical and perpendicular moves.

4.3. 3D octile heuristic

```
def h_Octile3D(nextPoint, goal)
    dx = abs(goal.x – nextPoint.x)
    dy = abs(goal.y – nextPoint.y)
    da = abs(goal.a – nextPoint.a)
    deltas = [dx, dy, da]
    deltas.sort()
    return dx + dy + da - (3 - sqrt(3)) * deltas[0] - (2 - sqrt(2)) *
deltas[1]
```

3D octile heuristic pseudo code

3D octile heuristic applies the octile distance between two points as the heuristic value by using (x,y,a) as coordinates in 3D coordinate system. 3D octile distance can be referred as an upgrade of Manhattan distance in 3D space. 3D octile distance allows diagonal moves that significantly reduce the path length. In particular, in 3D coordinate, a diagonal move replaces a combination of 3 basic moves used in Manhattan distance - horizontal, vertical and perpendicular, which allows to save up a great length each move. The formular for the 3D octile distance can be defined as the equation:

$$d = \Delta x + \Delta y + \Delta z - (3 - \sqrt{3}) * deltas[0] - (2 - \sqrt{2}) * deltas[1],$$

where Δx, Δy, Δz are differences coordinates, deltas[ ] is a sorted list of Δx, Δy, Δz. In the case of this problem, 3D octile distance defines the shortest path from nextPoint to goal that consists of diagonal moves.

5. References

- Stuart Russell and Peter Norvig. 2009. Artificial Intelligence: A Modern Approach (3rd ed.). Prentice Hall Press, Upper Saddle River, NJ, USA.
- Lê Hoài Bắc, Tô Hoài Việt. 2014. Giáo trình Cơ sở Trí tuệ nhân tạo. Khoa Công nghệ Thông tin.
- A* searching algorithm: https://www.geeksforgeeks.org/a-search-algorithm/
- 3D octile heuristic: http://jaylanzafane.com/wp-content/uploads/2017/06/Lanzafane_3D_Path_Planning.pdf