Search Tree Node ADT

The search tree node contains the following:

- State: initially as an object representing the state then we have replaced it with a string encoding of the state to optimize memory usage
- Parent pointer for the parent search tree node.
- The operator which is represented by string to store the operator applied on the parent to get the current search tree node.
- Depth representing the depth of the current search tree node in the search tree.
- A Comparable pathCost representing the actual cost from the root (the initial state) to the current search tree node.

Matrix problem

The matrix problem extends the abstract generic search problem and contains the following functionalities and attributes:

- InitialState which is initialized whenever the input grid is parsed in the static solve function. The description of the state in general will be on the following subsection.
- Successors function which takes a state, and expands it then returns a list of the successor states.
- isGoal takes a state and check whether or not it is a goal state. A state is considered a goal state iff all initial hostages are either returned to the telephone booth or killed after transforming to agents and Neo's location is the same as the telephone booth's location.
- PathCostFunction which takes the current state and return the cost of the path from the root till this state. The cost is defined as 1000*Number of hostages transformed to agents + number of agents killed. We have selected this cost function since we want to rescue as many hostages as possible which means that we need to minimize the number of transformed hostages to agents.

Then as a second priority (tiebreaker) we want to minimize on the number of agents killed. Consequently, we multiplied by the number of hostages transformed by 1000 which is a number greater than the maximum possible number of agents killed (at worst case your grid will be fill of agents 15*15=225 agents and 1000>225) so we give more weight for rescuing more hostages than killing less agents. However, if we have two plans and both of them rescue the same number of hostages then the cost will be minimum for the one which killed less agents. We consider a hostage to be transformed when its damage reaches 100 regardless whether this hostage was carried by neo or not unless it was dropped at the telephone booth. After that this total cost is multiplied by a big factor (1000,000,000) then we add to it an incremental index (we use the number of expanded nodes so far). We have done this because we want a third tie breaker in case of equal Number of hostages transformed to agents, and number of agents killed of two nodes this third factor, which is the index, will be a tie breaker. This is done to be consistent when dealing with equal costs since Java implementation of Priority Queue doesn't dequeue in order of insertion in case of equal costs.

State representation:

The state contains the following:

- Neo which is represented by the neo's location and damage.
- C which is a constant representing the number of hostages that neo can carry.
- Hostages which is a list containing all hostages that are alive or transformed but not yet carried or rescued.
- CarriedHostages which is a list containing all currently carried hostages even if they are transformed on the back of neo.

- TelephoneboothHostages which is a list containing all the hostages that have been rescued and dropped at the telephone booth since the beginning of the game.
- Number of hostages transformed to agents.
- Number of killed agents which is a general counter for all killed agents included those who are transformed from hostages.
- Agents which is a list of agents that are currently alive. An agent is represented by its location.
- Pills which is a list of pills that are currently available. A pill is represented by its location.
- Pads which is a list of pads. A pad is represented by its location and destination pads.
- Telephonebooth which is represented by its location in the grid.
- Matrix which is a grid that consists of cells that contain any one of the above mentioned objects. This is used only for the help of implementation and for the visualization of the grid.

Main functions

The following are the main functions that are not mentioned previously:

- Parse function which splits the input string representation of the grid on ";" then on each partition we have split it on "," to get individual values according to the schema described in the project description. Then we used these values to create the initial state that we pass to the search problem to be solved using the strategies (more about this to come).
- Expand function which expands the current state to a list of successors states by applying one of the operators to that state (if applicable). The operators are:

  - **carry hostage** if Neo and the hostage are on the same cell and remove that hostage from hostages list and add it to carried hostages list.
  - **drop hostages** if Neo is at the telephone booth cell and carries hostages that removes hostages from carried hostages list and adds it the telephone booth hostages list.
  - **take pill** if Neo is in a cell that contains a pill and decrease the damage of neo and all alive hostages by 20 and remove this pill from pills list.
  - **use pad** if Neo is in a cell that contains a pad.
  - **kill all agents** in adjacent cells if any and increase Neo's damage by 20 and remove these agents from agent list or from hostage list if they were transformed hostages.

after any of the previously mentioned actions except for taking a pill we increase the damage of all alive hostages by 2.

Algorithms Implementation
for all search algorithms, we handle the repeating visited states by adding the string encoding of the state to a hashset and before we add a new state to the queue we check if its string encoding doesn't exist in the hash set to avoid repeating states.

- **Breadth-First Search**:

  In this search strategy we initialize FIFO queue with the initial state then we keep on poping from the queue until either we find a goal state or the queue is empty. With every poped stated from the queue, we expand it and for all its sucessors if they are not in the already visited states, we add these states to the queue.
- **Depth First Search**:

In this search strategy we solve it recursively by first calling the function on the initial state. Then, if this state is the goal state or doesn't have any successors we return. otherwise, we expand this state and for each of the successors we check if it is not visited before. If so, we call the function recursively on that node if it returned an answer so we return it otherwise we keep on calling it on the other siblings.

- **Iterative Deepening Search**:

  we did excatly what we did with Depth First Search. However, we have added an extra terminating condition which terminates the recursive call of DFS if the depth has exceeded a certain threshold which is given as input. So, we call this enhanced version for all depths from 0 to infinity (MAX possible Integer)

- **Uniform Cost Search, Greedy Search, A* Search**:

  for these search strategies, we have implemented a single search function that sorts the nodes inside the queue according to a cost function that is passed as an input. We did excatly like we did with Breadth First Search. The only difference is that instead of using FIFO queue we used priority queue that gets the minimum value of the given cost function first.
  For uniform cost search we have passed the cost function as the path cost function (g(n)) explained above. For gready search, we have passed the cost function as the heurisitic function (h(n)) (more on that later). For A* search we have passed the cost function as the sum of the heuristic function and path cost function (g(n)+h(n)).

Heuristic functions

We have implemented two heuristics.

1. For heuristic function 1, we have chosen the number of hostages who have transformed into agents but not carried yet to be the value of the heuristic function. This function is an admissible heuristic function because the hostages transformed into agents but not carried must be killed to be able to reach a goal state. Moreover, since our cost function is 1000* number of hostages transformed(carried, dropped, or not taken yet)+ the number of agents killed so far at the current state. and since we won't reach a goal state without killing these transformed hostages, so we know that our goal state has a cost at least greater than the current cost by the number of hostages that must be killed because they will be counted as agents killed. We can know the number of hostages who have transformed into agents and have not killed or carried yet by going over the hostages list and check their damage. This is polynomial time in the size of the state representations O(length of hostages list)

2. For heuristic function 2, We have considered a relaxed problem which is the same as the current problem but without any agents and assuming that Neo can take all the pills without making any time step and for each alive hostage we check if we can not rescue this hostage alone without taking into consideration the other hostages. if we can not rescue it, then we can't rescue it along with other hostages. We count the number of hostages that we can't rescue in this relaxed problem. Since we know that if we can't them in the relaxed problem, then it is impossible to rescue them in the actual problem.
    - ➢ Let the number of hostages who have transformed (death) be at the current state H and Hg at the optimal goal.

- ➢ The number of agents killed be A and Ag at the optimal goal.
- ➢ The total number of hostages who can't be rescued in the relaxed problem X.

Since the cost at any node is 1000*H+A for any node. And since at the current node there is at least X hostages that will be transformed so H+X <= Hg since the actual cost at the goal is 1000*Hg+Ag and since so the actual cost from the current node to the optimal goal is (Hg-H)*1000+ (Ag-A). Since X+H<=Hg then Hg-H>= X then X which is our value for the heuristic is less than the cost from the current node to the optimal goal which is (Hg-H)*1000+ (Ag-A).

We first make a preprocessing step one time for the whole problem by computing the shortest path between any two pair of cells including the use of pads. To compute the shortest path, we have used Floyd Warshal Algorithm which is O(n^3) where n is the number of cells. Let the number of pills at the current state is p. For every hostage, we are sure that this hostage will be transformed if hostage damage - p* 20 + d*2 >= 100 where d is the shortest path from Neo's location to the hostage + the shortest path from the hostage location to the telephone both location + 1 which represents the action of carrying the hostage. This d is the minimum number of steps required to rescue the hostage. This computation is polynomial in the state representation. Since for computing the shortest path preprocessing O(n^3) where n is the number of cell and we iterate over all alive hostages O(length of hostages list)

Running Examples

To save space, we have uploaded the running example trace for two grids for each of the search strategies the output of grid 0 is here, and for grid 1 it will be here.

Performance Comparison

Completeness and Optimality

1. BFS is complete and not optimal since it doesn't take cost function into consideration.
2. DFS is complete in our implementation since we don't allow for repeated states and we have a finite branching factor. It is not optimal since it doesn't take cost function into consideration.
3. IDS is complete and not optimal for the same reason mentioned previously.
4. UCS is complete and optimal since our cost function is non decreasing positive function as we go deeper in the tree.
5. AS1, AS2 are complete and optimal efficient since the heuristic function is admissible.

6. GR1, GR2 are complete in our implementation since we don't allow for repeated states. They are not optimal since it doesn't take the actual cost function into consideration but only the heuristic which is not enough since it's not actual cost.

We will use the same two grids like the ones mentioned in the running examples. In all the following we will represent RAM usage as increase in usage from the initial RAM usage before running the program. For CPU utilization, we will present the difference in CPU utilization between before and after running the program.

- Grid 0
  - BFS
    RAM usage has increased by 6.94%. CPU utilization difference 22%. Number of expanded nodes 458.
  - DFS
    RAM usage has increased by 4.22%. CPU utilization difference 19%. Number of expanded nodes 14.
  - IDS
    RAM usage has increased by 5.55%. CPU utilization difference 40%. Number of expanded nodes 833.
  - UCS
    RAM usage has increased by 2.78%. CPU utilization difference 20%. Number of expanded nodes 151.
  - AS1
    RAM usage has increased by 6.94%. CPU utilization difference 18%. Number of expanded nodes 151.
  - AS2

RAM usage has increased by 5.17%. CPU utilization difference 16%. Number of expanded nodes 151.

  - GR1
    RAM usage has increased by 6.94%. CPU utilization difference 21%. Number of expanded nodes 203.

  - GR2
    RAM usage has increased by 6.94%. CPU utilization difference 15%. Number of expanded nodes 58.

- grid 1
  - BFS
    RAM usage has increased by 11.11%. CPU utilization difference 20%. Number of expanded nodes 2319.
  - DFS
    RAM usage has increased by 6.94%. CPU utilization difference 13%. Number of expanded nodes 70.

  - IDS
    RAM usage has increased by 6.94%. CPU utilization difference 30%. Number of expanded nodes 7782.

  - UCS
    RAM usage has increased by 11.11%. CPU utilization difference 25%. Number of expanded nodes 1255.

  - AS1
    RAM usage has increased by 6.94%. CPU utilization difference 15%. Number of expanded nodes 1255.

- AS2
  RAM usage has increased by 6.94%. CPU utilization difference 23%. Number of expanded nodes 1255.

- GR1
  RAM usage has increased by 4.17%. CPU utilization difference 35%. Number of expanded nodes 2376.

- GR2
  RAM usage has increased by 7.24%. CPU utilization difference 23%. Number of expanded nodes 101.

Comments

The number of expanded nodes in IDS in both grids is more than any other strategy including BFS since IDS applies depth limited search for the tree level by level so it will expand nodes more than that of BFS.

The number of expanded nodes in AS1, AS2 is less than or equal to the number of UCS which is consistent with that A* algorithm is optimally efficient.

In both running examples, the RAM usage, and CPU utilization of BFS is larger than that of DFS which is consistent with the space, and time analysis of the two algorithms since BFS is O(b^d) space and time complexity and DFS is O(b*d) in space and time complexity.

In both running examples, the RAM usage of IDS is less than that of BFS and greater than or equal to that of DFS. which is also consistent with the space complexity analysis of IDS since IDS space complexity is O(b*d).

In both running examples, the number of expanded nodes in DFS is less than that of BFS. We can conclude that the goal state is reached before expanding all nodes in the tree with levels shallower than that of the goal state.

In both running examples, the number of expanded nodes, and CPU utilization in GR2 is less than that of GR1. We may conclude that the heuristic function used in GR2 performed better than the one used in GR1 in these two examples.