National University of Singapore
School of Computing
CS3243 Introduction to AI

Tutorial 1: Agents, Problems, and Uninformed Search

SOLUTIONS

1. You are given an n-piece unassembled jigsaw puzzle set (you may assume that each jigsaw piece can be properly connected to either 2, 3, or 4 pieces), which assembles into an $m \times k$ rectangle (i.e., $n = m \times k$. There may be multiple valid final configurations of the puzzle. Figure 1 illustrates an example.

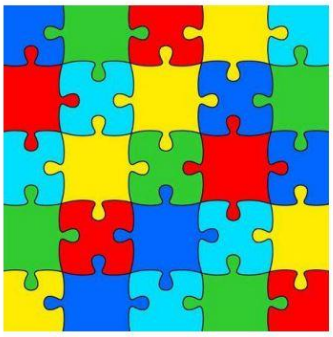


Figure 1: A sample configuration of the jigsaw puzzle.

Formulate the above as a search problem. More specifically, define the following:

- State representation
- Initial state
- Actions
- Transition model
- Action cost
- Goal test

If necessary, you may identify the assumptions you have made. However, assumptions that are contradictory to any instruction in the question, or that are unreasonable, will be invalid.

Solution:

- State representation:
  - Keeps track of the connections (i.e., 2/3/4 per piece) across all the puzzle pieces.
  - Maintain either the set of connected connections or unconnected connections.
- Initial state:
  - Depending on the representation, either the set of connected puzzle pieces is empty, or the set of unconnected puzzle pieces is full. There should initially be no connections between puzzle pieces.
- Actions:
  - Taking two unconnected puzzle pieces and establishing a connection between them. Note that you must mention that the pair of puzzle pieces picked can be legally connected (i.e., you cannot simply take ***any two*** puzzle pieces).
- Transition model:
  - Depending on the representation, can either add to the connected set or remove from the unconnected set.

- Action cost:
  - Action costs are 1 (or any positive constant value; cannot be 0).
- Goal test:
  - Depending on the representation, either the connected set is full, or the unconnected set if empty.
  - Check that for every puzzle piece, there should not be more connections than its number of available legal connections.

2. Prove that the **Uniform-Cost Search** algorithm is optimal as long as each action cost exceeds some small positive constant ε. You may also assume the same constraints that make Breadth-First Search complete.

Solution:

We are given the following assumptions.

- The branching factor, *b*, is finite.
- There is either a solution (i.e., reachable goal state), or the maximum depth, *m*, is finite.
- Each action cost exceeds some small positive constant ε.

We may therefore assume that completeness may be assumed[1]. Consequently:

- Whenever UCS expands a node *n*, the optimal path to that node has been found. If this was not the case (i.e., the path to *n* is not optimal – let us denote this path ***T***), there would have to be another frontier node *n*' on the optimal path from the start node to *n* (denote this optimal path ***U***). By definition, g(*n*) via ***U*** would be less than g(*n*) via ***T***, which implies that *n*' should have been selected first.
- If action costs are non-negative, path costs, i.e., g values never get smaller as nodes are added.

The above two points together imply that UCS expands nodes in the order of the optimal path cost.

[1] Do you recall why from the lecture?

3. (a) Describe the difference between **Tree Search** and **Graph Search** implementations.

Solution:

Graph-search algorithms will only explore non-redundant paths – i.e., they only explore nodes (i) with associated states that have not been visited, or (ii) have been visited, but previously via less optimal paths. There are many ways to implement this, with each implementation restricting some or all the redundant paths visited. However, tree-based algorithms have no such restriction; they consider all paths, including all redundant ones.

(b) Assuming that ties (when pushing to the frontier) are broken based on ascending alphabetical order (e.g., *A* before *B*), specify the order of the nodes checked (i.e., via the goal test) by the following algorithms. Assume that *S* is the initial state, while *G* is the goal state.

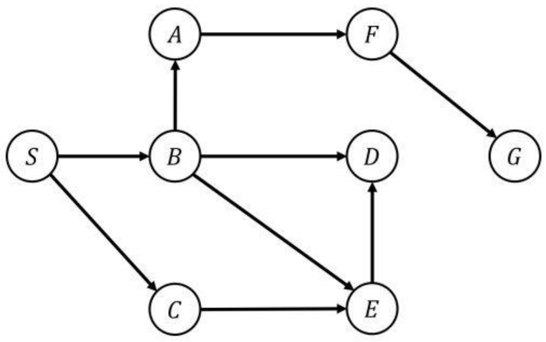


Figure 2: Graph representing the search space for Question **2b**.

You should express your answer in the form *S–B–A–F–G* (i.e., no spaces, all uppercase letters, delimited by the dash (–) character), which, for example, corresponds to the order, *S*, *B*, *A*, *F*, *G*.

i. **Depth-First Search** using a **tree-search** implementation.
ii. **Depth-First Search** using a **graph-search** implementation.
iii. **Breadth-First Search** using a **tree-search** implementation.
iv. **Breadth-First Search** using a **graph-search** implementation.

Solution:

i. S–C–E–D–B–E–D–D–A–F–G
ii. S–C–E–D–B–A–F–G
iii. S–B–C–A–D–E–E–F–D–D–G
iv. S–B–C–A–D–E–F–G

4. You have just moved to a strange new city, depicted via the graph in Figure 3, and you are trying to learn your way around. More specifically, you wish to learn how to get from your home at *S* to the train station at *G*.

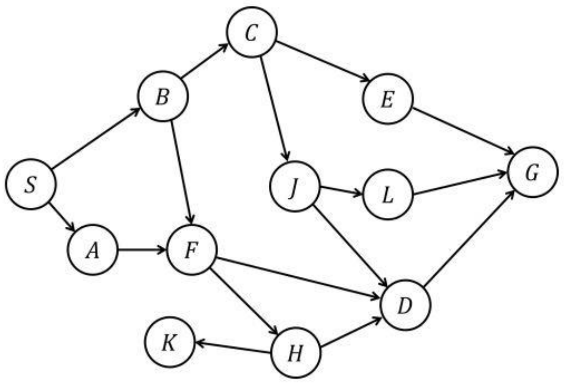


Figure 3: A graphical representation of the strange new city.

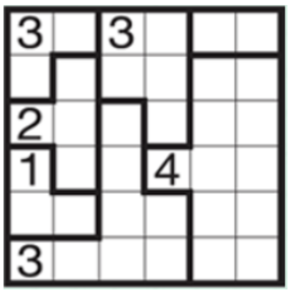
Apply the **Depth-First Search** algorithm with a **tree-search** implementation. Use ascending alphabetical order to break ties when deciding the priority for pushing nodes into the frontier (e.g., *A* before *B*).

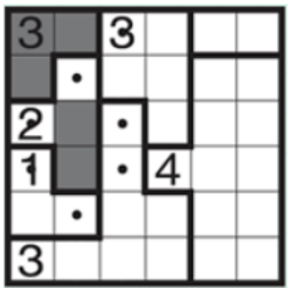
Determine the final path found from the start *S* to the goal *G*.

Note that you ***MUST*** express your answer in the form *S–B–C–J–L–G* (i.e., no spaces, all uppercase letters, delimited by the dash (–) character), which, for example, corresponds to the exploration order of *S*, *B*, *C*, *J*, *L*, then *G*.

Solution:
S–B–F–H–D–G

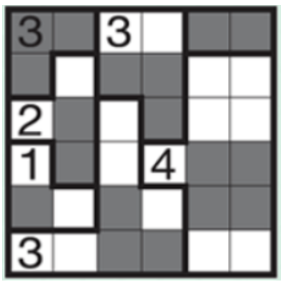
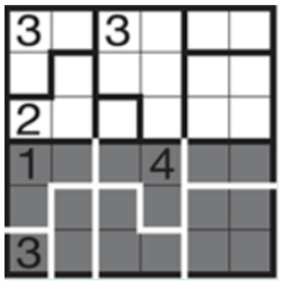
5. **[AY22/23 S3 Midterm Q1]** A certain puzzle game is played on a square *n* by *n* grid (i.e., a grid with *n* rows and *n* columns). The grid is separated into *k* contiguous regions. Each region, $\boldsymbol{R}_i$ (where $i \in [1, k]$), has an associated positive integer value, v_i. An example of this puzzle where $n = 6$ and $k = 7$ is depicted on the right. Notice that each of the *k* regions is bounded by a thick line, with the value for the region stated in the region. However, notice that some regions (e.g., the region in the top-right of the grid) have no associated value – such regions have variable values ranging from 0 to the size of the region (i.e., the number of cells in the region).





To solve this puzzle, one must fill contiguous cells in the region such that for region $\boldsymbol{R}_i$, the number of contiguous cells filled is v_i. When filling cells in this manner, there is a further constraint: filled cells cannot be contiguous across the regions. Continuing from the example puzzle above, suppose that we fill the first two regions in the top-left of the grid. The figure on the left shows a possible partial solution. The shaded cells denote filled cells. Notice that these filled cells only exist within the specified regions and are contiguous. The cells containing dots denote the cells that cannot be filled, since filling them would cause the solution to become inconsistent with the constraint that requires filled cells to not be contiguous across different regions.

Next, we must first fill all the regions (though you may choose to leave regions with no value empty). The puzzle is solved if the following constraint is satisfied. When all filled cells "fall" to the bottom of the grid, retaining their shape and not coexisting in cells with other filled cells, the bottom half of the grid must be completely filled. A solution to the example puzzle shown above is thus as follows.

 ← After all regions are filled but before the filled cells have fallen	 ← After cells have fallen, notice that the fallen cells are perfectly tessellated

Assume that the puzzle input is a list of 2-tuples, $\boldsymbol{P} = \{(\boldsymbol{R}_1, v_1), (\boldsymbol{R}_2, v_2), \ldots, (\boldsymbol{R}_k, v_k)\}$, where each $\boldsymbol{R}_i = \{(r_{i,1}, c_{i,1}), (r_{i,2}, c_{i,2}), \ldots\}$, and where each $(r_{i,j}, c_{i,j})$ refers to the row and column coordinates respectively for the *j*-th cell in the *i*-th region $\boldsymbol{R}_i$. Note that the sequence of row-column coordinates in each $\boldsymbol{R}_i$ is sorted primarily in terms of descending order of row coordinate value, and secondarily in terms of ascending order of column coordinate value – i.e., cells on the top row will be ordered before cells on lower rows, and within each row, cells are ordered from left to right; however, note that there is no specific order over the regions. Finally, note the following properties for $\boldsymbol{P}$.

- For the bottom row, the leftmost cell is (1, 1), while the rightmost is (1, *n*).
- For the top row, the leftmost cell is (*n*, 1), while the rightmost is (*n*, *n*).
- Each $v_i \leq |\boldsymbol{R}_i|$, $\sum_{i=1}^{k} v_i$, $= n^2/2$, and for cases where $\boldsymbol{R}_i$ has no value (i.e., is variable from 0 to $|\boldsymbol{R}_i|$), then $v_i = null$.

Thus, for the example puzzle above, we have $\boldsymbol{P} = \{(\boldsymbol{R}_1, 3), (\boldsymbol{R}_2, 3), (\boldsymbol{R}_3, \text{null}), (\boldsymbol{R}_4, 2), (\boldsymbol{R}_5, 4), (\boldsymbol{R}_6, 3), (\boldsymbol{R}_7, 1)\}$, where: $\boldsymbol{R}_1 = \{(6, 1), (6, 2), (5, 1)\}$; $\boldsymbol{R}_2 = \{(6, 3), (6, 4), (5, 3), (5, 4), (4, 4)\}$; $\boldsymbol{R}_3 = \{(6, 5), (6, 6)\}$; $\boldsymbol{R}_4 = \{(5, 2), (4, 1), (4, 2), (3, 2)\}$; $\boldsymbol{R}_5 = \{(5, 5), (5, 6), (4, 5), (4, 6), (3, 4), (3, 5), (3, 6), (2, 5), (2, 6), (1, 5), (1, 6)\}$; $\boldsymbol{R}_6 = \{(4, 3), (3, 3), (2, 3), (2, 4), (1, 1), (1, 2), (1, 3), (1, 4)\}$; and $\boldsymbol{R}_7 = \{(3, 1), (2, 1), (2, 2)\}$.

Your task is to model the puzzle as a search problem so that we may apply uninformed, informed, or local search algorithms to solve the problem. In doing this, you are to assume the following partial search problem formulation.

- **State**: each state corresponds to partially assigned solutions, $\boldsymbol{S} = \{\boldsymbol{F}_1, \boldsymbol{F}_2, \ldots, \boldsymbol{F}_k\}$, where each $\boldsymbol{F}_i \subseteq \boldsymbol{R}_i$, $|\boldsymbol{F}_i| \leq v_i$, and such that all coordinates in $\boldsymbol{F}_i$ are contiguous.
- **Initial State**: with the initial state, each $\boldsymbol{F}_i \in \boldsymbol{S}$ is empty, i.e., $\forall \boldsymbol{F}_i \in \boldsymbol{S}, |\boldsymbol{F}_i| = \emptyset$.
- **Action Costs**: all action costs correspond to a positive constant – e.g., 1.
- **Goal Check**: this function, *isGoal*($\boldsymbol{S}$), returns *True* if $\forall \boldsymbol{F}_i \in \boldsymbol{S}, |\boldsymbol{F}_i| = v_i$, and after all the filled cells have fallen, the bottom half of the grid is completely filled. Note that this function automatically returns *False* if $\exists \boldsymbol{F}_i \in \boldsymbol{S}$ where $|\boldsymbol{F}_i| \neq v_i$.

(a) Complete the search problem formulation by defining the **actions** and **transition model**. Your formulation must be efficient (i.e., significantly better than brute-force search).

Solution:

Actions

- This function, *actions*($\boldsymbol{S}$), must determine the specific v_i coordinates to add to each $\boldsymbol{F}_i$.
- This is a ***goal search*** problem, so a path is not important. Consequently, the size of the search tree may be reduced by not permuting over assignment order (similar to CSPs). Let the ordering be $\boldsymbol{F}_1$ on level 1, $\boldsymbol{F}_2$ on level 2, and so on. This also means that *actions*($\boldsymbol{S}$) must be able to access the current depth.
- For each $\boldsymbol{F}_i$, there are two general cases:
  - Case 1: $v_i \neq null$
    find all possible $\boldsymbol{F}_i \subseteq \boldsymbol{R}_i$ where $|\boldsymbol{F}_i| = v_i$ and the coordinates are contiguous.
  - Case 2: $v_i = null$
    find all possible $\boldsymbol{F}_i \subseteq \boldsymbol{R}_i$ where $|\boldsymbol{F}_i| \leq |\boldsymbol{R}_i|$ and the coordinates are contiguous.
- Decompose *actions*($\boldsymbol{S}$) such that it will utilise a *candidates*($\boldsymbol{R}_i$, v_i) function that finds all legal combinations for $\boldsymbol{F}_i$ under Case 1 above. Consequently, when Case 2 is observed, *candidates*($\boldsymbol{R}_i$, v_i) must be called iteratively, replacing v_i with the range $[0, |\boldsymbol{R}_i|]$.
- The function *candidates*($\boldsymbol{R}_i$, v_i) can be implemented as a nested search.
  - First, determine the adjacency matrix over the coordinates in $\boldsymbol{R}_i$. This can be done by linking any coordinate with another if the row difference or the column difference, exclusively, is exactly 1. This requires $|\boldsymbol{R}_i|(|\boldsymbol{R}_i|+1)/2$ comparisons.
  - Next, perform a Depth-limited Search (DLS) with each coordinate in $\boldsymbol{R}_i$ as the root (i.e., $|\boldsymbol{R}_i|$ DLS executions). The depth limit is v_i. The paths at all leaf nodes at depth $v_i - 1$ (since the root is at depth 0) correspond to candidates for $\boldsymbol{F}_i$. Note that each such execution of DLS must return all paths at depth $v_i - 1$.
  - Note that this function is only called exactly k times (as preprocessing, since we can simply reference all possible legal permutations of $\boldsymbol{F}_i$ after generation).

Transition Model

- Simply update the current $\boldsymbol{F}_i$ (where $\boldsymbol{F}_i = \emptyset$; recall the CSP-like ordering) with a valid set of coordinates (i.e., one of the sets output by *candidates*($\boldsymbol{R}_i$, v_i)).

(b) Define the runtime complexity of the ***actions*** function you specified in Part **(a)** and define the size of the resultant search tree.

Solution:

Let N be the size of the grid (i.e., $N = n^2$).

The runtime complexity of the actions function requires the following.

- Generation of the adjacency matrix: $|\boldsymbol{R}_i|(|\boldsymbol{R}_i|+1)/2$ comparisons or $O(N^2)$.
- Generation of all candidates for $\boldsymbol{F}_i$ via DLS:
  - Let M be the size of the region in question (i.e., $M = v_i$).
  - Each $\boldsymbol{F}_i$ requires M DLS executions.
  - Each DLS execution has complexity $O(4^{M-1})$ since the depth limit is $M - 1$.
  - Runtime complexity is thus: $O(M \cdot 4^{M-1})$ or $O(N \cdot 4^{N-1})$.

Thus, the ***actions*** function has a runtime complexity of $O(N^2) + O(N \cdot 4^{N-1})$, or just $O(N \cdot 4^{N-1})$.

Note that this has lower complexity than brute-force search, which has complexity $O(k \cdot N!)$. This is because you must find the $|\boldsymbol{R}_i|$ choose v_i subsets.

The size of the search tree is bounded by $O((N \cdot 4^N)^k)$.

- At each level there are $O(N \cdot 4^N)$ possible options for $\boldsymbol{F}_i$; this is the branching factor.
- There are k levels in this tree.

Note that the bounds above can be defined more tightly by considering that $\sum|\boldsymbol{R}_i| = N$.