CSC361: Bridge And Flashlight Problem

King Saud University

Authors

Mohand Al-Rasheed,
Abdulaziz Al-Jamhour,
Abdulmalik Al-Argani

Contents

1 Introduction

1.1 Problem Description

A group of K (K > 1) people wish to cross a bridge from the east to the west, the bridge is narrow and limits only two people to cross concurrently, furthermore, it's night and crossing the bridge requires a flashlight that the group has one of. Each person has a speed S_k, when more than one person crosses the bridge, they go with the speed of the slowest person. The flashlight has to be carried from west to east for more people to use it. Find the sequence of steps that minimize the crossing time.
One instance of the problem could be $K = 3, S = [3, 2, 5]$ (Note: we will reference the person with speed S_k as a_k, starting from k=0), the output should be

1. a_0, a_1 *move to the west side*
2. a_1 *returns with the flashlight*
3. a_1, a_2 *move to the west side*

The output should include a summary of evaluation metrics as well.

1.2 Problem State Space

The state space of this problem is all the possible positions of people and the position of the flashlight. We can calculate the number of states by 2^{k+1} since every person can be on the east or the west and the flashlight can be on the left or on the right.

2 Representation

2.1 State Representation

Given the problem structure, we can represent any one person being on the east or the west by a Boolean value, where *False* represents the person being on the east and *True* represents the person being on the west. The same logic can be applied to the flashlight position. We bundle all the positions of people in an array and include the flashlight position, this structure can fully represent the state at any given position.

Initial State The initial state for this problem is *(False, False, ..), False)*

Goal State The goal state for this problem is *(True, True, ..), True)*

2.2 Data Structures

In order for us to solve this problem using searching algorithms, we have to define specific data structures that allow us to traverse the state space efficiently, we create a *Node* and a *Problem* structures to house the state and problem inputs respectively.

2.2.1 BridgeNode

In each node we store a couple of different values that are useful to us

1. State
2. Parent Node
3. Depth
4. Path Cost

We also define the comparison operators between nodes to compare the path cost of each node

2.2.2 BridgeProblem

In the problem structure we store the inputs of the problem as well as the initial node and also several valuable functions.

Successor function By expanding a node we return nodes that success it, for example given the initial node with state *[False, False], False*, states that success it are:

1. *[True, False], True*
2. *[False, True], True*
3. *[True, True], True*

An algorithm for finding successor states can be formulated as follows:

```
# Input: State
# Output: List of States
m <- min_num_ppl_crossing
n <- max_num_ppl_crossing

positions, flash_is_west <- start_state
result <- []
legal_positions <- []

for i in 0..length(positions)-1:
    if positions[i] = flash_is_west:
        legal_positions.add(i)

for comb_indices in
    combinations(legal_positions, r) for r in
    m..n:
    newPositions <- newPositions
    for index in comb_indices:
        newPositions[index] <-
            !newPositions[index]
    result.add(State((newPositions, not
        flash_is_west)))
return result
```

By modifying m and n we can generalize the problem such that the minimum number of people moving each turn isn't 1 but m, the same thing for the maximum number of people moving each turn.

Goal Test We have to check weather a node is the goal state or not. Checking for goal state involves checking that every person made it to the left of west, i.e. every position in the node's state is True. Here's the algorithm for goal test

```
        positions <- node.state[0] # Positions
        for position in positions:
            if position is False:
                return False
        return True
```

2.2.3 Additional Data Structures

We used *Priority Heaps* to organize our search based on the priority of any given node (more on that in the Algorithms section). We also used structures to keep & count all the evaluation data (e.g. *Solution Cost*, etc.).

2.3 Repeated States

By expanding any given node all the nodes inside can be expanded to return to the original node, which can turn a linear problem to an exponential one. We use this fact to optimize some of our algorithms and compare the running time of each later.

3 Algorithms

In each of the following algorithms we used the implementation detailed in *A Modern Approach, Acting rationally*[1].

Uniform Cost Search Uniform Cost Search (abbreviated UCS) is a node expansion strategy that expands the nodes with the least cost, we used a *Priority Heap* data structure in organizing the nodes in the queue, where the priority is gotten from the *path cost* in the node's data.

Iterative Deepening Search Iterative Deepening Search (abbreviated IDS) is a depth first search variant that expands nodes starting from $Depth = 1$ until $Depth = \infty$ recursively, which saves up memory cost immensely.

A* Search A* Search or A* uses the *path cost* as well as a value $h(node.state)$ which allows it predict the node with the most potential. We arrived at a heuristic function h by relaxing the flashlight constraint then calculated the cost. This allows for an admissible heuristic.
The function approximates the remaining cost by taking the minimum cost possible on the east side of the bridge. This ensures optimally of the result[1].
Here's the algorithm

```
# Input: State, problem
# Output: Integer (Cost)
costs <- []
for index in 0..length(state.positions)-1:
    if !state.positions[index]:
        costs.add(problem.cost_arr[index])

if length(costs) = 0:
    return 0

costs = sort_ascending(costs)
if length(costs) % 2 != 0:
    costs.append(0)
cost = 0

for chunk in chunks_of_two(costs):
    cost += chunk[0]

return cost
```

4 Results

4.1 Performance

We calculated the performance of each algorithm on three dimensions

1. *Solution Cost*: The number of nodes in the solution
2. *Search Cost*: The number of nodes generated during search
3. *Space Requirement*: The number of nodes concurrently in the priority queue during search

We then optimized UCS, IDS and A* to ignore repeated states since they make no difference in the search and take a lot of unnecessary time and memory.

4.1.1 Performance

		UCS	IDS	A*
Test Case	Metric			
(1 2)	Solution Cost	2	2	2
	Search Cost	1	3	1
	Space Requirement	1	2	1
	Time (s)	3.82e-05	3.49e-05	4.71e-05
(3 2 5)	Solution Cost	10	11	10
	Search Cost	15	22	14
	Space Requirement	4	4	4
	Time (s)	0.0001595	0.000154	0.0001857
(1 3 2 5)	Solution Cost	12	12	12
	Search Cost	66	122	63
	Space Requirement	9	6	8
	Time (s)	0.0004984	0.0007511	0.0007958
(1 9 2 1)	Solution Cost	13	14	13
	Search Cost	66	122	61
	Space Requirement	9	6	9
	Time (s)	0.0005281	0.0008551	0.0006944
(3 3 3 3)	Solution Cost	15	15	15
	Search Cost	66	122	66
	Space Requirement	7	6	7
	Time (s)	0.0005001	0.0007254	0.0008629

4.1.2 Solutions

4.1.2.1 UCS

	(1 2)	(3 2 5)	(1 3 2 5)	(1 9 2 1)	(3 3 3 3)
0	Move a0, a1	Move a0, a1	Move a0, a2	Move a0, a3	Move a0, a2
1		Return a1	Return a0	Return a3	Return a2
2		Move a1, a2	Move a0, a1	Move a1, a2	Move a1, a2
3			Return a0	Return a0	Return a1
4			Move a0, a3	Move a0, a3	Move a1, a3

4.1.2.2 IDS

	(1 2)	(3 2 5)	(1 3 2 5)	(1 9 2 1)	(3 3 3 3)
0	Move a0, a1	Move a0, a1	Move a0, a1	Move a0, a1	Move a0, a1
1		Return a0	Return a0	Return a0	Return a0
2		Move a0, a2	Move a0, a2	Move a0, a2	Move a0, a2
3			Return a0	Return a0	Return a0
4			Move a0, a3	Move a0, a3	Move a0, a3

4.1.2.3 A*

	(1 2)	(3 2 5)	(1 3 2 5)	(1 9 2 1)	(3 3 3 3)
0	Move a0, a1	Move a0, a1	Move a0, a2	Move a0, a3	Move a0, a2
1		Return a1	Return a0	Return a0	Return a2
2		Move a1, a2	Move a0, a1	Move a1, a2	Move a1, a2
3			Return a0	Return a3	Return a1
4			Move a0, a3	Move a0, a3	Move a1, a3

4.2 Additional Test Cases

Test Case	Metric	UCS	IDS	A*
(9 9 9 9 9 9 30)	Solution Cost	120	120	120
	Search Cost	1757	3772	1757
	Space Requirement	62	12	52
	Time (s)	0.0328977	0.0215698	0.0320076
(44 55 66 33 8 1)	Solution Cost	152	429	152
	Search Cost	640	1398	345
	Space Requirement	41	10	64
	Time (s)	0.0088175	0.0077256	0.008262
(5 6 4 3 3 4 5 6000)	Solution Cost	6039	6061	6039
	Search Cost	4564	9359	4519
	Space Requirement	146	14	125
	Time (s)	0.145964	0.0540114	0.165986
(50 60 40 30 30 40 50 6)	Solution Cost	336	660	336
	Search Cost	4564	9359	4456
	Space Requirement	116	14	136
	Time (s)	0.146448	0.0553102	0.17063
(10 100 1000 10000 1000 100 10 1)	Solution Cost	11164	12280	11164
	Search Cost	4564	9359	4458
	Space Requirement	186	14	129
	Time (s)	0.150916	0.0546698	0.175499
(100 2 3 4 5 60 7 8 9)	Solution Cost	149	1500	149
	Search Cost	11465	22875	5780
	Space Requirement	381	16	560
	Time (s)	0.710664	0.13129	0.764477
(3 3 3 4 4 4 4 8 8)	Solution Cost	52	59	52
	Search Cost	11465	22875	11235
	Space Requirement	230	16	325
	Time (s)	0.703197	0.131648	0.838294
(10 20 30 40 50 60 70 80 90 100)	Solution Cost	500	620	500
	Search Cost	28093	51812	27511
	Space Requirement	472	18	529
	Time (s)	3.22846	0.343196	4.01714
(100 90 80 70 60 60 70 80 90 100)	Solution Cost	1120	1700	1120
	Search Cost	28095	51812	28045
	Space Requirement	346	18	503
	Time (s)	2.61024	0.303756	3.23587
(5 5 5 5 5 5 5 5 5 5)	Solution Cost	85	85	85
	Search Cost	28095	51812	28095
	Space Requirement	368	18	368
	Time (s)	2.29129	0.304314	2.29675

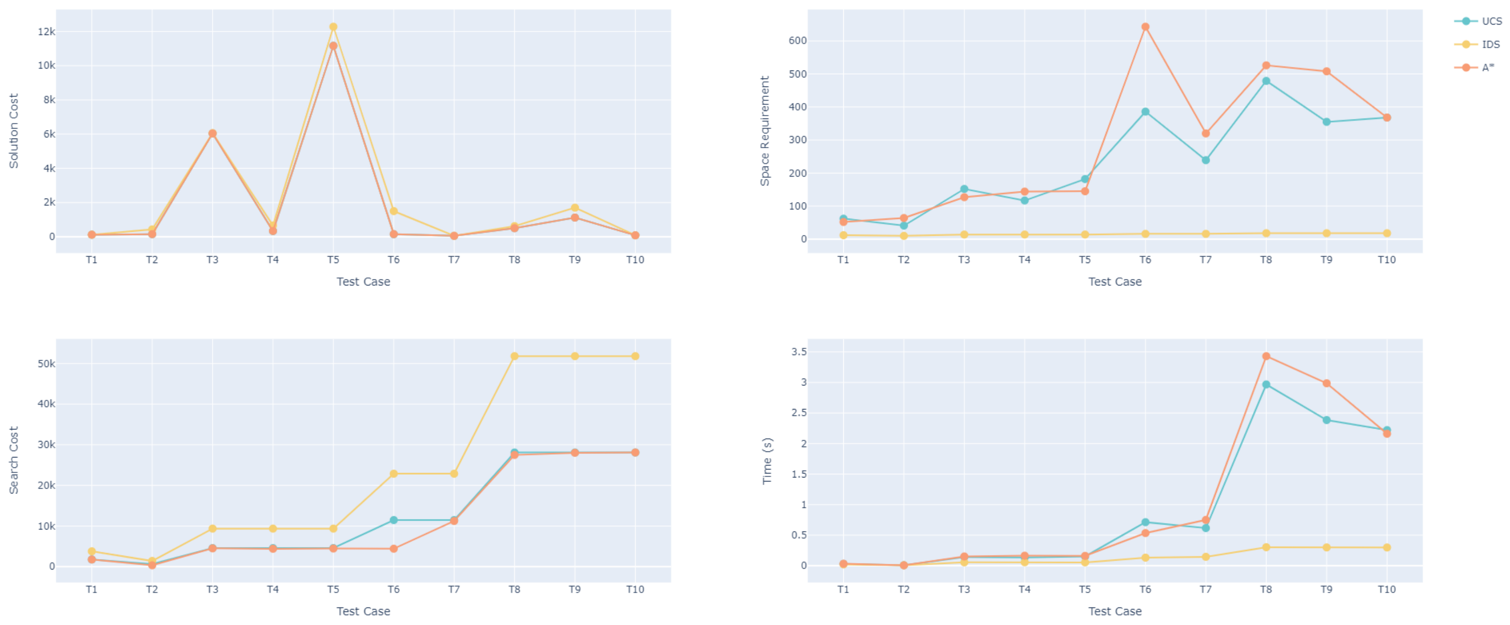


Figure 1: Plot additional Test Cases

4.3 Visualizations

Here we will visualize some of the different subsets of the state space. The label of any node is two parts, the number before the underscore signifies when the algorithm generated this node and what's after the underscore is a compact representation of the state at the current node, such that 0 at the ith index means the a_0 is at the east, and a 1 at the ith index means a_i is at the west.

4.3.1 Solution Tree Visualization

We can visualize a subset of the state space for us to have a better idea of what the algorithm is trying. Here is a visualization of all nodes visited by each algorithm.

UCS The test case visualized is *(3 2 5)*

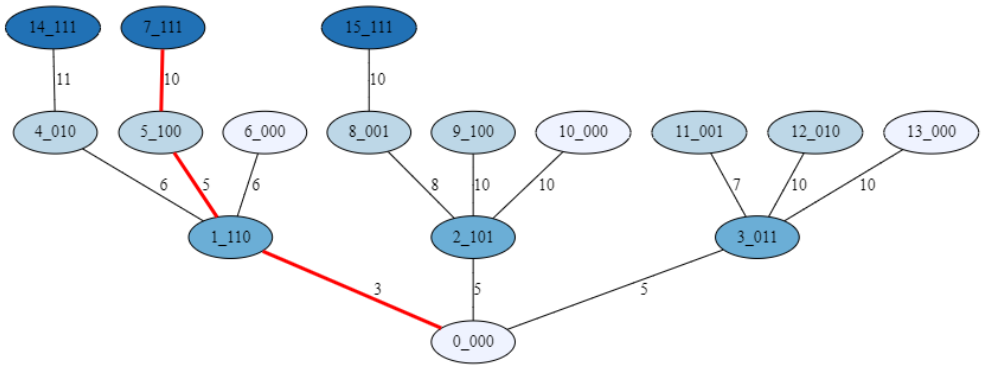


Figure 2: Uniform Cost Search expands the least cost node

IDS The test case visualized is *(3 2 5)*

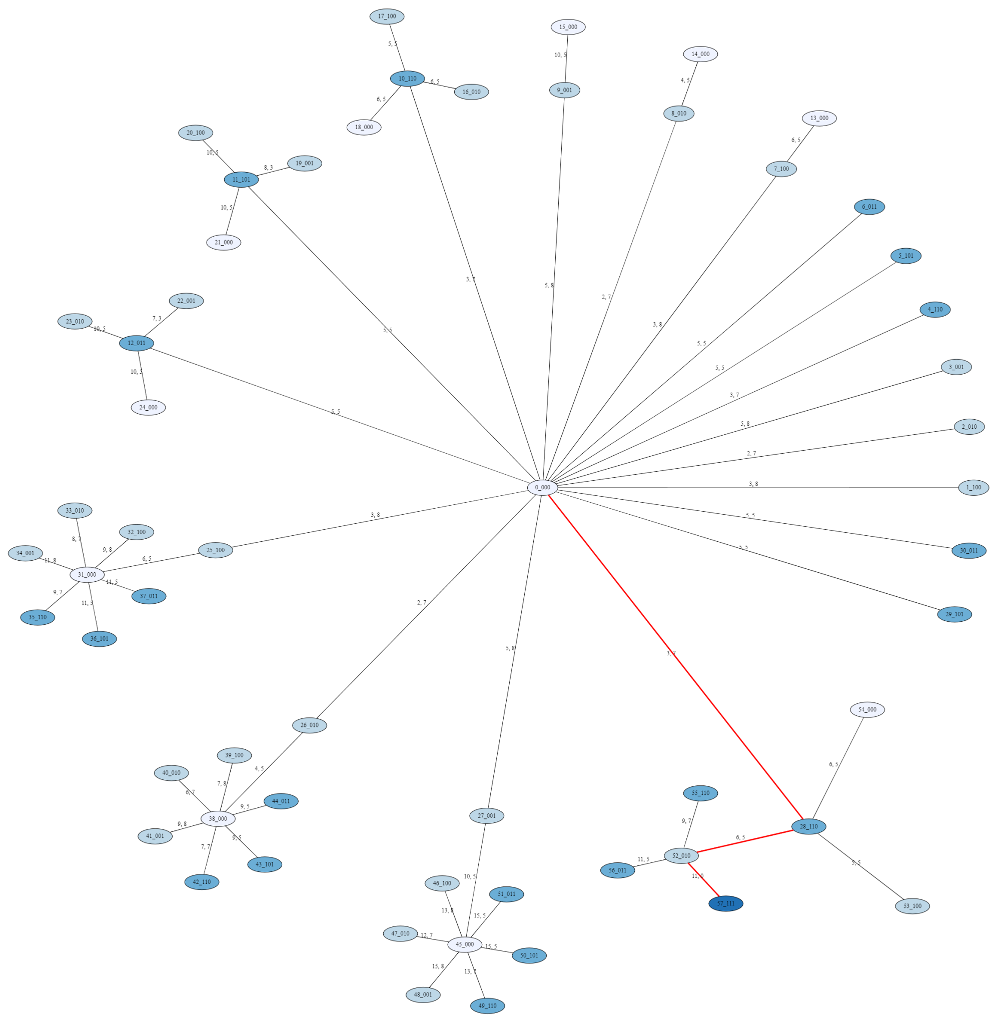


Figure 3: Iterated Deepening Search recursively repeats searching to conserve memory

A* The test case visualized is *(3 2 5)*

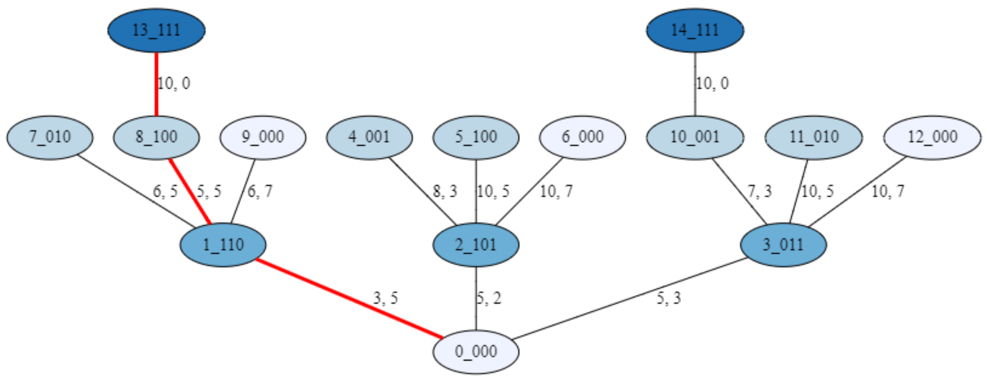
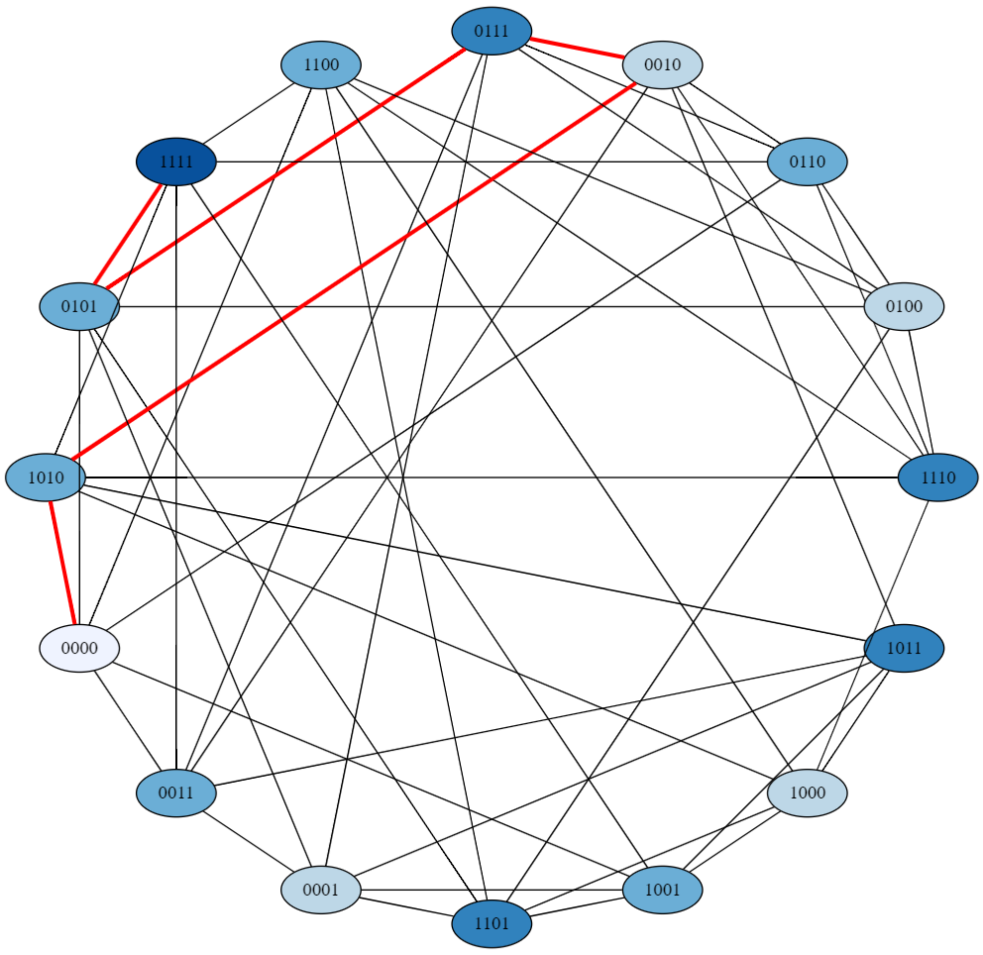


Figure 4: A* Uses a heuristic to predict the real cost

we can see the left most node is not visited by A* even though UCS visits that node, the reason for that is that the predicted cost of visiting it heavily outweighs the cost of visiting other nodes.

4.3.2 State Space Visualization

We can merge nodes with similar states to have a better idea of the problem the algorithms are traversing. The test case visualized is *(2 3 2 5)*



5 Conclusions & Future work

Searching algorithms vary in their performance, we concluded that A* has the slow run-time (given our heuristic function), and Iterated Deepening Search has a low memory usage which is sometimes essential, though heavily lacking in speed. Optimizing for repeated states adds enormous amount of speedup to our algorithms.

5.1 Future Work

Generalized Bridge And Flashlight Problem We can generalize the Bridge And Flashlight problem by changing the minimum number of people crossing the bridge m or increasing the number of maximum number of people crossing n, not all instances of this generalization are solvable, though some pose an interesting problem.

Bidirectional Search Given this problem's goal, we can run a bidirectional search that backtracks from the solution and also traverses the tree from the initial node, this can massively cut-down on the complexity growth.

References

1. Artificial Intelligence: A Modern Approach, *Searching Strategies*. Peter Norvig and Stuart J. Russell